Einführung in die Informatik

Stand
20.7.2009

Volker Claus, Universität Stuttgart, Studienjahr 2008/09

Die Grundvorlesung Informatik bleibt in diesem Jahr noch "klassisch" ausgerichtet. Sie betont die sequenzielle Herangehensweise und stellt Vorgehensweisen zum Problemlösen, die Begriffe Algorithmus und Datenstruktur sowie deren Darstellung, Ausführung, Verwaltung und Eigenschaften in den Mittelpunkt.

Die Erfahrung hat gezeigt, dass in der Vorlesung das "handwerkliche" Programmieren für die meisten Teilnehmer(innen) den Lernerfolg wesentlich bestimmt. Daher startet die Vorlesung mit einer Einführung in eine Programmiersprache (hier: Ada), wobei bis zum Ende des ersten Semesters etwas mehr als der sog. "PASCAL-Kern" vermittelt wird. Dadurch können die meisten Konzepte an konkreten Aufgaben nachvollzogen und Lösungen mit dem Rechner verifiziert werden.

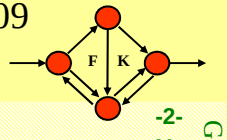
Großer Wert wird auf die Abstraktionsfähigkeit und Formalisierungen, auf "Modelle im Kopf" sowie auf die Kenntnis vieler Beispiele gelegt, an denen sich die Ideen und Konzepte konkretisieren.

Der in dieser Ausarbeitung vorliegende Stoff wird zu etwa 70% in den 112 Vorlesungsstunden des Studienjahrs 2008/2009 behandelt. Er wird auf rund 2200 Folien dargestellt, die Ihnen den Einstieg in die Wissenschaft Informatik vermitteln sollen.

Einführung in die Informatik

Universität Stuttgart, Studienjahr 2008/09

Gliederung der Vorlesung



0. Organisation und Studium

0.0 Vorbemerkungen

In 9 Monaten (und heute ist der 13.10.2008) werden Sie

- mehr über Programmierung und Datenstrukturen,
- mehr über mathematische und theoretische Grundlagen,
- mehr über den Aufbau von Rechnern wissen;
- gelernt haben, sich selbst besser einzuschätzen;
- eigene Schwächen/Stärken genauer kennen.

Es gibt fachliches Wissen und Können,
und es gibt die fachübergreifenden Fähigkeiten.

Das bringen wir Ihnen bei.

Diese müssen Sie überwiegend selbst entwickeln, aber wir fördern dies.

Hierzu gehören:

Arbeitsorganisation: Eigener Arbeitsstil, Disziplin, die Zeit genau einteilen können, Verlässlichkeit, Planung und Vorbereitung, Zusammenarbeit, Arbeit strukturieren, ...

Lernumgebung: Zeit reservieren, sich überwachen und testen, kooperatives Lernen, selbst lernen können, eine angemessene Atmosphäre zum Lernen herstellen und möglichst mit allen Sinnen lernen, ...

Persönlichkeitsmerkmale: Ausdrucksfähigkeit, Zuhören können, Bewerten können, Fremdsprache, konsequent sein, in die Breite und Tiefe denken, Durchhalten können, Verantwortung übernehmen, ...

Prioritäten setzen: Wichtiges im Studium identifizieren, Ziele festlegen, Mitstudierende suchen, sich mit ihnen austauschen und Lerngruppen bilden, eigene Motivation erzeugen, Studium als Persönlichkeitsbildung begreifen, ...

Wir sprechen es sofort an: Viele scheitern in ihrem Studium, nicht nur in der Informatik. Zu den Gründen gehören:

- Falsche Vorstellungen von den Inhalten des Studiums,

- nur Interesse an der heutigen Berufspraxis,

- Faulheit oder kein innerer Antrieb,

- Unorganisiertheit beim Arbeiten und Lernen.

Organisieren Sie deshalb Ihr Studium. Studium ist nämlich ein "Beruf", in dem Sie sich an Ihre Arbeitszeiten genau so halten und Leistungen erbringen müssen wie in jedem anderen Beruf. Überwachen Sie sich daher und vermeiden Sie Ignoranz, Vertagen des Lernens auf spätere Termine, falsche Prioritätensetzung usw.

Nach unserer Erfahrung scheitern die meisten nicht am Fach, sondern an der Unfähigkeit, sich schnell und konsequent einen eigenen Lernstil und eine eigene Arbeitsumgebung zu schaffen.

0.1 Zum Lernen

Wie lernen Sie?

Kennen Sie Ihre "optimale" Lernmethode?

Ist je nach Stoff eine andere Methode sinnvoll? Zum Beispiel:

Vokabeln: pauken
Bedienung einer Maschine: im Dialog mit Lehrpersonen
Mathematikaufgaben: alleine lösen durch Nachdenken
Umweltprobleme: in der Gruppe mit Arbeitsteilung
politische Fragen: in Gruppendiskussionen, Stammtisch

Geht das Lernen so weiter wie bisher? **Nein!**

Ab jetzt lernen Sie für *Ihre* Zukunft.

Lernen ist extrem komplex.

Lernen durch Instruktion
Lernen durch Vormachen und Nachvollziehen
Lernen durch Selbst Entdecken und Ausprobieren
Lernen durch Beobachten der Umweltreaktionen
Lernen durch Fehler und systematische Korrektur
Lernen durch Deduktion aus Abstraktem
Lernen durch Konstruieren, Modellieren, Simulieren
Lernen durch Weitervermitteln

Lernen durch schrittweise Vertiefung
Lernen durch Wiederholen
Lernen durch Anwenden
Lernen durch Variantenbildung

Lernen allein, Lernen in der Gruppe, Lernen gegen die Gruppe, Lernen durch asynchronen Austausch, Lernen durch Nachfragen, ...
Lernen durch Vereinfachen
Lernen durch Rückführen auf Bekanntes
Lernen durch Vorführen
Lernen durch Nachahmen, Vorbilder

Was soll gelernt werden?
Überprüfen des Gelernten (alleine, mit anderen, durch Prüfungen (aber die sind in der Uni selten), durch Üben mit Büchern, ...)
Welche Voraussetzungen (Inhalt, Fähigkeiten)?
Welche (stoffabhängige) Vorgehensweise?
Welche Lernumgebung? (Hilfsmittel, Werkzeuge, Gruppen, ...)

Brauche ich Hilfe? Lerne ich das Richtige? Schätze ich mich und mein Wissen richtig ein? Wie gut sind andere? Sollte ich ein anderes Fach studieren?
Warum lerne ich? Motivation, Gelderwerb, anderen helfen, Zwang/Bestrafung, Belohnung, ...

In jedem Studiengang stehen andere Lernstrategien im Vordergrund

Zum Beispiel *Softwaretechnik:*

Lernen durch Konstruieren

Lernen durch Modellieren, Experimentieren, Simulieren

Lernen durch Fehler-Machen !!

Lernen im Team (≠ in einer unverbindlichen Gruppe)

Lernen an realistischen Problemen (keine „Comicwelten“, Projekte, Konkurrenz, Kunden, Ressourcen ...)

Lernen von Regeln, Normen, Protokollen, ... und "Handwerk".

Machen Sie sich selber klar, wie Sie den Stoff am besten lernen. Z.B.: Begriffe lernt man durch eigenes Nachdenken und anschließende Diskussion mit anderen. Programmieren lernt man meist durch Fehler und ständiges Überprüfen. Definitionen lernt man durch Beispielkonstruktionen.

Sie müssen "einen eigenen Plan" entwickeln und einhalten!

Ziele und Zeiten setzen.
Lernzeiten und "Muße" planen.
Selbst in das Stoffgebiet eindringen (es gibt Bücher, eine Bibliothek, andere Studierende, ...)
Auswerten, sich selbst beurteilen, Lernmethoden verändern usw.

Hilfen bieten Ihnen die Übungen und Unterrichtsmaterialien.

Machen Sie sich Pläne!

Erkenntnisse:
- Wer etwas verstehen will, muss es anderen erklären!
- Wer sein Wissen langlebig nutzen will, muss es durch Diskussion mit anderen, durch eigene Arbeit, Nachvollziehen und Durchdenken selbst in seinem Gehirn "konstruieren".

Die meisten (etwa 80% ?) lernen erfolgreicher durch Diskussionen:
- Der Stoff wird besser verstanden,
- man bemerkt seine eigenen Fehler und Ungenauigkeiten,
- durch Nachfragen wird das eigene Mitdenken geschärft,
- durch Fragen denkt der Sprecher genauer nach, was er sagt.

Wir empfehlen und realisieren das **"Kooperative Lernen"** in dieser Vorlesung als eine Ausprägung des "Konstruktivismus", wodurch das Lernen interessanter und für die meisten effektiver gestaltet und die Arbeit in Teams angeregt wird.

Hauptproblem: Innerer Antrieb.

Wir wissen nicht, woran es genau liegt, aber die Studierenden machen von Jahr zu Jahr einen zunehmend hilfloseren Eindruck. Zum Beispiel nimmt seit Jahren das kritische Nachfragen, das Interesse an Inhalten, die nicht zur Prüfung gehören, der Besuch von fachfremden Veranstaltungen, das Studium im Ausland oder der Wechsel an andere Hochschulen stetig nach.

Unser Ziel ist es, neben den fachlichen Qualifikationen in Ihnen Kompetenzen und Persönlichkeitsmerkmale zu entfalten. Dies geht aber nur, wenn Sie mitmachen. Zunächst müssen Sie sich die vielen Grundlagen in der Mathematik, Elektrotechnik, Statistik und Informatik aneignen und, selbst wenn Sie diese Grundlagen nicht sonderlich mögen, eine Motivation aufbauen und durch Lernen verstärken.

Ihr Ziel ist es, eine Informatik-Expertise zu erwerben. Genau für dieses Ziel muss in Ihnen ständig ein "innerer Motor" laufen. Diesen Motor müssen Sie selbst betreiben! Kommen Sie zur Beratung, sobald Sie merken, dass dieser innere Antrieb zu sehr nachlässt.

0.2 Arbeitsleistung

Wenn Sie die Veranstaltung erfolgreich bestehen wollen, dann müssen Sie regelmäßig mitarbeiten sowohl in der Vorlesung als auch in den Übungen.

Wir bewegen uns in eine Zeit der Evaluationen hinein: Alles und Jedes wird analysiert und bewertet. Auch Sie. Für die, die kontinuierlich mitmachen, wird es kaum Probleme geben.

Zum Beispiel: Ihr Aufwand für die Veranstaltung "Einführung in die Informatik I und II" *alles inklusive:* Hierfür sind während der Vorlesungszeit pro Woche 13 Zeitstunden aufzuwenden. Hinzu kommen evtl. die Programmierübungen, die weitere 4 bis 5 Stunden pro Woche erfordern usw. Reservieren Sie feste Zeiten in jeder Woche ***und halten Sie diese auch ein***.

Kontrollieren Sie sich, und zwar was Sie tun, wie Sie es tun und wie effizient Sie es tun - und analysieren Sie die Ursachen. Hüten Sie sich vor allem vor zu viel "Rumsitzen" am Rechner, Small-Talk zwischen den Veranstaltungen und sinnlosen Fahrten.

Planen Sie Ihre Zeit! Zum Beispiel durch Einträge in eine Tabelle. Erwartet werden in der Vorlesungszeit 2700 Minuten/Woche. Es folgt ein Formular zum Ausdrucken und Benutzen:

Beispieleintragungen für jede Woche (nicht schummeln!!) Zeiten in Minuten, Soll: 2700 Minuten/Woche

Name: ___*ich*_____ Woche vom 27.10. bis 31.10.08

	Mo	Di	Mi	Do	Fr	Sa	So	Summe
Informatik	90	125	0	280	50	130	20	695
ProgrKurs	160	0	150	0	0	0	0	310
Mathematik	60	120	105	0	230	0	115	630
Logik	0	140	60	90	0	50	0	340
ETG	95	90	90	90	60	0	0	425
(Rechner ☺)	*(70)*	*(30)*	*(30)*	*(0)*	*(95)*	*(0)*	*(0)*	*(225)*
(Infos ☺)	*(20)*	*(15)*	*(10)*	*(35)*	*(0)*	*(45)*	*(0)*	*(125)*
Summe	405	475	405	460	340	180	135	**2400**
Soll	480	480	480	480	480	300	0	2700

Trotz "gefühlter" Arbeitsamkeit (2750) ist das zu wenig, weil Rumsitzen am Rechner und Kommunikationen nicht zählen! Planen Sie neu und halten Sie sich an den Plan!!

Name: _____ **Minuten pro Woche** Woche vom _____ bis _____

	Mo	Di	Mi	Do	Fr	Sa	So	Summe
Informatik								**780**
ProgrKurs								**300**
Mathematik								**780**
Logik								**420**
ETG								**420**
X								
Y								
Summe								
Soll								**2700**

0.3 Unerwünschtes

An einer vergleichbaren Universität in den USA müssten Sie über 6000 Dollar (4500 Euro) allein für unsere zweisemestrige 6-SWS-Grundvorlesung bezahlen. Dieser Gegenwert muss im "noch preiswerten" Deutschland allen Interessierten zugute kommen.
Das bedeutet vor allem:

Keine Störungen während der Veranstaltungen.

Speziell: Kein Lärm, keine Unterhaltungen, kommen Sie pünktlich und gehen Sie erst am Ende der Veranstaltungen.

Keine Einnahme kalter und warmer Mahlzeiten, keine herumliegende Flaschen, weggeworfene Zettel usw.

Computer sind (auch in diesem Jahr) während der Vorlesungen Informatik I + II nicht erlaubt. Sie würden hiermit Ihre gesamte Umgebung belästigen.

Seien Sie unbedingt rücksichtsvoll gegenüber anderen!

Wenn andere stören, können Sie nicht lernen. Folglich:

Werden Sie intolerant! Fordern Sie Störer auf, still zu sein.

Es ist Ihr Studium und es gehört zu Ihrer Selbstentwicklung, sich nicht alles gefallen zu lassen.

Wir helfen gerne, sehen aber immer nur kleine Ausschnitte ... .

Wir möchten, dass alle den Stoff erlernen. Das erreicht man nicht durch Abschreiben. Also: **keine identischen oder sehr ähnlichen Abgaben** in den Übungen (außer im Rahmen der vorab festgelegten Lern-Gruppen) und auf keinen Fall bei Klausuren, Hausarbeiten, Vorträgen usw.

In der Wissenschaft ist Abschreiben völlig inakzeptabel.

0.4 Goldene Regeln

Erstens: Glauben Sie keinem Gerücht.

Fast alles, was unter Studierenden über Professoren, Mitarbeiter, Prüfungen, Abläufe usw. kursiert, hat (teilweise grobe) Fehler! Folgerung: Suchen Sie nach Original-Informationen oder gehen Sie in eine Sprechstunde und lassen sich vom Zuständigen beraten!

Zweitens: Vermeiden Sie Oberflächliches.

Arbeiten Sie erst weiter, wenn Sie das Bisherige wirklich verstanden und sich an Beispielen, Programmen usw. klar gemacht haben. Fast nichts ist „trivial“, sondern besitzt verborgene Feinheiten.

Drittens: Üben Sie Durchhalten.

Nichts fällt einem zu oder kommt als Eingebung auf Spaziergängen. Manche Themen lassen sich nicht häppchenweise zerlegen und müssen daher im Ganzen verdaut werden, z. B. auch mal fünf Stunden am Stück am gleichen Problem arbeiten (oder kontinuierlich jeden Tag mehrere Stunden nacheinander und vor allem ungestört). In der Schule war es oft so, dass man alles Wesentliche mitbekam, wenn man nur anwesend war. Ab sofort gilt für Sie: **Anwesenheit allein reicht nicht!**

Viertens: Abstraktion bedarf der konkreten Beispiele.

Modelle stellen das Wesentliche vieler verschiedener Ausprägungen dar. Modellbildung und Abstraktion kann man nur verstehen, wenn man diverse typische Beispiele, Anwendungen, Abläufe usw. kennt. Abstraktion, Präzision und Formales sind zentral für die Informatik.

Fünftens: Arbeiten Sie in Lern-Gruppen (zu 3 bis 4 Personen)**.**

Zum einen bleibt der Stoff, der in Gesprächen erarbeitet wurde, besser haften. Zum anderen lernt man andere Sichtweisen kennen und wie man auf sie reagiert. Weiterhin „justieren“ Sie sich automatisch, da Sie im Vergleich mit anderen Ihren eigenen Wissensstand einzuschätzen lernen. Merke: Von Langzeitbekannten kann man nichts mehr lernen.

Sechstens: Überwachen und planen Sie Ihre Arbeit.

Die Lehrenden gehen davon aus, dass Sie pro Semesterwochenstunde etwa 2,25 Zeitstunden aufwenden (davon 0,75 für die Lehrveranstaltungsstunde, d.h. zusätzlich 1,5 Zeitstunden je SWS). Kontrollieren Sie sich. Spüren Sie sinnloses Arbeiten (am Rechner, in Vorlesungen,...) auf. Spalten Sie ggf. Teile ab, die Sie erst in der vorlesungsfreien Zeit weiter vertiefen (aber beachten Sie "zweitens").

Siebtens: Seien Sie stofforientiert

im Gegensatz zu „prüfungsorientiert“. Arbeiten Sie den Stoff und die Übungen durch, legen Sie sich Beispiele zurecht und schauen Sie öfters in eines der empfohlenen Bücher. Natürlich sollen Sie sich nach Prüfungsfragen erkundigen, wer aber sein ganzes Lernen hiernach ausrichtet, lernt falsch.

Achtens: Werden Sie frühzeitig kritisch zu Inhalten.

Lernen Sie „Bewerten“. Stimmen die Aussagen? Decken sie sich mit denen aus Büchern? Gibt es im Detail vielleicht Gegenbeispiele? Wie genau werden Anwendungen hierdurch erfasst? Gibt es andere Lösungswege? usw. Diskutieren Sie dies mit anderen, üben Sie dies vor allem an Stoffgebieten, die Sie weniger interessieren.

Neuntens: Werden Sie kritisch zu sich und gegenüber anderen.

Ab jetzt haben nur Sie die Verantwortung darüber, was aus Ihnen wird. Die Universität kann Ihnen helfen, sich zu orientieren, Ziele zu setzen, Wege zu finden. Doch alle Entscheidungen über Richtungen, Geschwindigkeiten, Studienwechsel usw. müssen Sie selbst fällen. Bewerten Sie sich selbst. Und: feilen Sie auf diese Weise an Ihrer Persönlichkeit.

Zehntens: Bilden Sie sich weiter.

Schauen Sie sich auch in anderen Fakultäten um. Lernen Sie Englisch, Kosten-Nutzen-Rechnung, Patentrecht oder was sonst für Ihre Zukunft interessant sein könnte. Fördern Sie auch Ihre Allgemeinbildung und Ihre "soft skills" (überfachliche Schlüsselqualifikationen). Diskutieren Sie auch mit Studierenden anderer Fachrichtungen, um deren Denkweisen und Fragstellungen kennen zu lernen. Besuchen Sie Vorträge, die die Universität oder die Fakultät anbieten. Nutzen Sie, solange es das noch gibt, das Neben- oder Anwendungsfach. Und üben Sie jetzt schon ein wenig LLL (= "LebensLanges Lernen").

Überwachen Sie sich (und zwar Ihr Können und Ihre Zeiten)!

Freiheit bedeutet mehr Verantwortung! Das heißt:
Sie sind schuld, wenn Sie etwas nicht verstehen,
und nicht eine als schlecht empfundene Lehre, nicht die Professor(inn)en, nicht der Lärm im Hörsaal, nein Sie!

0.5 Voraussetzung für die Vorlesung "Informatik I"

< Siehe hierzu das spezielle Material auf der Vorlesungsseite. >

Informatik hat wenig mit der Beherrschung eines konkreten Computers oder mit auf dem Markt verfügbaren Systemen zu tun. Daher ist solches Wissen auch nicht Voraussetzung für das Studium. Programmierkenntnisse können dagegen sehr hilfreich sein.

Gute *Mathematikkenntnisse* aus der Schule sind wichtig: Vieles muss formalisiert und modelliert und mit Logiken oder Kalkülen beschrieben werden. Für die Grundvorlesung muss man kein Mathematik-Fan sein; wer jedoch eine Abneigung gegen die Mathematik hat, wird Probleme in jedem universitären Informatikstudium in Deutschland bekommen.

Gute Deutsch- bzw. Muttersprach-Kenntnisse sind wichtig, um sich klar und umfassend ausdrücken zu können.

Weltweite Kommunikation basiert heute auf der englischen Sprache. Im Laufe des Studiums werden Sie deshalb lernen müssen, sich in Englisch in Schrift und Sprache klar ausdrücken zu können.

0.6 Zur Informatik

Informatik ist die Wissenschaft vom systematischen Darstellen, Verarbeiten, Übertragen und Speichern von Information und von der Beherrschung von Informationsprozessen unter besonderer Berücksichtigung digitaler Datenverarbeitungsanlagen und deren Vernetzung.

Sie besitzt Aspekte der
- Grundlagenwissenschaften, vor allem der Mathematik,
- Ingenieurwissenschaften und
- Naturwissenschaften.

Zurzeit dominieren die ingenieurwissenschaftlichen Aspekte. Beachten Sie, dass sich die Informatik nicht auf sinnlich wahrnehmbare, sondern auf abstrakte Gegenstände bezieht.

Information ist Kommunikations-, Steuerungs- und Entscheidungsmittel in allen Lebensbereichen. Informatik nimmt daher eine zunehmend zentrale Stellung ein.

Information ist heute verfügbar und zwar

überall	(ubiquitär)
zu jeder Zeit	(allgegenwärtig)
sofort	(auf Knopfdruck)
dauerhaft	(persistent)
vernetzt	(semantische Netze)
in der jeweiligen Ausprägung	
alles durchdringend	(eingebettet)
weiter verarbeitbar	(sehr viele Werkzeuge)

Die Entwicklung ist zurzeit rauschhaft schnell:
- Horrorvisionen der totalen Abhängigkeit
- Visionen von Paradiesen
- Alltag der nächsten Generation

Wichtig ist die Beherrschbarkeit fast unüberschaubarer Systeme
⇒ Wissenschaft Informatik
⇒ Methoden der Softwaretechnik
Arbeit wird vollkommen neu gestaltet
alle Informationsquellen sind für alle da
steuern statt selbst agieren
eine neue Welt der Dienste
neue Formen des Lernens
Freizeit und **Kommunikation** verändern sich laufend.
Informatik ist ein wichtiger Motor für **Innovationen**.

Informatik kann man nicht direkt sehen, aber indirekt.
Sichtbar wird Informatik bei ...?

- **Auto:** ABS, Parkhilfe, Tempomat, ESP, Einfädeln, Navigationssystem
- **Flugzeug:** Ausgleich von Turbulenzen, Autopilot, Kollisionswarnung
- **Handy:** Telefonbuch, Notrufe, Rufumleitung, Bilder, Internetaufruf
- **Roboter:** Ablaufsteuerung, Qualitätskontrolle, Situationsanpassung
- **Information:** Zeitung, Fernsehen, elektronische Post, Internet
- **Optimierungsprobleme:** < in jedem Gebiet >
- **Eingebettete Systeme:** < in jedem Gerät möglich >:
- **Visualisierung:** Architektur, Archäologie, Medizin, Crashtest, ...
- **Gesundheit, Sicherheit, Ausbildung, Urlaub, ...**

0.7 Hinweise zu Zielen der Vorlesung

Ziele der Veranstaltung sind die Vermittlung grundlegender Darstellungen und Prinzipien der Informatik, insbesondere
- Formalismen und Beschreibungssprachen, Modellbildung,
- Algorithmen und ihre Eigenschaften (Zeitkomplexität, Verifikation, Terminierung),
- Datenstrukturen und die Effizienz zugehöriger Verfahren,
- Prozesse und Netze,
- Grundbegriffe und Denkweisen der Programmierung,
- Abstraktion und Theorie, auf denen diese Begriffe basieren,
- Vorgehensweisen zur Problemlösung und die Realisierung und Implementierung der Konzepte,
- Beherrschung möglichst vieler wichtiger Standardalgorithmen,
- genaue Kenntnis typischer Beispiele und ihrer Programmierung,
- genaue Kenntnis einer Programmiersprache (hier: Ada 05).

Die Grundvorlesung ist von zentraler Bedeutung für die gesamte Informatik, da sie die Basisstrukturen und Konstruktionsprinzipien (heutiger) informationsverarbeitender Systeme behandelt, wichtige Denkweisen, langlebige Ideen, Konzepte und Methoden vermittelt und Beurteilungskriterien über Vor- und Nachteile von Prozessen und allgemein von Problemlösungen diskutiert.

Beachte:

Für die Informatik sind die *Abstraktion* und die *Formalisierung* besonders wichtig. Abstraktion bezeichnet das Herausarbeiten des Wesentlichen, wobei man sich von vielen konkreten Details befreit. Unter Abstraktion versteht man auch den Übergang zu einer Metaebene; sie umfasst das Erarbeiten gemeinsamer Ideen oder Prinzipien aus den unterschiedlichsten (Anwendungs-) Gebieten, z.B. die einheitliche Darstellung von Strukturen in Programmiersprachen, von syntaktischen Gebilden bei natürlichen Sprachen oder von Rechenbäumen mit Hilfe von (kontextfreien) Grammatiken. Abstraktes Denken ist eine wichtige Voraussetzung für eine adäquate *Modellierung*, wobei die Dinge der realen Welt in eine Modellierungssprache übertragen werden, aus der heraus man die Modelle weiter in Programmiersprachen abbildet. Um diese Modelle maschinell verarbeiten zu können, müssen sie formalisiert werden. Formalisierung beschreibt den Prozess, Geschehnisse der realen Welt oder (unscharfe) Informationen zu präzisieren und zugleich in Kalkülen zu notieren.

Wie verteilen sich Lernzielbereiche in der Vorlesung?

Ein Beispiel:

- Wissen und Fähigkeiten mit Theorieunterbau	(ca. 60%)
- Fertigkeiten, Handwerk, Routinewissen	(ca. 26%)
- Fachübergreifendes, Persönlichkeitsentwicklung	(ca. 6%)
- Entwicklung eines Beurteilungsvermögens	(ca. 4%)
- Sonstiges (Lärm, Aufbereitung, Vortragstechnik, ...)	(ca. 4%)

Hinzu kommen gute "handwerkliche Fähigkeiten" des Programmierens für diejenigen, die sich viel mit Informatikmethoden und deren Implementierungen befassen müssen.

Diese Vorlesung bereitet viele andere Vorlesungen vor. Stellvertretend für diese Lehre finden Sie auf der nächsten Folie die dafür zuständigen Hochschullehrer und ihre Arbeitsgebiete an der Universität Stuttgart (Stand Ende Sommersemester 2008; an dieser Stelle sei die aktuelle Zusammensetzung des Lehrkörpers, den Sie in den kommenden Jahren genauer kennen lernen werden, unauffällig "versteckt").

Bereich Informatik

Institutsverbund Informatik

Institut für Parallele und Verteilte Systeme (IPVS)

Rechnerarchitektur, eingebettete Systeme
Prof. Wunderlich, Prof. Radetzki
Formale Konzepte, Prof. Claus
Theoretische Informatik, Prof. Diekert
Zuverl. u. sich. Softwaresyst., N.N.
Programmiersprachen & ihre Übersetzer,
Prof. Plödereder
Software Engineering, Prof. Ludewig
Intelligente Systeme, Prof. Heidemann
Grundlagen der Informatik, CAD,
Prof. Roller, Prof. Eggenberger
Architektur von Anwendungssystemen,
Prof. Leymann
Visualisierung, interakt. Systeme,
Prof. Ertl, Prof. Weiskopf

Anwendersoftware, Prof. Mitschang
Parallele Systeme, Prof. Simon
Verteilte Systeme, Prof. Rothermel
Bildverstehen, Prof. Levi
Simulation großer Systeme, N.N.

Institut für Maschinelle Sprachverarbeitung (IMS)
(mit 4 weiteren Professuren)

0.8 Weitere Hinweise

Vorlesung, Übungen, Vortragsübungen, Zwischenklausuren: kontinuierlich mitmachen und durchhalten.

Wie können Sie sich stärker äußern?
- In den Übungen, leider *kaum* in der (Massen-) Vorlesung.
- Kontakte zu Tutoren und Mitarbeitern halten!
- Elektronische Medien (E-Mail, Forum) nutzen.
- Wahl von Vorlesungssprecher(inne)n und Rückmeldungen über diesen Personenkreis an Assistenten und Dozenten.
- Fachschaft, Tutoren und Studienberatung aufsuchen.

Nutzen Sie Ihre Lerngruppen!

Für jede Lehrveranstaltung ist ein **Lehrbuch** wichtig. Lehrbücher sind ausgereift, werden auch an anderen Universitäten eingesetzt, vereinheitlichen das Wissen und erleichtern Ihnen später den fachlichen Gedankenaustausch über Stuttgart hinaus. (Literatur zur Grundvorlesung siehe Hinweise ganz am Ende dieses Skripts.)

Zusätzlich hierzu können Präsentationen, Folien und weitere Unterlagen elektronisch abgerufen oder bei der Fachschaft kopiert werden.

Schreiben Sie dennoch manches mit, vor allem was an die Tafel geschrieben wird. (Ideen kann man schriftlich kaum darstellen; aber Sie erinnern sich über Ihre Notizen daran.) Bereiten Sie Ihre Notizen noch am gleichen Tag auf. Lernen verbessert sich mit der Zahl der beteiligten Sinne, d.h. auch, dass man beim Mit-Notieren mehr behält, als wenn man nur in der Vorlesung zuhört.

Erkundigen Sie sich nach Übungs- und Klausuraufgaben der vergangenen Jahre. Deren Bearbeitung liefert Ihnen Anhaltspunkte zum Stand Ihres Wissens.

0.9 Beispielthema: Mandatszuteilung bei Wahlen

Im Jahre 2008 hat das Bundesverfassungsgericht das derzeitige Wahlverfahren des Bundes zur auf Länder bezogenen Verteilung der Abgeordneten-Mandate auf die Parteien als verfassungswidrig erklärt. Dass die gängigen Berechnungsverfahren prinzipielle Mängel haben, ist seit langem bekannt.

Aufgabe für Sie: Informieren Sie sich
(a) über die folgenden Verfahren zur Mandate-Berechnung:
d'Hondt-Verfahren (= Jefferson-Verfahren),
Hare-Niemeyer-Verfahren (= Hamilton-Verfahren),
Sainte-Lague-Verfahren (= Webster-Verfahren),
Adams-Verfahren und Hill-Huntington-Verfahren;
(b) über den Grund, warum das Wahlrecht geändert werden muss.

Wie geht man vor, um das Wahlverfahren zu verbessern? Wer ein Zuteilungsverfahren der Abgeordneten-Mandate an die Parteien entwickelt, sollte die erforderlichen Daten genau kennen und dann einige Bedingungen festlegen, die solche Verfahren erfüllen müssen.

Folgende Werte müssen gegeben sein:
p = Anzahl der Parteien (nur solche oberhalb einer möglichen x%-Hürde),
s_i = Anzahl der gültigen Stimmen für die Partei i (für $1 \le i \le p$),
g = Anzahl aller gültigen Stimmen ($g = s_1 + s_2 + ... + s_p$),
v = Anzahl aller zu verteilenden Mandate (= Anzahl der Sitze).

Hieraus sind folgende Werte zu berechnen:
q_i = "Quote" für die Partei i (also: $q_i = v \cdot s_i / g$), dieser Wert entspricht dem proportionalen Stimmenanteil, den die Partei i erhalten hat. Es gilt stets: $v = q_1 + q_2 + ... + q_p$.
m_i = Anzahl der Sitze, die die Partei i erhalten wird. Es muss gelten: $v = m_1 + m_2 + ... + m_p$.

Dann sollten folgende Eigenschaften erfüllt sein:

Bedingung 1: Die Zahl der Mandate liegt nahe bei der Quote.
Für eine reelle Zahl x sei $\lfloor x \rfloor$ der ganzzahlige Anteil von x, also die größte ganze Zahl, die kleiner oder gleich x ist. Dann muss für jede Partei i gelten: $\lfloor q_i \rfloor \le m_i \le \lfloor q_i \rfloor + 1$.

Bedingung 2: Gleichwertigkeit der Stimmen.
Der Quotient q_i/m_i sollte für alle i sehr ähnlich sein und dicht bei 1 liegen.

Bedingung 3: Monotonie-Bedingung.
Aus $s_i > s_j$ folgt stets $m_i \ge m_j$.

Bedingung 4: Verträglichkeit mit Koalitionsaussagen.
Die Zahl der Mandate für eine Koalition ist unabhängig von der Aufteilung der Stimmen auf die beiden Koalitionsparteien; d.h., für je zwei Parteien i und j soll gelten: Erhöht man die Zahl der Stimmen s_i für Partei i um k und erniedrigt zugleich die Zahl der Stimmen s_j für Partei j um k, so soll sich die Summe der zugeteilten Mandate $m_i + m_j$ (= die Zahl der Mandate für die Koalition von i und j) nicht verändern.

Man kann weitere Bedingungen "erfinden". Zum Beispiel legt die denkbare Bedingung

Für alle Parteien i und j gilt: $s_i / m_i > s_j / (m_j + 1)$.

das d'Hondt-Verfahren nahe: Dividiere alle Stimmenzahlen s_i durch die Zahlen 1, 2, 3, ..., nimm die v größten dieser Zahlen und teile der Partei i so viele Mandate zu, wie sie hierbei auftritt.

Beispiel für d'Hondt: 4 Parteien mit den Stimmenzahlen 380, 370, 150, 100 (g = 1000 Stimmen). Bilde dann die Quotienten und vergib die Mandate nach der Rangfolge dieser Quotienten:

	1	2	3	4	5	6	7	8	9	10
380	380	190	126,6	95	76	63,3	54,2...	47,5	42,2	38
370	370	185	123,3	92,5	74	61,6	52,8...	46,25	41,1	37
150	150	75	50	37,5	30	25	21,4...	18,75	16,6...	15
100	100	50	33,3	25	20	16,6	14,2...	12,25	11,1...	10

Es sollen v = 11 Mandate vergeben werden, dann erhalten "Partei 1": 5, "Partei 2": 4, "Partei 3": 1 und "Partei 4" 1 Mandat.
Ist dies eine "gerechte Zuteilung"? Wir prüfen die 4 Eigenschaften.

	1	2	3	4	5	6	7	8	9	10
380	380	190	126,6	95	76	63,3	54,2...	47,5	42,2	38
370	370	185	123,3	92,5	74	61,6	52,8...	46,25	41,1	37
150	150	75	50	37,5	30	25	21,4...	18,75	16,6...	15
100	100	50	33,3	25	20	16,6	14,2...	12,25	11,1...	10

Partei:	1	2	3	4
Quote:	4,18	4,07	1,65	1,10
$\lfloor$Quote$\rfloor$:	4	4	1	1
Sitze:	5	4	1	1
w_i (s. nächste Folie):	76	92,5	150	100

Bedingung 3 ist erfüllt.
Bedingung 1 ist erfüllt.

Quote/Sitze:	0,836	1,0175	1,65	1,10

Bedingung 2 scheint nicht erfüllt zu sein.
Bedingung 4 ist nicht erfüllt. Man fasse z.B. die Parteien 2 und 3 zusammen; man erhält drei Parteien mit 380, 520 und 100 Stimmen, an die nach d'Hondt 4, 6 und 1 Sitz verteilt werden müssten.

Bedingung 5: Minderheitsbedingung.

Eine Partei, die nicht die absolute Mehrheit der gültigen Stimmen erreicht hat, darf nicht die absolute Mehrheit der Sitze erhalten. Das heißt: Aus $s_i/g < 0{,}5$ folgt stets $m_i/v < 0{,}5$ (oder zumindest $m_i/v \leq 0{,}5$).

Diese Bedingung ist beim d'Hondt-Verfahren nicht erfüllt. Man betrachte hierzu 4 Parteien mit 460, 181, 180 und 179 Stimmen, die sich um v = 7 Sitze bewerben, so erhält die erste Partei 4 und die anderen drei Parteien je einen Sitz.

Stimmenzahl pro Sitz (Wert der Stimmen):
Die Zahl $w_i = s_i/m_i$ gibt an, wie viele Stimmen der Partei i für die Erringung eines Sitzes reichten.

Bedingung 6: Diese Werte w_i sollten möglichst eng zusammen liegen.

In obigem Beispiel unterscheiden sich diese Werte fast um den Faktor 2.

Probleme: Prüfe nach, ob sich diese Bedingungen widersprechen. Formuliere die Berechnungsverfahren unmissverständlich.

Man sieht an diesen Überlegungen bereits, dass das d'Hondt-Verfahren Nachteile besitzt, indem es beispielweise die größeren Parteien bevorzugt. Es wurde in den USA daher nur bis 1840 verwendet; in der Bundesrepublik Deutschland wurde das Verfahren bis 1985 benutzt und danach bis zum Jahre 2005 durch das Hare-Niemeyer-Verfahren ersetzt (die Bundestagswahl im Jahre 2009 verwendet das Sainte-Lague-Verfahren).

Hare-Niemeyer gibt zunächst jeder Partei die Zahl $\lfloor q_i \rfloor$ an Sitzen. Ein weiterer Sitz wird nach der Größe der Nachkommastellen von q_i vergeben.

Dieses Verfahren begünstigt die kleineren Parteien bei der Zuteilung des über $\lfloor q_i \rfloor$ hinaus gehenden Sitzes.

Obiges Beispiel nach dem Hare-Niemeyer-Verfahren:

Partei:	1	2	3	4
Stimmenzahl:	380	370	150	100
Quote:	4,18	4,07	1,65	1,10
Vorab vergebene Sitze:	4	4	1	1
Nachkommastellen:	0,18	0,07	0,65	0,10
weitere Sitze:	0	0	1	0
Sitze:	4	4	2	1
w_i:	95	92,5	75	100

Das Hare-Niemeyer-Verfahren hat u. a. den Nachteil, dass die Zahl der Sitze einer Partei geringer werden kann, wenn man im Nachhinein die Gesamtzahl der Sitze erhöht.
Dieses Phänomen ist als "Alabama-Paradoxon" bekannt, weil es bei einer Wahl zum Repräsentantenhaus für die dem Staat Alabama zustehenden Sitze erstmals auftrat. Informieren Sie sich hierüber (Internet, Literatur, ...).

Wir brechen unsere einführenden Überlegungen hier ab und halten fest: Wer ein Verfahren, das letztlich einem Informatiksystem übertragen werden soll, entwickeln will,

- kläre zuerst genau die Problemstellung,
- identifiziere die erforderlichen Daten,
- lege zu erfüllenden Bedingungen fest ("Spezifikationen") und weise nach, dass diese in sich widerspruchsfrei sind (bzw. wenn nicht, welche Bedingungen Priorität erhalten sollen),
- beschreibe den Ablauf des Verfahrens und notiere ihn in einer unmissverständlichen Form ("Programm").

Erst danach folgen die Installation/Implementierung, Tests, Experimente, kritische Überprüfungen, Erweiterungen usw. Unverzichtbar ist hierbei eine nachvollziehbare Dokumentation.

Behalten Sie dieses Beispiel "Sitzzuteilungsverfahren" im Hinterkopf, wenn von Informatik-Verfahren, deren Präzisierung, Programmierung und Einsatz die Rede ist. Die Vorlesung wird sich nun als erstes der Formulierung von Abläufen als Programme in einer vorgegebenen Programmiersprache zuwenden.
Jetzt geht es also endlich mit der eigentlichen Vorlesung los.

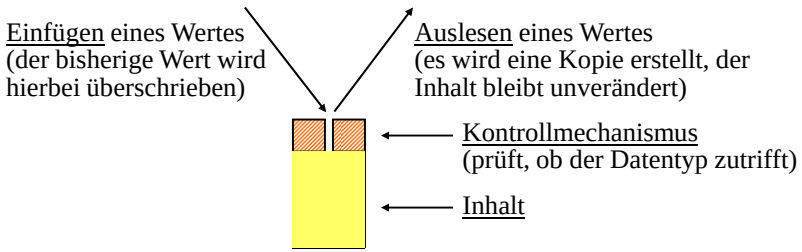
1. Kurze Einführung in die Sprache Ada

1. Kurze Einführung in die Sprache Ada

1.1 Grundsätzliches

Techniken des imperativen Programmierens:

- Das Verfahren wird durch eine Folge von Befehlen (genannt „Anweisungen“ oder „statements“) beschrieben.
- Daten werden in (strukturierten) Behältern, genannt „Variablen“ abgelegt. Zu jedem Behälter wird durch einen Datentyp angegeben, welche Daten man hineinlegen darf.
- Die grundlegende Anweisung ist die Zuweisung („assignment“), mit der einer Variablen ein Wert zugeordnet wird.
- Die Bearbeitungsreihenfolge wird durch Kontrollstrukturen festgelegt, insbesondere durch Hintereinanderausführen (";"), Alternativen ("if") und Schleifen ("while", "for").
- Teilverfahren werden zu Prozeduren zusammengefasst.



Modellvorstellung: Die Variable ist ein Behälter (gelb), in den genau die Werte eines Datentyps gelegt werden dürfen. Man stelle sich vor, dass zu dem Behälter ein Kontrollmechanismus (rot schraffiert) gehört, der bei jedem einzufügenden Wert prüft, ob der Datentyp vorliegt (falls nicht: Abbruch der Berechnung).

"Elementare" Datentypen sind:
Integer: nur ganze Zahlen dürfen in die Variable gelegt werden.
Natural: nur natürliche Zahlen (einschließlich 0).
Float: nur reelle Zahlen.
Char: nur Zeichen (z.B. die Zeichen auf der Tastatur).
Boolean: nur Wahrheitswerte {wahr, falsch} (= {true, false}).

1.2 Diszipliniertes Vorgehen

Das Programm steht nie am Anfang der Entwicklung.
Empfohlenes Vorgehen bei der Lösung von Problemen beim "Programmieren im Kleinen" (damit meint man Problemlösungen, die i. W. ein Einzelner erarbeiten kann):

(1) Ideen umgangssprachlich aufschreiben.
(2) Präzise Formulierung (formale Darstellung).
(3) Ermitteln von Eigenschaften (Analysieren des Problems).
(4) Hieraus einen Algorithmus entwickeln (algorithmische Lösung des Problems, Synthese einer Lösung).
(5) Diesen in ein Programm übertragen (Realisierung der Lösung und Implementierung).
(6) Testen, Verifizieren, Messen usw. des Programms.
(7) Kritisches Überdenken (Verfahren, Einsatz, Auswirkung)

Regel 1: Jeder Schritt von der Idee bis zum einsatzfähigen Programm muss dokumentiert werden.
Das heißt: Ideen, Eigenschaften, Testbeispiele, verwendete Methoden usw. sind während der schrittweisen Entstehung des Programms möglichst präzise aufzuschreiben.

Anders ausgedrückt: Das zu erstellende Produkt "Software" oder "Informatik-System" besteht aus:
- Ziele, Notwendigkeiten, Problembeschreibung
- Zusammenfassung der zugehörigen Literatur
- Formalisierungen
- Beschreibung von Eigenschaften, z.B. mathematische Sätze
- algorithmische Lösung mit Begründungen
- Programm (mit Hinweisen zur Implementierung)
- Testdaten, Zeitmessungen, Wartungshinweise, ...
- Einsatzbereiche, Gefahren, Erweiterungen, Verbesserungen usw.

Regel 2: Ein wichtiges Qualitätskriterium für richtiges Vorgehen beim Schreiben kleinerer Programme lautet: Der Entwicklungsprozess war gut, wenn das Programm auf Anhieb korrekt läuft.

Viele halten einen Entwicklungsprozess für gut, wenn das entstandene Programm die Besonderheiten eines Rechners gut ausnutzt oder besonders schnell oder platzsparend ist. Solche Programmeigenschaften mögen erstrebenswert sein, jedoch sind sie meist zeitgebunden und machen keine Aussage darüber, ob der Entwicklungsprozess systematisch abgelaufen ist.
Klarheit und Verständlichkeit bestimmen stärker als Tricks und Basteln die Güte von Software. Man versuche stets, ein gut lesbares und nachvollziehbares Lösungsprogramm zu schreiben.

Regel 3: Die Strukturen eines Programms müssen dem Problem entsprechen.

Dies ist anfangs schwer zu verstehen. Gemeint sind zwei Aspekte:
(1) Man wähle Datenstrukturen, Lösungsverfahren, Ein- und Ausgabe usw. möglichst einfach, aber so, dass das Problem angemessen gelöst ist. Hierbei müssen sich die Eigenschaften der Problemlösung im Programm klar widerspiegeln!
(2) Das Programm soll eine "innere Schönheit" besitzen (was das sein mag, versteht man erst mit einiger Erfahrung). Dazu gehören: Wer Felder verwendet, sollte auch for-Schleifen verwenden; wer Bäume einsetzt, benutze die Rekursion; wer hierarchische Strukturen bearbeitet, nutze Stacks usw. Unter den richtig arbeitenden Programmen gibt es "elegante" und "holprige" Lösungen. (Anzustreben sind die eleganten.)

1.3 Beispiel "Fakultät"

Die Vorgehensweise wird an diesem Beispiel erläutert.

(1) Die Ideen umgangssprachlich aufschreiben.

Gegeben sind n verschiedene Elemente a_1, a_2, ..., a_n.
Gesucht ist die Anzahl aller möglicher Anordnungen.

Bei zwei Elementen gibt es zwei Anordnungen, nämlich a_1a_2 und a_2a_1, bei drei Elementen sind es 6, nämlich $a_1a_2a_3$, $a_1a_3a_2$, $a_2a_1a_3$, $a_2a_3a_1$, $a_3a_1a_2$, $a_3a_2a_1$ usw.

Gesucht: Allgemeine Lösung plus Berechnungsverfahren.

(2) Präzise Formulierung.

Gesucht ist folgende Funktion f: $\mathbb{N} \to \mathbb{N}$

$f(n) = |\{a_{i_1} a_{i_2} a_{i_3} \ldots a_{i_n} \mid i_1, i_2, \ldots, i_n \in \{1, 2, \ldots, n\}$ und alle Indizes sind paarweise verschieden, d.h., $i_j \neq i_k$ für $j \neq k\}|$.

Beachte: Die Elemente selbst sind unwichtig, außer dass sie paarweise verschieden sein müssen.

Man kann nun einzelne Werte durch Probieren ermitteln, z.B. f(4) = 24. Dabei erahnt man bereits Eigenschaften von f.

Weiterhin kann man f auch auf $\mathbb{N}_0$ erweitern, indem man f(0) = 1 setzt.

(3) Ermitteln von Eigenschaften (Analysieren des Problems)

Offenbar gilt
f(0) = 1, f(1) = 1, f(2) = 2, f(3) = 6, f(4) = 24.

Ist eine Anordnung $a_{i_1} a_{i_2} a_{i_3} \ldots a_{i_n}$ gegeben, so kann man hieraus eine Anordnung von n+1 Elementen erzeugen, indem man das (n+1)-te Element vorne, an der zweiten Position, an der dritten Position, ... einfügt. Es gibt n+1 mögliche Positionen, wobei sich aus jeder Anordnung von n Elementen genau n+1 neue Anordnungen ergeben.

Es folgt daher: $f(n+1) = (n+1) \cdot f(n)$. Statt "f(n)" schreiben wir nun n! (gesprochen "n Fakultät"). Es gilt also:

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n-1)! & \text{für } n > 0 \end{cases}$$

noch: (3) Ermitteln von Eigenschaften

Wie lässt sich n! berechnen?

Indem man nacheinander 0!, 1!, 2!, ..., (n-1)! und hieraus n! berechnet.

Durch Einsetzen sieht man: $n! = 1 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$.

n! ist also das Produkt der ersten n Zahlen.

Diese Eigenschaft lässt sich für eine Berechnungsvorschrift verwenden.

(4) Lösungs-Algorithmus

Gegeben sei eine natürliche Zahl n ≥ 0. Diese wird in einer Variablen N abgelegt. In der Variablen Fak sollen die Zwischenergebnisse und am Ende das Ergebnis stehen.
Lies n in die Variable N ein.
Setze Fak anfangs auf den Wert 1.
Für i = 1, 2, ..., N wiederhole: multipliziere Fak mit dem Faktor i.
Gib den Wert von Fak als Ergebnis aus.

```
procedure Fakultaet is
    N, Fak, i: Natural;
begin
    Get(N);
    Fak := 1;
    for i := 1 to N do  Fak := Fak*i  od;
    Put(Fak)
end
```

(5) Programm in Ada

```
with Ada.Integer_Text_IO;  use Ada.Integer_Text_IO;

procedure Fakultaet is
    N, Fak: Natural;  -- in Fak wird nacheinander 0!, 1!, 2!, ..., N! abgelegt
begin
    Get(N);
    Fak := 1;
    for i in 1..N loop  Fak := Fak*i;  end loop;
    Put(Fak);
end ;
```

Die erste Zeile sorgt dafür, dass die Ein- und Ausgabe so funktioniert, wie man es erwartet. Erläuterungen in der Vorlesung.

(6) Testen, Verifizieren, Messen

Lassen Sie das Programm laufen für die Eingabewerte 0, 1, 3, 8 und 12. Sie erhalten jeweils die korrekten Ausgabewerte 1, 1, 6, 40320 und 479001600. Doch bei der Eingabe 13 erscheint:

raised CONSTRAINT_ERROR: ...

Der Grund: Normale PCs können eine natürliche Zahl nur mit 31 Binärstellen darstellen, d.h., die größte darstellbare Zahl ist $2^{31}-1 = 2\,147\,483\,647$.

Es ist aber: $13! = 13 \cdot 479\,001\,600 = 6\,227\,020\,800$.

Folglich erscheint eine Mitteilung, dass die Einschränkung (="constraint") auf den Bereich von 0 bis $2^{31}-1$ verletzt wurde und somit ein Fehler (error) ausgelöst (erweckt = raised) wurde.

noch: (6) Testen, Verifizieren, Messen

Schade - doch das lässt sich entweder durch "doppelte Genauigkeit" überwinden oder dadurch, dass man sich seine eigene Arithmetik schreibt.

Verifizieren bedeutet, möglichst mit formalen Methoden nachzuweisen, dass das Programm genau das tut, was es (laut "Spezifikation") soll. In unserem Beispiel ist nichts zu tun, weil wir exakt die Eigenschaft "Bilde das Produkt der ersten n Zahlen" nachvollzogen haben.

Messen: Wir benötigen nur 3 Variablen (Speicherplätze für N, Fak und i). Die Rechnung erfolgt für unsere erlaubten Zahlen von 0 bis 12 so schnell, dass man die Rechenzeit als Mensch kaum wahrnehmen kann.

noch: (6) Testen, Verifizieren, Messen

Wir wollen hartnäckig sein und schreiben folgendes Programm:

```
with Ada.Integer_Text_IO;  use Ada.Integer_Text_IO;
procedure Fakultaet is
    N, Fak: Natural;
begin
    for k in 1..100000 loop          -- wiederhole die Berechnung 100000 Mal
        N := 12;  Fak := 1;          -- statt Eingabe: fester Wert 12 für N
        for i in 1..N loop  Fak := Fak*i;  end loop;
        Put(Fak);                    -- viel Ausgabe
    end loop;
end ;
```

Wir wiederholen also die Berechnung 100000 mal. Mein Computer benötigte hierfür 12,5 Sekunden, aber er läuft ebenfalls 12 Sekunden, wenn man die Zahl 12 durch 1 ersetzt. Das bedeutet: Die eigentliche Rechenzeit ist verschwindend gering; die "sichtbare" Rechenzeit wird durch die Ausgabe Put(Fak) bestimmt.

noch: (6) Testen, Verifizieren, Messen

Wir lassen also auch noch die Ausgabe weg:

```
procedure Fakultaet is
N, Fak: Natural;
begin
    for k in 1..10_000_000 loop        -- wiederhole 10 Millionen Mal
        N := 12;  Fak := 1;            -- statt Eingabe: fester Wert 12 für N
        for i in 1..N loop  Fak := Fak*i;  end loop;
    end loop;
end ;
```

Nun wiederholen wir die Berechnung 10 Millionen Mal. Mein Computer benötigte hierfür 5 Sekunden, aber er läuft nur eine knappe Sekunde, wenn die Zahl 12 durch 1 ersetzt wurde. Folglich benötigt das ursprüngliche Programm für die Eingabe 12 weniger als eine Mikrosekunde (10^{-6} Sekunden).

So ein heutiger Rechner ist schon ziemlich schnell ...

Anforderung an Sie:

- Zu jedem Programm gehören gut aufbereitete Erläuterungen!
- Halten Sie sich an ein systematisches Vorgehen!
- Strukturieren Sie Ihre Programme durch Einrückungen, durch Verwenden einprägsamer Namen, durch Kommentare und durch viele weitere Maßnahmen, die das Lesen von Programmen meist erst ermöglichen. (Im Studiengang Softwaretechnik müssen Normen und Standards sehr genau eingehalten werden!)

Dies wird Ihnen sehr schwer fallen. Wenn Sie es aber nicht tun, schreiben Sie Software, die später kaum zu verstehen, nicht zu erweitern, nicht anzupassen und damit nicht mehr einsetzbar ist. Deshalb ziehen wir in den Übungen für unsystematisch geschriebene Programme viele Punkte ab ☺.

(7) Kritisches Überdenken (Verfahren, Einsatz, Auswirkung)

Der Bereich "Einsatz und Auswirkungen" spielt bei diesem kleinen Problem keine Rolle.

Aber man sollte kritisch überdenken, ob die Funktion in der angegebenen Weise berechnet werden soll.

Wie unter (3) erwähnt definiert man die Fakultät in der Regel rekursiv, also

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

Gibt es hier "bessere" Darstellungsmöglichkeiten? Hierauf gehen wir unter dem Stichwort "Rekursion" ein.

1.4 Programmaufbau in Ada

```
<with und use Bereich>
procedure <Name des Programms> is
    <Deklarationsteil>
begin
    <Anweisungsteil>
end;
```

Vorbemerkung: Um eine Struktur zu beschreiben, verwenden wir einen Text in spitzen Klammern: **<Text>**
Dies ist ein *Platzhalter!*
Damit meinen wir: An diese Stelle muss etwas eingesetzt werden, das von der Form "Text" ist. Z.B. muss in einem Ada-Programm zwischen die Wörter procedure und is ein "Name des Programms" geschrieben werden. Was <Name des Programms> genau ist, wird dann später erläutert.

Kommentare: von -- bis zum Zeilenende.
Diese Kommentare dürfen überall im Programm stehen.

Namen: Ein Name bezeichnet eine Einheit oder eine Größe in einem Programm. Sie besitzen stets mindestens einen Bezeichner (engl.: identifier), um sie identifizieren zu können. Insbesondere werden Variablen und Konstanten durch einen eindeutigen Bezeichner charakterisiert, d.h., jeder Bezeichner darf nur für eine Größe benutzt werden. Ein Bezeichner ist eine Folge aus Buchstaben, Ziffern und dem Unterstrich "_", die mit einem Buchstaben beginnen muss. Z. B.:

A, A15, A_und_B, Informatik_Neubau_38_01

aber nicht

_A3, 3Aplus2B, 100, N#23.

Ein **<Name des Programms>** ist ein solcher Name.

Besonderheit in Ada:

Es gibt bei Bezeichnern *keinen Unterschied zwischen großen und kleinen Buchstaben.*

Zum Beispiel werden die beiden Bezeichner

Haus
hAuS

als der gleiche Bezeichner aufgefasst.

Übliche Konvention: Ein Bezeichner für eine Variable oder Konstante sollte immer mit einem Großbuchstaben beginnen; alle anderen vorkommenden Buchstaben sollten klein geschrieben werden, außer eventuell hinter einem Unterstrich:

Haus7ab Zweite_Laufvariable Kleinster_x50_Wert

Es sind fast alle Bezeichner für Namen zugelassen mit Ausnahme der sog. Ada-Schlüsselwörter, auch Ada-Wortsymbole oder reservierte Wörter genannt. Dies sind Bezeichner, die eine feste Bedeutung in Ada haben und nicht für andere Zwecke benutzt werden dürfen. In Ada 2005 gibt es 72 solcher Schlüsselwörter, siehe nächste Folie. Diese Wörter werden in unseren Programmen oft (aber nicht immer) in blauer Schrift oder durch Unterstreichung hervorgehoben.

In den folgenden neun Wochen werden Sie mindestens folgende 42 Ada-Schlüsselwörter kennen und verstehen lernen (ungefähr auch in dieser Reihenfolge):

procedure, is, begin, end, constant, if, then, else, while, loop, for, in, not, and, or, xor, abs, mod, rem, elsif, exit, function, return, out, type, subtype, declare, others, with, use, array, of, range, record, access, null, all, new, case, when, digits, delta.

Es folgen die 72 Schlüsselwörter, die Sie nicht für eigene Größen benutzen dürfen, in alphabetischer Reihenfolge:

abort, abs, abstract, accept, access, aliased, all, and, array, at, begin, body, case, constant, declare, delay, delta, digits, do, else, elsif, end, entry, exception, exit, for, function, generic, goto, if, in, interface, is, limited, loop, mod, new, not, null, of, or, others, out, overriding, package, pragma, private, procedure, protected, raise, range, record, rem, renames, requeue, return, reverse, select, separate, subtype, synchronized, tagged, task, terminate, then, type, until, use, when, while, with, xor.

Beachten Sie, dass Bezeichner für die Datentypen (also Boolean, Integer usw., siehe unten) nicht geschützt sind. In der Praxis sollten Sie diese nur verwenden, wenn Sie die entsprechenden Datentypen umdefinieren wollen.

<Deklarationsteil> ist eine endliche Folge von Vereinba-rungen (genannt "Deklarationen"). Wir betrachten zunächst nur die <Variablendeklaration> und die <Konstantendeklaration>. Die **<Variablendeklaration>** ist eine endliche Folge von Vereinbarungen der Form

<Liste von Variablen> **:** <Datentyp> **;**

zum Beispiel folgende Vereinbarungen von sechs Variablen:

A, B, C24: Integer; Q17_2: Boolean; X: Character;
H11_oder_B22: Float;

Der "Datentyp" gibt für die davor stehenden Variablen an, welche Werte in ihnen abgelegt (und welche Operationen mit ihnen ausgeführt) werden dürfen. Ada ist hier sehr konsequent und zwingt die Programmierer zum Beispiel, klar zwischen der ganzen Zahl 1 und der reellen Zahl 1.0 zu unterscheiden.

Ein **<Datentyp>** kann ein <skalarer Datentyp> oder ein <zusammengesetzter Datentyp> sein. Uns interessieren zunächst nur die skalaren Datentypen. Hierzu gehören:

Wahrheitswerte: Boolean
Wertebereich: **IB** = {False, True} (oft auch {0, 1})

Ganze Zahlen: Integer
Wertebereich: **Z** = { ..., -2, -1, 0, 1, 2, 3, ... }

Reelle Zahlen: Float
Wertebereich: **IR**, alle reellen Zahlen, geschrieben in der Form <ganzzahliger Anteil>**.**<Nachkommastellen>
zum Beispiel: -4.13 -22.001 0.0 2.718281828459

Alphabetzeichen: Character
Wertebereich: Menge der Tastaturzeichen (sowie einige Steuerzeichen, die nicht auf der Tastatur stehen).
Schreibweise: in Apostrophe einschließen, also
'A' '1' '+' '.'

Diese sog. Standard-Datentypen werden ausführlich in 1.8.2 behandelt!

Statt der Variablen, deren Werte sich im Laufe der Berechnung ändern können, darf man auch Konstanten im Deklarationsteil vereinbaren, deren Werte nicht abgeändert werden dürfen. Hierfür schreibt man vor den Datentyp das Wort constant. und fügt den Wert (durch "**:=**" getrennt) an.

Beispiele:

Eins: constant Integer := 1;

Pi: constant Float := 3.1415926;

E: constant Float := 2.718281828459;

Pi_mal_E: constant Float := Pi*E;

Dies ist erlaubt, weil Pi und E *zuvor* bereits deklariert wurden!

Eine **<Konstantendeklaration>** hat also die Form:

<Bezeichner>**:** constant <Datentyp> **:=** <Ausdruck> **;**

<Anweisungsteil> ist eine endliche Folge von elementaren und strukturierten Anweisungen. Die wichtigste *elementare Anweisung* ist die Zuweisung. Sie hat die Form

<Variable> **:=** <Ausdruck> **;**

Eine weitere elementare Anweisung ist das "nichts tun": null**;**

Weiterhin benötigen wir elementare Anweisungen für die Ein- und Ausgabe:

Get(<Variable>) **;**

Put(<Ausdruck>) **;**

Die in Get eingesetzte "Variable" muss selbstverständlich im Deklarationsteil deklariert worden sein. Der <Ausdruck> ist im einfachsten Fall eine Variable; es kann auch eine Textkonstante der Form **"**<Text>**"** sein wie in: Put("N = "); Put(N);

Wir behandeln zu Beginn der Vorlesung zunächst folgende fünf *strukturierte Anweisungen:*

<Anweisung> <Anweisung>

if <Bedingung> then <Anweisung> end if;

if <Bedingung> then >Anweisung> else <Anweisung> end if;

while <Bedingung> loop <Anweisung> end loop;

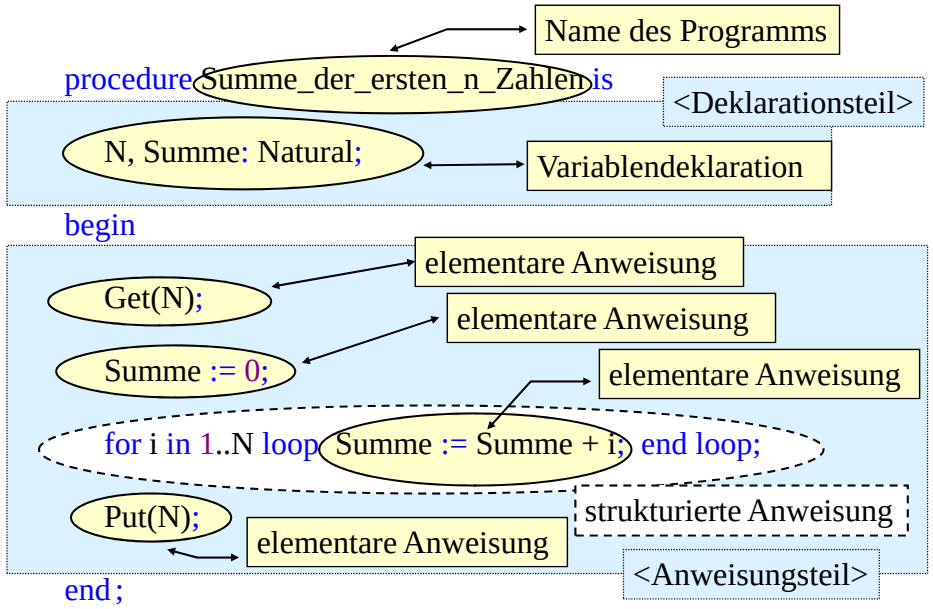
for <Bezeichner> in <Bereich> loop <Anweisung> end loop;

Bezeichnungen: sequentielle Ausführung von Anweisungen oder Nacheinanderausführung, einseitige Alternative, zweiseitige Alternative, while-Schleife und for-Schleife (oder Laufschleife). Die "Alternativen" gehören zu den sog. bedingten Anweisungen.

Beispiel: Berechne die Summe der ersten n Zahlen.
Das Programm ist wie die Fakultaet aufgebaut, nur wird die dortige Multiplikation durch die Addition und die Variable Fak durch die Variable Summe (anfangs 0) ersetzt:

```
procedure Summe_der_ersten_n_Zahlen is
    N, Summe: Natural;
begin
    Get(N);
    Summe := 0;
    for i in 1..N loop  Summe := Summe + i;  end loop;
    Put(N);
end;
```

Prüfen Sie genau nach, dass dieses Programm den geforderten Aufbau eines Ada-Programms besitzt.



Überprüfung, dass das Programm korrekt aufgebaut ist.

Lässt man das Programm nun laufen:

```
with Ada.Integer_Text_IO;  use Ada.Integer_Text_IO;
procedure Summe_der_ersten_n_Zahlen is
    N, Summe: Natural;
begin
    Get(N);
    Summe := 0;
    for i in 1..N loop  Summe := Summe + i;  end loop;
    Put(N);
end;
```

so liefert die Eingabe 5 die Ausgabe 5, anstatt 15. Dieses Programm enthält also einen Fehler. Da das Programm korrekt aufgebaut ist, handelt es sich um einen "logischen Fehler", der nicht in der Syntax (= formal korrekter Aufbau), sondern in den gewählten Anweisungen enthalten ist. Diesen Fehler erkennt man hier schnell, da er nur in der Ausgabe-Anweisung steckt: Man muss einfach Put(N) durch Put(Summe) ersetzen.

1.5 Beispiele

Vorbemerkung "Felder" (arrays)

Vektoren sind Ihnen aus der Schule geläufig. Oft muss man eine Folge von Variablen verwalten, die der Reihe nach durchnummeriert sind: $x = (x_1, x_2, ..., x_n)$ soll ein Vektor reeller Werte sein. Solch einen Vektor nennt man "Feld" (engl.: array). Wir stellen ihn in Ada dar, indem wir den Namen (hier: "x"), den "Indexbereich" (hier: "von 1 bis n") und den Datentyp (hier: "Float") angeben. Die allgemeine Deklaration lautet somit <Name>: array(<Indexbereich>) of <Inhaltsbereich>). Konkret:

X: array (1..n) of Float;

und man verwendet x_i in der Form X(i) im Programm. Genau so definiert man Matrizen ganzer Zahlen (Vektoren von Vektoren):

A: array (1..n,1..m) of Integer;

und verwendet die einzelnen Variablen in der Form A(i,j).

Vorbemerkung "Modulo-Rechnen" (mod)

Eine wichtige Funktion auf den natürlichen Zahlen (einschließlich der Null) ist die "Modulo"-Funktion mod: $\mathbb{N}_0 \times \mathbb{N} \to \mathbb{N}_0$:
a mod b = Rest, der bei der Division von a durch b übrig bleibt.

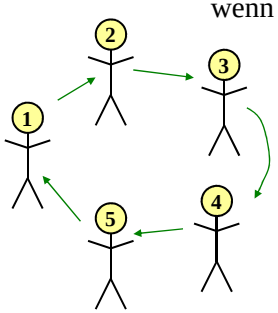
Beispiele:
7 mod 2 = 1, 7 mod 3 = 2, 12 mod 3 = 0, 3 mod 7 = 3.

Eigenschaften für alle $a, a_1, a_2 \in \mathbb{N}_0$ und $b \in \mathbb{N}$:
(i) $0 \le a \bmod b \le b-1$
(ii) $(a_1+a_2) \bmod b = ((a_1 \bmod b) + (a_2 \bmod b)) \bmod b$
(iii) $(a_1 \cdot a_2) \bmod b = ((a_1 \bmod b) \cdot (a_2 \bmod b)) \bmod b$

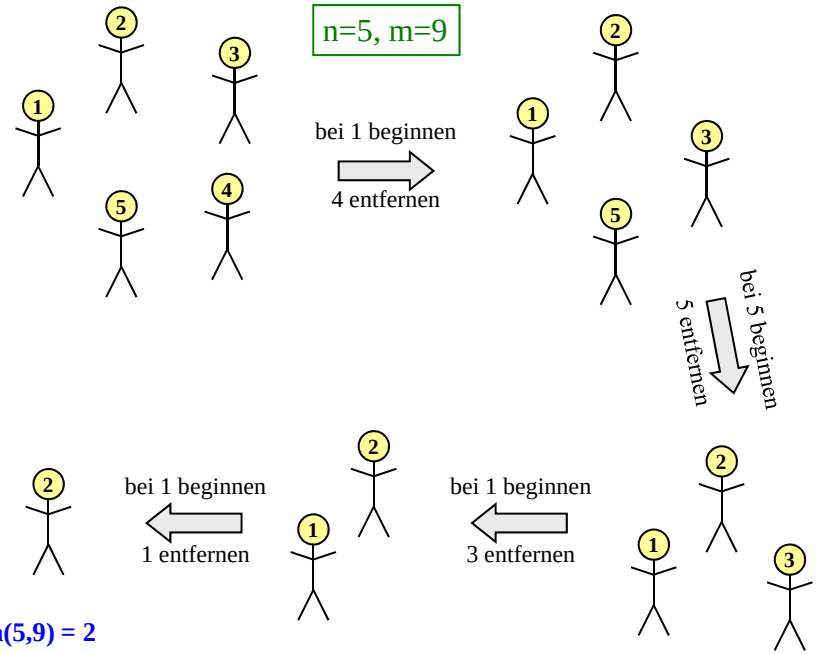
Beispiele:
Die Modulo-Funktion wird im Euklidischen Algorithmus (ggT-Berechnung) verwendet, siehe Abschnitt 1.6. Sie wird bei Kontrollrechnungen benutzt, da die Summe vieler Zahlen modulo einer Zahl gleich der viel einfacher zu bestimmenden Summe der Modulo-Werte ist. Man kann sie bei Zahlensystemen einsetzen. (→ Abschnitt 2.4.8 chinesischer Restklassensatz und Kapitel 14.1)

1.5.1 Beispiel "Abzählreim"

(1) Die Ideen umgangssprachlich aufschreiben.
n Kinder stehen im Kreis. Mit einem Reim aus m Silben wird (beginnend mit dem Kind 1) Kind nach Kind aus dem Kreis entfernt, bis nur noch ein Kind übrig ist. Dies definiert eine Funktion a: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ durch
a(n,m) = k ⇔ Das Kind mit der Nummer k bleibt von n Kindern beim Abzählen mit einem Reim der Länge m übrig, wenn das Abzählen mit Kind 1 beginnt.



In dieser Skizze ist n = 5. Der Abzählreim habe die Länge m = 9 (z.B. "Eene meene muh und raus bist du"). Man beginne bei Kind 1. Dann wird als erstes das Kind mit der Nummer 4 aus dem Kreis entfernt.



(2) Präzise Formulierung.

Definition 1.5.1.1: Die Auswahlfunktion a: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ wird definiert durch:

Stelle n Personen im Kreis auf. Beginne mit der Person i = 1. Ermittle ab hier die m-te Person, wobei die Person i mitzählt und im Kreis zur jeweils nächsten Person gegangen wird. Entferne diese m-te Person aus dem Kreis; i sei die Nummer der nächsten Person hinter der entfernten Person. Wiederhole dieses Verfahren dann mit den verbleibenden n-1, n-2, ... Personen, bis nur noch eine Person übrig ist.

Setze a(n,m) = die Nummer dieser verbleibenden Person.

Dies ist noch "zu anschaulich". Wir wollen präzise die Nummer der zu entfernenden Person ermitteln. Wir gehen von k Personen aus, anfangs ist k = n. Hierzu schreiben wir die Nummern der Personen in einen Vektor. Anfangs lautet dieser Vektor: P = (1, 2, 3, 4, ..., n). Wir starten mit der Person an der i-ten Position; diese hat die Nummer P(i).

Um die Position j der zu entfernenden Person zu berechnen, müssen wir m zu (i-1) addieren, weil das Zählen mit der Position i als erster (und nicht als nullter) Position beginnt. Es genügt jedoch, nur (m mod k) Positionen weiter zu zählen, weil man nach k Schritten ja immer wieder bei der gleichen Position im Kreis ankommt. Wir bilden also $j = (i-1) + (m \bmod k)$. Wenn $1 \leq j \leq k$ ist, sind wir fertig. Falls j = 0 ist, so ist im Kreis die Position k gemeint, ist j > k, dann muss einfach nur k von j subtrahiert werden.

Seien also: i die Nummer der Position, ab der die Zählung mit 1 beginnt, k die Anzahl der Personen im Kreis und m die Länge des Abzählreims, dann erhalten wir die Position der zu entfernenden Person j durch das Programmstück

```
j := (i-1) + (m mod k);
if j = 0 then j := k; end if;
if j > k then j := j-k; end if;
```

Der Vektor P = (P(1),P(2),...,P(k)) ist nun zu ersetzen durch (P(1),...,P(j-1),P(j+1),...,P(k)), also einen Vektor aus nur noch k-1 Komponenten. Dann erniedrigt man k um 1 und setzt i auf j, denn dort steht ja nun die Person, ab der weitergezählt werden muss). Diese Berechnung wiederholt man, bis k = 1 geworden ist.

Wir geben nun sofort den Algorithmus in Ada an.

```
procedure Abzaehlfunktion is
N: constant Integer := 5; M: constant Integer := 9;
I, J, K: Integer;  P: array(1..N) of Integer;
begin
  for Z in 1..N loop P(Z) := Z; end loop;
  I := 1; K := N;
  while K > 1 loop
    J := (I-1) + (M mod K);
    if J = 0 then J := K; end if;
    if J > K then J := J-K; end if;  -- hier gilt: 1 <= J <= K
    for Z in J+1..K loop P(Z-1) := P(Z); end loop;
    I := J; K := K-1;
  end loop;
  New_Line; Put("N ="); Put(N,4); Put(", M ="); Put(M,4);
  Put("; Letzter ="); Put(P(1),4);
end;
```

(3) Ermitteln von Eigenschaften.

Eine Eigenschaft erkennt man sogleich: Wenn

m mod k = (m+s) mod k für alle k = 1, 2, ..., n

ist, dann kommt auf jeden Fall für m und für m+s der gleiche Wert heraus, da dann das gleiche j berechnet wird.

Das kleinste s mit dieser Eigenschaft ist das kgV(1,...,n) = kleinstes gemeinsames Vielfaches der Zahlen von 1 bis n.

Also gilt für alle natürlichen Zahlen n :

a(n, m) = a(n, m+kgV(1,...,n)) für alle natürlichen Zahlen m.

(4) Lösungs-Algorithmus
(5) Programm in Ada
} bereits erledigt

(6) Testen, Verifizieren, Messen

Sie können Messungen wie in Abschnitt 1.3 vornehmen. Auch können Sie die Deklarationen für N und M entfernen und dafür zwei Schleifen

```
for N in 1..12 loop
   for M in 1..15 loop
```

um die Anweisungen legen, die P(1)-Werte in einem Feld A: array (1..12,1..15) of Integer ablegen und diese Werte am Ende ausdrucken, wodurch eine Tabelle der a-Funktion entsteht, siehe nächste Folie. Details siehe unter 1.7.

Die Zeit hierfür ist wiederum kaum messbar; am längsten dauert die Ausgabe.

m

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2:	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2
3:	3	3	2	2	1	1	3	3	2	2	1	1	3	3	2
4:	4	1	1	2	2	3	2	3	3	4	4	1	4	1	1
5:	5	3	4	1	2	4	4	1	2	4	5	3	2	5	1
6:	6	5	1	5	1	4	5	3	5	2	4	3	3	1	4
7:	7	7	4	2	6	3	5	4	7	5	1	1	2	1	5
8:	8	1	7	6	3	1	4	4	8	7	4	5	7	7	4
9:	9	3	1	1	8	7	2	3	8	8	6	8	2	3	1
10:	10	5	4	5	3	3	9	1	7	8	7	10	5	7	6
11:	11	7	7	9	8	9	5	9	5	7	7	11	7	10	10
12:	12	9	10	1	1	3	12	5	2	5	6	11	8	12	1

Die Abzählfunktion a(n,m) für kleinere Werte.

Hinweis: 1.5.2 wird in der Vorlesung nicht behandelt.

1.5.2 Beispiel "Harmonische Funktion"

(1) Die Ideen umgangssprachlich aufschreiben.

In vielen Anwendungen benötigt man den natürlichen Logarithmus ln(y), also die Umkehrfunktion der Exponentialfunktion e^z für die Zahl e ≈ 2,718281828459. Oft benötigt man den Logarithmus nur für die natürlichen Zahlen.
Wegen $\ln(n) = \int_1^n \frac{1}{x}\,dx$
kann man versuchen, die Summe $\sum_{i=1}^{n} \frac{1}{i}$ als Näherung zu verwenden. Wir streben für diese Summe möglichst genaue Werte an.

(2) Präzise Formulierung.

Definition 1.5.2.1: Die Harmonische Funktion H: $\mathbb{N}_0 \to \mathbb{R}$ wird definiert durch:

$H(0) = 0$

$H(n) = H(n-1) + \frac{1}{n}$ für $n > 0$.

Offenbar gilt (schrittweise einsetzen):

$$H(n) = \sum_{i=1}^{n} \frac{1}{i}$$

Wir wollen zunächst die Beziehung zwischen H(n) und ln(n) klären und dann H(n) berechnen.

(3) Ermitteln von Eigenschaften.

Funktionstabelle der ersten zwölf Werte:

n	H(n) exakt	Näherungswert
1	1/1	1.000000000000
2	3/2	1.500000000000
3	11/6	1.833333333333
4	25/12	2.083333333333
5	137/60	2.283333333333
6	49/20	2.450000000000
7	363/140	2.592857142857
8	761/280	2.717857142857
9	7129/2520	2.828968253968
10	7381/2520	2.928968253968
11	83711/27720	3.019877344877
12	86021/27720	3.103210678211

Aus der Definition folgt: H(n+1) > H(n), d.h., H ist eine streng monoton wachsende Funktion.

$$H(2^n) = \sum_{i=1}^{2^n} \frac{1}{i} = \sum_{i=1}^{2^{n-1}} \frac{1}{i} + \sum_{i=2^{n-1}+1}^{2^n} \frac{1}{i}$$

$$> H(2^{n-1}) + \sum_{i=2^{n-1}+1}^{2^n} \frac{1}{2^n} = H(2^{n-1}) + \frac{1}{2}$$

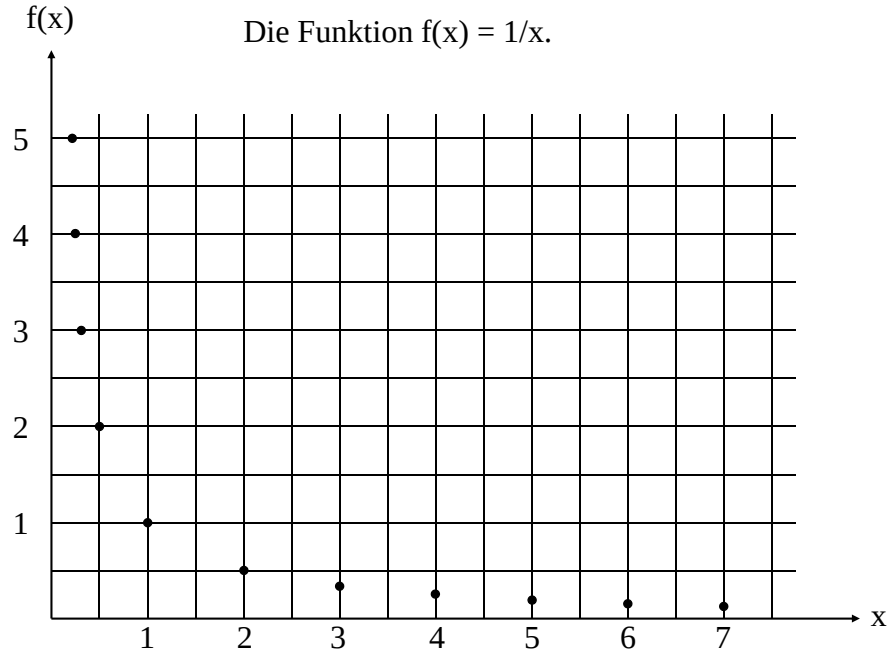
Es gilt für jedes n:

$H(2^n) > H(2^{n-1}) + 1/2 > H(2^{n-2}) + 1 > H(2^{n-3}) + 3/2 > ...$
$> H(2^{n-n}) + n/2 = H(2^0) + n/2 > n/2.$

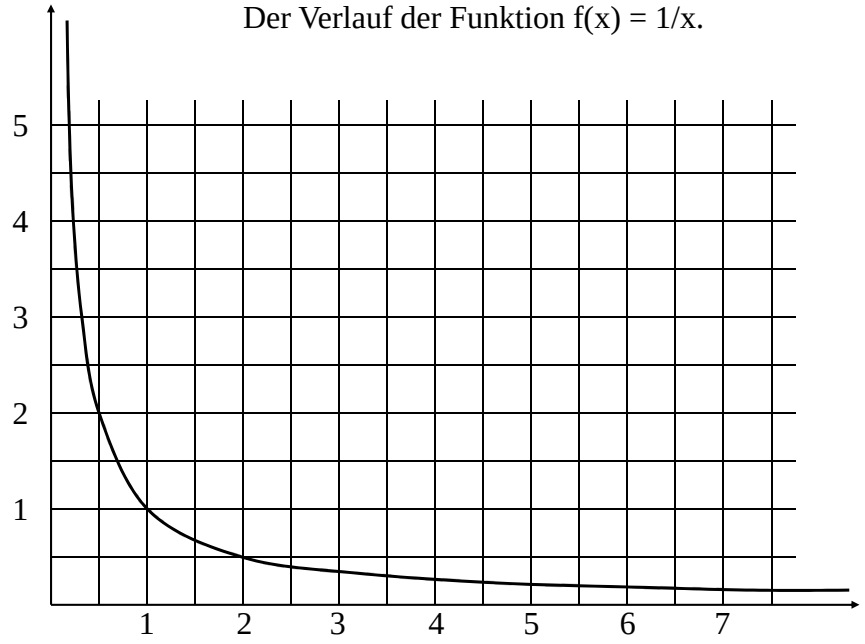
Zeigen Sie auf die gleiche Weise: $H(2^n) < n+1$ für alle $n > 0$.

Feststellung 1.5.2.2:
Die Funktion H ist streng monoton und wächst unbeschränkt.
Speziell gilt für alle $n > 0$: $n/2 < H(2^n) < n+1$.

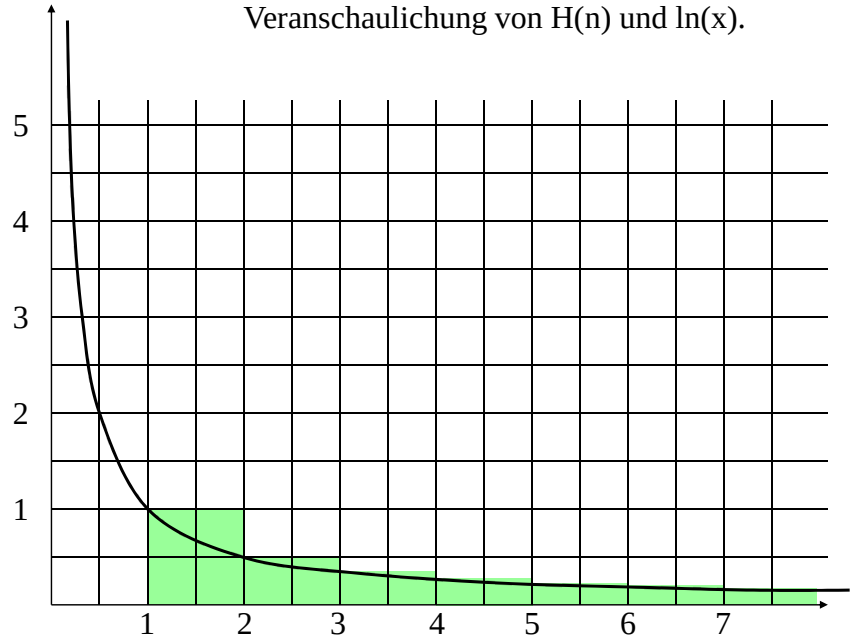
Die Funktion f(x) = 1/x.



Der Verlauf der Funktion f(x) = 1/x.



Veranschaulichung von H(n) und ln(x).



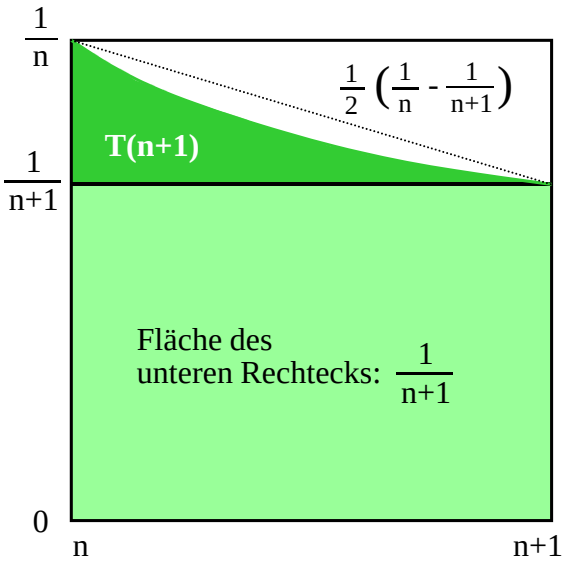
Betrachte den Bereich von n bis n+1.

T(n+1) sei der Anteil, um den der natürliche Logarithmus beim Übergang von n nach n+1 mehr als 1/(n+1) wächst.

Setze also
T(0) = 0, T(1) = 0,
ln(n+1) - ln(n)
= T(n+1) + 1/(n+1)
für n > 0.

Es gilt offenbar:

$$T(n+1) < \frac{1}{2}\left(\frac{1}{n} - \frac{1}{n+1}\right)$$



Fortsetzung: (3) Eigenschaften:

$$\ln(n) = (H(n) - 1) + \sum_{i=2}^{n} T(i) .$$

Hieraus folgt $0.5 < H(n) - \ln(n) \le 1$ wegen:

$$H(n) - \ln(n) = 1 - \sum_{i=2}^{n} T(i), \text{ d.h.: } H(n) - \ln(n) \le 1.$$

$$H(n) - \ln(n) = 1 - \sum_{i=2}^{n} T(i) > 1 - \frac{1}{2}\sum_{i=2}^{n}\left(\frac{1}{i-1} - \frac{1}{i}\right)$$

$$= 1 - \frac{1}{2}\left(1 - \frac{1}{n}\right) > 0.5$$

Die Differenz H(n) - ln(n) bewegt sich also im Intervall von 0.5 bis 1. Diese Differenz ist streng monoton fallend, denn es gilt für jedes n:

$$(H(n+1) - \ln(n+1)) - (H(n) - \ln(n)) =$$

$$(1 - \sum_{i=2}^{n+1} T(i)) - (1 - \sum_{i=2}^{n} T(i)) = -T(n+1) < 0.$$

Da H(n) - ln(n) mit wachsendem n in jedem Schritt kleiner wird, andererseits aber durch 0.5 nach unten beschränkt ist, muss es einen Grenzwert C geben mit

$$C = \lim_{n \to \infty} \left(H(n) - \ln(n)\right)$$ (sog. Eulersche Konstante).

Hinweis: C wurde vor 200 Jahren von Euler als 0.5772156649... berechnet.

(4) Lösungs-Algorithmus

Vorgehensweise:
Wir berechnen nacheinander H(i) für i = 0, 1, 2, ..., n.

Gegeben sei also eine natürliche Zahl n≥0. Diese wird in einer Variablen N abgelegt. In der Variablen H werden die Zwischenergebnisse H(0), H(1), H(2), ..., H(i) und am Ende das Ergebnis H(n) stehen.

Lies n in die Variable N ein.
Setze H anfangs auf den Wert 0.
Für i = 1, 2, ..., N wiederhole: Erhöhe H um 1/i.
Gib den Wert von H als Ergebnis aus.


```
procedure Harmonische_Funktion is
N, i: Natural; H: rationale_Zahl;
begin
        Get(N);  H := 0;
        for i := 1 to N do  H := H + 1/i  od;
        Put(H)
end
```

Das Verfahren entspricht genau der Definition 1.5.2.1.

Das Problem besteht in dem Begriff "rationale_Zahl".
Was ist das genau? Wie stellt man solche Zahlen dar?
Wie bildet man 1/i und wie addiert man rationale Zahlen?
(Das behandeln wir in Abschnitt 1.6.)

(5) Übertragen in ein Ada-Programm

Wir entscheiden uns zunächst dafür, die Zahlen als reelle Zahlen aufzufassen. Hierfür gibt es in Ada den Datentyp Float. Die Schnittstelle nach außen (with, use) muss um die reellen Zahlen erweitert werden.

Wie schon erwähnt, achtet Ada peinlich genau darauf, dass man Datentypen nicht durcheinander wirft. Integer wird nicht automatisch als Teilmenge von Float angesehen, sondern die Daten müssen explizit umgewandelt (konvertiert) werden. Die Umwandlung eines Integer-Wertes X in einen Float-Wert geschieht durch: Float(X). Das Umgekehrte, also die Umwandlung einer reellen Zahl Y in die nächst liegende ganze Zahl, erfolgt durch Integer(Y). Weiterhin dürfen wir nicht die ganze Zahl 1, sondern wir müssen die reelle Zahl 1.0 verwenden.

Somit erhalten wir das Programm

```
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Ada.Float_Text_Io; use Ada.Float_Text_Io;

procedure Harmonische_Funktion is
N: Natural; H: Float;
begin
        Get(N);
        H := 0.0;
        for I in 1..N loop  H := H + 1.0/ Float(I); end loop;
        Put(H);
end;
```


Wir wollen das Programm nun ein wenig modifizieren.

(1) Es sollen die H-Werte tabellenartig von 1 bis N ausgegeben werden. Dies erreicht man durch

New_Line; Put(I); Put(H);

in der for-Schleife. Man muss aber eine weitere with-Zeile hinzufügen, um New_Line verwenden zu können.

(2) Die Tabelle soll auch "schön" aussehen. Insbesondere sollen alle Zahlen ordentlich untereinander stehen. Dies erreicht man, indem man in die Put-Anweisung weitere Zahlen einfügt und zwar:

Put(I,a) bewirkt bei einer Integer-Variablen I, dass der Wert von I mit genau a Stellen ausgegeben wird.

Put(X,a,b,c) bewirkt, dass eine reelle Zahl mit a Stellen vor dem Dezimalpunkt, b Stellen nach dem Dezimalpunkt und c Stellen für einen Exponententeil ausgedruckt wird.

So entsteht das Programm

```
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Ada.Float_Text_Io; use Ada.Float_Text_Io;
with Text_Io; use Text_Io;

procedure Harmonische_Funktion is
N: Natural; H: Float;
begin
        Get(N);
        H := 0.0;
        New_Line;
        for I in 1..N loop
                H := H + 1.0/ Float(I);
                Put(I,6); Put(H,4,9,0); New_Line;
        end loop;
end;
```


Gibt man 60 ein, so erhält man eine Tabelle mit 60 Zeilen und viel freiem Platz. Man sollte die Tabelle daher in drei Spalten drucken, indem man New_Line nur jedes dritte Mal benutzt: if (I mod 3) = 0 then New_Line; end if;
Dies führt zu folgendem Programm:

```
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Ada.Float_Text_Io; use Ada.Float_Text_Io;
with Text_Io; use Text_Io;

procedure Harmonische_Funktion_mit_Tabellenausgabe is
N: Natural; H: Float;
begin
   Get(N);  H := 0.0;  New_Line;
   for I in 1..N loop
      H := H + 1.0/ Float(I);
      Put(I,6); Put(H,4,9,0); if  (I mod 3) = 0  then New_Line; end if;
   end loop;
end;
```

(6) Testen, Verifizieren, Messen

Die Ausgabe dieses Programms bei Eingabe 60:

1	1.000000000	2	1.500000000	3	1.833333373
4	2.083333492	5	2.283333540	6	2.450000286
7	2.592857361	8	2.717857361	9	2.828968525
10	2.928968430	11	3.019877434	12	3.103210688
13	3.180133820	14	3.251562357	15	3.318228960
16	3.380728960	17	3.439552546	18	3.495108128
19	3.547739744	20	3.597739697	21	3.645358801
22	3.690813303	23	3.734291553	24	3.775958300
25	3.815958261	26	3.854419708	27	3.891456842
28	3.927171230	29	3.961653948	30	3.994987249
31	4.027245522	32	4.058495522	33	4.088798523
34	4.118210316	35	4.146781921	36	4.174559593
37	4.201586723	38	4.227902412	39	4.253543377
40	4.278543472	41	4.302933693	42	4.326743126
43	4.349998951	44	4.372726440	45	4.394948483
46	4.416687489	47	4.437963963	48	4.458797455
49	4.479205608	50	4.499205589	51	4.518813610
52	4.538044453	53	4.556912422	54	4.575430870
55	4.593612671	56	4.611469746	57	4.629013538
58	4.646255016	59	4.663204193	60	4.679871082

Die Ergebnisse sind manchmal ab der 7. Stelle nicht mehr korrekt. Das liegt an den "Rundungsfehlern" beim Datentyp Float. Denn der Wert 1/i kann im Rechner nur auf etwa 7 Stellen nach dem Komma genau dargestellt werden und bei der Summation erhöhen sich diese Fehler meist noch.

Die Ergebnisse wären genau, wenn man die Werte H(i) als rationale Zahlen, also als ein Paar (Zähler, Nenner) schreiben würde. Dann muss man aber die arithmetischen Operationen auf rationalen Zahlen exakt nachvollziehen.
(Ende des Beispiels 1.5.2)

Genau dies wollen wir nun tun. Wir definieren den Datentyp "rationale Zahl" und geben die Addition an. Dies wird bereits ausreichen, um eine rationale Zahl als Ergebnis der harmonischen Funktion zu erhalten.

1.6 Beispiel "Rationale Zahlen"

(1) Die Ideen umgangssprachlich aufschreiben.

Es soll ein Programm geschrieben werden, das rationale Zahlen verarbeiten kann. Eine rationale Zahl ist ein Paar (z,n) mit einer ganzen Zahl z und einer natürlichen Zahl n.

Diese Darstellung ist *nicht eindeutig*, da (z, n) und (r·z, r·n) für jede natürliche Zahl r > 0 die gleiche rationale Zahl bezeichnen. Kürzt man jedoch so weit wie möglich, so ist die Darstellung eindeutig, insbesondere wird die Zahl Null durch (0,1) dargestellt.

Im Alltag lässt man auch negative Zahlen im Nenner zu. Dies wollen wir hier aber nicht erlauben.

(2) Präzise Formulierung.

Definition 1.6.1: Bilde die Menge von Paaren

$$Q' = \{(z, n) \mid z \in \mathbf{Z} \text{ und } n \in \mathbf{IN}\}.$$

Zwei Darstellungen (z_1, n_1) und (z_2, n_2) heißen gleich, wenn $z_1 \cdot n_2 = z_2 \cdot n_1$ gilt. Es sei

$$[(z, n)] = \{(z', n') \mid (z', n') \text{ und } (z, n) \text{ sind gleich}\}$$

die Klasse der zu (z, n) gleichen Paare.
Dann heißt

$$\mathbf{Q} = \{[(z, n)] \mid z \in \mathbf{Z} \text{ und } n \in \mathbf{IN}\}$$

die Menge der rationalen Zahlen.

1.6.2 Normalform: Zu jeder Klasse [(z, n)] kann man das Paar

(z/k, n/k), falls $z \neq 0$ ist, mit $k = ggT(z,n)$, bzw.
(0,1), falls $z = 0$ ist,

als eindeutigen Repräsentanten wählen.

größter gemeinsamer Teiler von z und n

Definition 1.6.3: Auf **Q** sind die üblichen Operationen ein- und zweistelliges Plus und Minus, Absolutbetrag, Multiplikation, Division sowie die Vergleiche definiert:

$-[(z,n)] = [(-z,n)]$, $+[(z,n)] = [(z,n)]$,

$\underline{abs}([(z,n)]) = [(\underline{abs}(z),n)]$,

$[(z_1,n_1)]+[(z_2,n_2)] = [z_1 \cdot n_2 + z_2 \cdot n_1, n_1 \cdot n_2)]$

$[(z_1,n_1)]-[(z_2,n_2)] = [z_1 n_2 - z_2 n_1, n_1 \cdot n_2)]$

$[(z_1,n_1)] \cdot [(z_2,n_2)] = [(z_1 \cdot z_2, n_1 \cdot n_2)]$

$[(z_1,n_1)] / [(z_2,n_2)] = [(z_1 \cdot n_2, n_1 \cdot z_2)]$ für $z_2 \neq 0$ (sonst undefiniert)

$[(z_1,n_1)] < [(z_2,n_2)] \Leftrightarrow z_1 \cdot n_2 < n_1 \cdot z_2$

$[(z_1,n_1)] = [(z_2,n_2)] \Leftrightarrow z_1 \cdot n_2 = n_1 \cdot z_2$ und analog für $>, \leq, \geq, \neq$.

(Siehe Mathematik: Mit diesen Operationen ist **Q** ein Körper.)

(3) Ermitteln von Eigenschaften (Analysieren des Problems)

Dieser Teil soll hier entfallen, da wir die rationalen Zahlen als bekannt vorausgesetzen und die Definitionen direkt in Programmstücke übertragen werden.

(4) Lösungs-Algorithmus

Anstelle von **Q** verwenden wir die Menge Q' und benutzen die Normalform 1.6.2, außer in Zwischenrechnungen. Somit wird der Datentyp "rationale_Zahl" durch Paare aus Integer und Natural beschrieben.
Die Operationen werden mit Hilfe der bereits bekannten Operationen der ganzen Zahlen unmittelbar so durchgeführt, wie es in der Definition 1.6.3 angegeben ist.
Wir formulieren alles sofort in Ada.

(5) Programm in Ada

Definition des Datentyps der rationalen Zahlen. Zwischen record und end record werden die einzelnen Komponenten aufgeschrieben; jede Komponente wird durch einen Bezeichner ("Selektor") identifiziert (der Datentyp "Positive" bezeichnte alle natürlichen Zahlen ab der Zahl 1):

```
type Rational is record        -- Definition von Rational
      Zaehler: Integer;        -- die Komponente "Zaehler": ganze Zahl
      Nenner: Positive;        -- die Komponente "Nenner": nat. Zahl > 0
end record;
```

Nun folgt ein *Programmstück zur Multiplikation:*

```
X, Y: Rational;          -- zwei Variablen vom Datentyp Rational
Z: Rational;             -- Ergebnis der Multiplikation
begin                    -- Berechne die beiden Komponenten
   Z.Zaehler := X.Zaehler * Y.Zaehler;
   Z.Nenner := X.Nenner * Y.Nenner;
end;
```

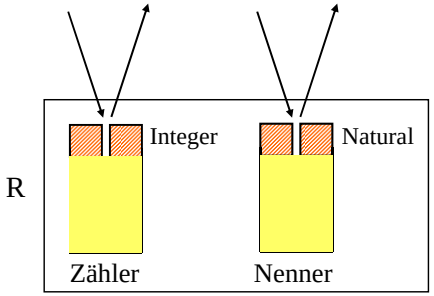

Es wurden also Variablen X und Y eingeführt, die aus zwei Komponenten bestehen. Die eine Komponente wird durch X.Zaehler, die andere durch X.Nenner angesprochen. Man spricht von der "Punkt-Notation" und meint damit, dass man auf die Komponenten einer Variablen über die jeweiligen Bezeichner (die sog. "Selektoren" Zaehler und Nenner) zugreifen kann, indem man

X.<Selektor>

schreibt.

Wenn X den Wert (4,13) besitzt, also die rationale Zahl $\frac{4}{13}$ bezeichnet, so kann man den Wert 4 durch X.Zaehler und den Wert 13 durch X.Nenner ansprechen und ggf. verändern.

Modellvorstellung: Variablen lassen sich ineinander schachteln. Gedankliche Zusammenfassung mehrerer Behälter zu einer Variablen zu einem "strukturierten Behälter" (wie eine Werkzeugkiste oder eine Besteckschublade). Die Operationen Einfügen und Auslesen und der Kontrollmechanismus bleiben auf die "innersten" Behälter beschränkt, in denen die Werte gespeichert sind. Diese werden über R.Zaehler und R.Nenner angesprochen.



Nun wollen wir das Programmstück für die Multiplikation Mult: $\mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q}$ so deklarieren, dass wir diese überall im Programm benutzen können. In Ada formuliert man dies als Funktion (Schlüsselwort function) mit den Parametern A und B, beide vom Datentyp Rational; auch das Ergebnis ist vom Typ Rational. Formulierung in Ada:

```
function Mult(X, Y: Rational) return Rational is
Z: Rational;
begin
        Z.Zaehler := X.Zaehler * Y.Zaehler;
        Z.Nenner := X.Nenner * Y.Nenner;
        return Z;
end;
```


Ähnlich können wir die Funktion Negation: $\mathbb{Q} \rightarrow \mathbb{Q}$ deklarieren, die zu einer rationalen Zahl x die Zahl -x liefert:

```
function Negativ(X: Rational) return Rational is
Z: Rational;
begin
        Z.Zaehler := -X.Zaehler;
        Z.Nenner := X.Nenner;
        return Z;
end;
```

Hinweis: Man hätte auch die Multiplikation mit der rationalen Zahl (-1,1) durchführen, also folgende Funktion Neg deklarieren können:

```
function Neg(X: Rational) return Rational is
Z: Rational;
begin   Z.Zaehler := -1; Z.Nenner := 1;
        return Mult(X, Z);
end;
```


Funktionen werden im <Deklarationsteil> definiert.

Funktionen können in Ausdrücken wie gewohnt eingesetzt werden. Z.B. kann man Mult mit zwei konkreten rationalen Zahlen verwenden ("aufrufen") und erhält deren Produkt als Ergebnis.

Das heißt: Funktionen besitzen ("formale") Parameter, die bei ihrer Verwendung durch aktuelle Werte ("aktuelle Parameter") ersetzt werden. Mit diesen aktuellen Werten wird die Funktion dann durchgerechnet. Eine solche Verwendung nennt man einen Aufruf.

Der Aufruf Mult((20,17),(34,12)) liefert also das Ergebnis (680,204).

Die Normalform für (680,204) lautet (10,3). Man muss also noch durch den größten gemeinsamen Teiler, den ggT, teilen! Dies wollen wir sofort nachholen.

Wir brauchen also eine Funktion ggT: $\mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, die zu zwei natürlichen Zahlen deren größten gemeinsamen Teiler liefert. Diese Funktion ist aus der Schule bekannt. Man berechnet sie mit dem "Euklidischen Algorithmus", indem man die Eigenschaften (wird erläutert später in 2.4.2-2.4.4.)

ggT(a,b) = ggT(b, a mod b) für b > 0 und
ggT(a,0) = a

ausnutzt. Als Funktion in Ada geschrieben:

```
function ggT(A, B: Natural) return Natural is
R, S, T: Natural;
begin  S := A; T := B;
       while T /= 0 loop
            R := S mod T; S := T; T := R;
       end loop;
       return S;
end;
```

Prüfen Sie, was geschieht, wenn a = 0 oder b = 0 oder beide Zahlen gleich 0 sind!

Mit Hilfe der Funktion ggT formulieren wir die Funktion Mult nun so, dass sie stets eine Normalform als Ergebnis liefert, wobei wir den Sonderfall Null zu beachten haben:

```
function Mult(X, Y: Rational) return Rational is
Z: Rational; G: Natural;
begin
      Z.Zaehler := X.Zaehler * Y.Zaehler;
      Z.Nenner := X.Nenner * Y.Nenner;
      if Z.Zaehler = 0 then Z.Nenner := 1;
      else  G := ggT(abs(Z.Zaehler), Z.Nenner);
            Z.Zaehler := Z.Zaehler/G;
            Z.Nenner := Z.Nenner/G;
      end if;
      return Z;
end;
```


Nun ist klar, wie Addition und Subtraktion zu deklarieren sind:

```
function Addition(X, Y: Rational) return Rational is
Z: Rational; G: Natural;
begin
    Z.Zaehler := X.Zaehler * Y.Nenner + Y.Zaehler * X.Nenner;
    Z.Nenner := X.Nenner * Y.Nenner;
    if Z.Zaehler = 0 then Z.Nenner := 1;
    else G := ggT(abs(Z.Zaehler), Z.Nenner);
         Z.Zaehler := Z.Zaehler/G;
         Z.Nenner := Z.Nenner/G;
    end if;
    return Z;
end;
```



```
function Subtraktion(X, Y: Rational) return Rational is
Z: Rational; G: Natural;
begin
    Z.Zaehler := X.Zaehler * Y.Nenner - Y.Zaehler * X.Nenner;
    Z.Nenner := X.Nenner * Y.Nenner;
    if Z.Zaehler = 0 then Z.Nenner := 1;
    else G := ggT(abs(Z.Zaehler), Z.Nenner);
         Z.Zaehler := Z.Zaehler/G;
         Z.Nenner := Z.Nenner/G;
    end if;
    return Z;
end;
```

Hinweis: Man hätte die Subtraktion auch auf die Addition zurückführen können mittels der Funktion Subtrakt:

```
function Subtrakt(X, Y: Rational) return Rational is
begin return Addition(X, Negation(Y)); end;
```

Nun haben wir alles zusammen, um die harmonische Funktion H als rationale Zahl berechnen zu können. Schema:

```
with ...; use ....;
procedure Harmonische_Funktion_Rational is
  type Rational is ...
  N: Natural; H, Bruch: Rational;
  function ggT(A, B: Natural) return Natural is ...
  function Addition(X, Y: Rational) return Rational is ...
begin
        Get(N);  H.Zaehler := 0;  H.Nenner := 1;
        for I in 1..N loop
                Bruch.Zaehler:=1; Bruch.Nenner:=I;
                H := Addition(H, Bruch);
        end loop;
        New_Line; Put("Zähler: "); Put(H.Zaehler,9);
        New_Line; Put("Nenner: "); Put(H.Nenner,9);
end;
```


Es folgt das Programm für die harmonische Funktion.

Am Ende drucken wir noch H.Zaehler/H.Nenner als reelle Zahl zum Vergleich mit den Ergebnissen des Programms in Abschnitt 1.5.2 aus.

```
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Ada.Float_Text_Io; use Ada.Float_Text_Io;
with Text_Io; use Text_Io;

procedure Harmonische_Funktion_Rational is

type Rational is record      -- Definition von Rationalen Zahlen
   Zaehler: Integer;         -- die Komponente "Zaehler": ganze Zahl
   Nenner: Positive;         -- die Komponente "Nenner": nat. Zahl > 0
end record;

N: Natural; H, Bruch: Rational;
                             -- Bruch speichert die Zahl 1/i
```



```
function ggT(A, B: Natural) return Natural is
R, S, T: Natural;            -- größter gemeinsamer Teiler
begin   S := A; T := B;      -- Verfahren: Euklidischer Algorithmus
   while T /= 0 loop  R := S mod T; S := T; T := R;
   end loop;                 -- Hinweis: hier ist ggT(0,0) = 0, aber:
   return S;                 -- (0,0) tritt bei der harmonischen Funktion
end;                         -- nie als aktueller Parameter auf

function Addition(X, Y: Rational) return Rational is
Z: Rational; G: Natural;     -- X + Y = Z.   G speichert einen ggT
begin                        -- Berechnung gemäß Definition der Addition
   Z.Zaehler := X.Zaehler * Y.Nenner + Y.Zaehler * X.Nenner;
   Z.Nenner := X.Nenner * Y.Nenner;
   if Z.Zaehler = 0 then Z.Nenner := 1;
   else G := ggT(abs(Z.Zaehler), Z.Nenner);
        Z.Zaehler := Z.Zaehler/G;  Z.Nenner := Z.Nenner/G;
   end if;
   return Z;                 -- das Ergebnis ist in Normalform!
end;
```



```
-- Beginn des eigentlichen Programms zur Berechnung von H
begin
   Get(N);  H.Zaehler := 0;  H.Nenner := 1;
   for I in 1..N loop   -- Addiere den Bruch 1/i zu H, für i von 1 bis N.
        Bruch.Zaehler := 1; Bruch.Nenner := I;
        H := Addition(H, Bruch);
   end loop;
   New_Line; Put("Zähler: "); Put(H.Zaehler,9);
   Put("  Nenner: "); Put(H.Nenner,9);
   New_Line; Put("Zum Vergleich: ");
   Put(Float(H.Zaehler)/Float(H.Nenner),4,9,0);
end;
```

Beispiel: Das Programm liefert bei Eingabe 19 die Ausgabe:

```
Zähler: 275295799
Nenner:  77597520
Zum Vergleich:    3.547739744
```

(6) Testen, Verifizieren, Messen

Beispiel: Obiges Programm liefert bei Eingabe 19 die Ausgabe:

```
Zähler: 275295799      Nenner:  77597520
Zum Vergleich:    3.547739744
```

Ab N=20 erfolgt ein Constraint_Error wie in 1.3 (6). Dies lässt sich hin zu größeren Eingabezahlen hinausschieben, indem man durch den ggT bereits teilt, *bevor* die Produkte, die größer als 2^{31}-1 werden, gebildet werden.

Aufgabe für Sie: Verändern Sie das Programm so, dass es für größere Eingabezahlen als 19 noch funktioniert. Sie können auch mit dem kgV (= kleinstes gemeinsames Vielfaches) arbeiten. Fügen Sie rationale Operatoren hinzu, experimentieren Sie hiermit usw.

Wir wenden uns nun der Beschreibung des Deklarationsteils eines Ada-Programms und hier zunächst den Funktionen und Prozeduren zu.

1.7 Funktionen und Prozeduren

Der <Deklarationsteil> besteht aus einer Folge von Deklarationen. (Evtl. ist der <Deklarationsteil> leer.)

Es gibt verschiedene Arten von Deklarationen, z.B.:

<Variablendeklaration>	siehe 1.4
<Konstantendeklaration>	siehe 1.4
<Typdeklaration>	z.B. type Rational in 1.6
<Untertypdeklaration>	siehe 1.9.1
<Funktionsdeklaration>	z.B. function ggT in 1.6
<Prozedurdeklaration>	siehe im Folgenden

1.7.1 Eine Funktionsdeklaration ist von der Form

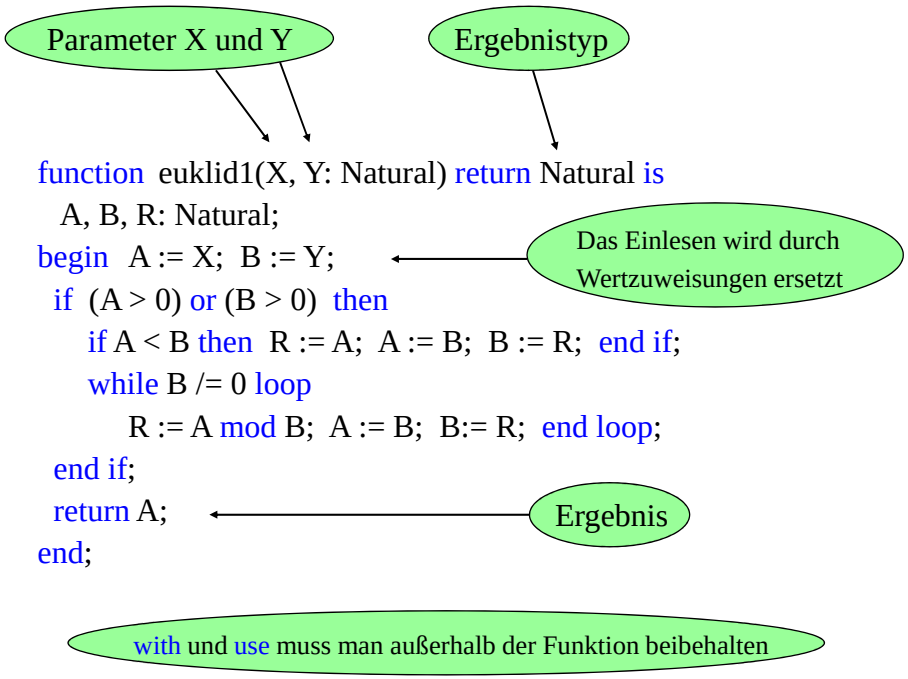
```
function <Name der Funktion> ( <Parameterteil> )      } Kopf
         return <Datentyp> is
  <Deklarationsteil>                                  }
begin                                                 } Rumpf
  <Anweisungsteil>                                    }
end <optional: Name der Funktion> ;
```

Zusätzliche Forderung: Durchläuft man <Anweisungsteil> in dieser Funktion, so muss man stets auf eine elementare Anweisung der Form return <Ausdruck>; stoßen. Der Wert dieses Ausdrucks ist das Ergebnis der Funktion. Dieser Wert muss genau von dem Datentyp sein, der in der Kopfzeile der Funktion hinter "return" steht.

Standardbeispiel ggT. Euklidischer Algorithmus (dieses Mal mit Abfragen am Anfang; hier ist ebenfalls ggT(0,0) = 0):

```
with Ada.Integer_Text_IO; use Ada.Integer.Text_IO;
procedure euklid1 is
  A, B, R: Natural;
begin  Get (A); Get (B);
  if  (A > 0) or (B > 0)  then
     if A < B then  R := A;  A := B;  B := R;  end if;
     while B /= 0 loop
        R := A mod B;  A := B;  B:= R;  end loop;
  end if;
  Put (A);
end;
```

Wir schreiben diesen Algorithmus in eine Funktion von $\mathbb{N}_0 \times \mathbb{N}_0$ nach $\mathbb{N}_0$ um, wobei die Eingabe durch "Parameter" und die Ausgabe durch "return" erfolgt. Für diese Umwandlung ist also nur sehr wenig zu tun.



Die Funktion euklid1 wird im Deklarationsteil definiert. Ab dieser Stelle ist der Name bis zum Ende des zugehörigen Anweisungsteils bekannt ("Sichtbarkeitsbereich"). Hier kann euklid1 in allen ganzzahligen Ausdrücken verwendet werden, wobei die aktuellen Parameterwerte natürliche Zahlen sein müssen, z.B. (K sei vom Typ Natural; I, J, M: Integer):

...

M := (7 + euklid1(K, 720)) * (I + J);

...

Generell kann eine Funktion der Form
function ... return T is ....
wie jeder Operand vom Datentyp T in Ausdrücken verwendet werden.

Der <Parameterteil> ist meist eine
<Liste von formalen Parametern>
Falls es keinen Parameter gibt, entfällt (<Parameterteil>) , anderenfalls werden die "formalen" Parameter mit ihren Datentypen aufgelistet (je Datentyp getrennt durch Semikolon, die einzelnen Parameter für jeden Datentyp getrennt durch Kommata).
Funktionen werden nur in Ausdrücken verwendet. Hierbei spricht man von einem "Funktionsaufruf". Dieser "Funktionsaufruf" hat die Form

<Name der Funktion> (<Liste von aktuellen Parametern>)

wobei es genau so viele aktuelle wie formale Parameter geben und jeder aktuelle Parameter den gleichen Datentyp wie sein zugehöriger formaler Parameter haben muss. Aktuelle Parameter sind meist Ausdrücke, z.B.:
Z := euklid1(I+J*4, (K+I)*(I+J)) + I;

Verwendung in einem Programm:

```
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Testprogramm_ggT is

  function euklid1(X, Y: Natural) return Natural is
    A, B, R: Natural;
  begin A := X; B := Y;
    if (A > 0) or (B > 0) then
      if A < B then R := A; A := B; B := R; end if;
      while B /= 0 loop
        R := A mod B; A := B; B:= R; end loop;
    end if;
    return A;
  end;

  m, n: Natural;
begin Get(m); Get(n);
  Put(euklid1(m,n)); Put(euklid1(m+1,n+1));
end;
```

1.7.2 Prozedurdeklaration

```
procedure <Name der Prozedur> ( <Parameterteil> ) is
   <Deklarationsteil>
begin
   <Anweisungsteil>
end <optional: Name der Prozedur> ;
```

Unterschied zur Funktionsdeklaration:
Eine Funktion muss stets über ein "return <Ausdruck>" verlassen werden. Sie berechnet einen Wert. Sie wird daher in Ausdrücken verwendet.
Eine Prozedur beschreibt einen Algorithmus und liefert daher keinen einzelnen Wert. Der Prozeduraufruf wird wie eine Anweisung verwendet.
Die Prozedur endet damit, dass man auf das letzte "end" trifft. Will man die Prozedur früher verlassen, so kann man dies mit der elementaren Anweisung "return;" erreichen.
Hinweis: Wie bei der Funktion nennt man den Teil von procedure bis is "Kopf der Prozedur", den Rest "Rumpf".

Beispiel: Herausziehen eines Teilalgorithmus als Prozedur

```
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Text_Io; use Text_Io;
procedure Ordne_drei_Zahlen_a is
    A, B, C, H: Integer;
begin  Get(A); Get(B); Get(C);
    if B < A then  H:=A; A:=B; B:=H; end if;    -- nun ist A <= B
    if C < A then  H:=A; A:=C; C:=H;      -- A < C <= B
                   H:=B; B:=C; C:=H;      -- A < B <= C
    else  if B < C then null;             -- A <= B < C
          else                            -- hier ist A <= C <= B
             H:=B; B:=C; C:=H;
          end if;                         -- nun ist A <= B <= C
    end if;                               -- stets: A <= B <= C
    Put(A,7); Put(B,7); Put(C,7);
end;
```


Wir ziehen das Vertauschen als Prozedur heraus:

```
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Text_Io; use Text_Io;
procedure Ordne_drei_Zahlen_b is
  procedure Vertausche (X, Y: in out Integer) is
  H : Integer;
  begin H:=X; X:=Y; Y:=H; end;
  A, B, C: Integer;
begin  Get(A); Get(B); Get(C);
    if B < A then Vertausche (A, B); end if;
    if C < A then Vertausche (A, C); Vertausche (B, C);
    else  if B < C then null;
          else Vertausche (B,C); end if;
    end if;
    Put(A,7); Put(B,7); Put(C,7);
end;
```


1.7.3 Rekursion

Im Inneren einer Funktion ist der Name der Funktion bekannt (man sagt auch "sichtbar"), denn er wurde ja bereits durch function .... eingeführt. Man darf daher dort die Funktion selbst verwenden. Die (direkte oder indirekte) Verwendung einer Funktion in ihrem eigenen Rumpf nennt man Rekursion.

Erstes Beispiel: Die Fakultätsfunktion

$$n! = \begin{cases} 1 & \text{für } n = 0; \\ n \cdot (n-1)! & \text{für } n > 0 \end{cases}$$

```
function Fak(n: Natural) return Natural is
begin
   if n=0 then return 1; else return n*Fak(n-1); end if;
end Fak;
```

Vorteil: Diese rekursive Formulierung übernimmt direkt die Definition und ist daher korrekt.

Zweites Beispiel: ggT

```
function ggT(A, B: Natural) return Natural is
begin
      if  (A=0) and (B=0)  then  Put("Fehler");
      else if  B=0  then  return A;
           else  return ggT(B, A mod B);  end if;
      end if;
end ggT;
```

Beachten Sie: Für a < b gilt a mod b = a, so dass der Aufruf ggT(a,b) zum Aufruf ggT(b,a) führt. Daher kann man die Abfrage, ob A<B ist, weglassen, vgl. Funktion euklid1.

Drittes Beispiel für Rekursion (siehe Definition 1.4.1)*:*

```
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Ada.Float_Text_Io; use Ada.Float_Text_Io;
with Text_Io; use Text_Io;
procedure Harmonische_Funktion_rekursiv is
  function Harmon (I: in Natural) return Float is
  begin
     if I > 0 then return (Harmon(I-1) + 1.0/Float(I));
     else return 0.0;
     end if;
  end;
  N: Natural;
begin Get (N);              -- alle H-Werte von 1 bis N ausdrucken
     for I in 1..N loop
          New_Line; Put(I, 7); Put (Harmon(I),5,9,0);
     end loop;
end;
```


Viertes Beispiel: Wechselseitiger Aufruf zweier Funktionen Gerade und Ungerade

```
function Ungerade(A: Natural) return Boolean;

function Gerade (A: Natural) return Boolean is
begin
     if A = 0 then return True;
     else return Ungerade(A-1); end if;
end Gerade;

function Ungerade (A: Natural) return Boolean is
begin
     if A = 0 then return False;
     else return Gerade(A-1); end if;
end Ungerade;
```

Hier ist der Bezeichner "Ungerade" nicht bekannt; daher kündigt man ihn zuvor durch eine Zeile an. Seine Deklaration erfolgt dann später.

Übliche Bezeichnungen (für Prozeduren genauso)

Funktionsspezifikation: Bezeichnung für die Angabe von Name, Urbild- und Wertebereich einer Funktion (oder einer Prozedur). Man fügt noch die Bezeichner für die formalen Parameter hinzu.
Eine Funktion h: $\mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$ hat also z.B. die Spezifikation

```
function h (X, Y: Natural) return Boolean;
```

Funktionsdeklaration (in Ada wird dies als "function-body" bezeichnet): Spezifikation zusammen mit dem Programmstück, welches die Funktion realisiert. Wurde die Spezifikation zusätzlich vorgezogen, so muss der Anfang (der *Funktions-"Kopf"*) der Funktionsdeklaration identisch mit der Spezifikation sein.

Die Funktionsdeklaration ohne den Kopf (also den Rest nach is) bezeichnet man meist als Rumpf oder Implementierungsteil.

Obiges Beispiel: Spezifikation vorab angeben, damit die Funktion "Gerade" die Funktion "Ungerade" verwenden kann.

Funktionsspezifikation

```
function Ungerade(A: Natural) return Boolean;
```

zugehörige Funktionsdeklaration

```
function Ungerade (A: Natural) return Boolean is
begin
      if A = 0 then return false;
      else return Gerade(A-1); end if;
end Ungerade;
```

Kopf

Rumpf

1.7.4 Operatoren

Operatoren sind spezielle Funktionen. Sie sind in der Regel ein- oder zweistellig und besitzen statt eines Namens ein Operatorsymbol.

In Ada kann man solche Operatorsymbole wie eine Funktion deklarieren. Das Deklarationsschema ist identisch, nur steht anstelle des Funktionsnamens eines der folgenden Operatorsymbole (nur diese sind in Ada erlaubt!)

abs, and, mod, not, or, rem, xor,
<, <=, >, >=, =, /=, +, -, *, /, **, &

zwischen zwei Anführungszeichen.

Beispiel:

```
function "+" (X: Float; K: Integer) return Float is
begin return X + Float(K); end "+";
```

Hierdurch wird ein Operator + definiert, der die Funktionalität $\mathbb{R} \times \mathbb{Z} \to \mathbb{R}$ besitzt.

Bisher war der Operator + nur für zwei ganze oder für zwei reelle Werte definiert; wir haben seine Möglichkeiten mit obiger Deklaration erweitert. Aber woher weiß das Programm eigentlich, welche der vielen Möglichkeiten in einem konkreten Fall gemeint ist?
Wir werden diese Frage später unter dem Stichwort "Overloading" (Überladen) behandeln.

1.7.5 Beispiel: Ausdrucken von ganzen Zahlen

Die Ada-Anweisung Put(A) druckt den Wert einer Integer-Variablen A in der Regel 9-stellig aus. Will man ihn 4-stellig ausdrucken, so muss man Put(A,4) schreiben. Die Stelligkeit kann man berechnen, indem man durch fortgesetztes Dividieren durch 10 die Zahl der Stellen ermittelt; eine negative Zahl braucht eine weitere Stelle für das "-".

```
function Laenge(Z: Integer) return Natural is
   L, K: Integer;
begin
   if Z < 0 then  K := -Z; L := 2;  else  K := Z; L := 1;  end if;
   while K > 9 loop K := K/10; L := L+1; end loop;
   return L;
end;
```

Verwendung: Put(A,Laenge(A)); Der Wert von A wird mit genau der Anzahl ihrer Ziffern ausgedruckt, also ohne zusätzliche Zwischenräume davor oder dahinter.

1.7.6 Beispiel: Stellenwertsysteme (siehe später 2.4.8)

Es sei $b \geq 2$ eine natürliche Zahl. Dann kann man jede natürliche Zahl $z > 0$ eindeutig in der Form

$$z = z_n \cdot b^n + z_{n-1} \cdot b^{n-1} + ... + z_1 \cdot b^1 + z_0 \cdot b^0$$
$$\text{mit } 0 \leq z_i < b \text{ für alle } i = 0, 1, ..., n.$$

darstellen. Man schreibt $z = z_n z_{n-1} ... z_1 z_0$ oder, falls b nicht unmittelbar klar ist, $z = (z_n z_{n-1} ... z_1 z_0)_b$ und nennt dies die Darstellung der Zahl z im *Stellenwertsystem zur Basis b*.

Die Werte 0, 1, ..., b-1 heißen *Ziffern* des Stellenwertsystems. Ist $b \leq 10$, so nimmt man hierfür die Ziffern von 0 bis b-1. Ist $b > 10$, so verwendet man für die Ziffern, die ≥ 10 sind, neue Symbole, und zwar die Großbuchstaben A (für 10), B (für 11), C (für 12) usw. Das Hexadezimalsystem hat die Basis b = 16 und die Ziffern {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. Im Falle von b = 2 spricht man von Binärsystem, bei b = 8 vom Oktalsystem und bei b = 10 vom Dezimalsystem.

Beispiele:
$(3)_{10} = (11)_2$, $(20)_5 = (10)_{10}$, $(33)_6 = (30)_7 = (25)_8 = (21)_{10}$,
$(11001101)_2 = (205)_{10} = (B3)_{16} = (A5)_{20} = (9G)_{21} = (7N)_{26}$.

Oft muss man Zahlen im Stellenwertsystem zu einer anderen Basis darstellen. Diese Darstellung erfolgt mit Hilfe der Division und Modulo-Bildung. Wenn z und eine Basis b gegeben sind, so ist (z mod b) die letzte Ziffer bzgl. der Basis b und man führe dann das Verfahren rekursiv für die Zahl z div b durch.

Beispiel: 1566 soll zur Basis 16 dargestellt werden.
1566 mod 16 = 14 (= Ziffer E), 1566 div 16 = 97.
97 mod 16 = 1 (= Ziffer 1), 97 div 16 = 6.
6 mod 16 = 6 (= Ziffer 6), 6 div 16 = 0. Fertig.
Somit ist 61E die Hexadezimaldarstellung von 1566.

Wir programmieren diesen Algorithmus nicht, sondern überlassen das Ausdrucken der Ziffern (in umgekehrter Reihenfolge) zu einer Zahl z und die Formulierung als Prozedur den Leser(inne)n.

1.7.7 Beispiel: ggT (erweiterter Euklidischer Algorithmus)

Der Euklidische Algorithmus beruhend auf
$ggT(a, 0) = 0$ und $ggT(a, b) = ggT(b, a \bmod b)$
wurde bereits mehrfach vorgestellt.

In der Praxis braucht man oft die Darstellung des ggT(a, b) als Linearkombination der Zahlen a und b. Vor ca. 250 Jahren bewies der französische Mathematiker Étienne Bézout:

Lemma von Bézout:
$\forall a, b \in \mathbb{N}\ \exists x, y \in \mathbb{Z}: ggT(a, b) = x \cdot a + y \cdot b.$

Das heißt: Der ggT zweier Zahlen lässt sich stets als deren Linearkombination (mit ganzen Zahlen als Faktoren) schreiben.

Der Beweis ist recht einfach, wenn man den Euklidischen Algorithmus genau nachvollzieht.

ggT-Berechnung als Folge von Zahlen $a_0, a_1, a_2, a_3, ...$:
Gegeben seien zwei natürliche Zahlen a und b mit $a > b > 0$.
(Der Fall a = b ist trivial und wird daher nicht betrachtet.)

> Initialisiere $a_0 := a$; $a_1 := b$;
> Berechne $a_{i+2} := a_i \bmod a_{i+1}$ solange, bis $a_{i+2} = 0$ ist.
> Dann ist a_{i+1} der $ggT(a, b) = ggT(a_0, a_1)$.
> Beachte: $a_0 > a_1 > a_2 > a_3 > ... > a_{i+2} = 0$. Folglich endet dieses Verfahren tatsächlich.

Beispiel: a = 448, b = 175. Bilde die Folge der a_i:

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
448	175	98	77	21	14	7	0

Also gilt ggT(448,175) = 7.

Schreibweise:
$\lfloor z \rfloor$ ist der ganzzahlige Anteil der Zahl z in der mathematisch üblichen Schreibweise ("untere Gauß-Klammern"). Speziell gilt $\lfloor a/b \rfloor = a \text{ div } b$ sowie $a \bmod b = a - \lfloor a/b \rfloor \cdot b$.

Jedes a_i ist offenbar eine Linearkombination von a und b:

$a_0 = 1 \cdot a + 0 \cdot b = x_0 \cdot a + y_0 \cdot b$ (mit $x_0 = 1$ und $y_0 = 0$)

$a_1 = 0 \cdot a + 1 \cdot b = x_1 \cdot a + y_1 \cdot b$ (mit $x_1 = 0$ und $y_1 = 1$)

$a_2 = a_0 \bmod a_1 = a \bmod b = a - \lfloor a/b \rfloor \cdot b = 1 \cdot a - \lfloor a/b \rfloor \cdot b$
$= x_2 \cdot a + y_2 \cdot b$ (mit $x_2 = 1$ und $y_2 = -\lfloor a/b \rfloor$)

$a_3 = a_1 \bmod a_2 = b - \lfloor b/a_2 \rfloor \cdot a_2 = 1 \cdot b - \lfloor b/a_2 \rfloor \cdot (1 \cdot a - \lfloor a/b \rfloor \cdot b)$
$= - \lfloor b/a_2 \rfloor \cdot a + (1 + \lfloor b/a_2 \rfloor \cdot \lfloor a/b \rfloor) \cdot b$
$= x_3 \cdot a + y_3 \cdot b$ (mit $x_3 = -\lfloor b/a_2 \rfloor$ und $y_3 = 1 + \lfloor b/a_2 \rfloor \cdot \lfloor a/b \rfloor$)

$a_4 = a_2 \bmod a_3 = a_2 - \lfloor a_2/a_3 \rfloor \cdot a_3 = ... = x_4 \cdot a + y_4 \cdot b$ usw.

Dies gilt auch für a_{i+1}, d. h., der ggT(a,b) ist als ganzzahlige Linearkombination der Zahlen a und b darstellbar. Unser Ziel ist es nun, diese Darstellung anzugeben, also gleichzeitig mit a_i auch x_i und y_i zu berechnen.

Wir berechnen also die drei Folgen

$a_0, a_1, a_2, a_3, ...$
$x_0, x_1, x_2, x_3, ...$
$y_0, y_1, y_2, y_3, ...$

mit der Nebenbedingung $a_i = x_i \cdot a + y_i \cdot b$ für alle i.

Initialisierung:

$a_0 := a;\quad a_1 := b;\quad x_0 := 1;\quad x_1 := 0;\quad y_0 := 0;\quad y_1 := 1;$

Für alle $i \geq 0$: $a_{i+2} := a_i \bmod a_{i+1} = a_i - \lfloor a_i/a_{i+1} \rfloor \cdot a_{i+1}$

Aus

$a_{i+2} = a_i \bmod a_{i+1} = a_i - \lfloor a_i/a_{i+1} \rfloor \cdot a_{i+1}$
$= (x_i \cdot a + y_i \cdot b) - \lfloor a_i/a_{i+1} \rfloor \cdot (x_{i+1} \cdot a + y_{i+1} \cdot b)$
$= (x_i - \lfloor a_i/a_{i+1} \rfloor \cdot x_{i+1}) \cdot a + (y_i - \lfloor a_i/a_{i+1} \rfloor \cdot y_{i+1}) \cdot b$

folgt

$x_{i+2} := x_i - \lfloor a_i/a_{i+1} \rfloor \cdot x_{i+1}$ und $y_{i+2} := y_i - \lfloor a_i/a_{i+1} \rfloor \cdot y_{i+1}$

Man muss also jedes Mal nur $\lfloor a_i/a_{i+1} \rfloor$ berechnen und kann dann die Folgen der a, x und y *auf die gleiche Art* behandeln.

Schema (am Beispiel a = 448 und b = 175)

i	a_i	$\lfloor a_{i-2}/a_{i-1} \rfloor$	x_i	y_i	$x_i \cdot a + y_i \cdot b$
0	448	-	1	0	448
1	175	-	0	1	175
2	98	2	1	-2	98
3	77	1	-1	3	77
4	21	1	2	-5	21
5	14	3	-7	18	14
6	7	1	9	-23	7
7	0	2	-25	64	0

Hier erhalten wir: ggT(448,175) = 7 = 9 · 448 - 23 · 175.

Nun zur Ada-Prozedur.

Wir haben bereits gesehen, dass man nicht die ganze a-Folge $a_0, a_1, a_2, a_3, ...$ speichern muss, sondern dass man mit zwei Variablen A und B (jetzt mit A0 und A1 bezeichnet) und einer Hilfsvariablen R auskommt. Wir benötigen hier aber eine weitere Variable Q für $\lfloor a_i/a_{i+1} \rfloor = \lfloor A0/A1 \rfloor$.

```
Get(A0); Get(A1);        -- Dies wird später zur Paramterübergabe
while A1 /= 0 loop
    Q := A0 / A1;        -- ganzzahlige Division
    R := A0 - Q*A1;      -- dies ist gleich dem früheren  R := A mod B
    A0 := A1;  A1 := R;
end loop;
```

Wir müssen nun aber noch die Variablen für die x- und die y-Folge hinzufügen. Wir nennen sie X0, X1, Y0, Y1. Wir unterscheiden noch die Fälle A < B und A = B und erhalten die folgende Prozedur.

```
procedure ggT_erweitert (A, B: in Natural; G, X, Y: in out Integer) is
   R, Q, A0, A1, X0, X1, Y0, Y1: Integer;
begin
   if  A = B  then G := B; X := 1; Y := -1; return; end if;
   if  A < B  then A0 := B; A1 := B; else A0 := A; A1 := B; end if;
   X0 := 1; X1 := 0; Y0 := 0; Y1 := 1;
   while (A1 /= 0) loop
      Q := A0 / A1;
      R := A0 - Q*A1;  A0 := A1;  A1 := R;
      R := X0 - Q*X1;  X0 := X1;  X1 := R;
      R := Y0 - Q*Y1;  Y0 := Y1;  Y1 := R;
   end loop;
   G := A0;
   if  A < B  then X := Y0; Y := X0; else X := X0; Y := Y0; end if;
end;
```

Wir fügen noch eine Ausgabe einschließlich der Funktion "Laenge" aus 1.7.5 hinzu und erhalten:

Kompakte Darstellung des Programms

```
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Text_Io; use Text_Io;
procedure ggT_neu is
   A, B: Natural; GAB, XA, YB: Integer;
   function Laenge (Z: Integer) return Natural is    -- siehe 1.7.5
      L, K: Integer;
   begin  if Z < 0 then  K := -Z; L := 2;  else  K := Z; L := 1;  end if;
      while K > 9 loop K := K/10; L := L+1; end loop; return L;
   end;
   procedure ggT_erweitert (A, B: in Natural; G, X, Y: in out Integer) is
      R, Q, A0, A1, X0, X1, Y0, Y1: Integer;
   begin
      if  A = B  then G := B; X := 1; Y := 0; return; end if;
      if  A < B  then A0 := B; A1 := A; else A0 := A; A1 := B; end if;
      X0 := 1; X1 := 0; Y0 := 0; Y1 := 1;
      while (A1 /= 0) loop  Q := A0 / A1;
         R := A0 - Q*A1;  A0 := A1;  A1 := R;
         R := X0 - Q*X1;  X0 := X1;  X1 := R;
         R := Y0 - Q*Y1;  Y0 := Y1;  Y1 := R;
      end loop;
      G := A0;  if  A < B  then X := Y0; Y := X0; else X := X0; Y := Y0; end if;
   end;
begin  Get(A); Get(B); ggT_erweitert(A,B,GAB,XA,YB);    -- Es folgt die Ausgabe
   Put("Der ggT von "); Put(A,Laenge(A)); Put(" und "); Put(B,Laenge(B)); Put(" ist ");
   Put(GAB,Laenge(GAB)); Put("."); New_Line; Put("Es gilt: ");
   Put(GAB,Laenge(GAB)); Put(" = "); Put(XA,Laenge(XA)); Put(" * ");
   Put(A,Laenge(A)); if YB < 0 then Put(" - "); YB := abs(YB); else Put(" + "); end if;
   Put(YB,Laenge(YB)); Put(" * "); Put(B,Laenge(B)); Put("."); New_Line;
end;
```

Ohne Dokumentation ist dieses kurze Programm schwer durchschaubar!

Hinweis: Diese Seite wird in der Vorlesung nicht behandelt.

Hinweis: Die Folge der $\lfloor a_{i-1}/a_{i-2} \rfloor$ definiert die rationale Zahl a/b eindeutig. Denn aus
$a_2 = a_0 \bmod a_1 = a_0 - \lfloor a_0/a_1 \rfloor \cdot a_1$
erhalten wir mit der Abkürzung $h_i = \lfloor a_i/a_{i+1} \rfloor$:

$$\frac{a}{b} = \frac{a_0}{a_1} = h_0 + \frac{a_2}{a_1} = h_0 + \frac{1}{\frac{a_1}{a_2}} = h_0 + \frac{1}{h_1 + \frac{1}{\frac{a_2}{a_3}}}$$

$$= h_0 + \frac{1}{h_1 + \frac{1}{h_2 + \frac{1}{\frac{a_3}{a_4}}}} = \ldots$$

Dieser **Kettenbruch** bricht ab, sobald $a_{i+2} = 0$ ist.

Hinweis: Diese Seite wird in der Vorlesung nicht behandelt.

Für obiges Beispiel gilt also:

$$\frac{448}{175} = 2 + \frac{1}{1 + \frac{1}{1 + \frac{1}{3 + \frac{1}{1 + \frac{1}{2}}}}}$$

Man schreibt hierfür: (2; 1, 1, 3, 1, 2).

Rechnet man diesen iterierten Bruch aus, so erhält man 64/25, also die gekürzte Darstellung von 448/175.

Jede rationale Zahl lässt sich als endlicher Kettenbruch schreiben. Nimmt man alle unendlichen Kettenbrüche hinzu, so erhält man die reellen Zahlen.

Hinweis: Diese Seite wird in der Vorlesung nicht behandelt.

Anwendungsbeispiel: RSA = Asymmetrisches Verschlüsselungsverfahren.
Vorgehensweise:
1. Finde zwei große Primzahlen p und q (z. B. 200-stellige Zahlen).
2. Bilde n = p · q und h = (p-1) · (q-1).
3. Wähle eine Zahl e mit der Eigenschaft ggT(e,h) = 1; empfehlenswert: e sollte mindestens 7-stellig sein.
4. Suche eine Zahl d mit d · e ≡ 1 mod h; empfehlenswert: d sollte mindestens ein Viertel der Stellenzahl von h haben. (Evtl. e neu festlegen.)

Veröffentliche die Zahlen n und e. Jeder kann dann eine Nachricht N (als Zahl codiert, 0 < N < n) verschlüsselt an Sie schicken, indem er $C = N^e \bmod n$ berechnet und abschickt; diese Modulo-Rechnung erfolgt relativ schnell mit einem Rechner. Es gilt dann: $N = C^d \bmod n$, d.h., nur wer die Zahl d kennt, kann die Nachricht entschlüsseln. d aus e und n zu berechnen, erfordert nach heutiger Kenntnis aber Jahrmillionen.

In diesem Verfahren muss d · e ≡ 1 mod h gelöst werden; hierzu muss eine Zahl j existieren mit d · e = 1 + j · h. Dies bedeutet: e und h müssen den ggT(e,h) = 1 besitzen und es muss eine Darstellung ggT(e,h) = d · e - j · h gefunden werden. Mit dem erweiterten euklidischen Algorithmus lässt sich dies leicht realisieren. Prüfen Sie dies für kleine Zahlen p und q einmal nach!

Hinweis: Diese Seite wird in der Vorlesung nicht behandelt.

Nur für Interessierte ein Beispiel zum Ausfüllen: Wähle p = 1009, q = 1007.

Berechne n = p · q = 1016063 und h = (p-1) · (q-1) = 1008 · 1006 = 10140448.

Wähle willkürlich e = 151 (dies ist eine Primzahl).

Berechne nun die Zahl d mit obigem erweiterten Eulidischen Algorithmus: ggT(e,h) = 1 = x * 10140448 + d * 151. Es ist d = _____ .

Wähle nun eine Nachricht, z. B. "Du". "D" hat die interne Nummer 68 (binär: 01000100) und "u" die Nummer 117 (binär: 01110101), siehe 1.8.4. Die Nachricht N sei die Konkatenation 0100010001110101, also ist N die Zahl $68 \cdot 2^8 + 117 = 17525$.

Jetzt kommt man ohne ein Programm nicht mehr weiter. Denn man muss $C = 17525^{151} \bmod 1016063$ berechnen. (Schreiben Sie hierfür ein Programm ☺, Vorsicht wegen Überlauf). Es ist C = _____.

Berechne anschließend mit dem gleichen Programm $C^d \bmod 1016063$.

Hinweise: 200-stellige Primzahlen kann man mit heutigen Rechnern in Sekundenschnelle finden. Ein exakter Algorithmus ist recht kompliziert; ein randomisierter Algorithmus ist leichter verständlich, kann aber (ganz selten) einen Fehler liefern. Die Potenzierung ist "einfach", weil man nach jeder Multiplikation sofort " mod n " durchführen darf, ohne dass das Ergebnis falsch wird. - Warum das gesamte Verfahren korrekt ist, wird in der Vorlesung "Diskrete Mathematik" oder in Vorlesungen zur Kryptographie erklärt.

1.8 Skalare Datentypen

Definition: ***Datentyp*** *(= Mengen und erlaubte Operationen)*

Eine Menge (oder mehrere Mengen, "Wertebereich" genannt) zusammen mit den hierauf definierten Operationen nennt man einen (konkreten) Datentyp.

Einen Datentyp, den man nicht auf andere Datentypen zurückführt, nennt man in Ada einen skalaren Datentyp. Alle anderen heißen zusammengesetzte Datentypen.

Skalare Datentypen in Ada sind die vordefinierten Standard-datentypen und die selbst definierten Aufzählungstypen.

Neue Datentypen werden durch das Schlüsselwort type eingeführt. Will man nur den Wertebereich eines Datentyps einschränken, so verwendet man das Schlüsselwort subtype.

Anschaulich: Ein Datentyp besteht aus
- einem *Wertebereich*,
- zusätzlichen Wertebereichen (oftmals Boolean) und
- den zugehörigen *Operationen*.

Schema:

NameT
Wertebereich: M = ...
Weitere Wertebereiche
Operationen: 0-stellig: ... 1-stellig: ... 2-stellig: ... ...

Beispiel:

Integer
$\{-2^{31},...,-1,0,1,2,...,2^{31}-1\} \subset \mathbf{Z}$
B, **R**
Operationen: 0-stellig: alle Konstanten, Integer'Last, ... 1-stellig: +, abs, -, Wurzel, round, ... 2-stellig: +, -, *, /, mod, Min, Max, <, <=, =, ...

1.8.1 Aufzählungstypen

Ein neuer skalarer Datentyp wird im Deklarationsteil mit Hilfe des Schlüsselwortes "type" eingeführt in der Form:

type <Name des Datentyps> is (<Liste der Elemente>);

Die Reihenfolge in der Liste bildet zugleich eine Anordnung der Elemente.

Beispiele:

type Wochentage is (Mo, Di, Mi, Do, Fr, Sa, So);

type Freie_Tage is (Sa, So);

type Farbe is (weiß, gelb, grün, rot, blau, schwarz);

type Erste_zehn_Primzahlen is (2,3,5,7,11,13,17,19,23,29);

type Nur_die_Null is (0);

Ein Datentyp besteht aus *Wertebereichen* und den zugehörigen *Operationen*. Für Aufzählungstypen sind dies:

Wertebereich (= zugrunde liegende Menge): die endliche Menge $M = \{m_1, m_2, ..., m_n\}$, die man selbst definiert hat. Die Menge M wird wie oben angegeben eingeführt, also:

type <Name des Datentyps> is (m_1, m_2, ..., m_n);

Ein Aufzählungstyp ist in Ada stets angeordnet! Die Reihen-folge in der Liste gibt die *Anordnung* der Elemente an (auch wenn die Menge $\{m_1, m_2, ..., m_n\}$ gar nicht angeordnet war).

Es kommt hierbei per Definition nicht zu Namenskonflikten! Das heißt: Wenn ein Element in mehreren Mengen liegt, so muss man selbst darauf achten, dass bei jeder Verwendung dieses Elementes eindeutig klar ist, zu welchem Datentyp es gehört, s. u.

Mit jedem Aufzählungstyp sind automatisch folgende Operationen (in Ada auch "Attribute" genannt) verbunden:
First, Last, Pred, Succ, Pos, Val, <, <=, >, >=, =, /=, Min, Max.
Attributen wird in Ada ein Apostroph vorangestellt.

Nullstellige Operationen (= besondere Konstanten) sind alle Elemente m_i der Menge M. Weiterhin sind First das erste und Last das letzte Element der Menge.
Man muss hierbei den Datentyp angeben, d.h., wenn T der Name des Datentyps ist, so schreibt man T'First bzw. T'Last.
Ist ein Element m in mehreren Datentypen, so gibt man das Element, das man verwenden will, durch <Datentyp>'(m) an.
Beispiele:
Wochentage'Last = So. Freie_Tage'First = Sa.
Freie_Tage'(Sa) ist verschieden von Wochentage'(Sa).

Einstellige Operationen: (Vorgänger, Nachfolger, Position, Wert)

"predecessor" (=Vorgänger) und "successor" (=Nachfolger)
Pred, Succ: $M \to M$ mit
Pred(X) = das Zeichen vor X in der Auflistungs-Reihenfolge,
Succ(X) = das Zeichen nach X in der Auflistungs-Reihenfolge.
$Pred(m_i) = m_{i-1}$ für $i>1$ und $Pred(m_1)$ = undefiniert,
$Succ(m_i) = m_{i+1}$ für $i<n$ und $Succ(m_n)$ = undefiniert.
Position in der Anordnung der Zeichen: Pos: $M \to \mathbb{N}_0$
Pos(X) = n bedeutet: X ist das n-te Zeichen in der Auflistungsreihenfolge
Beachte! (*beginnend mit 0*, d.h., das erste Zeichen hat die Position 0, das zweite die Position 1 usw.).
n-tes Element in der Anordnung: Val: $\mathbb{N}_0 \to M$
Val(n) = X bedeutet: X ist das n-te Zeichen in der Auflistungsreihenfolge *(beginnend mit 0)*. **Beachte!**

Beispiele:

type Stuhlteil is (bein, sitzfläche, lehne);
type Körperteil is (arm, bein, rücken, sitzfläche, kopf);

Hiermit kann man folgende Ausdrücke bilden:

Körperteil'(bein) *(dies ist* bein *des Datentyps Körperteil),*

Stuhlteil'(bein) *(dies ist* bein *des Datentyps Stuhlteil),*

Körperteil'Pos(bein) *(dies ist* 1*),*

Stuhlteil'Pos(bein) *(dies ist* 0*),*

Körperteil'Val(Stuhlbein'Pos(lehne)) *(dies ist* rücken*),*

Stuhlteil'(Pred(bein)) *(undefiniert, dies führt zu einem Fehler),*

Körperteil'(Pred(bein)) *(dies ist* arm*).*

Zweistellige Operationen (Vergleiche und Min, Max):
Alle sechs Vergleichsoperationen bezüglich der Anordnung der Menge "=", "/=", "<", "<=", ">" und ">=" sind zugelassen. Das Ergebnis ist vom Typ Boolean.

Für alle Aufzählungstypen T sind die zweistelligen Abbildungen Min und Max definiert: Min, Max: $M \times M \to M$ mit

Min(X,Y) = X $\Leftrightarrow$ T'Pos(X) $\leq$ T'Pos(Y) ($\Leftrightarrow$ T'(X) $\leq$ T'(Y)),
anderenfalls ist Min(X,Y) = Y.

Max(X,Y) = X $\Leftrightarrow$ T'Pos(X) > T'Pos(Y),
anderenfalls ist Max(X,Y) = Y.

Im Falle $X \neq Y$ gilt stets T'Min(X,Y) /= T'Max(X,Y).

Einige Beziehungen, Gesetzmäßigkeiten:

Es gilt zum Beispiel stets:

Pos(Val(k)) = k für $0 \leq k \leq n-1$,

Val(Pos(X)) = X für alle Elemente $X \in M$,

Succ(Pred(m_i)) = m_i für $1 < i \leq n$,

Pred(Succ(m_j)) = m_j für $1 \leq j < n$,

Min(Min(X,Y), Max(X,Y)) = Min(X, Y),

Min(Max(X,Y), Max(X,Y)) = Max(X, Y),

X = Y im Datentyp T $\Leftrightarrow$ Pos(X) = Pos(Y) in $\mathbb{N}_0$,

X < Y im Datentyp T $\Leftrightarrow$ Pos(X) < Pos(Y) in $\mathbb{N}_0$.

1.8.2 Standard-Datentypen (Überblick, vgl. 1.4):

Boolean	**IB** = {false, true}
Character	**A** = Latin1-Alphabet aus 256 Zeichen
Integer	**Z** = Menge der ganzen Zahlen
Float	**IR** = Menge der reellen Zahlen

Unterbereiche (subtypes) von Integer:

Natural	$\mathbb{N}_0$ (Natürliche Zahlen mit der Null)
Positive	$\mathbb{N}$ (Natürliche Zahlen ohne die Null)

Wichtig: Mit jedem Datentyp sind zugleich die erlaubten Operationen definiert! Diese listen wir für die vier Standard-datentypen nun auf. Lesen Sie dies bitte genau durch, da wir die folgenden Folien in der Vorlesung nur knapp behandeln; Sie müssen die Operationen und Wertebereiche gut kennen.

1.8.3 Der Datentyp Boolean in Ada

1.8.3 bis 1.8.6: bitte alle Details selbst erarbeiten!

Wertebereich: **IB** = {false, true} (= {falsch, wahr})
Boolean wird in Ada zugleich als Aufzählungstyp aufgefasst.

Nullstellige Operationen (=besondere Konstanten): False und True.

Einstellige Operationen (NICHT):
Negation ¬: **IB** → **IB** (statt ¬ schreibt man in Ada not).

Zweistellige Operationen (UND, ODER, EXKLUSIVES ODER):
Konjunktion ∧: **IB**×**IB** → **IB** (statt ∧ schreibt man and)
Disjunktion ∨: **IB**×**IB** → **IB** (statt ∨ schreibt man or)
Ungleichheit (oder auch "Exklusives Oder" genannt)
/= : **IB**×**IB** → **IB** (statt a /= b schreibt man a xor b)
Man kann auch die Gleichheit sowie andere Vergleiche auf **IB** verwenden (wie auf allen Aufzählungstypen): = : **IB**×**IB** → **IB**

In Ada gibt es neben and und or die beiden „Kaskadenoperationen“ AND THEN und OR ELSE, die man aber "sehr bewusst" einsetzen sollte, da man sich hierdurch leicht Fehler einfangen kann:

a and then b entspricht:
falls a nicht zutrifft, ist das Ergebnis false, anderenfalls b.

a or else b entspricht:
falls a zutrifft, ist das Ergebnis true, anderenfalls b.

Im Normalfall, dass a und b einen der Werte false oder true besitzen, stimmen and und and then bzw. or und or else überein. Den Unterschied zu and und or erkennt man nur, wenn man die Funktionstabellen für diese Operationen zusätzlich mit dem Wert "undefiniert" aufschreibt:

AND	false	true	undef
false	false	false	undef
true	false	true	undef
undef	undef	undef	undef

AND THEN	false	true	undef
false	false	false	**false**
true	false	true	undef
undef	undef	undef	undef

OR	false	true	undef
false	false	true	undef
true	true	true	undef
undef	undef	undef	undef

OR ELSE	false	true	undef
false	false	true	undef
true	true	true	**true**
undef	undef	undef	undef

Prioritäten in Ada für Boolean (um Klammerungen zu sparen):

In Ada werden nur zwei Prioritätsstufen für Boolesche Operationen festgelegt:

Der Operator not erhält eine höhere Priorität, die anderen Operatoren and, or und xor erhalten eine gleiche, aber niedrigere Priorität. Um Fehler zu vermeiden, gilt:

In Ada müssen alle Booleschen Ausdrücke, in denen mindestens zwei der drei Operatoren and, or und xor vorkommen, geklammert werden!

a and b or c ist also verboten; man muss (a and b) or c oder a and (b or c) schreiben.

Erlaubt ist dagegen a or b or c ; dies wird von links nach rechts ausgewertet.

In Booleschen Ausdrücken können auch Vergleiche auftreten.

In Ada erhalten *alle* Vergleichsoperatoren eine höhere Priorität als die Booleschen Operationen außer not. Statt

(17 < 10) and (not (X=Y))

kann man also auch schreiben:

17 < 10 and not (X=Y).

Beachte: Bzgl. der Prioritätsregeln unterscheidet sich Ada von den meisten anderen Programmiersprachen.

1.8.4 Der Datentyp Character in Ada

Der Typ Character wird als Aufzählungstyp mit dem Alphabet "Latin-1" als Wertebereich aufgefasst:
type Character is (*NUL*, ..., *US*, ' ', '!', '"', '#', '$', ..., 'ÿ');
Die einzelnen Zeichen werden in Apostroph eingeschlossen, sofern sie graphisch darstellbar sind (das Apostroph selbst wird hierbei durch zwei Apostrophe dargestellt).
Es stehen somit für Character alle allgemeinen Operationen der Aufzählungstypen zur Verfügung (1.8.1).

Zur Kenntnisnahme geben wir den vollständigen Latin-1 Code, definiert in der ISO-Norm 8859-1, auf den nächsten Folien an.
In der Tabelle wird die hexadezimale und die (mit dem Nummerzeichen '#' versehene) dezimale Nummerierung vorangestellt und dann das zugehörige Zeichen genannt.

ISO 8859, Latin-1 Alphabet (festgelegt sind nur die schwarzen Werte; die kursiven blauen bilden eine übliche Ergänzung)

00 #0 NULL	*01 #1 START OF HEADING*	*02 #2 START OF TEXT*	*03 #3 END OF TEXT*	*04 #4 END OF TRANSM.*	*05 #5 ENQUIRY*	*06 #6 ACKN.*	*07 #7 BELL*	*08 #8 BACK-SPACE*	09 #9 HORIZ. TAB.	0A #10 LINE FEED	*0B #11 VERTICAL TAB.*	*0C #12 FORM FEED*	0D #13 CARRIAG RETURN	*0E #14 SHIFT OUT*	*0F #15 SHIFT IN*
10 #16 DATA LINK ESC.	*11 #17 DEVICE CONTR.1*	*12 #18 DEVICE CONTR.2*	*13 #19 DEVICE CONTR.3*	*14 #20 DEVICE CONTR.4*	*15 #21 NEGATIV ACKN.*	*16 #22 SYNCHR. IDLE*	*17 #23 END OF T.BLOCK*	*18 #24 CANCEL*	*19 #25 END OF MEDIUM*	*1A #26 SUBSTI-TUTE*	*1B #27 ESCAPE*	*1C #28 FILE SE-PARATOR*	*1D #29 GROUP SEPAR.*	*1E #30 RECORD SEPAR.*	*1F #31 UNIT SEPAR.*
20 #32 SPACE	21 #33 !	22 #34 "	23 #35 #	24 #36 $	25 #37 %	26 #38 &	27 #39 '	28 #40 (	29 #41)	2A # 42 *	2B #43 +	2C #44 ,	2D #45 -	2E #46 .	2F #47 /
30 #48 0	31 #49 1	32 #50 2	33 #51 3	34 #52 4	35 #53 5	36 #54 6	37 #55 7	38 #56 8	39 #57 9	3A #58 :	3B #59 ;	3C #60 <	3D #61 =	3E #62 >	3F #63 ?
40 #64 @	41 #65 A	42 #66 B	43 #67 C	44 #68 D	45 #69 E	46 #70 F	47 #71 G	48 #72 H	49 #73 I	4A #74 J	4B #75 K	4C #76 L	4D #77 M	4E #78 N	4F #79 O
50 #80 P	51 #81 Q	52 #82 R	53 #83 S	54 #84 T	55 #85 U	56 #86 V	57 #87 W	58 #88 X	59 #89 Y	5A #90 Z	5B #91 [	5C #92 \	5D #93]	5E #94 ^	5F #95 _
60 #96 `	61 #97 a	62 #98 b	63 #99 c	64 #100 d	65 #101 e	66 #102 f	67 #103 g	68 #104 h	69 #105 i	6A #106 j	6B #107 k	6C #108 l	6D #109 m	6E #110 n	6F #111 o
70 #112 p	71 #113 q	72 #114 r	73 #115 s	74 #116 t	75 #117 u	76 #118 v	77 #119 w	78 #120 x	79 #121 y	7A #122 z	7B #123 {	7C #124 |	7D #125 }	7E #126 ~	*7F #127 DELETE*
80 #128 €	*81 #129*	*82 #130 ‚*	*83 #131 ƒ*	*84 #132 „*	*85 #133 …*	*86 #134 †*	*87 #135 ‡*	*88 #136 ˆ*	*89 #137 ‰*	*8A #138 Š*	*8B #139 ‹*	*8C #140 Œ*	*8D #141*	*8E #142 Ž*	*8F #143*
90 #144	*91 #145 ‘*	*92 #146 ’*	*93 #147 “*	*94 #148 ”*	*95 #149 •*	*96 #150 –*	*97 #151 —*	*98 #152 ˜*	*99 #153 ™*	*9A #154 š*	*9B #155 ›*	*9C #156 œ*	*9D #157*	*9E #158 ž*	*9F #159 Ÿ*
A0 #160 NO-BREAK SPACE	A1 #161 ¡	A2 #162 ¢ CENT	A3 #163 £ POUND	A4 #164 ¤	A5 #165 ¥ YEN	A6 #166 ¦	A7 #167 §	A8 #168 ¨	A9 #169 ©	AA #170 ª	AB #171 «	AC #172 ¬	AD #173 - SOFT HYPHEN	AE #174 ® REGIS-TERED	AF #175 ¯ MACRON
B0 #176 °	B1 #177 ±	B2 #178 ²	B3 #179 ³	B4 #180 ´ ACCENT	B5 #181 µ MICRO	B6 #182 ¶	B7 #183 · MID.DOT	B8 #184 ¸ CEDILLA	B9 #185 ¹	BA #186 º	BB #187 »	BC #188 ¼	BD #189 ½	BE #190 ¾	BF #191 ¿
C0 #192 À	C1 #193 Á	C2 #194 Â	C3 #195 Ã	C4 #196 Ä	C5 #197 Å	C6 #198 Æ	C7 #199 Ç	C8 #200 È	C9 #201 É	CA #202 Ê	CB #203 Ë	CC #204 Ì	CD #205 Í	CE #206 Î	CF #207 Ï
D0 #208 Ð	D1 #209 Ñ	D2 #210 Ò	D3 #211 Ó	D4 #212 Ô	D5 #213 Õ	D6 #214 Ö	D7 #215 ×	D8 #216 Ø	D9 #217 Ù	DA #218 Ú	DB #219 Û	DC #220 Ü	DD #221 Ý	DE #222 Þ	DF #223 ß
E0 #224 à	E1 #225 á	E2 #226 â	E3 #227 ã	E4 #228 ä	E5 #229 å	E6 #230 æ	E7 #231 ç	E8 #232 è	E9 #233 é	EA #234 ê	EB #235 ë	EC #236 ì	ED #237 í	EE #238 î	EF #239 ï
F0 #240 ð	F1 #241 ñ	F2 #242 ò	F3 #243 ó	F4 #244 ô	F5 #245 õ	F6 #246 ö	F7 #247 ÷	F8 #248 ø	F9 #249 ù	FA #250 ú	FB #251 û	FC #252 ü	FD #253 ý	FE #254 þ	FF #255 ÿ

Latin-1 enthält ASCII als die ersten 128 Zeichen. Die Zeichen mit den Nummern 128 bis 159 wurden in Latin-1 frei gelassen; sie werden von verschiedenen Softwareherstellern unterschiedlich verwendet. Oben sind die Zeichen von Microsoft Windows eingetragen (die Nummern 128, 129, 141-144, 157, 158 werden dabei nicht festgelegt; 128 wird meist für das Eurozeichen, 142 und 158 für das modifizierte Z benutzt), in Ada kann man auf diese Zeichen mit symbolischen Bezeichnungen zugreifen, siehe nächste Folie.

Die Werte mit den Nummern 0 bis 31 und 127 bis 159 können zwar verwendet, aber oft nicht angezeigt werden. Sie werden durch zwei- oder dreibuchstabige Kürzel bezeichnet und zwar:

Nr.	Kürzel	Nr.	Kürzel	Nr.	Kürzel	Nr.	Kürzel
0	NUL	16	DLE	127	DEL	144	DCS
1	SOH	17	DC1	128	128	145	PU1
2	STX	18	DC2	129	129	146	PU2
3	ETX	19	DC3	130	BPH	147	STS
4	EOT	20	DC4	131	NBH	148	CCH
5	ENQ	21	NAK	132	132	149	MW
6	ACK	22	SYN	133	NEL	150	SPA
7	BEL	23	ETB	134	SSA	151	EPA
8	BS	24	CAN	135	ESA	152	SOS
9	HT	25	EM	136	HTS	153	153
10	LF	26	SUB	137	HTJ	154	SCI
11	VT	27	ESC	138	VTS	155	CSI
12	FF	28	FS	139	PLD	156	ST
13	CR	29	GS	140	PLU	157	OSC
14	SO	30	RS	141	RI	158	PM
15	SI	31	US	142	SS2	159	APC
				143	SS3		

First, Last, die Funktionen Pred, Succ, Pos, Val, Min, Max und die Vergleichsoperationen sind auf Character wie für alle Aufzählungstypen definiert. Z.B. gilt Character'First = NUL, Character'Val(51) = '3', Character'Pos(Pred(Pred(DEL))) = 125, 'C' < '©', aber: Character'Succ(Character'Last) ist undefiniert.

Hinweis: Es gibt viele verschiedene Zeichensätze. Daher gibt es auch in Ada mehrere vordefinierte Character-Datentypen. Diese werden angesprochen durch

Ada.Characters.<Name des Zeichensatzes>

Beispiel: Der oben beschriebene übliche Zeichensatz ist

Ada.Characters.Latin_1

Man kann über

Ada.Characters.ASCII
Ada.Characters.Wide_Character

andere Zeichensätze nutzen. Details siehe spezielle Literatur.

1.8.5 Der Datentyp Integer in Ada

Theoretisch lautet die zugrunde liegende Wertemenge:
Z = { ..., -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, ... } mit üblicher Ordnung.

In Ada ist auch der Datentyp Integer ein Aufzählungstyp, der von der kleinsten bis zur größten darstellbaren Zahl reicht:

type Integer is (Integer'First, Integer'First+1, ..., Integer'Last);

Nullstellige Operationen: Alle Zahlen.

Hinweis: Im Prinzip genügen die beiden Konstanten 0 und 1. Jede Zahl lässt sich dann durch Anwenden der Addition und der Bildung der negativen Zahl beschreiben:

3 durch 1 + 1 + 1 oder -4 = -(1+1+1+1).

Auch die 0 könnte man wegen 0 = 1 - 1 noch weglassen.

Einstellige Operationen:

+	einstelliges Plus (wie Identität, verändert also nichts)
-	einstelliges Minus, Bildung der negativen Zahl, Negation
abs	Absolutbetrag einer Zahl

Zweistellige Operationen:

+	Addition zweier Zahlen
-	Subtraktion (=Differenz) zweier Zahlen
*	Multiplikation zweier Zahlen
mod	Modulo-Funktion (beachte: $0 \le x \bmod y < \text{abs } y$)
/	ganzzahlige Division (wie bei reellen Zahlen, aber dann nächste ganze Zahl in Richtung zur Null nehmen)
rem	Rest bei der Division mit /, d.h.: x rem y = x - (x/y)*y.
**	Exponentiation (x**y = x^y, nur für $y \ge 0$ definiert)

Hinzu kommen die sechs Vergleichsoperationen "=", "/=", "<", "<=", ">" und ">=" der Funktionalität $\mathbf{Z} \times \mathbf{Z} \to \mathbf{IB}$.

Da Integer als Aufzählungstyp aufgefasst wird, sind auch Min, Max, Pred, Succ, Pos und Val für Integer definiert, allerdings *mit der Ausnahme*, dass die Zahl 0 die Positionsnummer 0 erhält und Pos und Val auch negative Zahlen sein können, d.h., in Ada gilt für alle ganzen Zahlen a:

Pred(a) = a-1; Succ(a) = a+1; Pos(a) = a, Val(a) = a.

In der Praxis orientiere man sich stets zuvor, welche Zahlen auf dem jeweiligen Rechner Integer'First und Integer'Last sind.

Heute sind es oft die Zahlen
$-2\_147\_483\_648 = -2^{31}$ und
$2\_147\_483\_647 = 2^{31}-1$
(dies entspricht einer 32-Bit-Darstellung).

Einschränkungen in Ada:

Gewisse Kombinationen sind in Ausdrücken ohne zusätzliche Klammerung verboten, insbesondere:
Folgt ein einstelliger Operator nach einem höher- oder gleichrangigen zweistelligen Operator, so müssen Klammern verwendet werden.
Da die Exponentiation nicht assoziativ ist, muss bei Mehrfachverwendung von ** geklammert werden.
Kombinationen aus not, abs und ** dürfen nur mit Klammern verwendet werden.

Verboten sind also: x / -y, x**abs y, x**y**z, abs X**Y, not abs(X) > 17.

Prioritäten der gängigen Operatoren in Ada:

1. Priorität:	abs, ** , not
2. Priorität:	*, mod, /, rem
3. Priorität:	+, - (als einstellige Operationen)
4. Priorität:	+, - (als zweistellige Operationen)
5. Priorität:	=, /=, <, <=, >, >=
6. Priorität:	and, or, xor

Bei Integer ist (im Gegensatz zu Boolean) die Verwendung gleichrangiger Operatoren ohne Klammern erlaubt (siehe jedoch vorige Folie). Dies gilt auch für Float.

Empfehlung: Ausdrücke in Ada, aber auch in in anderen Programmiersprachen, möglichst vollständig klammern!

1.8.6 Der Datentyp Float in Ada

In Ada werden die reellen Werte durch endlich viele Zeichen angenähert dargestellt und zwar entweder als Gleitkommazahl oder als Festkommazahl.

Darstellung der Festkommazahlen: Man gibt in Ada den Bereich von der kleinsten ("unten") bis zur größten ("oben") darstellbaren Zahl in der Form *range unten .. oben* an und legt durch das reservierte Wort *delta* die feste Differenz "d" zwischen je zwei aufeinander folgenden Zahlen fest. Deklaration:

type <Datentypname> is delta d range unten .. oben;

Beispiel: type km is delta 0.001 range -1000.0 .. 1000.0;
Dies legt das Intervall von -1000.0 bis 1000.0 als Wertemenge fest, das mit der Schrittweite 0.001 durchlaufen wird. Dies sind insgesamt 2_000_001 Werte.

Darstellung von Festkommazahlen durch Dezimalziffern

Eine andere Möglichkeit, wie sie zum Beispiel im Geldverkehr gefordert wird, ist die Angabe der Nachkommastellen und der Gesamtzahl an Dezimalziffern. In Ada beschreibt man diesen durch Dezimalziffern festgelegten Festkommazahlentyp wie folgt (a ist eine natürliche Zahl, p eine Zehnerpotenz):

type <Datentypname> is delta p digits a;

Der Wertebereich sind alle positiven und negativen Zahlen, die mit a Dezimalziffern beschrieben werden können, wobei das Komma so zu setzen ist, dass zwei aufeinander folgende Zahlen den Abstand p voneinander haben.

Beispiel: type kontostand is delta 0.01 digits 8;
Dies legt als Wertemenge genau die Zahlen des Intervalls von -999999.99 bis 999999.99 mit der Schrittweite 0.01 fest.

Darstellung von Gleitkommazahlen (Details siehe später 2.4.14)

Hier muss man zunächst nichts tun, denn in Ada ist der Typ Float vordefiniert, der der Gleitkommadarstellung entspricht und mindestens eine Genauigkeit von 6 Dezimalziffern besitzen muss; dies entspricht einer Mantisse (Ziffernfolge der Zahl ohne führende Nullen) von mindestens 21 Binärstellen. Weiterhin gibt es einen Typ LONG_FLOAT mit mindestens 11 Dezimalziffern Genauigkeit.

Man kann die Genauigkeit vorgeben, z.B. durch

type MY_FLOAT is digits 14;

wodurch die Mantisse (= Folge der gültigen Ziffern) durch mindestens 48 Binärstellen dargestellt werden muss.

(Faustformel: Wegen $2^{10} \approx 10^3$ kann man von "digits a" auf rund $10 \cdot a/3 + 1$ Binärstellen schließen; "+1" wegen des Vorzeichens.)

Operationen auf den reellen Typen

Für Gleitkomma- und Festkommadarstellungen gibt es die einstelligen Operationen +, - und Absolutbetrag und die zweistelligen Operationen Addition +, Subtraktion -, Multiplikation *, Division / und Exponentiation ** (rechts von ** darf in Ada nur eine ganze Zahl stehen).

Weiterhin gibt es die sechs Vergleichsoperatoren: =, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist. Man sollte wegen der unvermeidlichen Rundungsfehler die Vergleichsoperatoren = und /= bei reellen Zahlen aber *nicht* verwenden.

Hinweis: Darüber hinaus gibt es eine Vielzahl weiterer Funktionen (Sinus, Cosinus, Exponentialfunktion, Logarithmus, ...) in speziellen Paketen für die reellen Datentypen, z.B. in Ada.Numerics.Generic_Elementary_Functions .

Rundung von $\mathbb{R}$ nach $\mathbb{Z}$

Erinnerung: Runden zur nächsten ganzen Zahl:
round: $\mathbb{R} \to \mathbb{Z}$ ("round" = runden) mit

für $x \geq 0$: $\text{round}(x) = z \Leftrightarrow z \in \mathbb{Z}$ und $\text{abs}(z-x) < 0.5$ oder
$z \in \mathbb{Z}$, $x < z$ und $\text{abs}(z-x) = 0.5$,

für $x < 0$: $\text{round}(x) = -\text{round}(-x)$.

Z.B.: round(2.4) = 2, round(2.5) = 3, round(-2.4) = -2, round(-2.5) = -3.

round wandelt eine reelle Zahl zur nächsten ganzen Zahl. In Ada erfolgt diese Typumwandlung von Float nach Integer durch

Integer(<Ausdruck vom Ergebnistyp Float>)

Integer(X) ist also in Ada gleich obigem round(X).

Weitere Hinweise

Ausnahmsweise ist in Ada die Multiplikation von ganzen mit reellen Zahlen erlaubt, ebenso die Division einer reellen Zahl durch eine ganze Zahl.

Die Prioritäten für die Operationen sind die gleichen wie die, die am Ende von 1.8.5 angegeben sind.

Auf weitere Details gehen wir hier nicht ein. Sie sind in jedem Buch über Ada aufgelistet.

Hinweis: Da letztlich reelle Zahlen nur durch eine feste Zahl von Binärstellen angenähert werden können, kann man diese Menge auch als Aufzählungstyp auffassen. Dies ist in Ada 95 (aber nicht in früheren Ada-Definitionen) auch der Fall. Ab Ada 95 sind daher die Funktionen Pred und Succ auch auf den reellwertigen Datentypen zugelassen. Sie liefern die vorherige bzw. nächst größere darstellbare Zahl. (Man sollte dies aber nur benutzen, wenn man sich mit Ada wirklich gut auskennt.)

1.8.7 Konversion (Typumwandlung)

Ada ist "streng typisiert", was insbesondere bedeutet, dass die Datentypen der Operanden in Operationen genau festgelegt sind und beachtet werden müssen. Zum Beispiel ist in

```
X, Z: Float;  K, L: Integer;  ...
Z := X + K;
```

der Ausdruck X+K nicht zulässig, da der Operator "+" entweder für zwei ganze Zahlen oder für zwei reelle Werte definiert ist, aber nicht für einen Mix hieraus. Man muss zunächst die Datentypen der Operanden anpassen:

```
Z := X + Float(K);     oder       L := Integer(X) + K;
```

Float bewirkt die Umwandlung einer ganzen Zahl in die entsprechende reelle Zahl; Integer entspricht genau der Rundungsfunktion "round".

1.8.8 Initialisierung von Variablen

Variablen werden wie folgt deklariert (siehe 1.4):

<Liste von Bezeichnern> **:** <Datentyp> **;**

Sehr oft will man einer Variablen zu Beginn einen Wert zuweisen (man sagt: die Variable wir initialisiert). Dies ist bereits in der Deklaration erlaubt durch

<Liste von Bezeichnern> **:** <Datentyp> **:=** <Ausdruck> **;**

Jeder Bezeichner in der <Liste von Bezeichnern> erhält hierdurch den Wert, den <Ausdruck> liefert, als Anfangswert zugewiesen. Stimmen bei der tatsächlichen Berechnung <Datentyp> und der Ergebnistyp von <Ausdruck> nicht überein, so bricht das Programm hier mit einem Fehler ab.

Diese Initialisierung ist auch für strukturierte Datentypen zulässig. Wir hatten den Datentyp Rational in 1.6 eingeführt. Die Deklaration von Konstanten und die Initialisierung von Variablen können nun in der Reihenfolge der Komponenten oder durch explizite Angabe <Selektor> => <Ausdruck> erfolgen. Beispiel:

```
type Rational is record
      Zaehler: Integer;
      Nenner: Positive;
end record;
Ein_Drittel: constant Rational := (1,3);
Zwei_Drittel: constant Rational := (Nenner => 3, Zaehler => 2);
X, Y: Rational := (1,1);
Z: Rational := (X.Zaehler+Ein_Drittel.Nenner,Y.Nenner);
Q: Rational := (1,-1);
```

Fehler, weil -1 nicht vom Datentyp Positive ist

Dies ist auch für Aufzählungstypen und für Felder ("array", siehe 1.9) zugelassen: Beispiele:

```
Zehn, Ten, Dix, Dieci, Diez, Deka, On, Shi, To: Integer := 10;
EinK: constant := 1024;     -- bei Integer und Float darf man diesen
                            -- Datentyp ausnahmsweise in einer
                            -- Konstantendeklaration weglassen
type Wochentag is (Mo, Di, Mi, Do, Fr, Sa, So);
Mitte_der_Woche: constant Wochentag := Mi;
Tag: Wochentag := Wochentag'Succ(Mitte_der_Woche);
```

Vorgriff auf Abschnitt 1.9:

```
Ueberschrift: constant array (1..5) of Wochentag
                          := (Mo, Di, Mi, Do, Fr);
X: array (1..3) of Float := (1.0, 2.0, 3.0);
Y: array (1..EinK) of Character := (1=>1, 2=>7, others=>0);
```

(Durch die Schreibweise <Index> => ... wird man unabhängig von einer zufällig gewählten Reihenfolge.)

1.8.9 Ausdrücke und Ein-Ausgabe

Ausdrücke werden aus Konstanten, Variablen und Operationen "wie üblich" gebildet. Wir erwarten, dass dies bekannt ist.

Beispiele:

Arithmetische Ausdrücke:

A + 23.0 - (X + 3*Z)**2

Hierin kommt 23.0 vor, welches nur vom Typ Float sein kann. Daher liegt genau dann ein korrekter Ausdruck vor, wenn A, X und Z Ausdrücke (z.B. Variablen oder Konstanten) vom Typ Float sind. Der Ergebnistyp dieses Ausdrucks ist Float. (Beachte: 3*Z ist ausnahmsweise erlaubt, siehe Ende von 1.8.6)

(Jahr-1) * 365 + (Monat-1) * 30 + Tag

ist ein ganzzahliger Ausdruck, aber nur, falls Jahr, Monat und Tag vom Typ Integer sind.

Beispiele:

Character-Ausdruck:

Pred(Val(23+J))

ist ein Ausdruck, sofern J vom Typ Integer ist und der Wert von (23+J) im Intervall von 0 bis 255 bleibt.

Boolesche Ausdrücke:

A
not A and (Integer(365.25*59) > 20_850 or K = N-1)
(B and not C) or (A xor B)
(B and not (-K = N+3)) or ((K < N) xor (A or B))

sind Boolesche Ausdrücke, sofern A, B und C Boolesche Ausdrücke (z.B. Boolesche Konstanten oder Variablen) und K und N ganzzahlige Ausdrücke sind. Beachten Sie die Pflicht, bei den gleichwertigen Booleschen Operationen zu klammern.

Die Ein- und Ausgabe erfolgt über die Standardfunktionen Get und Put.

Dies sind Funktionen, die von der Ada-Bibliothek bereitgestellt werden. Man muss die zugehörigen Pakete, in denen sie definiert sind, daher explizit angeben.

```
with Ada.Text_IO;                (sofern man nur Texte behandelt)
with Ada.Integer_Text_IO;  with Ada.Float_Text_IO
```

sofern man bei der Ein-/Ausgabe ganzzahlige Werte bzw. reellwertige Datenbereiche benutzt.

Die Ein-/Ausgabe wird dann z.B. mittels

Ada.Integer_Text_IO.Put(X)

angegeben. Will man die explizite Angabe des Pakets sparen, so muss ein use Ada.Integer_Text_IO usw. hinzugefügt werden. Details siehe Übungen und selbst anlesen.

1.9 Felder

1.9.1 Unterbereiche/Untertypen

Ist M eine angeordnete Menge und sind a, b ∈ M mit a <=b, so ist der Unterbereich von a bis b der Menge M

a .. b = {x ∈ M | a ≤ x ≤ b}.

Der Unterbereich übernimmt alle Operationen von M.

Unterbereiche werden in Ada als "subtype" (Untertyp) eines anderen Datentyps bezeichnet. Schema hierzu:

subtype <Name> is <Datentyp> <constraint>

Die Einschränkung <constraint> gibt den Bereich ("range") von unterer bis oberer Grenze an: <constraint> hat in Ada die Form

range <Ausdruck1> .. <Ausdruck2>

Jeder Ausdruck muss einen Wert mit <Ausdruck1> ≤ <Ausdruck2> liefern. Die Einschränkung darf aber fehlen. In diesem Fall hat der Untertyp die gleiche Wertemenge wie der <Datentyp>.

Basistyp

In der Definition

subtype U is T <constraint>

ist U der Untertyp von T und T kann man als den "Obertyp" von U bezeichnen. Der Datentyp, von dem U letztlich abgeleitet wurde, heißt Basisdatentyp oder kurz Basistyp zu U. Es sei T ein skalarer Datentyp. Im Falle

subtype U is T <constraint>
subtype V is U <constraint>
subtype W is V <constraint>

ist W Untertyp von V, von U und von T. U ist Obertyp von V und von W. Der Basistyp von U, V und W ist T.
Alle Operationen des Basistyps T sind automatisch auch für jeden Untertyp U, V und W gültig.

Operationen

"subtype" bezieht sich ausschließlich auf die Wertebereiche. Alle Operationen des Basistyps werden vom Unterdatentyp übernommen.

In Ada wird jede Variable eines Untertyps stets auch als Variable des Basistyps aufgefasst, so dass man also rechnen kann, als ob man im Obertyp wäre. Erst bei der Zuweisung des Ergebnisses wird geprüft, ob das Ergebnis die Einschränkung erfüllt, also im Bereich range <Ausdruck1> .. <Ausdruck2> ist.

(Ein Untertyp wird in Ada also als der Basistyp mit zusätzlicher Einschränkung des Wertebereichs aufgefasst.)

Beispiel

subtype Monatstage is Integer range 1..31;

subtype MinMonatstage is Monatstage range 1..28;

Will man zu einem Tag vom Typ Monatstage den Tag der folgenden Woche (plus 7) berechnen, so kann man schreiben:

```
X, Z: Monatstage;   .....
Z := X + 7;
if Z > 31 then Z := Z-31; end if;
```

Dies führt natürlich zu einem Fehler, falls X mindestens den Wert 25 besaß, da dann Z den Unterbereich verlassen würde. Dagegen führt if X > 24 then Z := X + 7 - 31; end if; nicht zu einem Fehler, auch wenn zwischenzeitlich der Bereich verlassen wurde, da Ada den Ausdruck X+7-31 im Basistyp Integer berechnet.

Zuweisungen an Variablen der Untertypen sind immer erlaubt, wenn der zuzuweisende Wert im Unterbereich liegt:

subtype Monatstage is Integer range 1..31;

subtype MinMonatstage is Monatstage range 1..28;

X, Y, Z: Monatstage; A: MinMonatstage; H, J: Integer; .....
H := 50; X := 30; Z := 20;

Beachte: (H*Z) mod 30 +1 ist ein Ausdruck, der in den ganzen Zahlen berechnet wird. Erst das Ergebnis wird der Variablen Y zugewiesen. Folglich sind erlaubt:

A := Z; Y := H-20; Y := (H*Z) mod 30 +1; J := X; A := X-Z;

Verboten sind aber (d.h., zu Bereichsüberschreitungen führen):

A := X; Z := H; A := Z*Z; Z := Z*Z;

Wichtige vordefinierte Untertypen in Ada sind:

subtype **Natural** is Integer range 0..Integer'Last;

subtype **Positive** is Integer range 1..Integer'Last;

Unterbereiche kann man auch für den Typ Float einführen, sofern die Grenzen Ausdrücke sind, deren Ergebnisse reellwertig sind:

subtype Einheitsintervall is Float range 0.0 .. 1.0;

1.9.2 Felder (statisch)

Für Elemente der Wertemengen M^r benötigt man r Variablen der gleichen Art, auf die mit einem Index zugegriffen werden kann. Diese fasst man unter einem Namen zu einem "Feld" (oder Vektor) zusammen. Man muss den Indexdatentyp (meist ist dies ein Unterbereich) und den Werte-Datentyp angeben. Schema:

array (<Indexdatentyp>) of <Datentyp>.

In der Regel muss man in Ada jedem Datentyp (aber nicht unbedingt den Untertypen) einen eigenen Namen geben.

(Ausnahme: Bei Feldern ist dies nicht notwendig, was jedoch zu Unverträglichkeiten zweier Deklarationen führen kann, die man als gleich auffasst, die es aber in Ada nicht sind, siehe später.)

Wir empfehlen daher, Feldern stets einen Namen zu geben:

type <Feldname> is array (<Indexdatentyp>) of <Datentyp>;

Hierbei darf der <Datentyp> der Komponenten des Feldes erneut eine Felddeklaration sein (das ergibt dann eine Matrix):

array <Datentyp des 1.Index> of
array <Datentyp des 2.Index> of <Datentyp>

Dies kürzt man in Ada ab durch
array <Datentyp des 1.Index>, <Datentyp des 2.Index> of <Datentyp>

und nennt dies ein zwei-dimensionales Feld. Wiederholt man dies, so lassen sich d-dimensionale Felder deklarieren, d ≥ 1. Zum Beispiel braucht man dreidimensionale Felder zur Darstellung von Volumendaten.

Allgemeines Schema zur Einführung eines Feld-Datentyps:

type <name> is
array (<Liste_der_Indexdatentypen>) of <Datentyp>

Beispiel: Stundenplan in einer Schule

```
type Lehrer is (Hinz, Kunz, Maier, Müller, Schmidt);
type Stunde is Integer range 1..9;
type Schultag is (Mo, Di, Mi, Do, Fr, Sa);
type Fach is (Deu, Mathe, Engl, Phy, Bio, Gesch, Musik, frei);
type Klasse is Integer range 5..13;
type Stundenplan is array (Klasse, Schultag, Stunde) of Fach;
type Lehrerplan is array (Lehrer, Schultag, Stunde) of Klasse;
```

(Erkennen Sie eine Schwäche dieser Modellierung?!)

Beispiele:

```
type Codierung is array (Character) of Character;
type Matrix is array (1..500, 1..500) of Float;
type Farbe is (weiss,rot,gelb,gruen,hellblau,blau,violett,schwarz);
type Farbvoxel is array (1..X, 1..Y, 1..Z) of Farbe;
```

Beispiel Skalarprodukt

```
with ...; use ...;
procedure skalarprodukt1 is      -- Berechne die Summe (X(i) mal Y(i))
Max: constant Integer := 1000;
type Vektor is array (1..Max) of Float;
X, Y: Vektor;  Sk: Float := 0.0; N: Natural;
begin  Get(N);                   -- Anzahl der Komponenten je Vektor
  if (N > 0) and (N <= Max) then
        for i in 1..N loop Get (X(i)); end loop;
        for i in 1..N loop Get (Y(i)); end loop;
        for i in 1..N loop Sk := Sk + X(i) * Y(i); end loop;
        Put(Sk);
  else Put ("Anzahl N ist zu klein oder zu groß.");
  end if;
end;
```


Statt Konstanten dürfen in den Indexdatentypen auch Ausdrücke verwendet werden, sofern jeder Ausdruck in dem Augenblick, in dem die Deklaration erreicht wird, auch tatsächlich ausgerechnet werden kann (vgl. 1.9.1).

Beispiele (wobei f und g Funktionen vom Ergebnistyp Integer sein sollen):

```
type Hilfs_Vektor is array (1..N, x..I*J) of Boolean;
type Ausschnitt is array (f(unten)..g(oben)) of Integer;
type Unklar is array ((oben-unten)/2..f(g(unten*oben))) of Float;
```

In der Regel verwendet man Felder in der Form

type F is array (u_1..o_1, u_2..o_2, .., u_n..o_n) of ...;

Die Werte u_1 und o_1 heißen untere und obere Feldgrenze der ersten Dimension von F, u_2 und o_2 heißen untere und obere Feldgrenze der zweiten Dimension von F usw.

Ein Feld heißt *statisch*, wenn zur Übersetzungszeit (also unabhängig von Eingabewerten) alle Feldgrenzen bekannt sind. Wenn man das Programm also als Text liest, kann man die Feldgrenzen bereits genau angeben.

Wird mindestens eine Feldgrenze aber erst zur Laufzeit des Programms berechnet, so heißt das Feld *dynamisch*. In diesem Fall muss man Blöcke (siehe später) verwenden, mit denen man die Speicherplatz-Reservierung zur Laufzeit vornehmen kann. Ein Beispiel finden Sie in 1.11.5.

Dürfen Feldgrenzen zur Laufzeit verändert werden, so heißt das Feld "*dynamisierbar*". (Dies ist in Ada nicht erlaubt.)

1.9.3 Unspezifizierte Feldgrenzen

In Ada kann man die Feldgrenzen auch undefiniert lassen und erst zur Laufzeit, also wenn das Programm erstmals an die betreffende Verwendung des Feldes gelangt, die konkreten Werte einsetzen. Man trägt in der Deklaration dann nur den Datentyp des Index ein und fügt dahinter ein

range <> ("<>" sprich "box")

hinzu. Solche unspezifizierten array-Deklarationen *muss* man bei ihrer Verwendung spezifizieren, z. B. bei der Deklaration von zugehörigen Variablen oder bei einem Parameteraufruf.

Beispiele:

```
type Text is array (Natural range <>) of Character;
type Raster is array (Integer range <>, Integer range <>) of Boolean;
type Ganzzahl_Vektor is array (Integer range <>) of Integer;
```

Bei der Deklaration von Variablen gibt man dann die konkreten Grenzen statisch oder dynamisch (I und J müssen dann Werte haben) vor, z. B.:

X: Ganzzahl_Vektor (-10..10); oder Y: Ganzzahl_Vektor (I..J);

In Ada sind weitere Operationen mit arrays verbunden (in Ada werden auch sie als *Attribute* bezeichnet):

Das Attribut "Range(i)" gibt zu jeder Feld-Variablen den Indexdatentyp ihrer i-ten Dimension an. Analog bezeichnet "Length(i)" die Anzahl der Elemente des Indexdatentyps in der i-ten Dimension. Mittels First(i) und Last(i) erhält man das erste bzw. letzte Element des Indexdatentyps der i-ten Dimension. Hat man nur eine Dimension, so darf man "(1)" weglassen.

Beispiel:

```
type Codes is array (Character range 'A'..'Z', 1..32) of
                                    Character range 'B'..'z';
W: Codes;
```

Dann gilt:

W'Range(2) = 1..32, W'Length(1) = 26,
W'First(1) = 'A' , W'Last(2) = 32.

Weiterhin darf man Feldvariablen, die *vom gleichen Datentyp* sind, einander zuweisen.

```
type H_Vektor is array (1..100) of Character;

X, Y: H_Vektor;

....

X := Y;
```

Bedeutung dieser Zuweisung: Der gesamte Inhalt von Y wird komponentenweise in die Variable X kopiert.

Dies ist auch für einzelne Dimensionen erlaubt, sog. Slices (Ausschnitte, "Scheiben"). Wir gehen hierauf nur in den Programmierübungen ein.

Mit unspezifizierten Feldgrenzen kann man z.B. Prozeduren und Funktionen unabhängig vom konkreten Indexdatentyp schreiben:

```
type Zahl_Vektor is array (Integer range <>) of Float;

function Skalarprodukt(X, Y: Zahl_Vektor) return Float is
    Sk: Float := 0.0;
begin
    if (X'First = Y'First) and (X'Last = Y'Last) then
        for i in X'Range loop Sk := Sk + X(i) * Y(i); end loop;
        return Sk;
    else raise Constraint_Error; end if;
end;

A, B: Zahl_Vektor (-5..115);

begin  ...                          -- die Vektoren A und B erhalten Werte ...
    if Skalarprodukt(A, B) > 0.0 then ....
...
```

Hier werden die tatsächlichen Grenzen festgelegt


```
with Ada.Text_IO; use Ada.Text_IO; with Text_IO; use Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;

procedure Summe_der_Quadrate is
type Zahl_Vektor is array (Integer range <>) of Float;
function Skalarprodukt(X, Y: Zahl_Vektor) return Float is
    Sk: Float := 0.0;
begin
    if (X'First = Y'First) and (X'Last = Y'Last) then
        for i in X'Range loop Sk := Sk + X(i) * Y(i); end loop;
        return Sk;
    else raise Constraint_Error; end if;
end;

A, B: Zahl_Vektor (1..20);

begin
    for I in A'Range loop A(I) := Float(I); B(I) := Float(I); end loop;
    if Skalarprodukt(A,B) > 0.0 then Put(Skalarprodukt(A,B),8,8,0);
            New_Line; Put(A'Last*(A'Last+1)*(2*A'Last+1)/6, 8);
    else null; end if;
end;
```

Einsatz dieses Beispiels in einem konkreten Programm.

Hier wird konkret der Bereich 1..20 festgelegt.

Ergebnis:

```
2870.00000000
2870
```

Hinweis: Die Summe der ersten n Quadratzahlen ist n(n+1)(2n+1)/6. Dies wird hier für n=20 berechnet.

Erläuterungen:

1. Einfacher wäre in der Funktion "Skalarprodukt" die Abfrage
   if X'Range = Y'Range then ...
   gewesen. In Ada ist jedoch die Verwendung des Attributs "Range" in Ausdrücken nicht zugelassen. Daher wurde die Formulierung mit X'First und X'Last gewählt.
2. In der Funktion "Skalarprodukt" wurde die elementare Anweisung raise <Ausnahmebehandlung> ;
   in der Form raise Constraint_Error;
   verwendet. Dies ist ein "kontrollierter Fehlerausgang". Das Programm bricht also nicht einfach zusammen und es ist unklar, was dabei geschieht, sondern es wird der Fehler "Constraint_Error" aufgerufen ("erweckt") und das Programm wird "sauber abgewickelt". In Ada gibt es vier voreingestellte Ausnahmebehandlungen, die man mittels raise aktivieren kann: Constraint_Error, Program_Error, Storage_Error, Tasking_Error.

1.10 BNF und EBNF

1.10.1 Beispiel

Wie beschreibt man die Struktur (das Schema) eines Programms und seiner Bestandteile? Bereits aus dem Jahre 1957 stammt die sog. BNF, Abkürzung für "Backus-Naur-Form", nach dem amerikanischen Informatiker J. Backus und dem dänischen Informatiker P. Naur, die diese Darstellung vorschlugen und hiermit die Programmiersprache ALGOL 60 definierten.

In der Theorie erweist sich die BNF als identisch mit den sog. "kontextfreien Grammatiken". Die BNF besteht aus Terminal- und Nichtterminalzeichen, einem Startsymbol und einer Menge von Regeln der Form

$$A ::= u_1 \mid u_2 \mid ... \mid u_k$$

wobei A ein Nichtterminalzeichen ist und die u_i $(i = 1, 2, ..., k)$ je eine Folge aus Terminal- und Nichtterminalzeichen sind.

Wir demonstrieren die BNF am Beispiel des Aufbaus eines Ada-95-Programms:

```
procedure <Name des Programms> is
   <Deklarationsteil>
begin
   <Anweisungsteil>
end;
```

Nichtterminalzeichen: <Name des Programms>
<Deklarationsteil>
<Anweisungsteil>

Sie dienen *zur Erzeugung* eines korrekten Programms und dürfen im Programm später nicht auftreten.
Terminalzeichen sind alle Zeichen, die im erzeugten Programm vorkommen dürfen. Hier sind dies:
procedure is begin end ;

Zu dem obigen Programmaufbau gehört die Regel:

<Programm> ::= procedure <Name *des Programms*> is
<Deklarationsteil> begin <Anweisungsteil> end;

Eine Regel der Form

$$A ::= u_1 \mid u_2 \mid ... \mid u_k$$

besagt: Beim Erzeugungsprozess darf das Nichtterminalzeichen A durch irgendeine der Zeichenfolgen u_i ersetzt werden. Für jedes Nichtterminalzeichen A gibt es genau eine Regel, in der A links von "::=" auftritt.

Ein Nichtterminalzeichen soll stets etwas bedeuten. Daher schreibt man eine geeignete Bezeichnung in spitze Klammern und verwendet diese Darstellung anstelle von A.

Statt <Name des Programms> schreiben wir <Name *des Programms*>. Mit dem Kursivdruck wird angedeutet, dass hier das Nichtterminalzeichen <Name> stehen muss, dass aber im späteren Programm dieser Name der Name des Programms sein muss.

Nichtterminalzeichen kann man beliebig wählen, während die Terminalzeichen, mit denen die Programme geschrieben werden, sich an den Wünschen der Kunden orientieren. Dennoch sollte man die Lesbarkeit der Syntax erhöhen, indem man verständliche Nichtterminalzeichen verwendet.

Beispiel: Wir hätten schreiben können:

α ::= procedure β is η begin φ end;

mit den Nichtterminalzeichen α, β, η, φ.

Lesbarer ist aber zwiefellos:

<Programm> ::= procedure <Name *des Programms*> is
<Deklarationsteil> begin <Anweisungsteil> end;

Nun müssen wir für die anderen Nichtterminalzeichen Regeln angeben. Treten hierbei neue Nichtterminalzeichen auf, so geben wir auch für diese Regeln an usw.

Vorsicht, nicht korrekt.

<Name *des Programms*> ::= <Bezeichner>

<Bezeichner> ::= <Buchstabe> | <Bezeichner> <Buchstabe> |
<Bezeichner> <Ziffer> | <Bezeichner> _

<Buchstabe> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N |
O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f |
g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<Ziffer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Ein Bezeichner ist jede Zeichenfolge aus Terminalzeichen, die man aus dem Nichtterminalzeichen <Bezeichner> erzeugen kann, also eine Folge von Buchstaben, Ziffern und dem Unterstrich, die mit einem Buchstaben beginnt. Prüfen Sie dies nach!

Haben Sie dies wirklich in Ada-95-Programmen ausprobiert?
Dann haben Sie entdeckt: *In Ada95 stimmt dies nicht!*
Beispiele:

Erlaubt sind: r45 ZZ6_T In_form_atik Drei_D_Video

Verboten sind: 3_D_Video 45r In__form_atik _Dx A_

Man erkennt: Der Unterstrich darf nicht zweimal nacheinander auftreten und er darf nicht am Ende stehen!

Also müssen wir unsere Regeln ändern!

Korrekte Festlegung von <Bezeichner> (engl.: <identifier>) in Ada. Man erzwinge, dass nach dem Unterstrich ein Buchstabe oder eine Ziffer folgen muss:

<Bezeichner> ::= <Buchstabe> |
<Bezeichner> <Buchstabe> |
<Bezeichner> <Ziffer> |
<Bezeichner> _ <Buchstabe> |
<Bezeichner> _ <Ziffer>

<Buchstabe> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N |
O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f |
g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<Ziffer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

In ähnlicher Weise behandeln wir nun den <Deklarationsteil>, vgl. hierzu den Anfang von 1.7, ε bedeutet: Hier steht nichts.

<Deklarationsteil> ::= <Folge_von_Deklarationen>

<Folge_von_Deklarationen> ::= ε |
<Deklaration> <Folge_von_Deklarationen>

<Deklaration> ::= <Variablendeklaration> | <Typdeklaration> |
<Konstantendeklaration> | <Untertypdeklaration> |
<Funktionsdeklaration> | <Prozedurdeklaration>

<Variablendeklaration> ::=
<Liste_von_Variablen> **:** <Datentyp> **;**

<Liste_von_Variablen> ::= <Variable> |
<Variable> **,** <Liste_von_Variablen>

<Variable> ::= <Bezeichner>

usw. Versuchen Sie, die weiteren fünf Deklarationen und den <Datentyp> selbst auszuformulieren.

1.10.2 BNF Definition

Eine BNF (Backus-Naur-Form) ist ein Viertupel (V, Σ, P, S) mit

(1) V ist eine nicht-leere endliche Menge (die Menge der Nichtterminalzeichen); alle Elemente sind von der Form < Zeichenkette > (in "Zeichenkette" treten '<' und '>' nicht auf),

(2) Σ ist eine nicht-leere endliche Menge (die Menge der Terminalzeichen) mit $V \cap \Sigma = \emptyset$ und $| \notin \Sigma$,

(3) $S \in V$ ist ein Nichtterminalzeichen (das Startsymbol),

(4) P ist eine endliche Menge von Regeln oder Produktionen. Zu jedem Nichtterminalzeichen A gibt es in P genau eine Regel der Form $A ::= u_1 \mid u_2 \mid ... \mid u_k$ $(k \geq 1)$, wobei jedes u_i eine Folge von Zeichen aus $(V \cup \Sigma)$ ist; die leere Zeichenfolge ε ist hierbei als ein u_i zugelassen.

Abstraktes Beispiel:

(V, Σ, P, S) sei gegeben durch $\Sigma = \{0, 1\}$, $V = \{S, A\}$ und P bestehe aus den beiden Regeln

$S ::= 1A \mid 0$ und $A ::= 1A \mid 0A \mid \varepsilon$

Welche Zeichenfolgen über Σ kann man aus S erzeugen?

$S \Rightarrow 0$
$S \Rightarrow 1A \Rightarrow 1$
$S \Rightarrow 1A \Rightarrow 10A \Rightarrow 10$
$S \Rightarrow 1A \Rightarrow 11A \Rightarrow 11$
$S \Rightarrow 1A \Rightarrow 10A \Rightarrow 100A \Rightarrow 100$
$S \Rightarrow 1A \Rightarrow 10A \Rightarrow 101A \Rightarrow 101$
$S \Rightarrow 1A \Rightarrow 11A \Rightarrow 110A \Rightarrow 110$
$S \Rightarrow 1A \Rightarrow 11A \Rightarrow 111A \Rightarrow 111$
...

Man erkennt: Aus S lassen sich alle beliebige langen Folgen aus Nullen und Einsen erzeugen, die mit einer 1 beginnen sowie die Zeichenfolge 0, d.h., alle Binärdarstellungen von $\mathbb{N}_0$, also genau die Menge $\{0\} \cup \{1u \mid u \in \{0,1\}^*\}$.

Programmiersprachen-Beispiel:

Durch die folgende BNF (V', Σ', P', S') sollen (ganzzahlige und vollständig geklammerte) arithmetische Ausdrücke beschrieben werden. Ein solcher arithmetischer Ausdruck ist eine ganze Zahl oder eine Variable oder es sind arithmetische Ausdrücke, die durch Operationen +, -, abs, *, / und mod verknüpft sind. Wir nehmen hier an, alle Variablen seien von der Form Xi, wobei i eine natürliche Zahl ist. Wir definieren:

V' = {<arithmAusdr>, <ganzeZahl>, <Variable>, <Zahl≠0>,
<Ziffer≠0>, <Ziffer>},
Σ' = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, X, (,), +, -, abs, *, /, mod},
S' = <arithmAusdr>

Die Menge P' muss aus genau sechs Regeln bestehen, da V' sechs Elemente besitzt.

P' besteht aus den Regeln:

<arithmAusdr> ::= <ganzeZahl> | <Variable> | (<arithmAusdr>) |
+ <arithmAusdr> | - <arithmAusdr> | abs(<arithmAusdr>) |
(<arithmAusdr> + <arithmAusdr>) | (<arithmAusdr> - <arithmAusdr>) |
(<arithmAusdr> * <arithmAusdr>) | (<arithmAusdr> / <arithmAusdr>) |
(<arithmAusdr> mod <arithmAusdr>)

<ganzeZahl> ::= 0 | <ganzeZahl≠0>

<ganzeZahl≠0> ::= <Ziffer≠0> | <ganzeZahl≠0> <Ziffer>

<Ziffer≠0> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Ziffer> ::= 0 | <Ziffer≠0>

<Variable> ::= X <ganzeZahl>

Machen Sie sich an Beispielen klar, dass Sie genau die geforderten arithmetischen Ausdrücke erzeugen können, z.B. 0 oder X2 oder (X13 + X0) oder (X1 - ((45 * 13) / X8)) usw.

Hinweis: Wir heben Terminalzeichen manchmal in blau hervor:

Σ' = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, X, (,), +, -, abs, *, /, mod}

<arithmAusdr> ::= <ganzeZahl> | <Variable> | (<arithmAusdr>) |
+ <arithmAusdr> | - <arithmAusdr> | abs(<arithmAusdr>) |
(<arithmAusdr> + <arithmAusdr>) | (<arithmAusdr> - <arithmAusdr>) |
(<arithmAusdr> * <arithmAusdr>) | (<arithmAusdr> / <arithmAusdr>) |
(<arithmAusdr> mod <arithmAusdr>)

<ganzeZahl> ::= 0 | <ganzeZahl≠0>

<ganzeZahl≠0> ::= <Ziffer≠0> | <ganzeZahl≠0> <Ziffer>

<Ziffer≠0> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Ziffer> ::= 0 | <Ziffer≠0>

<Variable> ::= X <ganzeZahl>

Erzeugbare arithmetische Ausdrücke:
45 oder X13 oder (X13 * X0) oder (X1 - ((45 * 13) / X8)) usw.

1.10.3 EBNF

Man erweitert die BNF meist um folgende Möglichkeiten.

a) Verwenden von Schlüsselwörtern:
Schlüsselwörter der Sprache werden als Folgen von Zeichen dargestellt, die in Apostrophe eingeschlossen und deutlich hervorgehoben werden. (Tritt im Schlüsselwort ein Apostroph auf, so wird dies durch zwei Apostrophe dargestellt.)

Beispiel: <Boolesche Operatoren> ::= 'and' | 'or'

Hierdurch wir ausgedrückt, dass die Operatoren and und or eigentlich einzelne Terminalzeichen sind, jedoch als Zeichenkette durch Hintereinanderschreiben anderer Zeichen dargestellt werden. Man könnte ansonsten in BNF auch schreiben:

<And-Operator> ::= and <Or-Operator> ::= or
<Boolesche Operatoren> ::= <And-Operator> | <Or-Operator>

b) Einführen von eckigen Klammern ("ein- oder keinmal"):
Symbole oder Folgen von Symbolen, die auch wegfallen dürfen, werden in eckige Klammern eingeschlossen.

Beispiele:

<Ziffernfolge> ::= <Ziffer> [<Ziffernfolge>]

(Dies ist die Abkürzung für
<Ziffernfolge> ::= <Ziffer> | <Ziffer> <Ziffernfolge>)

<Alternative> ::= 'if' <Bedingung>
'then' <Anweisung>
['else' <Anweisung>] 'end' 'if' ;

c) Einführen von geschweiften Klammern ("Iteration"):
Symbole oder Folgen von Symbolen, die beliebig oft (auch keinmal) wiederholt werden dürfen, werden in geschweifte Klammern eingeschlossen.

Beispiele:

<Ziffernfolge> ::= <Ziffer> {<Ziffer>}
ist die Abkürzung für
<Ziffernfolge> ::= ε | <Ziffer> <Ziffernfolge>

Definition von Bezeichnern in Ada, siehe 1.10.1:

<Bezeichner> ::=
<Buchstabe> { [_] <Buchstabe> | [_] <Ziffer> }

Beachte: "|" darf nun auch in den Ausdrücken stehen!

Definition: Erweiterte Backus-Naur-Form
Die EBNF ist die Erweiterung der BNF um die Möglichkeiten:

Apostrophe für Schlüsselwörter als Terminalzeichen. (Wenn wir die Terminalzeichen blau drucken, lassen wir die ' ' weg.)

Eckige Klammern mit der Bedeutung: Die eingeschlossenen Teile können hier ein Mal auftreten, müssen aber nicht.

Geschweifte Klammern mit der Bedeutung: Die hierdurch eingeschlossenen Teile können beliebig oft nacheinander auftreten.

Ergänzender Hinweis: Falls eckige oder geschweifte Klammern als Terminalzeichen in den Regeln auftreten können, so bleibt es dem Ersteller der Regeln überlassen, für Eindeutigkeit zu sorgen.

Hinweis: Es gibt Varianten der EBNF.

Oft schreibt man (...)* statt {...}
und führt auch das Konstrukt (...)+ ein, wenn man die 0-Iteration (= den leeren Fall) verbieten möchte.

Statt ::= wird manchmal auch einfach = geschrieben.

Einzelne Terminalzeichen müssen oftmals in Anführungsstriche eingeschlossen werden.
Schlüsselwörter werden dann meist fett gedruckt.
Die Nichtterminalzeichen werden dann nicht mehr in spitze Klammern '<' und '>' eingeschlossen.

Regeln werden mit einem besonderen Zeichen, meist mit einem Punkt, abgeschlossen.

(An solche Varianten gewöhnt man sich aber rasch.)

1.11 Kontrollstrukturen

1.11.1 Überblick

Als Kontrollstrukturen kennen wir bereits die Hintereinanderausführung, die ein- und zweiseitige Fallunterscheidung (if ... end if) und zwei Schleifenarten (while und for). Es gibt noch weitere Kontrollstrukturen in Ada.

Betrachte einen Ausschnitt aus der Ada-Syntax in BNF:

<Anweisungsfolge> ::= <Anweisung> |
<Anweisung> <Anweisungsfolge>

<Anweisung> ::= <elementare_Anweisung> |
<zusammengesetzte_Anweisung>

<elementare_Anweisung> ::= <leere_Anweisung> |
<Zuweisung> | <exit_Anweisung> |
<Prozedur_Aufruf> | <return_Anweisung> |
<Sprung_Anweisung> | <raise_Anweisung>

<leere_Anweisung> ::= null ;

<Zuweisung> ::= <*Variablen*_Name> := <Ausdruck> ;

<exit_Anweisung> ::= exit when <Bedingung> ; | exit ;

<Sprung_Anweisung> ::= goto <Marke> ;

<Marke> ::= <Bezeichner>

<Prozedur-Aufruf> ::= <Prozedurname> ; |
<Prozedurname> <Aktueller_Parameterteil> ;

<return_Anweisung> ::= return ; | return <Ausdruck> ;

<raise_Anweisung> ::= raise <Name_der_Ausnahme> ;

Bemerkung: Es gibt in Ada noch weitere einfache Anweisungen, die unter <elementare Anweisung> aufgelistet werden müssten. Einige behandeln wir später oder im Programmierkurs oder auch gar nicht. Auch die obige und die folgende Syntax sind nicht vollständig; vielmehr gibt es noch weitere Varianten. (=> Lesen Sie das Original der Ada-Syntax.)

Im Prinzip sind alle diese elementaren Anweisungen klar:

- leere_Anweisung: Tue nichts.
- Zuweisung: Werte den Ausdruck aus und lege den Wert in der Variablen ab (sofern der Datentyp übereinstimmt).
- exit_Anweisung: Verlasse sofort die Schleife.
- Sprung-Anweisung: Fahre mit der Anweisung fort, die mit der <Marke> bezeichnet ist. (Hier gibt es Einschränkungen; diese Anweisung führt oft zu undurchschaubaren Programmen, daher sollte man sie meiden.)
- Prozedur_Aufruf: Führe die Prozedur mit den aktuellen Parametern durch (sofern welche notwendig sind).
- return-Anweisung: Verlasse sofort die Funktion oder Prozedur.
- raise_Anweisung: Führe sofort die Ausnahmebehandlung durch.

Zu den Marken: Man darf unmittelbar vor jede Anweisung eine Marke (das ist ein in << ... >> eingeschlossener Bezeichner) setzen. Die Marke dient nur als Ziel für Sprunganweisungen.

<zusammengesetzte_Anweisung> ::= <if_Anweisung> |
<case_Anweisung> | <Schleife> | <Block>

<if_Anweisung> ::=
if <Bedingung> then <Anweisungsfolge> end if ; |
if <Bedingung> then <Anweisungsfolge>
else <Anweisungsfolge> end if ;

<Bedingung> ::= <Boolescher_Ausdruck>

<case_Anweisung> ::= case <Ausdruck> is
<case_Anweisungsfolge> end case ;
<case_Anweisungsfolge> ::= <case_Anweisung_Alternative> |
<case_Anweisung_Alternative> <case_Anweisungsfolge>
<case_Anweisung_Alternative> ::=
when <diskrete_Auswahlliste> => <Anweisungsfolge>

<Schleife> ::=
<Iterationsschema> loop <Anweisungsfolge> end loop ;

<Iterationsschema> ::= while <Bedingung> |
for <Zählschleifen_Angabe> | ε

<Zählschleifen_Angabe> ::= <Bezeichner> in <Unterbereich> |
<Bezeichner> in reverse <Unterbereich>
<Block> ::= declare <Deklarationsteil> begin
<volle_Anweisungsfolge> end ;
<Unterbereich> ::= <diskreter_Datentyp> |
<Datentyp> range <Ausdruck> .. <Ausdruck>

Hinweise: Ein <diskreter_Datentyp> muss eine endliche Wertemenge besitzen, z.B. Intervalle ganzer Zahlen oder ein Aufzählungstyp. Eine <volle_Anweisungsfolge> ist nicht leer und darf am Ende eigene Ausnahmebehandlungen besitzen.

Wir beschreiben nun einige dieser Anweisungen.

1.11.2 Fallunterscheidung (if)

Die vollständige Fallunterscheidung lautet in EBNF:

<if_Anweisung> ::=
    if <Bedingung> then <Anweisungsfolge>
    {elsif <Bedingung> then <Anweisungsfolge>}
    [else <Anweisungsfolge>] end if ;

Eine Kaskade von Fallunterscheidungen
(Bi sind Bedingungen, Ai sind Anweisungsfolgen):

```
if B1 then A1 else if B2 then A2 else if B3 then A3
else if B4 then A4 else A5 end if; end if; end if; end
if; end if;
```

lässt sich mit "elsif" also abkürzen zu

```
if B1 then A1 elsif B2 then A2 elsif B3 then A3
elsif B4 then A4 else A5 end if;
```


1.11.3 Auswahlanweisung ("case")

Mit der case_Anweisung kann man beliebig viele Alternativen vorgeben. Die Alternativen (<diskrete_Auswahlliste>) kennen wir ansatzweise schon von der Initialisierung:

<diskrete_Auswahlliste> ::= <diskrete_Auswahl> |
    <diskrete_Auswahl> <diskrete_Auswahlliste>
<diskrete_Auswahl> ::= <Ausdruck> | <diskreter_Bereich> | others

Die Auswahlmöglichkeiten müssen *disjunkt und vollständig* sein! Einzelne Alternativen werden durch einen Strich "|" getrennt. Beispiel (Tag sei eine Variable von Datentyp Wochentage, siehe 1.8.1; Schulbeginn sei eine Variable für Texte):

```
case Tag is
    when Mo | Di => Schulbeginn := "7:45";
    when Mi..Fr => Schulbeginn := "8:35";
    when Sa => Schulbeginn := "9:40";
    when others => Schulbeginn := "entfällt";
end case;
```


Zur Auswertung von case-Anweisungen:

Prinzipiell wertet Ada sämtliche Möglichkeiten aus, d.h., für *alle* Auswahlen hinter allen when wird geprüft, ob sie zutreffen. Die case-Anweisung wird nur dann ausgeführt, wenn hierbei *genau einmal* der Wert True auftritt.

Daher können case-Anweisungen in der Praxis deutlich mehr Zeit benötigen als iterierte / kaskadierte if-Anweisungen, mit denen man jede case-Anweisung leicht simulieren kann (bis auf die Überprüfung, dass nur genau eine Bedingung zutreffen darf). Wenngleich aus Gründen der Lesbarkeit die case-Anweisungen oft zu bevorzugen ist, so sollte man sie dennoch in Programmen, die diese Überprüfung nicht benötigen und die schnell ablaufen sollen, vermeiden.

1.11.4 Schleifen und exit

Eine Schleife ist in Ada zunächst eine "unendliche Schleife" der Form loop ... end loop; . Die Anweisungsfolge zwischen loop und end loop nennt man das *Schleifen-Innere* oder den *Schleifen-Rumpf*.

Vor dem Schleifenbeginn "loop" kann man eine Wiederholungsbedingung angeben entweder mit einer Laufvariablen (for) oder mit einer Fortsetzungsbedingung (while). Die Laufvariable darf ihren Bereich auch von hinten nach vorne durchlaufen ("reverse"). Bei einer Laufvariablen ist gesichert, dass die Schleife nur endlich oft ausgeführt wird; in anderen Fällen sind unendliche Schleifen möglich.

Man kann die Schleife auch mit Hilfe der exit-Anweisung beenden. "exit when B;" verlässt die aktuelle Schleife, sofern die Bedingung B an dieser Stelle zutrifft ("when B" darf fehlen, dann wird die Schleife auf jeden Fall verlassen).

Man darf eine Schleife mit einem Bezeichner versehen, der abgetrennt durch einen Doppelpunkt unmittelbar vor der Schleife stehen muss. Dies dient folgendem Zweck:

Will man mittels exit mehrere ineinander geschachtelte Schleifen auf einmal verlassen, so kann man den Namen der Schleife, die man beenden möchte, hinter exit angeben.

Mit exit darf man nicht beliebige Bereiche verlassen. Insbesondere darf man keine Bereiche verlassen, die einen Deklarationsteil besitzen oder besitzen könnten. Wenn man z.B. in einer Prozedur, die aber keine Schleife besitzt, ein exit verwendet mit dem Wunsch, dann aus der Schleife, in der die Prozedur aufgerufen wird, zu springen, so ist dieses "exit" nicht erlaubt.

Beispiel zur exit_Anweisung. Die Schleifen tragen hier die Namen Schleife1, Schleife2 und Schleife3. Falls beim Erreichen der exit-Anweisung Y=Z gilt, wird unmittelbar mit der Anweisung X := 0; fortgefahren .
Würde exit when Y=Z; statt exit Schleife1 when Y=Z; stehen, so würde nur Schleife3 im Falle, dass Y gleich Z ist, beendet, aber die for-Schleife würde hierdurch nicht verlassen.

```
Schleife1:
while X>Y loop
        Schleife2: for K in .. loop
                Schleife3: while Z > 0 loop
                        ...
                        exit Schleife1 when Y=Z;
                        ...
                end loop;  ...
        end loop; ...
end loop;
X := 0;   ...
```


1.11.5 Block

Ein Block ist eine Programmeinheit, die in der Regel aus einem Deklarationsteil und einem darauf folgenden Anweisungsteil besteht. Wie eine Schleife darf auch ein Block einen Namen erhalten, der durch Doppelpunkt abgetrennt vor den Block geschrieben wird.

Die vollständige Regel für einen Block in Ada lautet:

```
<Block> ::=  [<Blockname> :] [declare <Deklarationsteil>]
             begin <volle_Anweisungsfolge>
             end [<Blockname>] ;
```

Sofern ein Blockname am Ende eingefügt wird, muss dies genau der Name sein, der dem Block zu Beginn (durch Doppelpunkt abgetrennt) gegeben wurde.

Der Block ist eine zentrale Grundstruktur in Ada (und vielen anderen Programmiersprachen). Bedeutung:

Wird der Anfang eines Blockes (declare) erreicht, so werden der Reihe nach alle Deklarationen bearbeitet, die zugehörigen Speicherplätze angelegt und die Initialisierungen ausgeführt. Anschließend wird der Anweisungsteil des Blocks abgearbeitet. Sobald das end am Ende des Blocks erreicht wird, wird die Situation, die unmittelbar vor Betreten des Blockes gegeben war, wieder hergestellt. Betritt man diesen Block später erneut, dann sind keine früheren Daten mehr zugänglich.

Ein Block wird also bei jedem Erreichen neu aufgemacht und an seinem Ende wieder aufgelöst. Alles, was in seinem Inneren deklariert wurde und geschieht, ist nach außen nicht sichtbar. Ein Block bildet also eine in sich geschlossene Programmeinheit.

Durch Blöcke wird die Zuordnung zwischen einem Bezeichner und dem Objekt, das er bezeichnet, eindeutig hergestellt.

Jeder **Bezeichner** muss eine Bedeutung besitzen. Diese Bedeutung erhält er in Ada wie folgt:

Entweder ist ein Bezeichner vordefiniert (Schlüsselwort, Standarddatentypen und ihre Darstellungen oder durch with und use von außen übernommen) und somit im gesamten Programm bekannt

oder er wird in einem Deklarationsteil definiert und existiert dann genau in dem Programmstück von seiner definierenden Stelle bis zum end des zugehörigen Blockes

oder er wird bei seinem ersten Auftreten implizit deklariert (dies betrifft hier nur Namen von Blöcken und Schleifen, Marken und Laufvariablen) und existiert daher genau im Anweisungsteil des Blockes, in dem er steht, bzw. im Fall von Laufvariablen genau in der zugehörigen Schleife.

Alle Bezeichner und die durch sie bezeichneten Objekte unterliegen der Blockstruktur. Sie erhalten hierdurch eine Lebensdauer und einen Sichtbarkeitsbereich.

Die Lebensdauer eines Objekts, das in einem Block deklariert wurde, ist der Teil des Programms von der deklarierenden Stelle bis zum zugehörigen end des Blockes (statische Definition). Man definiert auch: Die Lebensdauer eines Objekts ist die Zeitspanne ab dem Zeitpunkt, zu dem die Deklaration des Objekts erreicht wird, bis zum end des zugehörigen Blockes (dynamische Definition); dies kann mehrfach geschehen, da man einen Block ja auch mehrfach betreten und verlassen kann, z.B. wenn der Block im Inneren einer Schleife steht. Ist das Objekt eine Variable, so besitzt es während dieser Zeit einen eigenen Speicherplatz, in dem seine Werte abgelegt sind.

Ein Objekt kann nur verwendet werden, wenn es *sichtbar* ist. Daher spielt der Sichtbarkeitsbereich in allen Programmiersprachen eine zentrale Rolle.

Der Sichtbarkeitsbereich eines Objekts ist der Teil der Lebensdauer, in der der Bezeichner des Objekts nicht umdeklariert wurde. Wird der Bezeichner nämlich in einem untergeordneten Block umdefiniert, so ist in diesem Block nur das neue Objekt, nicht aber das zuvor deklarierte alte Objekt durch den Bezeichner ansprechbar; man sagt, das alte Objekt ist nicht mehr sichtbar. In den meisten Fällen kann man dann auch nicht mehr auf das alte Objekt zugreifen, sondern muss warten, bis der untergeordnete Block verlassen wurde. (Tricks, um dies doch zu ermöglichen: siehe später.)

Wenn an einer Stelle s des Programms ein Bezeichner "X" steht, *welches Objekt ist hiermit gemeint?*

Ada geht wie folgt vor:

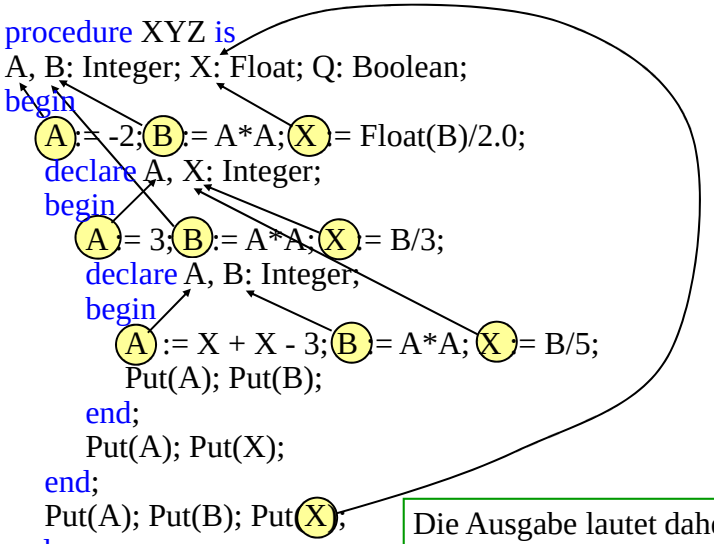
Wenn der innerste Block B, der die Stelle s umfasst, eine Deklaration für den Bezeichner X besitzt, so ist das Objekt, das durch diese Deklaration definiert wurde, gemeint. Gibt es keine solche Deklaration, so suche in dem kleinsten Block B' weiter, der B umfasst; falls auch hier keine Deklaration für X vorliegt, dann suche in dem kleinsten Block, der B' umfasst, weiter usw. (Gibt es vor einem Block B" with- und use-Klauseln, so wird nach dem Deklarationsteil von B" zunächst in diesen gesucht, bevor zum nächst höheren Block gegangen wird.) Aus diesem Zuordnungsalgorithmus ergibt sich auch, dass ein Bezeichner in jedem Block nur einmal neu deklariert werden darf (siehe jedoch "overloading"). Nun ein Beispiel.

Beispiel: Schema einer Blockstruktur:

```
procedure XYZ is
A, B: Integer; X: Float; Q: Boolean;
begin
    A := -2; B := A*A; X := Float(B)/2.0;
    declare A, X: Integer;
    begin
        A := 3; B := A*A; X := B/3;
        declare A, B: Integer;
        begin
            A  := X + X - 3; B := A*A; X := B/5;
            Put(A); Put(B);
        end;
        Put(A); Put(X);
    end;
    Put(A); Put(B); Put(X);
end;
```

Welche sieben Zahlen werden ausgegeben?

Zuordnung einiger Bezeichner zu den Objekten:



Die Ausgabe lautet daher:
3 9 3 1 -2 9 2.00000E+00

Man sieht hieran, dass eine Prozedur (und eine Funktion) ebenfalls wie ein Block aufgebaut ist, nämlich als "Prozedurkopf plus Block" (wegen des Wortes is darf declare fehlen):
procedure <Name> is <Block_ohne_das_Wort_declare_am_Anfang>

Auch die *Zählschleife* ist eigentlich ein Block. Wenn wir schreiben for K in 1..N loop A; end loop;
dann meinen wir hiermit die bedingte Anweisung mit Block:

```
if N >= 1 then
    declare K: Natural range 1..N := 1;
    begin
        loop A;     -- in A darf K aber nicht verändert werden
            if K < N then K := K+1; else exit; end if;
        end loop;
    end;
end if;
```

Wegen der Blockstruktur ist die Laufvariable daher nie identisch mit einer Variablen gleichen Namens.

Beispiel: Diese Zuordnungen gelten für alle Objekte. Wir demonstrieren dies auf der nächsten Folie anhand eines Operators "*", der außerhalb eines inneren Blockes eine andere Bedeutung hat als innerhalb, obwohl die verwendeten aktuellen Parameter die gleichen sind.
Hierzu betrachten wir erneut das Skalarprodukt aus 1.9.2 und beschreiben es durch den Operator "*":

```
function "*" (X, Y: Vektor) return Float is
        SP: Float := 0.0 ;
begin  for J in Y'Range loop  SP := SP + X(J) * Y(J) ;  end loop ;
        return SP ;
end "*";
```

In einem inneren Block definieren wir "*" in eine Summenbildung um. (Zu "Operator": siehe am Ende von 1.7)
Die drei Blöcke des Programms sind umrandet. Ihre Deklarationsteile sind farbig unterlegt.

```
procedure Irgendwas is
N: Positive;                                                    äußerer Block
begin Get (N);
   declare  type Vektor is array (1..N) of Float;               mittlerer Block
   Z: Float; A, B: Vektor;
   function "*" (X, Y: Vektor) return Float is
   SP: Float := 0.0 ;
   begin   for J in Y'Range loop  SP := SP + X(J) * Y(J);  end loop ;
           return SP ;
   end "*";
   begin
      for I in 1..N loop Get(A(I)); end loop;
      for I in 1..N loop Get(B(I)); end loop;
      Z := A*B;  Put(Z,6,6,0);
         declare                                                inerer Block
         function "*" (X, Y: Vektor) return Float is
         S: Float := 0.0 ;
         begin  for J in Y'Range loop  S := S + X(J) + Y(J);  end loop ;
                return S ;
         end "*";                                  Im inneren Block wird der
         begin                                     Operator "*" des mittleren
               Z := A*B;  Put(Z,6,6,0);            Blockes umdefiniert.
         end;
   end;
end;
```


An diesem Beispiel sieht man zugleich, wie man *dynamische Felder* behandelt: Sobald man die Länge des Feldes kennt, legt man einen neuen Block an, in dem das Feld deklariert wird.

Hinweis zum Zugriff auf nicht sichtbare Objekte: In Ada gibt es die Möglichkeit, auf ein altes Objekte, dessen Bezeichner umdefiniert wurde, zugreifen zu können, sofern der Block, in dem das alte Objekt deklariert wurde, einen Block-Namen besitzt: Man verwende dann einfach

<Block_Name>.<Bezeichner>

Beispiel: Maximale Häufigkeit

Gegeben ist eine Folge $a_1, a_2, ..., a_n$ von n ganzen Zahlen ($n \geq 1$). Hierunter können mehrere Zahlen mehrfach vorkommen.

Gesucht ist eine Zahl, die am häufigsten in der Folge vorkommt, sowie ihre Häufigkeit.

Hinweis: Wir verwenden hier eine Folge von Zahlen; das Problem und seine Lösung gelten selbstverständlich auch für jede Folge von anderen Objekten anstelle von Zahlen.

Einfache Lösung: Prüfe für die erste Zahl, wie oft sie vorkommt, prüfe dies für die zweite Zahl, die dritte Zahl usw. Teste jedes Mal, ob die soeben ermittelte Häufigkeit das bisherige Maximum darstellt und notiere gegebenenfalls die neue Zahl und ihre Häufigkeit.

Zur Illustration: 145 Zahlen. Welche Zahl kommt am häufigsten vor?

2301, 4892, 8197, 7823, 6541, 2639, 7891, 6883, 9211, 6738, 3371, 10892, 4394, 13823, 11741, 2663, 4852, 3197, 7623, 7841, 6383, 10512, 6938, 4092, 8144, 7823, 6741, 2639, 7391, 6884, 9291, 6735, 5171, 10892, 4994, 13623, 12742, 2662, 4432, 3857, 5623, 10395, 2394, 1823, 1751, 2263, 4152, 3647, 7635, 7741, 6383, 1022, 6938, 4992, 8744, 4823, 6641, 7739, 5191, 6294, 4971, 7035, 6631, 11542, 4794, 1373, 15542, 2362, 4412, 3707, 5323, 5371, 4892, 4294, 1373, 11940, 2664, 4252, 3737, 7913, 7221, 6373, 11512, 6928, 4492, 2144, 7433, 6641, 12799, 7341, 6284, 9201, 4735, 5441, 10852, 4984, 12223, 11741, 2632, 2432, 3657, 5629, 10355, 4394, 1823, 1751, 7263, 4452, 6647, 8645, 7641, 6383, 1322, 3938, 4022, 8441, 4323, 6941, 7832, 5121, 6354, 4931, 7235, 6431, 9542, 1794, 3273, 4542, 2662, 4812, 2707, 8323, 6484, 9251, 3795, 5071, 6362, 4812, 2747, 5422, 5371, 1592, 4294, 2723, 6242.

Programmiersprachliche Umsetzung:

Variablen: Man benötigt eine Variable N für die Zahl n und n Variablen A(1), ..., A(n) für die Folge der Zahlen. Sodann kann man die aktuell zu prüfende Zahl und ihre zu ermittelnde Häufigkeit in zwei Variablen Aktuell und Häufig notieren. Weiterhin braucht man zwei Variablen MaxZahl und MaxHäufig, in denen man die bisher am häufigsten vorkommende Zahl und deren Häufigkeit ablegt.

Vorbereiten der Variablen: Als erstes muss n eingelesen werden. Danach muss man in einem neuen Block die Variablen A(1), ..., A(n) deklarieren und ihre Werte einlesen. Dann sind die restlichen Variablen anzulegen und eventuell zu initialisieren (= mit Anfangswerten zu belegen). Dann kann der Lösungsalgorithmus beginnen. Wir realisieren dieses Vorgehen in Ada mit Hilfe von drei "Blöcken".

Datenbereiche und Blockstruktur:

```
procedure MaxHäufigkeit is
N: Positive;
begin
   Get(N);
   declare  A: array (1..N) of Integer;
   begin
      for i in 1..N loop Get(A(i)); end loop;
      declare Aktuell, MaxZahl: Integer;
         Häufig, MaxHäufig: Natural;
      begin
         MaxZahl := A(1); MaxHäufig := 1;
         -- Hier folgen die Anweisungen des Lösungsalgorithmus
      end;
   end;
end;
```


Datenbereiche und Blockstruktur:

```
procedure MaxHäufigkeit is
N: Positive;
begin
   Get(N);
   declare  A: array (1..N) of Integer;
   begin
      for i in 1..N loop Get(A(i)); end loop;
      declare Aktuell, MaxZahl: Integer;
         Häufig, MaxHäufig: Natural;
      begin
         MaxZahl := A(1); MaxHäufig := 1;
         -- Hier folgen die Anweisungen des Lösungsalgorithmus
      end;
   end;
end;
```


Datenbereiche und Blockstruktur:

3 ineinander geschachtelte Programmteile: "Blöcke"

```
procedure
N: Positive;
begin

   declare  A: array (1..N) of Integer;
   begin

      declare Aktuell, MaxZahl: Integer;
         Häufig, MaxHäufig: Natural;
      begin



      end;
   end;
end;
```

Kontrollstruktur zur Lösung:

```
-- Hier stehen Deklarationen und Einleseoperationen, s.o.
    begin                                   -- Initialisiere zunächst
      MaxZahl := A(1); MaxHäufig := 1;
      for i in 1..N loop                    -- Prüfe für jede Zahl
        Aktuell := A(i); Häufig := 0;       -- Initialisiere
        for j in 1..N loop                  -- Vergleiche mit jeder Zahl
          if A(j) = Aktuell then Häufig := Häufig + 1; end if;
        end loop;
        if Häufig > MaxHäufig then          -- Neues Maximum gefunden?
          MaxZahl := Aktuell; MaxHäufig := Häufig; end if;
      end loop;
      New_Line; Put(MaxZahl); Put(MaxHäufig);
    end;
  end;
end;
```


Gesamtprogramm (getestet, liefert für obige 145 Zahlen das Ergebnis: 6383 und 3).

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Text_Io; use Text_Io;
procedure MaxHäufigkeit is
N: Positive;
begin
   Get (N);  New_Line;
   declare  A: array (1..N) of Integer;
   begin
      for i in 1..N loop Get(A(i)); end loop;
      declare Aktuell, MaxZahl: Integer;
          Häufig, MaxHäufig: Natural;
      begin
          MaxZahl := A(1); MaxHäufig := 1;
          for i in 1..N loop
              Aktuell := A(i); Häufig := 0;
              for j in 1..N loop
                  if A(j) = Aktuell then Häufig := Häufig + 1; end if;
              end loop;
              if Häufig > MaxHäufig then
                  MaxZahl := Aktuell; MaxHäufig := Häufig; end if;
          end loop;
          New_Line; Put(MaxZahl); Put(MaxHäufig);
      end;
   end;
end;
```


Gesamtprogramm mit Struktur:

```
procedure MaxHäufigkeit is
N: Positive;
begin
   Get(N);  New_Line;                       -- Länge der Folge
   declare  A: array (1..N) of Integer;     -- zu untersuchende Folge
   begin
      for i in 1..N loop Get(A(i)); end loop;   -- Initialisieren durch Einlesen
      declare Aktuell, MaxZahl: Integer;        -- gesucht werden:
          Häufig, MaxHäufig: Natural;           --    MaxZahl, MaxHäufig
      begin                                     -- Initialisiere zunächst
          MaxZahl := A(1); MaxHäufig := 1;
          for i in 1..N loop                    -- Prüfe für jede Zahl
              Aktuell := A(i); Häufig := 0;     -- Initialisiere
              for j in 1..N loop                -- Vergleiche mit jeder Zahl
                  if A(j) = Aktuell then Häufig := Häufig + 1; end if;
              end loop;
              if Häufig > MaxHäufig then        -- Neues Maximum?
                  MaxZahl := Aktuell; MaxHäufig := Häufig; end if;
          end loop;
          New_Line; Put(MaxZahl); Put(MaxHäufig);
      end;
   end;
end;
```

■

Definition:

Ein Bezeichner (und speziell eine Variable) heißt lokal in dem Block, in dem er (explizit oder implizit) deklariert ist.

In allen echten Unterblöcken heißt der Bezeichner global.

Im Beispiel auf der vorigen Folie sind:

N lokal im äußeren Block und global für die beiden inneren Blöcke,

A lokal im mitteleren Block und global im inneren Block,

Aktuell lokal im inneren Block.

1.11.6 Sprunganweisung

Mit dem Sprung goto <Marke> kann man direkt mit der Anweisung im Programm, die durch diese Marke (vgl. 1.11.1) markiert ist, fortfahren. Es sind aber nicht beliebige Sprünge im Programm erlaubt.

Beispiel:

```
while X>Y loop
      for K in .. loop
            ...
            if Y=Z then  goto weiter;  end if;
            ...
      end loop;
      ...
      <<weiter>>  null;        -- Dies ist zugleich ein deklarierendes
      ...                      -- Auftreten für den Bezeichner "weiter".
end loop;
```


Der Sprung "goto" darf nicht beliebig verwendet werden:

Man darf nicht von außen in das Innere einer strukturierten Anweisung hineinspringen (insbesondere nicht in parallel liegende oder untergeordnete Prozeduren, Blöcke, Schleifen, then- oder else-Zweige oder case-Alternativen), sondern nur an deren Anfang.

Man darf nicht aus einem Block oder einer anderen Programmeinheit hinaus springen. Am Ende einer solchen Einheit müssen nämlich dort deklarierte Variablen wieder entfernt werden, was bei einem Überspringen nicht geschehen würde.

Hinweise:

Man kann jeden Algorithmus mit der einseitigen Fallunterscheidung und Sprüngen simulieren. Zum Beispiel lässt sich jede while-Schleife mit dem Schleifenrumpf A

```
while β loop A; end loop;
```

wie folgt darstellen:

```
<<Schleife>>  if β then A; goto Schleife; end if;
```

Sprünge sind in maschinennahen Sprachen eine wichtige Anweisung. Sie sind aber für Menschen schwer verständlich und führen daher zu kaum auffindbaren Fehlern. Sie sollten daher nicht benutzt werden. Sie führen zu sog. "Spaghetti-Code".

1.12 Records (Verbundtypen)

1.12.1 Records für kartesische Produkte

Zusammenfassung von n verschiedenen Mengen zu einem n-Tupel: Gegeben seien die Mengen M_1, M_2, ..., M_n, die zu den Datentypen T_1, T_2, ..., T_n gehören. Den Datentyp T, dessen Wertemenge die Menge $M = M_1 \times M_2 \times ... \times M_n$ ist, erhält man durch:

```
type T is record  S1: T1; S2: T2; ...; Sn: Tn;  end record;
```

S_1, S_2, ..., S_n sind Bezeichner, die so genannten "Selektoren", mit deren Hilfe man auf die einzelnen Komponenten des Datentyps T zugreifen kann. Man "selektiert" die i-te Komponente, indem man S_i durch einen Punkt getrennt hinter den Namen der Variable vom Typ T hängt ("Punkt-Notation", *dot-notation*).

Beispiele (Einzelheiten werden im Folgenden erläutert)

```
type Monatsname is (Januar, Februar, März, April, Mai, Juni,
        Juli, August, September, Oktober, November, Dezember);

type Datum is record
        Jahr: Integer;
        Monat: Monatsname;
        Tag: Integer range 1..31;
end record;

Start_Fussball_Weltmeisterschaft: Datum := (2010,Juni,11);
Heute, Ende, Nikolaus, Neujahr: Datum;      ...

Heute.Jahr := 2008; Heute.Monat := November; Heute.Tag := 13;

if Heute.Jahr = 2008 then Ende := Datum'(2009,Juli,23);
else Ende := Heute; end if;

Nikolaus := (Tag => 6, Jahr => 2008, Monat => Dezember);
Neujahr := Datum'(2009, Monat => Januar, Tag => 1);
```


Spezialfall: Leerer Verbund.

Oft ist es sinnvoll, keine Komponenten anzugeben, also einen "leeren Verbund" zu deklarieren. Dies kann in der Form

```
type Leerer_Verbund is record null; end record;
```

geschehen.
In Ada darf man dies abkürzen durch

```
type Leerer_Verbund is null record;
```


In diesem Beispiel wurden bereits Verbundaggregate verwendet. Diese haben die Form (<Auflistung>) entsprechend einem der folgenden drei Schemata:

(Liste der Werte) Hier werden die Werte genau in der Reihenfolge der Komponenten des Verbundtyps aufgelistet

(Liste der Werte mit Angabe der Selektoren)
Hier werden die Werte in der Form
<Selektor> => <Wert> aufgelistet

(Liste der Werte, danach Liste mit Angabe der Selektoren)
Hier werden die ersten Werte in der Reihenfolge der Komponenten des Verbundtyps zugeordnet, die restlichen Werte gemäß den noch fehlenden Selektoren.

Die Angaben müssen vollständig sein; es dürfen also keine Komponenten frei gelassen werden. Am Ende darf man "others" verwenden, sofern nur noch Komponenten des gleichen Typs folgen.

Manchmal ist es notwendig, den Datentyp des Aggregats anzugeben. Man spricht von "Typ-Qualifizierung". Hier stellt man den zugehörigen Datentyp des Verbunds voran und trennt das Aggregat durch ein Apostroph ab, wie dies oben im Beispiel

```
Neujahr := Datum'(2009, Monat => Januar, Tag => 1);
```

demonstriert wurde. Hier wäre die Qualifizierung nicht nötig gewesen, weil Neujahr von Typ "Datum" deklariert war.

Diese "Typ-Qualifizierung" gilt auch für andere (benannte) Datenstrukturen (vor allem Felder) und erfolgt in Ada nach diesem Schema. Wir haben sie bereits bei den Aufzählungstypen verwendet (siehe zum Beispiel Körperteil'(bein) und Stuhlteil'(bein) unter "einstellige Operationen" in 1.8.1).

Beispiele

Records spielen im täglichen Leben eine zentrale Rolle, weil sie die programmiersprachliche Beschreibung von Formularen sind. *Wer ein Formular ausfüllt, füllt einen Verbundtyp mit Werten.*

```
type Hotelformular is record
        Ankunftstag, Abreisetag: Datum;
        Name: array (1..30) of Character;
        Geburtsjahr: 1880..2008;
        Wohnort: array (1..30) of Character;
        PLZ: array (1..5) of Character range '0'..'9';
        Strasse: array (1..30) of Character;
        Hausnummer: Positive;
        Allein: Boolean;
        Raucherzimmer: Boolean;
end record;
```


Verbundtypen kann man bei ihrer Typdeklaration mit Vorbesetzungen ("default"-Werte) versehen, in Ada Verbund-Initialisierungen genannt. Alle Variablen, die von diesem Verbundtyp sind, werden mit den angegebenen Werten (implizit) initialisiert. Man kann bei der Deklaration aber wie gewohnt auch eine andere Initialisierung festsetzen.

```
type Rational_Init is record
        Zähler: Integer := 0;
        Nenner: Positive := 1;
end record;
P: Rational_Init; Q: Rational_Init := (Nenner=>3, Zähler=>5);
```

Hierbei ist P mit (0,1) initialisiert, während Q explizit auf (5,3) gesetzt wurde.

1.12.2 Records für Vereinigungen (variante Records)

Gegeben seien die Mengen M_1, M_2, ..., M_n, die zu den Datentypen T_1, T_2, ..., T_n gehören. Dann kann man hieraus den Datentyp T bilden, dessen Wertemenge M die disjunkte Vereinigung dieser Mengen ist:

$$M = M_1 \cup M_2 \cup ... \cup M_n$$

Formulierung in Ada:

```
type T (Index: Integer range 1..n) is record
    case Index is
        when 1 => S1: T1;
        when 2 => S2: T2;   ...

        when n => Sn: Tn;
    end case;
end record;
```


Statt des "Index", der hier aus der Menge {1, 2, ..., n} ist, kann auch ein anderer Name und ein anderer Datentyp gewählt werden. Dieses auswählende Element heißt Diskriminator oder Diskriminante. Es muss hinter dem Typnamen am Anfang der Deklaration genau spezifiziert werden.

Die Menge ist eine disjunkte Vereinigung, weil durch die Diskriminante "Index" genau eine Menge ausgewählt wird, die Mengen also als unterschiedlich angesehen werden, auch wenn verschiedene M_i gleiche Elemente enthalten sollten.

Natürlich können im record noch weitere Komponenten enthalten sein, so dass man bei der disjunkten Vereinigung in der Programmierung von einem "varianten Anteil" ("variant part") spricht. An folgendem Beispiel wird dies klar.

Bei Studierenden wird meist nach Inländern, ausländischen EU-Bürgern und Ausländern, die nicht zur EU gehören, unterschieden.

```
type SL is (D, EU, sonst);
type Student (Herkunft: SL) is record
    Name: array (1..30) of Character;
    Matrikel_Nummer: Positive;
    case Herkunft is
        when D => Geburtsort: array (1..40) of Character;
        when EU | sonst => Land: array (1..25) of Character;
    end case;
end record;
```

Die Fallunterscheidungen sind wie bei der case-Anweisung (1.11.3) aufgebaut.

Die Diskriminante darf eine *Vorbesetzung* besitzen, z.B.:

```
type SL is (D, EU, sonst);
type Student (Herkunft: SL := D) is record
    Name: array (1..30) of Character;
    Matrikel_Nummer: Positive;
    case Herkunft is
        when D => Geburtsort: array (1..40) of Character;
        when EU | sonst => Land: array (1..25) of Character;
    end case;
end record;
```

In diesem Fall dürfen Variablen dieses Typs im Laufe der Berechnungen verschiedene variante Ausprägungen annehmen. Ist die Diskriminante nicht implizit vorbesetzt, so muss ihr Wert bei der Deklaration jeder Variablen dieses Typs angegeben werden und kann dann nicht mehr geändert werden.

Hinweise:

Die (implizite) Initialisierung der Diskriminante ist zulässig.

Die einzelnen Komponenten des varianten Teils können wieder Verbünde sein, die ebenfalls variante Teile enthalten können.

Alle Selektoren, die in der gleichen Verbunddeklaration auftreten, müssen verschieden sein (auch in den varianten Teilen *und* in Unter-Verbünden!).

Die Auflistung im varianten Teil muss vollständig sein; jeder Wert des Datentyps der Diskriminante muss also unter den when-Alternativen genau einmal auftreten, wobei am Ende "others" erlaubt ist. Auswahl-alternativen der Form ...|... sind wie bei case-Anweisungen zulässig.

Die Diskriminante kann bei der Deklaration der Variablen einen Wert erhalten, der dann nicht mehr verändert werden darf. (Dies gilt auch, wenn die Diskriminante vorbesetzt ist).

Besitzt der Verbundtyp eine Vorbesetzung, so darf eine Variable dieses Typs ohne Wertangabe für die Diskriminante deklariert werden; genau solche Variablen dürfen im weiteren Verlauf verschiedene Ausprägungen erhalten, sofern mit jeder Änderung des Wertes der Diskriminante ein vollständiger Verbund zugewiesen wird (siehe Beispiel nächste Folie, die Diskriminante darf selbstverständlich nicht allein verändert werden!).

Beispiel:

Bei Fahrzeugen kann man an verschiedenen Dingen interessiert sein:

Wir nehmen an, dass für alle Fahrzeuge die Länge, die Breite und die Höhe vorliegen müssen, bei Bussen die Zahl der Sitzplätze und der Stehplätze, bei Lastkraftwagen die Größe der Ladefläche in qm und bei einem PKW die Zahl der Airbags. Bei Hand-Karren sind wir dagegen an keiner weiteren Information interessiert (hier braucht man den leeren Verbund).

Als Diskriminante wird eine Komponente mit Namen "Art" gewählt, der die Werte PKW, LKW, Bus oder Karren annehmen kann.

Dies führt zu folgendem Verbunddatentyp:

```
type mass is delta 0.001 range 0.0 .. 50.0;
type Kategorie is (PKW, LKW, Bus, Karren);
type Fahrzeug (Art: Kategorie := PKW) is record
      Länge, Breite, Höhe: mass;
      case Art is
            when Bus => record   Sitzplätze: 8 .. 60;
                                 Stehplätze: 0..80;  end record;
            when LKW => Ladefläche: Positive;
            when PKW => Airbagzahl: 0..10;
            when Karren => null record;
      end case;
end record;
A: Fahrzeug; B: Fahrzeug (Bus);
C: Fahrzeug := (Bus, 15.856, 2.95, 3.13, 55, 12);
A := (LKW, 12.4, 2.75, 3.542, 21.66);   -- zulässig
B := A;  -- verboten, da B nur die Variante "Bus" besitzen darf
B := C;  -- erlaubt, da C ebenfalls von der Variante "Bus" ist
A := B;  -- erlaubt, weil A Variable mit vorbesetzter Diskriminante
A.Art := LKW;  -- verboten, da Diskriminante nicht allein änderbar
A := (Karren, 2.9, 1.85, 1.02);  -- erlaubt, da Verbund vollständig.
```


Weiteres Beispiel: Mit varianten Records lassen sich Variablen deklarieren, die Werte von verschiedenen Datentypen besitzen können, zum Beispiel eine Variable X, die Zeichen und Zahlen enthalten kann:

```
type Zwei (Wahl: Boolean := True) is record
   case Wahl is
         when True => Zahl: Integer;
         when False => Z: Character;
   end case;
end record;

X: Ref_Zwei; L := Integer;   ...
X := (False, 'R');
X := (True, L+28);
X.Zahl := L*L;   ...
```


1.12.3 Diskriminanten für Größenangaben

Oft kennt man die aktuelle Größe von Formularen nicht. Wenn man zum Beispiel eine Reisegruppe von Stuttgart aus zusammenstellen will, so möchte man schreiben:

```
type Reisegruppe is record
     Ziel: array (1..30) of Character;
     Startort: constant array (1..9) of Character
                                  :=('S','t','u','t','t','g','a','r','t');
     Abfahrt, Rueckfahrt: Datum;
     Anzahl_Teilnehmer: Positive;
     Namen_Teilnehmer:
              array (1..Anzahl_Teilnehmer, 1..30) of Character;
end record;
```

Da aber Anzahl_Teilnehmer erst während des Programmablaufs bekannt ist, muss dieses Problem anders gelöst werden.

In Ada verwendet man für die Größenfestlegung ebenfalls eine Diskriminante, die in den Diskriminantenteil (unmittelbar hinter dem Namen des zu definierenden Verbundtyps) eingefügt wird.

```
type Reisegruppe (AT: Positive:=1; NL: Positive:=30) is record
     Ziel: array (1..30) of Character;
     Startort: constant array (1..9) of Character
                                  :=('S','t','u','t','t','g','a','r','t');
     Abfahrt, Rueckfahrt: Datum;
     Namen_Teilnehmer: array (1..AT, 1..NL) of Character;
end record;
```

Hier haben wir zugleich die Namenslänge NL zu einem Parameter gemacht, der mit 30 vorbesetzt ist. Eine konkrete Variable für 37 Personen, deren Namen jeweils durch 40 Zeichen dargestellt wird, kann dann deklariert werden durch:

```
R: Reisegruppe (37,40);
```

Weiteres Beispiel: Puffer

```
type Puffer (Laenge: Natural := 80) is record
     Position: 0..Laenge := 1;
     Inhalt: array(1..Laenge) of Character;
end record; ...
P1: Puffer;
P2: Puffer (30);
P3: Puffer (Laenge => 1000);
P4: Puffer (K+L);  ...        -- K und L seien globale ganzzahlige Variable
P2.Inhalt(P2.Position) := P3.Inhalt(P4.Position mod 20 + 1); ...
```

Den gesetzten Parameter kann man später nicht mehr ändern. Braucht man mehr Platz, so muss man ein neues größeres Element erzeugen und das alte an dessen Anfang umspeichern.

Hinweise:

Diskriminanten dürfen in Ada innerhalb der Verbunddeklaration nicht in Ausdrücken, sondern nur als einzeln auftretender Parameter verwendet werden.

```
Inhalt: array(1..Laenge+3) of Character;
```

ist in Ada also verboten, da Laenge+3 ein Ausdruck ist.

Im Diskriminantenteil können mehrere durch Semikolon getrennte Diskriminanten stehen. Erhält eine Diskriminante eine Vorbesetzung, so müssen *alle* eine Vorbesetzung bekommen.

Eine Diskriminante, die die Größe des Verbunds beeinflusst, darf nicht geändert werden (eine Diskriminante, die eine Variante festlegt, kann dagegen unter bestimmten Bedingungen verändert werden, siehe Hinweise in 1.12.2.).

Treten Verbundtypen mit Diskriminanten als Datentypen von formalen Parametern in einer Prozedurdeklaration auf, so werden die jeweiligen Diskriminanten aus den aktuellen Parametern genommen.

Man könnte sich jetzt wünschen, auch Datentypen über eine Diskriminante einfügen zu können. Dies ist jedoch in Ada verboten und muss durch generische Pakete realisiert werden, siehe später.

1.13 Weitere Sprachkonstrukte

Folgende Begriffe der Programmierung werden in den Programmierübungen und der Vorlesung noch behandelt:
- Zeiger (access-Typen, Listen, Bäume, Graphen)
- Pakete (Moduln)
- Generizität, Überladen, Polymorphie
- Ausnahmen (exceptions)
- Prozesse, Nebenläufigkeit (concurrency)
- Objektorientierung, Vererbung (tagged)

In der Programmierung muss man viele Prinzipien festlegen, um möglichst klare Software zu konstruieren und hiermit auch anpassbare und erweiterbare Programme zu bekommen. Wir betrachten zur Illustration im Folgenden ein Beispiel.

Beispiel: Gleichheit von Datentypen, speziell bei Feldern:

In Ada gibt es eine klare restriktive Regel, wann zwei Variablen A und B vom gleichen Datentyp sind: Sind die beiden Variablen von einem Untertyp, so müssen sie den gleichen Basistyp besitzen. Anderenfalls müssen sie den gleichen "benannten" Datentyp haben, d.h., sie besitzen entweder den gleichen vordefinierten Datentyp (Boolean, Character, Integer, Natural, Positive, Float usw.) oder es gibt eine Typ- oder Untertyp-Definition mit einem Namen und beide Variablen sind mit diesem Namen deklariert worden.

"Benannt" bedeutet, dass dem Datentyp ein Name per Datentypdefinition zugewiesen ist. In Ada kann man nämlich Feld-Variablen auch unbenannt deklarieren:

```
X: array (1..100) of Character;
```

Jede solche Deklaration wird stets *als neue, noch nicht vorhandene* Feld-Datentypdefinition aufgefasst!

Gleichen Datentyp haben z. B. A, B bzw. K, L bzw. X, Y:

```
type Zehnziffern is array(1..10) of Integer range 0..9;
A, B: Integer;  K, L: Float;
X, Y: Zehnziffern;
```

Dagegen haben D und E bzw. F und G bzw. H und I *nicht* den gleichen Datentyp in Ada:

```
D: Zehnziffern; E: array(1..10) of Integer range 0..9;
F, G: array(1..10) of Integer range 0..9;
H, I: Integer range 0..50;
```

denn Ada fasst H, I: 0..50 auf als die Deklarationsfolge

```
H: Integer range 0..50;
I: Integer range 0..50;
```

und diese beiden Datentypen sind verschieden, da sie keinen vom Benutzer vergebenen gemeinsamen Namen besitzen.
Vorsicht: Es gibt Ada-Compiler, die diese Datentypen trotzdem identifizieren und keinen Fehler melden!

Vorläufige Schlussbemerkung zur Ada-Einführung

Mit Kapitel 1 wurde der sog. *PASCAL-Teil von Ada* vorgestellt.

Man bezeichnet die Möglichkeiten, die diese Sprachelemente eröffnen, als Programmieren im Kleinen.

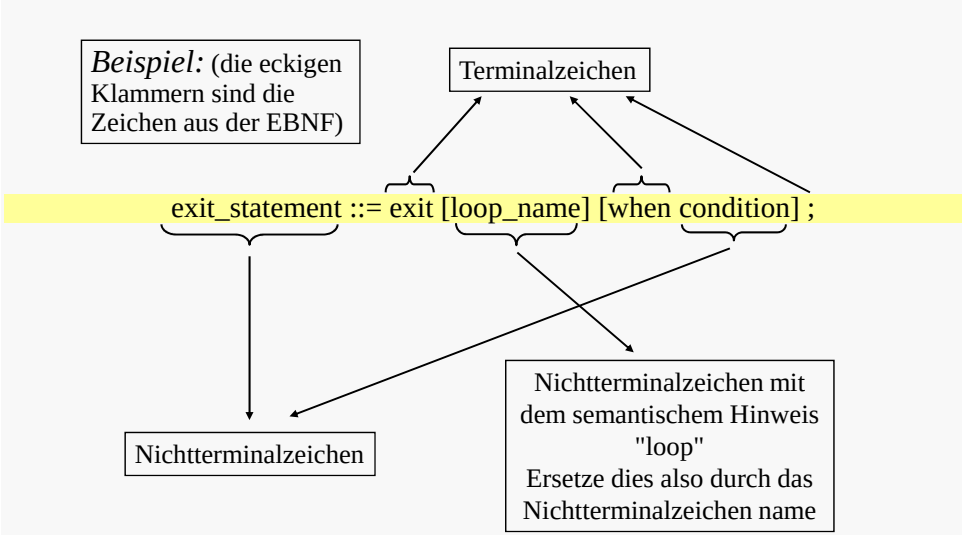
Mit weiteren Ada-Sprachelementen wie Pakete, generische Einheiten, Objektorientierung, getrennte Übersetzung und Bibliotheken beginnt das Programmieren im Großen. Wir werden hierauf und auf nebenläufige Systeme im Verlauf der Vorlesung eingehen und dort auch die jeweilige Formulierung in Ada vorstellen.

Im Netz finden Sie Ada 2005 einschl. der gesamten Ada-Syntax:
http://www.iste.uni-stuttgart.de/ps/ada-doc/rm2005/RM-TOC.html

1.14 Wie lese ich die Syntax von Ada?

Auf den folgenden beiden Folien finden Sie die Original-Syntax für Anweisungen ("statements") für Ada.
In der Praxis schreibt man die Syntax nicht so lesbar aufbereitet, wie wir es hier getan haben. Die Nichtterminal-zeichen erkennt man daran, dass sie irgendwo auf der linken Seite stehen; oft sind sie aus mehreren Wörtern zusammen-gesetzt und enthalten einen Unterstrich; dabei ist oft das erste Wort dieser Wörter nur ein semantischer Hinweis; meist sind sie in der Original-Syntax unterstrichen, und wenn man sie anklickt, springt man zu dieser zugehörigen Regel. Terminalzeichen werden optisch von den Nichttermi-nalzeichen nicht unterschieden, d.h., es wird vorausgesetzt, dass die Leser(innen) sie ohne Weiteres erkennen. Beispiel:

```
exit_statement ::= exit [loop_name] [when condition] ;
```

Zum Vergleich unsere bisherige Schreibweise:
<exit_statement> ::= exit [<*loop*_name>] [when <condition>] **;**

Beispiel: Die Syntax von <Anweisungsfolge> (= sequence_of_statements)

sequence_of_statements ::= statement {statement}

statement ::= {label} simple_statement | {label} compound_statement

simple_statement ::= null_statement | assignment_statement |
exit_statement | goto_statement | procedure_call_statement |
simple_return_statement | entry_call_statement |
requeue_statement | delay_statement | abort_statement |
raise_statement | code_statement

compound_statement ::= if_statement | case_statement | loop_statement |
block_statement | extended_return_statement | accept_statement |
select_statement

null_statement ::= null;

label ::= <<label_statement_identifier>>

statement_identifier ::= direct_name

assignment_statement ::= variable_name := expression;

if_statement ::= if condition then sequence_of_statements
{elsif condition then sequence_of_statements}
[else sequence_of_statements] end if;

condition ::= boolean_expression

case_statement ::= case expression is case_statement_alternative
{case_statement_alternative} end case;

case_statement_alternative ::=
when discrete_choice_list => sequence_of_statements

loop_statement ::= [loop_statement_identifier:] [iteration_scheme]
loop sequence_of_statements end loop [loop_identifier];

iteration_scheme ::= while condition | for loop_parameter_specification

loop_parameter_specification ::=
defining_identifier in [reverse] discrete_subtype_definition

block_statement ::= [block_statement_identifier:]
[declare declarative_part] begin
handled_sequence_of_statements end [block_identifier];

exit_statement ::= exit [loop_name] [when condition];

goto_statement ::= goto label_name;

Diese 18 Regel bilden den Teil 5 der Ada-Syntax, siehe obigen Link.

Vergleichen Sie diese Originalfassung mit den Regeln in 1.11.1. Schärfen Sie Ihren Blick und versuchen Sie anschließend, die Syntax für ganze Zahlen, für reelle Zahlen, für arithmetische Ausdrücke und für Boolesche Ausdrücke selbst zu formulieren. Schauen Sie dann in der Syntax von Ada nach und vollziehen Sie die dort gewählten Formulierungen nach, die manchmal nicht nahe liegend sind.

Zur Übung folgt auf der nächsten Folie ein Ausschnitt aus der Ada-Syntax für die Feld-Deklaration.
Machen Sie sich hieran klar, wie Felddeklarationen aufgebaut sind und dass die "normalen" Felddeklarationen alle unter diese Syntax fallen.

Teil der Syntax der Feld-Deklaration (nur array_type):

array_type_definition ::=
unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::= array (index_subtype_definition
{, index_subtype_definition}) of component_definition

index_subtype_definition ::= subtype_mark range <>

subtype_mark ::= *subtype*_name

component_definition ::= subtype_indication

subtype_indication ::= subtype_mark [constraint]

constrained_array_definition ::= array (discrete_subtype_definition
{, discrete_subtype_definition}) of component_definition

discrete_subtype_definition ::= *discrete*_subtype_indication | range

range ::= simple_expression .. simple_expression

(Hinweis: Ein constraint ist meist von der Form <Ausdruck> .. <Ausdruck>.)

1.15 Übungsaufgaben

Aufgabe 0: *Multiplikation*
Beschreiben Sie die Multiplikation zweier natürlicher Zahlen.

a) Verwenden Sie eine iterierte Addition.
b) Beschreiben Sie den zeichenweise arbeitenden Algorithmus, den Sie aus der Grundschule kennen.
c) Nehmen Sie an, Sie hätten als Operation die Bildung des Quadrats einer Zahl zur Verfügung, d.h., es gibt den Operator square(x), der zu einer natürlichen Zahl x das Quadrat x^2 liefert. Wie kann man dann die Multiplikation durchführen?

Aufgabe 1: *Berechnung von div und mod*
Der Euklidische Algorithmus (siehe 1.6 und 1.7) benutzt die Operation mod, die zu zwei natürlichen Zahlen a und b den Rest liefert, der bei der ganzzahligen Division a div b von a durch b übrig bleibt. Es gilt: $a = b \cdot (a \text{ div } b) + a \bmod b$ mit $0 \le a \bmod b < b$. Geben Sie zwei Algorithmen an, die a div b und a mod b gleichzeitig berechnen, und zwar

a) einmal zeichenweise (z.B. als array dargestellt) und
b) einmal ohne die Darstellung mit Ziffern zu verwenden.

Prüfen Sie Ihre Algorithmen mit einigen Ablaufprotokollen.

Aufgabe 2: *Darstellung zu einer Basis* (vgl. 1.7.6)
Geben Sie einen Algorithmus an, der zu zwei natürlichen Zahlen x und b die Ziffern der Darstellung von x zur Basis b ausgibt. ($b \ge 2$, und die Ziffern sollten Sie in der Form (i) ausgeben, wobei i die eigentliche Ziffer ist, $0 \le i < b$.)

Aufgabe 3: Primzahlen, Teilbarkeit

Eine natürliche Zahl $p > 1$ heißt Primzahl, wenn sie nur durch 1 und sich selbst teilbar ist.
(Man beachte, dass 1 *keine* Primzahl ist.)
Schreiben Sie einen Algorithmus, der eine natürliche Zahl n einliest und prüft, ob n eine Primzahl ist. Falls n keine Primzahl ist, so soll die Zahl n und der Text " ist keine Primzahl" ausgegeben werden; anderenfalls soll mindestens eine Zerlegung $n = a \cdot b$, d.h., es sollen zwei Teiler $n > a > 1$ und $n > b > 1$ ausgegeben werden.
Zusatz: Es ist die gesamte Primzahlzerlegung der Zahl n auszugeben.

Aufgabe 4: Sieb des Eratosthenes (ca. 280-200 v. Chr.)

Es sollen *alle* Primzahlen, die kleiner oder gleich n sind, gefunden werden. Eratosthenes schlug folgendes Verfahren vor:

Gegeben sei n. Sei w der ganzzahlige Anteil der Wurzel von n, d.h., die Zahl w mit $w^2 \le n < (w+1)^2$.
Schreibe alle Zahlen von 1 bis n hin. Streiche die Zahl 1.
for z in 2..w loop
  sofern z nicht gestrichen ist, streiche alle echten Vielfachen von z
end loop;
Die nicht gestrichenen Zahlen sind alle Primzahlen von 1 bis n.

Implementieren Sie diesen Algorithmus in Ada und lassen Sie alle Primzahlen von 1 bis 1_000_000 ermitteln.
Welchen Zeit- und Platzbedarf besitzt dieser Algorithmus?

Aufgabe 5: Kleine Teufelchen

In einem Hotel gibt es einen langen Flur mit n Türen. Alle Zimmer sind besetzt und alle Hotelgäste haben ihre Türen verschlossen.
Nun kommen nacheinander n kleine Teufelchen und verändern den Zustand jeder Tür, d.h., wenn eine Tür offen ist, machen sie sie zu, wenn die Tür geschlossen ist, machen sie sie auf. Teufelchen 1 macht dies für jede Tür, Teufelchen 2 für jede zweite Tür, Teufelchen 3 für jede dritte Tür usw. und Teufelchen n nur noch für die n-te Tür.
Programmieren Sie diesen Sachverhalt mit Hilfe eines Booleschen Feldes. Welche Türen sind am Ende offen und welche geschlossen? Erklären Sie das Ergebnis.

Aufgabe 6: *Zeichenverschiebung und quadratische Codierung*
Es sei Pos: **A** → {0, 1, 2, ..., 127} die Funktion, die jedem Zeichen seine Nummer im ASCII-Code zuordnet (siehe 1.8.4).
Es sei Val: {0, 1, 2, ..., 127} → **A** die Funktion, die jeder Zahl zwischen 0 und 127 das zugehörige Zeichen des ASCII-Codes zuordnet.

a) Schreiben Sie einen Algorithmus, der zunächst eine Zahl k und danach eine Folge von Zeichen (das Ende der Folge sei durch das Zeichen '&' bestimmt) einliest und die um k Stellen im ASCII-Code verschobene Folge dieser Zeichen ausgibt (nach dem letzten Zeichen wieder von vorn).
b) Wir wollen die Zeichen durch folgende Vorschrift verschlüsseln: Wenn das Zeichen α die Nummer Pos(α) = i besitzt, dann berechnen wir $j = i^2 \bmod 128$ und ersetzen α durch das Zeichen β = Val(j).
Beispiel: α = 'U', Pos(U) = i = 85, $j = 85^2 \bmod 128 = 7225 \bmod 128 = 57$, β = Val(57) = '9'.

Schreiben Sie einen Algorithmus, der diese Verschlüsselung vornimmt, der also die Abbildung α → β = Val ($(\text{Pos}(\alpha))^2 \bmod 128$) berechnet.
Ist diese Abbildung injektiv? (vgl. Tabelle in 1.8.4)
Diskutieren Sie Vor- und Nachteile dieser Abbildung.

Aufgabe 7: *Terminierung?*
Ermitteln Sie, für welche Eingabewerte folgende Programme terminieren und welche Funktion sie realisieren.

```
procedure P1 is
k, m: Integer;
begin  Get(m);  k := 1;
       while m > k loop  k := k+2; m := m+1; end loop;
       Put (k);
end;

procedure P2 is
k, m: Integer;
begin  Get(m);  k := -m;
       while m > 0 loop  k:=k+2; m:= k; end loop;
       Put (k);
end;
```


Aufgabe 8: *Terminierung?*
Ermitteln Sie, für welche Eingabewerte folgendes Programm terminiert (zu "Pos" siehe 1.8.1 Aufzählungstyp und 1.8.4).

```
procedure unbekannt is
k: Integer; X: Character; DB: array (0..3) of Character;
begin
     for R in 0..3 loop DB(R) := 'Z'; end loop;
     Get (X);
     while X /= 'Z' loop
          k := Character'Pos(X);
          while DB(k) /= 'Z' loop
                k := (k+1) mod 4;  end loop;
          DB (k) := X;
     end loop;
     for R in 0..3 loop Put(DB(R)); end loop;
end;
```

Aufgabe 9: *Erzeugte Sprachen*

Es seien V = {S, A} und Σ = {0, 1}. Welche Sprachen L_i werden von folgenden BNFs (V, Σ, P_i, S) erzeugt mit

P_1 = { S ::= 0 | 1 }
P_2 = { S ::= 1 | S | 0, A ::= 0 }
P_3 = { S ::= 0S | 00 }
P_4 = { S ::= S1 | 00 | 1, A ::= 1 | AS }
P_5 = { S ::= 0A0 | 1, A ::= 0S0 }
P_6 = { S ::= 0AS | 0, A ::= 1A1 | 1 }
P_7 = { S ::= AA, A ::= 0A0 | 1 }
P_8 = { S ::= AS | A, A ::= ε | 0S1 }
P_9 = { S ::= SAS | A, A ::= 10 | 0S1A }

Aufgabe 10: *BNFs konstruieren*

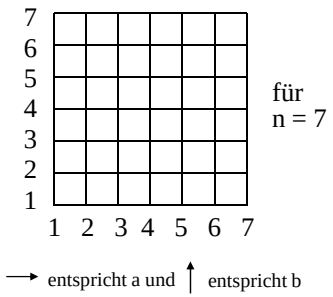
Konstruieren Sie je eine BNF (V_i, Σ, P_i, S) mit Σ = {0, 1}, die folgende Sprachen L_i erzeugen (hier ist w^R das gespiegelte Wort w, d.h., wenn $w = a_1a_2...a_n$ ist, dann ist $w^R = a_n..a_2a_1$):

L_1 = Menge aller Folgen von 0 und 1 ohne das leere Wort
= Σ* -{ε}
L_2 = { w∈ Σ* | w hat eine gerade Länge }
L_3 = { 0^n10^n | n > 0 }
L_4 = { $(0^n10^n)^m$ | n > 0, m > 0 }
L_5 = { ww^R | w∈ Σ* }
L_6 = { w^2 | w∈ Σ* }
L_7 = { $0^n10^n10^n$ | n > 0 }
L_8 = { w∈ A* | w besitzt gleich viele Zeichen 0 und 1 }
L_9 = { 0^n | Es gibt ein k≥0 mit $n = 2^k$ }

Aufgabe 11: *Charakterisierung von "diagonalen Wegen"*

Es sei ein n×n-Gitter gegeben:

Es sollen genau die Wege beschrieben werden, die vom Punkt (1,1) unten links zum Punkt (n,n) oben rechts führen. Hierfür bedeute das Zeichen a "gehe zum nächsten Punkt rechts" und b bedeute "gehe zum nächsten Punkt nach oben". Das Wort abab..ab = $(ab)^n$ beschreibt dann einen solchen "diagonalen Weg" ebenso wie das Wort a^nb^n.



a) Beschreiben Sie die Menge aller Wörter, die einen Weg vom Punkt (1,1) zum Punkt (n,n) beschreiben.
b) Geben Sie eine BNF an, die für alle n alle diese Wörter erzeugt.

Aufgabe 12: TSP = Travelling Sales Person Problem

Eine Umordnung oder Permutation $\pi = (\pi_1, \pi_2, ..., \pi_n)$ der ersten n Zahlen ist irgendeine Reihenfolge der Zahlen von 1 bis n.
Gegeben seien n Städte, die mit 1 bis n durchnummeriert sind. Es sei d_{ij} die Entfernung ("Distanz") von der Stadt i zur Stadt j.
Für eine Permutation $\pi = (\pi_1, \pi_2, ..., \pi_n)$ der ersten n Zahlen sei

$$d(\pi) = d_{\pi_1\pi_2} + d_{\pi_2\pi_3} + d_{\pi_3\pi_4} + ... + d_{\pi_{n-1}\pi_n} + d_{\pi_n\pi_1}$$

die Gesamtentfernung, wenn man die n Städte in der Reihenfolge der Permutation π durchreist.
Die Aufgabe lautet nun, zu gegeben n und d_{ij} (für $1 \le i \le n, 1 \le j \le n$) eine minimale Rundreise zu finden, d.h., eine Permutation π, so dass für alle anderen Permutationen π' gilt: $d(\pi) \le d(\pi')$.
Programmieren Sie dies in Ada. Man kann das Problem leicht lösen, indem man alle n! Permutationen durchprobiert (siehe Abschnitt 6.5.6). Programmieren Sie diese Lösung.

Aufgabe 13: Nächste Permutation (vgl. später 6.5.6)

Will man alle Anordnungen einer Folge aus n Elementen durchprobieren, so muss man die n! Anordnungen wiederholungsfrei erzeugen. Hierfür dient die Prozedur "Naechste_Permutation", die zu einer Permutation P die nächste Permutation berechnet.

Diese Prozedur soll nun von Ihnen programmiert werden. Meist erzeugt man die n! Permutationen in der Reihenfolge, in der sie nacheinander der Größe nach angeordnet wären, wenn man die Permutationen als Zahlen zur Basis n+1 aufgeschrieben hätte.

Z.B. für n = 3 und n = 4 senkrecht aufgeschrieben:

1 2 3	1 2 3 4	2 3 1 4	3 4 1 2
1 3 2	1 2 4 3	2 3 4 1	3 4 2 1
2 1 3	1 3 2 4	2 4 1 3	4 1 2 3
2 3 1	1 3 4 2	2 4 3 1	4 1 3 2
3 1 2	1 4 2 3	3 1 2 4	4 2 1 3
3 2 1	1 4 3 2	3 1 4 2	4 2 3 1
	2 1 3 4	3 2 1 4	4 3 1 2
	2 1 4 3	3 2 4 1	4 3 2 1

Wie erhält man aus einer Permutation $(\pi_1, \pi_2, ..., \pi_n)$ die nächst größere? Betrachten Sie hierzu folgendes Beispiel (n = 9):

1 7 4 6 9 8 5 3 2 $\xrightarrow{\text{nächste Permutation}}$ 1 7 4 8 2 3 5 6 9

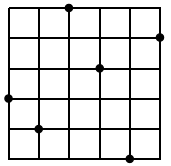
Man geht in drei Schritten vor (die Zahlen stehen im Feld P):
(1) Suche das größte i mit P(i) < P(i+1); speziell gilt: i < n.
(2) Suche in P(i+1) bis P(n) das kleinste Element z mit z > P(i). Dieses möge P(j) sein. Vertausche P(i) und P(j).
(3) Sortiere den Bereich P(i+1) bis P(n).

In obigem Beispiel (n = 9):
(1) i = 4 und P(4) = 6 < P(5) = 9.
(2) z = P(6) = 8, also j = 6. Vertauschen ergibt: 1 7 4 8 9 6 5 3 2.
(3) Sortieren von P(i+1..n) = 9 8 5 3 2 liefert: 1 7 4 8 2 3 5 6 9.

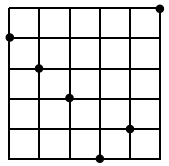
Damit man das Ende (also die Permutation 9 8 7 6 5 4 3 2 1) erkennt, sollte man das Feld P für die Permutationen ab dem Index 0 beginnen lassen und P(0) := 0 setzen. *Details selbst überlegen!*

Aufgabe 14: Die unordentlichste Permutation

Eine Permutation π der ersten n Zahlen kann man als Diagramm in der zweidimensionalen Ebene veranschaulichen, indem man dort die Punkte (i, π_i) für i = 1, 2, ..., n einträgt. Als Beispiele für n=6 betrachte die Permutationen 3 2 6 4 1 5 und 5 4 3 1 2 6:



3 2 6 4 1 5



5 4 3 1 2 6

Die linke Permutation sieht "unordentlicher" als die rechte aus. Wir messen dies durch die Funktion $C(\pi)$ = die Summe über alle Steigungen aller Strecken zwischen den n Punkten, allerdings müssen wir die Absolutbeträge summieren.

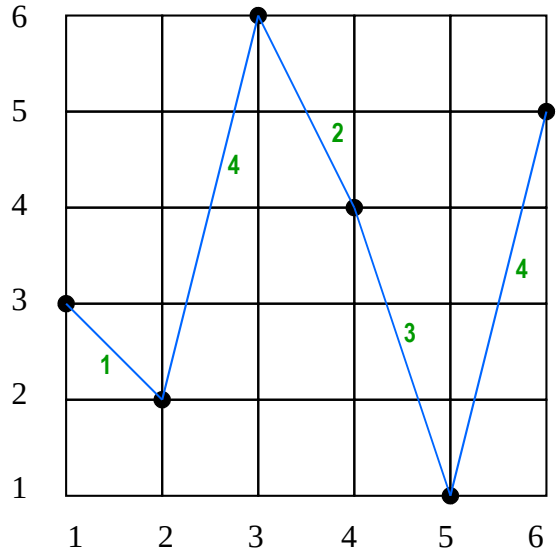
Man verbinde nun alle n Punkte paarweise miteinander durch Strecken. Für eine Permutation $\pi = (\pi_1, \pi_2, ..., \pi_n)$ der ersten n Zahlen sei $C(\pi)$ die Summe der Absolutbeträge aller Steigungen dieser Strecken (mal 2, da von jedem Punkt aus gerechnet wird):

$$C(\pi) = \sum_{i \neq j} \frac{|\pi_j - \pi_i|}{|j - i|}$$

$$= 2 \cdot \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{|\pi_j - \pi_i|}{j - i}$$

Dies ist ein Maß für die "Unordentlichkeit": Je größer $C(\pi)$ ist, um so unordentlicher ist die Permutation π.
Aufgabe: Schreiben Sie ein Ada-Programm, das zu einer Zahl n die unordentlichste Permutation berechnet, also ein π, dessen Wert $C(\pi)$ maximal ist für alle Permutationen der n Zahlen.

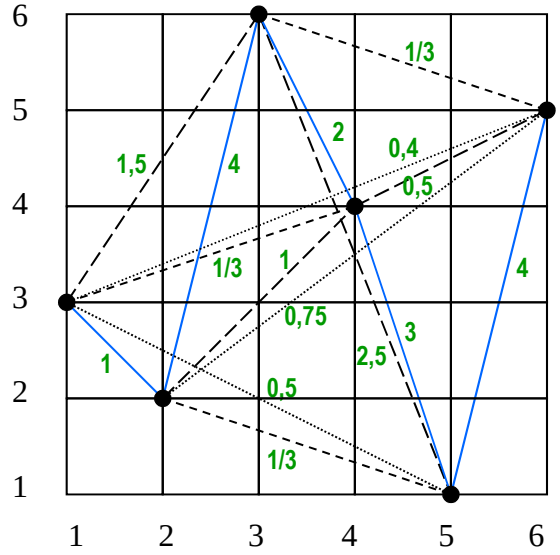
n=6. Veranschaulichung für die Permutation 3 2 6 4 1 5



Summe: 14

Hier sind nur benachbarte Punkte bisher verbunden.

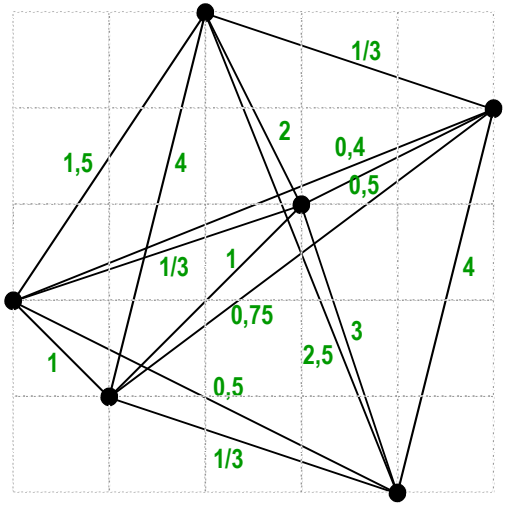
Permutation 3 2 6 4 1 5



Summe: 14
+ 5,5
+ 1
+ 1,25
+ 0,4
= 22,15

Hier sind nun alle Punkte miteinander verbunden. Jede Strecke ist aber nach Definition von C(π) zweimal zu nehmen!

Permutation 3 2 6 4 1 5 C(326415) = 44,3



Hinweis: Diese Permutation besitzt für n=6 *nicht* den größten C-Wert. Dieser lautet 47,4. Berechnen Sie die zugehörige Permutation!

Aufgabe 15: Die gerechte Erbschaft

Zwei Personen erben n Goldklumpen. Diese besitzen die Werte $g_1, g_2, g_3, ..., g_n$ (Euro oder Dollar oder sonst eine Währung). Ihr Gesamtwert beträgt $G = g_1 + g_2 + g_3 + ... + g_n$. Kann man diese Goldklumpen (ohne sie zu zerkleinern) so in zwei Hälften teilen, dass jede Person genau den Wert G/2 erbt?

Dieses Problem ist gleichwertig zum gleichmäßigen Auslasten zweier Maschinen. Gegeben seien zwei gleiche Maschinen (z.B. zwei Kopierer) und eine Menge von n Aufträgen, für deren Erledigung jede Maschine $a_1, a_2, a_3, ..., a_n$ Zeiteinheiten benötigt. Eine einzelne Maschine würde insgesamt $A = a_1 + a_2 + a_3 + ... + a_n$ Zeiteinheiten brauchen. Kann man die n Aufträge so auf die zwei Maschinen verteilen, dass alle Aufträge in A/2 Zeiteinheiten erledigt sind?

Die Aufgabe lautet also konkret:

Man schreibe einen Algorithmus, der zu einer natürlichen Zahl n und n rationalen Zahlen $g_1, g_2, g_3, ..., g_n$ (es sei $G = g_1 + ... + g_n$) feststellt, ob es eine Teilmenge $J = \{j_1, j_2, ..., j_m\} \subseteq \{1, 2, ..., n\}$ gibt mit $g_{j_1} + g_{j_2} + g_{j_3} + ... + g_{j_m} = G/2$? Im Falle, dass dies möglich ist, soll mindestens eine solche Menge J ausgegeben werden. Schätzen Sie die Laufzeit Ihres Verfahrens ab.

In der Regel löst man dieses Problem durch systematisches Durchprobieren aller möglichen Teilmengen J. Dabei baut man Erfolg versprechende Teilmengen schrittweise auf und bricht deren weiteren Ausbau immer dann ab, wenn sie sich nicht mehr zu einer Lösung erweitern lassen. (Rekursion verwenden!)

Da es 2^n Teilmengen der Menge {1, 2, ..., n} gibt, führt ein solches Vorgehen zu einer exponentiellen Laufzeit. (Wesentlich schneller arbeitende Verfahren sind bis heute nicht bekannt.)

2. Algorithmen und Sprachen

2. Algorithmen und Sprachen

Erinnerung: *Informatik ist keine Geheimwissenschaft schwer durchschaubarer Computer und ihrer konkreten Software-systeme.*

Es geht um das grundsätzliche Verständnis des Rohstoffs "Information" und um Konzepte, nach denen dieser dargestellt, aufbereitet, abgelegt, verarbeitet, implementiert, wiederver-wendet, eingesetzt, zugeschnitten oder anderweitig manipuliert werden kann. Dies erfordert eine theoretische Durchdringung. Als erstes klären wir umgangssprachlich, aber bereits recht exakt, den Begriff "Algorithmus".

Für die Syntax benötigen wir die "Formale Sprache". Ihre Beschreibung hängt eng mit Algorithmen zusammen, sodass wir die beiden Begriffe in diesem Kapitel zusammenfassen.

2.1 Darstellung von Algorithmen

Umgangssprachliche grobe Festlegung 2.1.1:
Ein Verfahren, das prinzipiell von einer mechanisch arbeitenden Maschine durchgeführt werden kann, nennen wir einen Algorithmus.

Etwas präzisere Festlegung 2.1.2:
Ein Algorithmus ist ein exakt beschriebenes Verfahren einschließlich der genauen Festlegung der Eingabe und der Ausgabe, der Zwischenspeicherung von Daten usw. Das Verfahren muss so genau ausformuliert sein, dass jede(r) den Algorithmus nachvollziehen und einem Computer übertragen kann, ohne Rücksprache mit den Verfassern zu nehmen.

Beispiel 2.1.3: Addiere 1 zu einer dezimal dargestellten Zahl.

Umgangssprachliche Formulierung: Eine Zahl sei als eine Folge von Ziffern aus der Menge {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} gegeben. Falls die letzte Ziffer nicht die 9 war, so ersetze sie durch die nächst größere Ziffer und beende das Verfahren, anderenfalls ersetze sie durch 0 und wiederhole das Verfahren für die zweitletzte Ziffer usw.

Hier wird erwartet, dass der, der das Verfahren ausführt, ein wenig mitdenkt oder ein Vorwissen besitzt. So muss man wissen, dass die Ziffern in der angegebenen Reihenfolge $0 < 1 < 2 < ... < 8 < 9$ angeordnet sind, man muss wissen, was das "usw." bedeutet, und man muss wissen, was man zu tun hat, wenn es keine zweitletzte Ziffer gibt.

Als Mensch macht man sich einen Algorithmus an konkreten Beispielen klar. So liefert der obige Algorithmus "+ 1":

2 + 1 = 3, 44 + 1 = 45, 103 + 1 = 104,
0 + 1 = 1, 19 + 1 = 20, 199 + 1 = 200.

Nicht klar ausformuliert wurden die Fälle, in denen die Folge nur aus Neunen besteht, wie 9 + 1 = 10, 99 + 1 = 100.

Weiterhin entspricht die Aussage, dass eine Zahl eine Folge von Ziffern sei, nicht der üblichen Anschauung, da man i.A. (außer im Falle der Null selbst) keine führenden Nullen zulässt. Das Verfahren liefert zwar 0068 + 1 = 0069, aber in der Regel schreibt man nicht 0068, sondern 68, da man nur eindeutige Darstellungen für Zahlen verwenden möchte.

Das Verfahren beschränkt die Addition der 1 auf natürliche Zahlen (einschl. der Null). Man kann 1 aber auch zu einer ganzen, einer rationalen, einer reellen oder einer komplexen Zahl addieren. Diese Erweiterung wird durch das angegebene Verfahren nicht erfasst, sondern muss durch einen neuen Algorithmus beschrieben werden.

Wie und wo man die Zahlen aufschreibt, bleibt offen. Wir denken sicher sogleich an Papier und Bleistift, doch kämen andere Völker möglicherweise nicht auf diese Realisierung. (Beachten Sie, dass die Dezimaldarstellung bzw. allgemein die Darstellung in einem Stellenwertsystem zu einer Basis keineswegs nahe liegend ist. Die Römer haben beispielsweise mit einem völlig anderen System gearbeitet.)

An diesem Beispiel erkennt man einige zentrale Sprachelemente, um Algorithmen zu beschreiben.

- Algorithmen bestehen aus einfachen Handlungen, sog. "elementaren Anweisungen". Im Beispiel sind dies: "ersetze Ziffer durch nächst größere Ziffer" oder "beende das Verfahren".
- Einzelne Handlungen können nacheinander ausgeführt werden. Im Beispiel: "ersetze Ziffer durch nächst größere Ziffer" und danach "beende das Verfahren".
- Die nächste Handlung kann von einer aktuellen Bedingung (Alternative oder Fallunterscheidung) abhängen. Im Beispiel: "Falls die Ziffer nicht die 9 war, dann ..." .

- Handlungen können wiederholt werden. Im Beispiel wird dies durch das "usw." beschrieben, welches besagt, man solle die Ersetzungen solange vornehmen, bis eine von 9 verschiedene Ziffer erreicht wird.
- Durch den Algorithmus werden irgendwelche Gebilde manipuliert. Deren anfängliche Darstellung ist anzugeben. In unserem Beispiel sind dies natürliche Zahlen und deren Darstellung als Ziffernfolgen.
- Die Gebilde, die das Ergebnis des Algorithmus sind, ergeben sich durch den Algorithmus selbst. Man sollte sie aber möglichst zuvor beschreiben können. In unserem Beispiel sind dies ebenfalls Dezimaldarstellungen.

Beispiel 2.1.4:
Addieren zweier dezimal dargestellter Zahlen.

Für diese Aufgabe gibt es eine einfache Formulierung eines Algorithmus: Wenn a und b zwei dezimal dargestellte Zahlen (also dargestellt als Ziffernfolgen) sind, so addiere b-mal 1 zu a.

Wie man 1 zu einer Zahl addiert, wissen wir ja bereits.

Hinweis: "addiere b-mal 1 zu a" kann missverstanden werden. Gemeint ist, dass man 1 zu a addiert, dann zum Ergebnis 1 addiert, danach zu dem neuen Ergebnis 1 addiert usw.

An diesem Beispiel erkennt man zwei weitere zentrale Sprachelemente zur Beschreibung von Algorithmen.

- Wiederhole eine Handlung b-mal. Die Anzahl der Wiederholungen ist hier also vor der ersten Ausführung bekannt und hängt nicht von einer aktuellen Bedingung ab.
- Man darf Algorithmen, die bereits anderweitig beschrieben wurden, in anderen Algorithmen verwenden. In unserem Beispiel darf man den "Algorithmus Addiere 1" für die Addition zweier Zahlen benutzen.

Weiterhin muss man sich Gedanken darüber machen, wo Zwischenergebnisse abgelegt und wie sie weiter verwendet werden.

Einige Beispiele aus dem Alltag:

- Kochrezepte.
- Bastelanleitungen.
- Ermittlung der Abiturnote aus den Leistungen der Oberstufe.
- Benutzungsalgorithmus des Übungssystems.
- Ablauf der Gesetzgebungsverfahren.
- Ermittlung kürzester Verbindungen zwischen zwei Orten.
- Feststellen von Rechtschreibfehlern in einem Text.
- In der Industrie eingesetzte Produktionsvorgänge.
- Abläufe in Verwaltungen, z.B. Genehmigung eines Bauantrags.
- Berechnung der Reisekostenerstattung durch eine Verwaltung.
- Erstellung einer Häufigkeitsstatistik aller Wörter, die Goethe in seinem Gesamtwerk verwendet hat (das Gleiche für Noten und Musiker, Farben und Maler, ...)

Festlegungen 2.1.5: (A1) bis (A9)
Diese und weitere Untersuchungen führten zu folgenden Vorgaben und Sprachelementen für die Beschreibung von Algorithmen und den von ihnen manipulierten Daten:

(A1) Ein Algorithmus ist eine Folge von Anweisungen. Eine Anweisung besteht aus elementaren Anweisungen, die nach den folgenden Regeln zu Anweisungen zusammengefügt werden können. Der Algorithmus erhält einen Bezeichner (oder einen Namen).

Ein Bezeichner ist eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt, z.B. X, i, B34Z, qqq. Meist lässt man auch noch den Unterstrich _ zu: Heute_ist_Donnerstag. In der Praxis begrenzt man die Länge eines Bezeichners, z.B. auf 32 Zeichen.

(A2) Ein Algorithmus arbeitet auf Daten. Daten werden in "Behältern", genannt Variablen, abgelegt. Variablen sind zu Beginn des Algorithmus aufzulisten einschließlich der Angabe, welche Daten in die Variable gelegt werden dürfen und welche nicht (dies nennt man "Deklaration" oder "Vereinbarung" der Variablen). Diese durch "," oder ";" getrennte Auflistung beginnt mit dem Wort declare.

Variablen werden ebenfalls durch Bezeichner dargestellt. In einem Algorithmus soll man die Bezeichner so wählen, dass man hieraus die Bedeutung der Variablen entnehmen kann, also z.B. Bezeichner wie
Eingabewert, Anzahl, Summe_der_Messwerte, Mittelwert_von_1_bis_100.

(A3) Elementare Anweisungen sind von der Form:

skip	*Bedeutung:* Tue nichts.
X := α	"Wertzuweisung". α ist ein Ausdruck. *Bedeutung:* Rechne den Ausdruck α aus und lege den erhaltenen Wert in der Variablen X ab.
read (X)	Leseanweisung. *Bedeutung:* Lies den nächsten Wert ein und lege ihn in der Variablen X ab.
write (α)	Schreibanweisung. *Bedeutung:* Drucke den Wert, den der Ausdruck α besitzt, aus.
halt	*Bedeutung:* Beende den Algorithmus.
$F(X_1, ..., X_n)$	*Bedeutung:* Führe den Algorithmus F mit den Werten der Variablen $X_1, ..., X_n$ aus.

(A4) Ausdrücke sind entweder übliche arithmetische Ausdrücke (aufgebaut aus Zahlen, Variablen, Klammern und Operatoren wie +, -, *, /, div, mod) oder logische Ausdrücke (aufgebaut aus den Wahrheitswerten true und false, Variablen, Klammern, Vergleichen und Operatoren wie and, or, not usw.) oder Zeichenausdrücke (aufgebaut aus den Zeichen eines Alphabets, Variablen und Operatoren wie append, empty, remove usw.). Logische Ausdrücke nennt man auch Boolesche Ausdrücke. Wir setzen voraus, dass jede(r) weiß, wie Ausdrücke aufgebaut sind und wie man Ausdrücke auswertet.

(A5) Jede elementare Anweisung ist auch eine Anweisung.

(A6) Hintereinanderausführung oder Sequenz: Wenn D und E Anweisungen sind, dann ist auch D;E eine Anweisung.
Bedeutung: Führe erst D und danach E aus.

(A7) Alternative oder Fallunterscheidung:

Wenn C und D Anweisungen und β ein Boolescher Ausdruck sind, dann ist auch

if β then C else D fi

eine Anweisung.

Bedeutung: Wenn zu dem Zeitpunkt, zu dem man auf diese Anweisung stößt, der Boolesche Ausdruck β den Wert true besitzt, so führe die Anweisung C aus, anderenfalls die Anweisung D.

Spezialfall: Falls D die Anweisung skip ist, so schreibt man kurz if β then C fi (*einseitige Alternative*).

Hinweis: fi ist wie die "Klammer Zu" zum Symbol if. Die Klammerpaare "(" und ")" oder "[" und "]" entstehen auseinander ebenfalls durch Spiegelung. Auch das in (A8) auftretende "od" bildet mit "do" eine solche Klammerung.

(A8a) while-Schleife:

Wenn C eine Anweisung und β ein Boolescher Ausdruck sind, dann ist while β do C od eine Anweisung.
Bedeutung: Solange der Boolesche Ausdruck β den Wert true ergibt, wiederhole die Anweisung C. Hierbei wird der Ausdruck β stets *vor* der Ausführung von C ausgewertet. Wenn also β zu dem Zeitpunkt, zu dem man auf die while-Schleife stößt, false ist, wird C überhaupt nicht ausgeführt.

(A8b) repeat-Schleife:

Wenn C eine Anweisung und β ein Boolescher Ausdruck sind, dann ist repeat C until β eine Anweisung.
Bedeutung: Solange der Boolesche Ausdruck β den Wert false ergibt, wiederhole die Anweisung C. Hierbei wird der Ausdruck β erst *nach* der Ausführung von C ausgewertet. C wird also stets mindestens einmal ausgeführt.

(A8c) for-Schleife oder Zählschleife:

Wenn C eine Anweisung, i eine Variable, in die ganze Zahlen gelegt werden dürfen, und a und e zwei ganze Zahlen sind, dann ist auch

for i := a to e do C od

eine Anweisung.

Bedeutung: Setze i auf den Wert a. Falls i nicht größer als e ist, führe C aus. Erhöhe nun i um 1 und wiederhole diesen Vorgang, bis i größer als e ist; anschließend führe die Anweisung, die auf die for-Schleife folgt, aus.

Präzisere Festlegung: Die for-Schleife besitzt genau die gleiche Bedeutung wie folgende Anweisung

i := a; while i ≤ e do C; i := i+1 od

Man verbietet, dass die Variable i durch C verändert werden darf. (Insbesondere gibt es in C keine Wertzuweisung der Form i := ... .)

(A8d) Allgemeine for-Schleife:

Wenn C eine Anweisung, i eine Variable, in die ganze Zahlen gelegt werden dürfen, und α, δ und ω arithmetische Ausdrücke, die eine ganze Zahl als Ergebnis liefern, sind, dann ist auch

for i := α by δ to ω do C od

eine Anweisung.

Bedeutung: Werte den Ausdruck α aus. Setze i auf diesen Wert. Nun wiederhole folgendes bis zum Abbruch: [Werte den Ausdruck ω aus. Falls i größer als dieser Wert ist, brich die for-Schleife ab. Anderenfalls führe C aus. Werte danach den Ausdruck δ aus und addiere diesen Wert zu i hinzu.]

Vermutung: Die allgemeine for-Schleife besitzt die gleiche Bedeutung wie folgende Anweisung

i := α; while i ≤ ω do C; i := i+δ od

(A8d) Allgemeine for-Schleife (Fortsetzung):

Das ist aber wohl nicht zutreffend, wenn δ negativ ist.

Betrachte: for i := 10 by -2 to 5 do X:=X+i od

Diese Schleife soll für die Werte i=10, i=8 und i=6 durchlaufen werden, da die Schrittweite abgesenkt wird.

Im Falle einer negativen Schrittweite muss man die Bedingung $i \le \omega$ also durch $i \ge \omega$ ersetzen. Man kann den Fall einer positiven und einer negativen Schrittweite zusammenfassen durch $i*sign(\delta) \le \omega*sign(\delta)$, wobei "$*$" die Multiplikation und sign(z) das Vorzeichen der Zahl z sind. Also:

Die allgemeine for-Schleife besitzt genau die gleiche Bedeutung wie folgende Anweisung

i := α; while $i*sign(\delta) \le \omega*sign(\delta)$ do C; i := i+δ od

Einschränkung in Programmiersprachen: Sofern in der Praxis die allgemeine Form der for-Schleife zugelassen wird, verbietet man meist, dass die Variable i sowie die Ausdrücke α, δ und ω in der Anweisung C verändert werden dürfen.

Man fügt also eine Zusatzbedingung an die Anweisung C in der for-Schleife for i := α by δ to ω do C od hinzu:

In C dürfen die Variable i und alle Variablen, die in den Ausdrücken α, δ und ω vorkommen, nicht verändert werden.

Man kann auch anders vorgehen, indem man Änderungen in α, δ und ω erlaubt, aber diese unwirksam für die Schleife macht (siehe nächste Folie).

Man kann jedoch auch die Werte von α, δ und ω vor dem Eintritt in die Schleife auswerten und diese Werte dann für die Schleifen-Kontrolle verwenden unabhängig von der Anweisung C.

Das heißt, man legt die Bedeutung der allgemeinen for-Schleife dann nicht wie in (A8d), sondern durch folgende Anweisung fest:

i := α; D := δ; E := ω; V := sign(δ);
while $i*V \le E*V$ do C; i := i+D od

wobei D, E und V drei neue Variable sind, die sonst nirgends im Algorithmus auftreten.

Machen Sie sich an Beispielen klar, dass wir somit drei verschiedene Bedeutungen der for-Schleife (A8d) angegeben haben.

Ein solches Beispiel lautet:

X := 1; Y := 0;

for i := 3 by X to 8 do

Y := Y+1; X := X+1; od ;

1) Wenn die Laufvariable i nicht verändert werden darf, so ist diese Anweisung fehlerhaft und der Algorithmus bricht ab.

2) Man könnte diese Anweisung auswerten wie die Anweisung (neue Variablen für Schrittweite und Ende einführen)

X := 1; Y := 0;
i := 3; D := X; E := 8 ; V := sign(X);
while $i*V \le E*V$ do Y := Y+1; X := X+1; i := i+D od

Rechnen Sie diese Anweisung durch. Am Ende haben die Variablen X und Y dann die Werte 7 bzw. 6.

Ein solches Beispiel lautet:

X := 1; Y := 0;

for i := 3 by X to 8 do

  Y := Y+1; X := X+1; od ;

3) Nach der allgemeinsten Definition wird diese Anweisung ausgewertet wie die Anweisung

X := 1; Y := 0; i := 3;
while i*sign(X) ≤ 8*sign(X) do Y := Y+1; X := X+1; i := i+X od

Rechnet man diese Anweisung durch, so haben am Ende die Variablen X und Y die Werte 4 bzw. 3.

Anmerkung: In der Praxis verwendet man meist nur die Zählschleife (A8c), bei der die Laufvariable nicht im Schleifenrumpf verändert werden darf, ergänzt um das "Herunterzählen":

for i := a downto e do C od

Hierbei wird i am Ende der Schleife nicht um 1 erhöht, sondern um 1 erniedrigt. Die Bedeutung lautet für "downto" also:

i := a; while i ≥ e do C; i := i-1 od

Auch wir werden in Zukunft nur diese Art der for-Schleife verwenden, wie es z.B. in Ada realisiert ist (mit dem Schlüsselwort "reverse" beim Abwärtszählen). Die allgemeine Form diente hier nur als Beispiel, dass man weitere Anweisungsformen einführen kann und dass man deren Bedeutung präzise definieren muss (in Java gibt es eine noch allgemeinere Variante). Achten Sie daher bei Algorithmen stets auf die genaue Definition ihrer Bestandteile.

(A9) Ergänzende Vorschriften

Aus Gründen der Übersichtlichkeit und Lesbarkeit legt man meist weitere Beschränkungen fest, *zum Beispiel*:

Die Deklarationen müssen stets am Anfang des Algorithmus (oder Teilalgorithmus, *Block*) angegeben werden.

Die auf die Deklarationen folgende Anweisung wird in begin ... end eingeklammert. (halt kann dann entfallen.)

Kommentare trennt man bis zum Zeilenende durch die Zeichen -- vom Algorithmus ab.

Jeder vorkommende Bezeichner muss vorher in einer Deklaration vereinbart worden sein.

Der Algorithmus beginnt stets mit einem Schlüsselwort (z.B. program), danach folgt der Name des Algorithmus, danach das Wort is, dann die Deklarationen und schließlich die in begin und end eingeschlossene Anweisung.

Einen nach den Vorschriften (A1) bis (A9) aufgeschriebenen Algorithmus bezeichnen wir als **Programm**.

Unsere Programme sind also folgendermaßen aufgebaut:

program <Name des Algorithmus> is
declare <Deklarationen>;
begin <Anweisung> end

Später werden wir dies erweitern. Insbesondere werden wir den Deklarationsteil ausgestalten und wir werden Parameter hinzufügen, um mehr Flexibilität zu erreichen.

Programme von imperativen Programmiersprachen sind in der Praxis meist nach diesem Muster aufgebaut.

Hinweise:

Wir ergänzen (A1) bis (A9) später um weitere Regelungen. Auf diese Weise erhalten wir dann eine "richtige Programmiersprache".

Zunächst ist die Deklaration von Variablen zu präzisieren, insbesondere wenn die Variablen nicht Zahlen, sondern komplexere Gebilde wie Vektoren, Matrizen oder Graphen als Werte besitzen.

Sodann ist die Wiederverwendung von bereits ausformulierten Programmen festzulegen. Hierfür werden wir Prozeduren, Funktionen, Moduln und Objekte einführen.

Schließlich ist das Zusammenspiel von Programmen zu regeln (Dialoge und andere Interaktionen).

Was ist der Sinn eines Algorithmus?

Er soll ein Problem lösen, indem er den Eingabewerten ihre zugehörigen Ausgabewerte zuordnet.

Eingabewerte $\rightarrow$ Ausgabewerte.

Wenn In die Menge aller möglichen Eingabewerte und Out die Menge aller möglichen Ausgabewerte des deternministischen Algorithmus A ist, dann beschreibt A also genau eine Abbildung

Res_A: In $\rightarrow$ Out mit

$Res_A(u) = v \quad \Leftrightarrow$ v ist die Ausgabe, die der Algorithmus A bei Eingabe von u liefert.

Man nennt Res_A die von A realisierte Abbildung. Meist ist diese Abbildung vorgegeben und man sucht einen Algorithmus, der diese Abbildung realisiert.

(Falls A für u nicht anhält, so sei $Res_A(u)$ = "undefiniert".)

Wie hängen Algorithmus und Programm zusammen?

Ein Programm ist die Formulierung eines Algorithmus in einer konkreten Programmiersprache. Hierzu muss die Programmiersprache hinreichend aussagekräftig sein; Ada ist z.B. eine solche Sprache.

Im Algorithmus können also Formulierungen vorkommen, die man nicht unmittelbar in ein Programm übertragen kann. Man spricht dann davon, dass der Algorithmus in der jeweiligen Programmiersprache realisiert oder implementiert werden muss.

Man beachte, dass es zu einem Algorithmus beliebig viele Programme geben kann, durch die er implementiert wird.

Beispiel 2.1.6: Stelle fest, ob eine natürliche Zahl a eine Quadratzahl ist. Falls ja, drucke 1 aus, sonst drucke 0 aus.

Verfahren: Prüfe für alle Zahlen von 0 bis a, ob deren Quadrat gleich a ist. Falls es eine solche Zahl gibt, dann ist a eine Quadratzahl, anderenfalls nicht. Übertragen in unsere Sprache:

```
program quadratzahl1 is
declare x, i, ergebnis: Variablen für natürliche Zahlen;
begin  read (x);                    -- Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;
       for i:=0 to x do             -- prüfe für i von 0 bis a, ob i² = a ist
           if i*i = x then ergebnis := 1 fi  od;
       write (ergebnis)
end
```

Überzeugen Sie sich, dass alle Anforderungen (A1) bis (A9) eingehalten wurden; dies ist also ein korrekt gebildetes Programm.

Fortsetzung Beispiel 2.1.6:
Rechnet man das Programm für eine Zahl, z.B. für a = 33 durch, so quadriert man alle Zahlen von 0 bis 33, wobei man jedes Mal feststellt, dass i^2 ungleich 33 ist. Man hätte bereits bei i = 6 aufhören können, da ab dann $i^2 > 33$ ist. Dies führt zu folgendem "effizienter" arbeitenden Programm:

```
program quadratzahl2 is
declare x, i, ergebnis: Variablen für natürliche Zahlen;
begin  read (x);                -- Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;  i := 0;
       while i*i ≤ x do         -- prüfe nur für i von 0 bis wurzel(a)
           if i*i = x then ergebnis := 1 fi; i := i+1 od;
       write (ergebnis)
end
```


Fortsetzung Beispiel 2.1.6:

Das Programm quadratzahl2 führt nicht mehr a, sondern nur noch 2·wurzel(a) Multiplikationen durch. Frage: Kann man auch noch die aufwendigen Multiplikationen sparen?

Ja, das geht. Beachten Sie, dass die Differenz zwischen zwei Quadratzahlen immer eine ungerade Zahl ist und dass man die n-te Quadratzahl erhält, indem man die ungeraden Zahlen von 1 bis 2n-1 aufsummiert.

1 = 1, 4 = 1+3, 9 = 1+3+5, 16 = 1+3+5+7, 25 = 1+3+5+7+9, ...

Wir legen die nächste ungerade Zahl in der Variablen u ab (erster Wert ist 1) und das aktuelle Quadrat in der Variablen q (deren erster Wert ist 0). Die nächste Quadratzahl ermitteln wir dann durch die Anweisungen q := q+u; u := u+2 .

Fortsetzung Beispiel 2.1.6:
Dies führt zu dem Programm

```
program quadratzahl3 is
declare x, u, q, ergebnis: Variablen für natürliche Zahlen;
begin  read (x);                -- Es wird a eingelesen und in x abgelegt.
       q := 0;  u := 1; ergebnis := 0;
       while q ≤ x do           -- Prüfe für alle Quadrate q ≤ a, ob q = a ist.
           if q = x then ergebnis := 1 fi; q := q+u; u := u+2  od;
       write (ergebnis)
end
```

Hier werden keine Multiplikationen mehr benötigt, sondern nur noch 2·wurzel(a) Additionen. Dieses Programm wird daher wesentlich schneller durchzurechnen sein als quadratzahl1.

Frage: Wie prüft man nach, was ein Programm macht?

Einfache Antwort: Man vollzieht es schrittweise nach, wobei man die Veränderungen aller Variablen notiert. Ein solches Schema nennt man ein Ablaufprotokoll. Das Schema hierfür lautet (man schreibe die Eingabe und Ausgabe gesondert auf):

Schritt	Aktion	<Var. 1>	<Var. 2>	<Var. 3>	<Var. 4>	···

Definition 2.1.7: Es sei ein Programm mit seinen aktuellen Eingabedaten gegeben. Bilde eine zweidimensionale Tabelle, die für jede im Programm vorkommende Variable eine Spalte, zwei Spalten für die fortlaufende (Zeilen-) Nummerierung und für die aktuelle Aktion (dies ist in der Regel eine Anweisung oder die Auswertung eines Ausdrucks) sowie eventuelle weitere Spalten für Hilfsinformationen besitzt.

Trage in die erste Zeile die Anfangssituation ein, also die erste Aktion des Programms und die Werte der Variablen nach Durchführung dieser Aktion. Trage in die jeweils nächste Zeile mit der Nummer k die im k-ten Schritt durchgeführte Aktion und die Werte der Variablen nach Durchführung dieser Aktion ein, solange bis halt oder end erreicht wird. Ein- und Ausgabe notiere man gesondert. Die so entstandene Tabelle heißt Ablaufprotokoll des Programms für die gegebenen Eingabedaten.


```
program quadratzahl3 is
declare x, u, q, erg: Variablen für natürliche Zahlen;
begin  read (x);
       q := 0;  u := 1; erg := 0;
       while q ≤ x do
           if q = x then erg := 1 fi; q := q+u; u := u+2  od;
       write (erg)
end
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

Schritt	Aktion	x	u	q	erg	Ausdrücke
1	read(x)	14	⊥	⊥	⊥	
2	q := 0	14	⊥	0	⊥	
3	u := 1	14	1	0	⊥	
4	erg := 0	14	1	0	0	
5	q ≤ x	14	1	0	0	true
6	q = x	14	1	0	0	false
7	q := q+u	14	1	1	0	
8	u := u+2	14	3	1	0	
9	q ≤ x	14	3	1	0	true


```
program quadratzahl3 is
declare x, u, q, erg: Variablen für natürliche Zahlen;
begin  read (x);
       q := 0;  u := 1; erg := 0;
       while q ≤ x do
           if q = x then erg := 1 fi; q := q+u; u := u+2  od;
       write (erg)
end
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

Schritt	Aktion	x	u	q	erg	Ausdrücke
10	q = x	14	3	1	0	false
11	q := q+u	14	3	4	0	
12	u := u+2	14	5	4	0	
13	q ≤ x	14	5	4	0	true
14	q = x	14	5	4	0	false
15	q := q+u	14	5	9	0	
16	u := u+2	14	7	9	0	
17	q ≤ x	14	7	9	0	true
18	q = x	14	7	9	0	false


```
program quadratzahl3 is
declare x, u, q, erg: Variablen für natürliche Zahlen;
begin  read (x);
       q := 0;  u := 1; erg := 0;
       while q ≤ x do
           if q = x then erg := 1 fi; q := q+u; u := u+2  od;
       write (erg)
end
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

Schritt	Aktion	x	u	q	erg	Ausdrücke
19	q := q+u	14	7	16	0	
20	u := u+2	14	9	16	0	
21	q ≤ x	14	9	16	0	false
22	write(erg)	14	9	16	0	**Ausgabe 0**
23	"end"	14	9	16	0	

Ausgabe ist 0, d.h., die Eingabe ist keine Quadratzahl.

Vergleich: Wie wird dieser Algorithmus in Ada formuliert?

```
program quadratzahl3 is
declare x, u, q, ergebnis:
  Variablen für natürliche Zahlen;
begin  read (x);
   q := 0;  u := 1; ergebnis := 0;
   while q ≤ x do
      if q = x then
          ergebnis := 1 fi;
      q := q+u; u := u+2
   od;
   write (ergebnis)
end
```

```
procedure  quadratzahl3  is
x, u, q, ergebnis: Natural;
          -- Natural steht für nat. Zahlen
begin  Get (x);
   q := 0;  u := 1; ergebnis := 0;
   while q ≤ x loop
      if q = x then
          ergebnis := 1; end if;
      q := q+u; u := u+2;
   end loop;
   Put (ergebnis);
end;
```


Beispiel 2.1.8: Klärung der Darstellung von Daten
Wir greifen nun noch einmal Beispiel 2.1.3 auf.

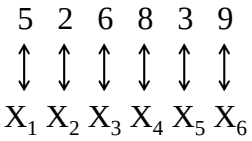
Erinnerung: Addiere 1 zu einer dezimal dargestellten Zahl.

Umgangssprachliche Formulierung: Eine (natürliche) Zahl sei als eine Folge von Ziffern aus der Menge {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} gegeben. Falls die letzte Ziffer nicht die 9 war, so ersetze sie durch die nächste größere Ziffer und beende das Verfahren, anderenfalls ersetze sie durch 0 und wiederhole das Verfahren für die zweitletzte Ziffer usw.

Aufgabe: Formuliere diesen Algorithmus mit Hilfe unserer Sprachelemente.

Problem: Wie beschreibt man eine Ziffernfolge?
Vorschlag: mit indizierten Variablen.

Ziffern sind Elemente der Menge {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.
Betrachte eine Folge solcher Ziffern:



"Indizierte Variable"

Vorläufig schreiben wir bei der Deklaration:
declare X_1, ..., X_n: Variablen für ...
und in Anweisungen X_i, wenn wir die i-te dieser Variablen verwenden wollen.

Formulierung des oben umgangssprachlich formulierten Algorithmus mit Hilfe unserer Sprachelemente:


```
program  add1_versuch1  is
declare  i, n: Variablen für natürliche Zahlen;
      x1, ..., xn: Variablen für Ziffern;
begin  read(n);
     for i := 1 to n do read(xi) od;
     for i := n downto 1 do
          if xi ≠ 9 then xi := nächsteziffer(xi)
          else xi := 0 fi
     od;
     for i := 1 to n do write(xi) od
end
```

Vorsicht: falsches Programm!

Hier treten alle Unklarheiten, die wir schon in Beispiel 2.1.3 besprochen hatten, deutlich auf. Zugleich befindet sich in der zweiten Zählschleife ein gravierender Fehler. Zum Beispiel liefert die Eingabe 1992 die Ausgabe 2003.

Klärung: Was bedeutet x_i := nächsteziffer(x_i) ? Antwort:
Diese Anweisung steht als Abkürzung für

if $x_i = 0$ then $x_i := 1$

else if $x_i = 1$ then $x_i := 2$

else if $x_i = 2$ then $x_i := 3$

else if $x_i = 3$ then $x_i := 4$

else if $x_i = 4$ then $x_i := 5$

else if $x_i = 5$ then $x_i := 6$

else if $x_i = 6$ then $x_i := 7$

else if $x_i = 7$ then $x_i := 8$

else if $x_i = 8$ then $x_i := 9$ fi fi fi fi fi fi fi fi fi

Beseitigung des gravierenden Fehlers: Die Anweisung
x_i := nächsteziffer(x_i) darf nur genau einmal ausgeführt werden.

```
for i:=n downto 1 do
      if x_i ≠ 9 then x_i := nächsteziffer(x_i)
      else x_i := 0 fi
od;
```

müsste also ersetzt werden durch:

```
for i:=n downto 1 do
      if x_i ≠ 9 then
          x_i := nächsteziffer(x_i);
          "for-Schleife abbrechen"
      else x_i := 0 fi
od;
```


Wir müssten also eine neue elementare Handlung, nennen wir sie wie in Ada "exit" (englisch "Ausgang"), einführen mit der *Bedeutung:* Verlasse die aktuelle Schleife, d.h., setze die Ausführung des Programms mit der Anweisung fort, die unmittelbar auf diese Schleife folgt.

Algorithmen mit solchen exit-Anweisungen sind für Menschen meist schwerer zu lesen und führen häufiger zu Fehlern. Daher wollen wir diesen Weg zunächst nicht beschreiten.

Statt dessen können wir eine Boolesche Variable "fertig" einführen, die anfangs false ist und die auf true gesetzt wird, sobald x_i := nächsteziffer(x_i) ausgeführt wurde. Wenn fertig den Wert true hat, darf kein x_i mehr verändert werden. Wir bauen diesen Gedanken in das Programm ein.

Die Ergänzungen sind blau und kursiv gekennzeichnet:

```
program  add1_versuch2  is
declare  i, n: Variablen für natürliche Zahlen;
     fertig: Variable für Boolesche Werte;
     x_1, ..., x_n: Variablen für Ziffern;
begin  read(n); fertig := false;
    for i:=1 to n do read(x_i) od;
    for i:=n downto 1 do
        if not fertig then
            if x_i ≠ 9 then x_i := nächsteziffer(x_i); fertig := true
            else x_i := 0 fi
        fi
    od;
    for i:=1 to n do write(x_i) od
end
```

Vorsicht: immer noch falsches Programm!

Was ist nun noch falsch?

Besteht die Eingabe nur aus Neunen, so werden nur Nullen ausgegeben. In diesem Fall muss also am Ende noch eine '1' als führende Ziffer ausgegeben werden.

Diesen Fall erkennen wir daran, dass nach Durchlaufen der zweiten for-Schleife die Variable fertig immer noch den Wert false besitzt. In diesem Fall geben wir also als erste Ziffer eine '1' aus.

Den Fall, dass anfangs n = 0 eingegeben wird, müssen wir noch berücksichtigen. In diesem Fall wird nichts eingelesen und nichts ausgegeben, so dass die Ziffer 1 nur im Fall $n \geq 1$ ausgedruckt werden darf.


```
program  add1_versuch3  is
declare  i, n: Variablen für natürliche Zahlen;
      fertig: Variable für Boolesche Werte;
      x1, ..., xn: Variablen für Ziffern;
begin  read(n); fertig := false;
    for i:=1 to n do read(xi) od;
    for i:=n downto 1 do
        if not fertig then
            if xi ≠ 9 then xi := nächsteziffer(xi); fertig := true
            else xi := 0 fi
        fi
    od;
    if (not fertig) and (n ≥ 1) then write(1) fi;
    for i:=1 to n do write(xi) od
end
```

Vorsicht: schlechter Programmierstil!

Warum ist dies ein "schlechter Programmierstil" gewesen?

Wichtig ist, dass sich im Algorithmus die Struktur des Problems und eine angemessene Lösung widerspiegelt und dass das Programm möglichst keine Anteile enthält, die nur wegen der Darstellung (also der vorgegebenen Sprachelemente) eingefügt werden müssen, die aber mit dem Problem nichts zu tun haben.

Wenn das Programm das Lösungsverfahren nicht klar zum Ausdruck bringt, sondern es verschleiert und wenn das Programm nicht "gut lesbar" und daher auch nicht "wartbar" ist (d.h., es lassen sich Fehler nur schwer finden und das Programm kann kaum an ähnliche Situationen angepasst werden), so liegt ein schlechter Programmierstil vor.

Unser Programm add1_versuch3 mag zwar korrekt arbeiten, aber es gibt die Lösung nicht angemessen wieder.

Auffälligste Abweichung: Unser Programm durchläuft in jedem Fall alle Variablen, obwohl die Lösungsmethode zu Ende ist, sobald ein $x_i \neq 9$ erreicht ist.

Grund für diese Abweichung: Wir haben eine Zählschleife verwendet, die aber die Lösungsmethode nicht angemessen widerspiegelt. Vielmehr besagt die Lösungsmethode: Solange man (von hinten beginnend) eine 9 vorfindet, ersetze sie durch eine 0. Die danach erreichte Ziffer wird durch ihre nächst größere Ziffer ersetzt bzw., wenn man am Anfang angelangt ist, wird die Ziffer 1 vorangestellt.

Wir müssen statt der for- also eine while-Schleife verwenden.

Diese lautet:

i := n;
while $x_i = 9$ do $x_i := 0$; i := i-1 od;

Nun sind wir bei einem $x_i \neq 9$ angelangt und können diese Ziffer durch ihre nächst größere Ziffer ersetzen.

Fehlerhaft ist noch, dass im Falle einer Folge aus Neunen die Variable i den Wert 0 erhält und dann die nicht vorhandene Variable x_0 in der while-Bedingung auftritt. Wir müssten also noch ein "i > 0" in die while-Bedingung aufnehmen:

i := n;
while $(x_i = 9)$ and (i > 0) do $x_i := 0$; i := i-1 od;

Dies ist aber wenig hilfreich, denn wenn i = 0 geworden ist, dann steht in dem Booleschen Ausdruck $(x_i = 9)$ and (i > 0) immer noch die nicht vorhandene Variable x_0.

Ada behilft sich hier folgendermaßen (siehe 1.8.3): Es wird neben dem symmetrischen Operator and ein weiterer Operator and then eingeführt. α and then β bedeutet: Werte erst α aus. Falls der Wert false ist, so ist der ganze Ausdruck false; anderenfalls werte β aus und dessen Wert ist dann der Wert des gesamten Ausdrucks. In Ada könnte man also die Bedingung formulieren (dort schreibt man x(i) an Stelle von x_i):

```
i := n;
while (i > 0) and then (x(i) = 9) loop
        x(i) := 0;  i := i-1  end loop;
```


Wir müssen aber nicht so vorgehen. Vielmehr können wir von hinten beginnend maximal nur bis zur zweiten Ziffer prüfen und anschließend den kritischen Fall abfangen:

i := n;
while $(x_i = 9)$ and (i > 1) do $x_i := 0$; i := i-1 od;
if $((i = 1)$ and $(x_1 = 9))$
then $x_1 := 0$; write (1)
else x_1 := nächsteziffer(x_1)
fi;
gib x_1 bis x_n aus.

Dies führt zu folgendem Programm (wir fangen noch den Fall n < 1 zu Beginn ab):


```
program  add1_version1  is
declare  i, n: Variablen für natürliche Zahlen;
      x1, ..., xn: Variablen für Ziffern;
begin  read(n);
  if  n > 0  then
     for i:=1 to n do  read(xi)  od;
     i := n;
     while  (xi = 9)  and  (i > 1)  do  xi := 0;  i := i-1  od;
     if  ((i = 1) and (x1 = 9))  then  x1 := 0; write (1)
           else  xi := nächsteziffer(xi)  fi;
     for i:=1 to n do  write(xi)  od
  fi
end
```

Eine andere Variante besteht darin, mit einem "Stopper" zu arbeiten. Hierzu erweitert man die Daten um ein Element, mit dem garantiert wird, dass die while-Schleife stets korrekt beendet wird.

Im Falle der "Addition einer 1" fügen wir die Variable x_0 hinzu und setzen sie auf 0. Dann ist garantiert, dass die while-Schleife spätestens für i = 0 abgebrochen wird.

Wurde nach der while-Schleife x_0 verändert, dann lag eine Folge aus Neunen vor und die Anzahl der Ziffern erhöht sich um eine Ziffer, anderenfalls bleibt die Anzahl gleich. Alle Abfragen bzgl. "i > 1" oder "i = 1" entfallen jetzt.

Mit dem Stopper x_0 erhalten wir folgendes Programm:


```
program  add1_version2  is
declare  i, n: Variablen für natürliche Zahlen;
      x0, x1, ..., xn: Variablen für Ziffern;
begin  read(n);
 if  n > 0  then
    for i := 1 to n do  read(xi)  od;
    x0 := 0;  i := n;
    while  xi = 9  do  xi := 0;  i := i-1  od;
    xi := nächsteziffer(xi);
    if x0 ≠ 0 then write(x0) fi;
    for i := 1 to n do  write(xi)  od
 fi
end
```


Dieses Programm add1_version2 ist (nach einiger Erfahrung mit Programmen unter Verwendung der hier vorgestellten Sprachelemente) leicht lesbar und folgt zugleich der üblichen Vorstellung, dass man zu jeder Zahl eine führende Null hinzufügen kann, die verändert werden muss, sofern ein Überlauf bei der Addition entsteht.

Unbefriedigend ist noch, dass die Datenbereiche und die Deklaration der Variablen umgangssprachliche Elemente enthält. (Aus Kapitel 1 wissen wir aber schon, wie man dies durch array und record beschreiben kann.)

2.2 Charakteristika von Algorithmen

Wie lauten notwendige Eigenschaften, die ein Algorithmus erfüllen muss, "damit er ein Algorithmus sein kann"?

Beispielsweise darf ein Algorithmus nicht in einem Schritt eine unendliche Menge von Möglichkeiten abprüfen oder unendlich viele Kopien von sich erzeugen können.

Auch darf ein Algorithmus nicht über unendlich viele verschiedene elementare Handlungen verfügen können; dies dürfen sogar nur beschränkt viele sein, da man anderenfalls keine mechanische Maschine zur Bearbeitung bauen könnte.

Wir listen einige Eigenschaften auf, die aus der Forderung an Algorithmen "realisierbar mit einer mechanischen Maschine" abgeleitet werden können.

2.2.1 Ein Algorithmus ist eine Vorschrift, die die Reihenfolge von durchzuführenden Handlungen (Operationen) auf Daten (Operanden) genau beschreibt. Hierbei muss gelten:

a) Die Daten sind "diskret" aufgebaut und es gibt beschränkt viele ("digitale") Zeichen $\{a_1, ...,a_k\}$, so dass jedes Datum eine endliche Folge dieser Zeichen ist.

b) Die Operationen sind "diskret" aufgebaut. Genauer: Es gibt beschränkt viele "Grundoperationen" $\{b_1, ...,b_m\}$, so dass jede Operation einschließlich ihrer Operanden hieraus zusammengesetzt werden kann.

c) Die Vorschrift ist eine endliche Folge von Operationen. Die Vorschrift wird schrittweise abgearbeitet (diskrete Zeitskala).

Hinweis: Eine Menge heißt "diskret", wenn ihre Elemente gut unterscheidbar und mit endlicher Länge darstellbar sind. Erfolgt die Darstellung mit beschränkt vielen Zeichen, so spricht man von einer digitalen Darstellung.

d) Eine der Operationen ist als Startoperation ausgezeichnet.

e) Für jede Operation ist unmittelbar nach ihrer Ausführung bekannt, welches die möglichen (endlich vielen) Folgeoperationen sind oder ob der Algorithmus abbricht (terminiert).

f) Die Eingabe für die Vorschrift ist eine (eventuell unendliche oder auch leere) Folge von Daten (vgl. a).

g) In jedem Schritt (d.h. zu jedem Zeitpunkt) gilt: Die bis dahin bearbeitete oder betrachtete Menge an Daten und durchgeführten Operationen ist endlich.

Hinweis: Ein Algorithmus verfügt über eigene Speicherbereiche für die Daten und für die Vorschrift. Beide Bereiche kann der Algorithmus während seiner Abarbeitung verändern, aber in jedem Schritt nur einen endlichen Bereich. Beide Bereiche sind prinzipiell unendlich groß, auch wenn zu jedem Zeitpunkt nur ein endlicher Teil betrachtet und verändert werden kann.

Hinweis: Obige Ausführungen bilden keine richtige Definition, sondern nur eine Liste von umgangssprachlich formulierten Forderungen.

Turingmaschinen erfüllen diese Forderungen, aber auch andere "Rechenmaschinen" und die bereits vorgestellten Grammatiken, siehe später.

Auch Programme beschreiben Algorithmen. Prüfen Sie die Forderungen an einer Programmiersprache und ihren Programmen nach.

2.2.2 Determinismus, Nichtdeterminismus

Oft weiß man nicht, welches die nachfolgende Operation sein soll. Dann gibt man nicht nur eine, sondern *mehrere mögliche Folgeoperationen* an. Beispiele sind Spiele: Bei Schach, Skat, Siedler von Catan usw. kann man in jedem Zug einen unter vielen auswählen. Ein Verfahren, welches zu irgendwelchen Situationen mehrere Operationen alternativ zulässt, bezeichnet man als nichtdeterministisch.

Solange diese Menge der zulässigen Operationen endlich ist, lässt sich das Problem durch einen "normalen" deterministischen Algorithmus simulieren (siehe später: BFS-Verfahren mit BFS = Breadth-First-Search). Ist diese Menge jedoch unendlich, so liegt kein Algorithmus mehr vor. "Nichtdeterminismus" darf also, sofern er auftritt, nur endlich viele Möglichkeiten zulassen.

In der Praxis verlangt man meist, dass Algorithmen deterministisch sein müssen, d.h., nach Abarbeitung jeder Operation steht eindeutig fest, welches die nachfolgende Operation ist oder ob die gesamte Berechnung hiermit beendet ist.

Diese Forderung lässt sich aber bei vielen Anwendungen nicht einhalten, etwa wenn Programme miteinander kommunizieren und die Reihenfolge der Nachrichtenübermittlungen von der Umwelt oder von Zufällen abhängt.

Beispiel 2.2.3: NIM-Spiel (einfache Variante).
Gegeben sei ein Haufen von n Hölzchen. Zwei Spieler A und B ziehen abwechselnd (A beginnt). In jedem Zug darf man 1, 2 oder 3 Hölzchen wegnehmen. Wer das letzte Hölzchen wegnimmt, hat verloren.

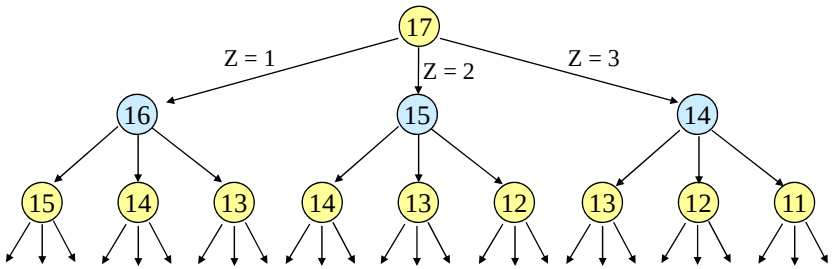
In jedem Zug werden also nichtdeterministisch 1, 2 oder 3 Hölzchen weggenommen. Ein nichtdeterministisches Programm für n=17 Hölzchen würde dann lauten:

```
program NIM is
Anzahl, Z: Integer; A_gewinnt: Boolean;
begin  Anzahl := 17; A_gewinnt := true;
   while Anzahl > 0 do
      wähle eine der drei Anweisungen: Z := 1 | Z:= 2 | Z:= 3;
      Anzahl := Anzahl - Z; A_gewinnt := not A_gewinnt;
      if A_gewinnt then write ("B nimmt ", Z," Hölzchen. ")
                   else write ("A nimmt ", Z," Hölzchen. ") fi
   od;
   if A_gewinnt then write ("A hat gewonnen.")
                else write ("B hat gewonnen.") fi
end
```


Aufgabe für Sie:
Vollziehen Sie dieses Beispiel nach und machen Sie sich klar, welche verschiedenen Abläufe möglich sind.

Obiges Programm enthält eine Unkorrektheit, die aber keinen Einfluss auf das Ergebnis hat. Welche ist dies?

Suchen Sie nach einer Darstellung für solche nichtdeterministischen Abläufe, z.B. baumartig (gelb entspricht "A ist am Zug", blau entspricht "B ist am Zug"; vereinfachen Sie den Baum zu einem Netz, in dem jede Zahl nur höchstens einmal auftritt):



Beachten Sie:
Wir haben nur die Abläufe des NIM-Spiels beschrieben. Dies gibt noch keinen Hinweis darauf, wie man das Spiel spielen muss, um zu gewinnen.
Hierzu muss man die Darstellung analysieren und nach Situationen suchen, die den Gewinn garantieren, und daraus eine Spiel-Strategie ableiten.
Wenn z.B. noch 8 Hölzchen vorhanden sind und B am Zug ist, so kann B den Sieg erzwingen, indem B 3 Hölzchen wegnimmt (Situation "5 Hölzchen") und nach dem Zug von A so viele Hölzchen wegnimmt, dass noch genau ein Hölzchen übrig bleibt.
Dies wäre ein "Zusatzalgorithmus", der zu jeder Situation den verheißungsvollsten Zug ermittelt.

■

2.2.3 Terminierung

Ein Algorithmus (oder ein Programm) *terminiert für eine Eingabe u*, wenn der Algorithmus bei Eingabe von u nach endlich vielen Schritten anhält. Ein Algorithmus "terminiert stets", wenn er für alle Eingaben terminiert.

[Die Abbildungen, die von terminierenden Algorithmen realisiert werden, bilden genau die Menge der total rekursiven Funktionen $\Re$, siehe später.]

In der Praxis wird die Terminierung oft gefordert, insbesondere von Benutzern, die auf jede Eingabe eine Antwort erwarten, und dies möglichst schon nach kurzer Zeit. Die Bedingung der Terminierung ist für Algorithmen aber *nicht notwendig* und ist auch nicht für alle Algorithmen erwünscht. Z.B. sollte ein Betriebssystem prinzipiell unendlich lange arbeiten.

Besonderheit von Algorithmen:

Algorithmen können Algorithmen als Daten einlesen.

Ein Algorithmus lässt sich als Programm formulieren und ist dann eine Folge von Zeichen über dem Terminalalphabet Σ der Programmiersprache, d.h., jeder Algorithmus lässt sich als ein Wort $w \in \Sigma^*$ auffassen. Es gibt Algorithmen, die solche Zeichenfolgen einlesen und verarbeiten können, folglich kann man Algorithmen als Eingabe für Algorithmen verwenden.

Dies ist nicht überraschend; denn ein Compiler ist ein Algorithmus, der Algorithmen in Form von Programmen einliest und sie in Programme der Maschinensprache übersetzt.

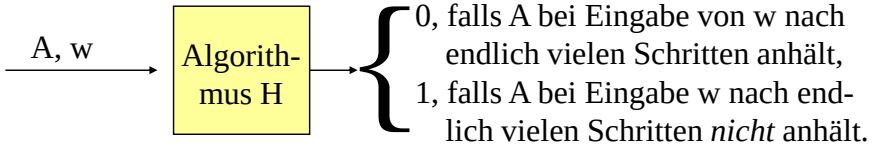
Diese Besonderheit nutzen wir nun aus, um zu zeigen, dass es Probleme gibt, die man mit Algorithmen nicht lösen kann.

2.3 Grenzen der Algorithmen, Unentscheidbarkeit

Was kann man mit Algorithmen ***nicht*** beschreiben oder lösen?

Betrachte folgende Aufgabe:
Konstruiere einen Algorithmus H, der beliebige Algorithmen und zugehörige Eingabedaten einlesen kann und der zu jedem beliebigen Algorithmus A und zu jeder Folge von Daten w in endlich vielen Schritten feststellt, ob der Algorithmus A für die Eingabedaten w nach endlich vielen Schritten anhält oder nicht.
Bezeichnung 2.3.1: Die Aufgabe, einen solchen stets terminierenden Algorithmus H zu finden, bezeichnet man als das Halteproblem für Algorithmen.

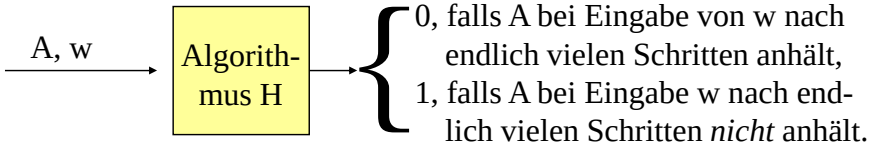
Gesucht wird also ein Algorithmus H



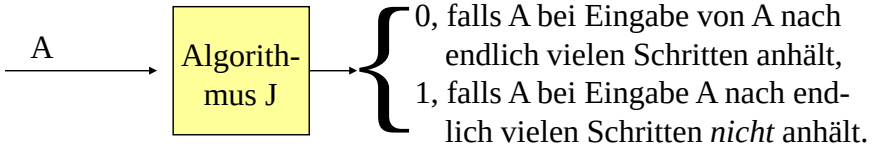
für alle Algorithmen A und für alle Eingabefolgen w.

Hinweis: Wie üblich verwenden wir im Folgenden das Zeichen $\forall$ für "für alle" und das Zeichen $\exists$ für "es existiert".

Angenommen, es gibt solch einen Algorithmus H mit ∀A ∀w

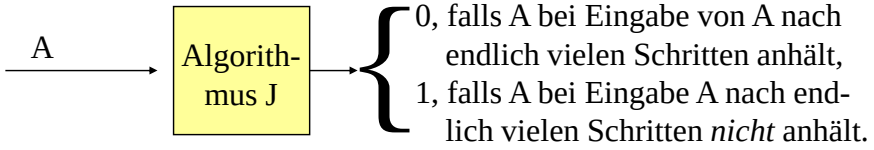


Speziell gibt es dann auch einen Algorithmus J mit ∀A

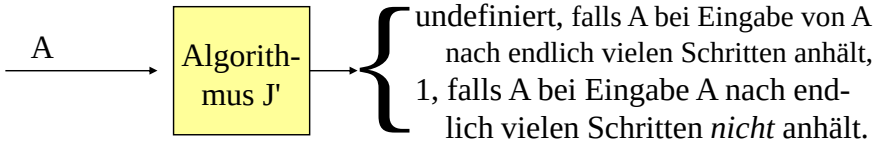


für alle Algorithmen A, indem man nur den Algorithmus A eingibt und dann H mit der Eingabe A und A (= w) ablaufen lässt.

Zu diesem Algorithmus J mit ∀A

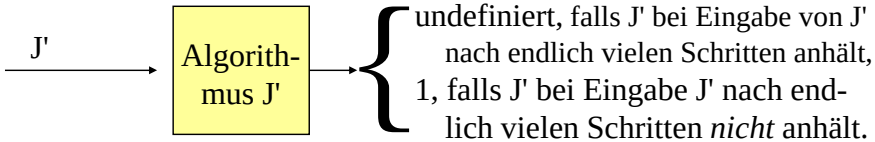


gibt es dann auch einen Algorithmus J' mit ∀A



Klar wegen: Modifiziere J so, dass der Algorithmus J anstelle der Ausgabe 0 in eine unendliche Schleife geht; so erhält man J'.

Was macht J' bei Eingabe von J'?



Fall 1: Bei Eingabe von J' hält J' an.
Dann liefert J' bei Eingabe von J' den Wert 1.
Nach Definition von J' hält dann J' bei Eingabe J' *nicht* an.
Widerspruch!

Fall 2: Bei Eingabe von J' hält J' *nicht* an.
Dann läuft J' bei Eingabe von J' in eine unendliche Schleife.
Nach Definition von J' hält dann J' bei Eingabe J' an und liefert 1.
Widerspruch!

Beide möglichen Fälle führen also auf einen Widerspruch.
Mehr Möglichkeiten gibt es aber nicht.
Folglich muss die Annahme, dass es den Algorithmus H gibt, falsch gewesen sein. Es gilt daher der

Satz 2.3.2 (Unlösbarkeit des Halteproblems)
Es gibt keinen Algorithmus H, der zu jedem beliebigen Algorithmus A und zu jeder Folge von Daten w in endlich vielen Schritten feststellt, ob der Algorithmus A für die Eingabedaten w nach endlich vielen Schritten anhält oder nicht.

Dieses Resultat formuliert man kurz in der Form:

Das Halteproblem ist algorithmisch nicht lösbar.

Anmerkung 2.3.3: Eine Menge $L \subseteq \Sigma^*$ heißt entscheidbar, wenn es einen Algorithmus mit einem stets terminierenden Programm π mit der Eingabemenge $E_\pi = \Sigma^*$ und der realisierten Funktion $f_\pi: \Sigma^* \to \{0,1\}$ gibt, so dass für alle $w \in \Sigma^*$ gilt:

$$f_\pi(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{falls } w \notin L \end{cases}$$

Anderenfalls heißt die Menge L unentscheidbar.

Der obige Satz besagt also:
Das Halteproblem für Algorithmen ist unentscheidbar.

Beachte: Das Halteproblem H lässt sich als Teilmenge $H \subseteq (\Sigma' \cup \{\eta\})^*$ für ein geeignetes Alphabet Σ' auffassen, in welchem sich alle Algorithmen und eingebbaren Daten beschreiben lassen, z.B. das Alphabet $\Sigma' = \mathbf{A}$. Sei $\eta \notin \Sigma'$. Dann gilt nämlich:
$H = \{u\eta v \mid u, v \in \Sigma'^*, u \text{ beschreibt einen Algorithmus und } u \text{ hält für die Eingabe } v \text{ an.}\} \subseteq (\Sigma' \cup \{\eta\})^*$.

Anmerkung 2.3.4: Für eine Menge $L \subseteq \Sigma^*$ heißt die Abbildung $\chi_L: \Sigma^* \to \{0,1\}$, die für alle $w \in \Sigma^*$ definiert wird durch

$$\chi_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{falls } w \notin L, \end{cases}$$

die charakteristische Funktion von L.

Die charakteristische Funktion verbindet die Begriffe "Teilmenge" und "Funktion" miteinander: Zu jeder Teilmenge gehört die charakteristische Funktion und zu jeder Funktion $g: \Sigma^* \to \{0,1\}$ gehört die Teilmenge $\{w \in \Sigma^* \mid g(w) = 1\} \subseteq \Sigma^*$.

Satz 2.3.2 besagt also $\chi_H \notin \mathfrak{R}$ (und natürlich auch $\chi_H \notin \wp$).

Das heißt: Die charakteristische Funktion des Halteproblems $\chi_H: (\Sigma' \cup \{\eta\})^* \to \{0,1\}$ ist nicht rekursiv.

(Zu den Symbolen siehe Abschnitt 2.5.)

Bezeichnung 2.3.5: Die Aufgabe, einen stets terminieren- den Algorithmus H_ε zu finden, der zu jedem beliebigen Algorithmus A nach endlich vielen Schritten feststellt, ob der Algorithmus A für die leere Eingabe ε nach endlich vielen Schritten anhält oder nicht, bezeichnet man als das spezielle oder als das ε-Halteproblem für Algorithmen.

Satz 2.3.6: Das ε-Halteproblem ist unentscheidbar.

Beweis durch Rückführung auf das Halteproblem. Wir konstruieren zu jedem Algorithmus A und zu jeder Eingabe w einen Algorithmus A_w, so dass gilt: A hält bei Eingabe von w genau dann an, wenn A_w mit der leeren Eingabe anhält. Wäre das ε-Halteproblem also entscheidbar, dann wäre auch das Halteproblem entscheidbar, im Widerspruch zu Satz 2.3.2. Folglich muss das ε-Halteproblem unentscheidbar sein.

Konstruktion von A_w aus A und w:
Es seien A ein Algorithmus und $w \in \Sigma^*$ eine Eingabefolge. Dann wandele A in folgenden Algorithmus A_w um: A_w ist wie A aufgebaut, besitzt jedoch eine weitere Zeichenketten-Variable X und an den Anfang des Anweisungsteils wird die Wertzuweisung X := "w"; gesetzt. Die read-Befehle werden jetzt dadurch ersetzt, dass immer die nächsten Zeichen von X (anstelle: "von der Eingabe") der jeweiligen Variablen zugewiesen werden. Der so erhaltene Algorithmus A_w liest also nichts ein und arbeitet genau so, wie A bei der Eingabe w arbeitet. Folglich gilt:

A hält bei Eingabe von w $\Leftrightarrow$ A_w hält bei Eingabe von ε.

Damit ist Satz 2.3.6 bewiesen. ■

Der folgende Abschnitt 2.4 soll Ihnen folgende Inhalte vermitteln:

Elementare Wertebereiche der Programmierung, also die Wertebereiche, die man nicht auf andere Bereiche zurückführt.

Für jeden dieser Bereiche muss eindeutig festgelegt sein, wie seine Elemente dargestellt (aufgeschrieben oder im Rechner repräsentiert) sind. Sie lernen diese Darstellungen kennen.

Zugleich stellen wir alternative Darstellungen vor. An diesen erkennen Sie einige Vor- und Nachteile von Darstellungen; diese hängen meist von den späteren Anwendungen ab, so dass Sie ein Gespür dafür erhalten sollen, in welchem Fall welche Darstellung nutzbringend ist.

Weiterhin lernen Sie einige Phänomene (wie z.B. Rundungsfehler) und mehrere typische Beispiele. Im Vorgriff auf später führen wir am Ende Unterstrukturen und arrays ein.

Diesen Abschnitt 2.4 kennen Sie weitgehend schon aus Ada, Kap. 1 ! Neu, aber nicht schwer sind die Zahldarstellungen, 2.4.8 bis 2.4.17. Vor allem sie werden im Hörsaal vorgetragen.

2.4 Grundlegende Datenbereiche

Programme verändern Daten. Diese Daten werden in (Informatik-) Variablen abgelegt. Eine solche Variable ist ein Behälter, wobei vorher festgelegt wird, welche Werte in den Behälter gelegt werden dürfen und welche nicht.

Durch diese *Festlegung der zulässigen Wertemenge* in den Deklarationen erhält eine Variable einen "Typ", den sog. Datentyp. Wenn eine Variable z.B. den Datentyp "Integer" erhält, so darf sie nur ganze Zahlen als Werte enthalten, erhält sie den Typ "Character", so darf sie nur Zeichen des Alphabets enthalten usw.

Bei den Datentypen ist es wie bei den Anweisungen: Es gibt "elementare Datentypen" (diese gehören zu Wertebereichen, die als grundlegend angesehen werden) und es gibt Regeln, wie man aus bereits definierten Datentypen neue Datentypen gewinnt.

Was sind "elementare Datentypen", also Wertebereiche, die man nicht aus noch einfacheren Mengen zusammensetzen kann oder will?

Die rationalen Zahlen z.B. gelten nicht als "elementar", weil man eine rationale Zahl stets in der Form p/q schreiben kann, wobei p eine ganze und q eine natürliche Zahl sind. Dagegen kann man ganze Zahlen nicht in mehrere Teile zerlegen.

Elementare Datentypen sind in den meisten Programmiersprachen:

- die Menge der Wahrheitswerte,
- die Menge der Zeichen (dies umfasst die Zeichen auf der Tastatur, die sog. alpha-numerischen Zeichen),
- die ganzen Zahlen (einschl. der natürlichen Zahlen)
- die reellen Zahlen
- alle endlichen Mengen, die man selbst durch Auflisten ihrer Elemente definiert (sog. Aufzählungstypen).

Hinweis: Grundsätzlich reicht der Formalismus der Mengenlehre aus: $0 \leftrightarrow \emptyset$, $1 \leftrightarrow \{\emptyset\}$, $2 \leftrightarrow \{\emptyset,\{\emptyset\}\}$, $3 \leftrightarrow \{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\}\}$ usw. Zeichen können dann codiert werden (siehe 1.8), Wahrheitswerte als 0 und 1, selbstdefinierte Mengen entsprechend ihrer Aufzählung. Letztlich kann man auch Folgen von 0 und 1 verwenden, wie es im Rechner geschieht.

Grundlegende Mengen in Mathematik und Informatik sind:
IB = {false, true} Boolesche Werte, Wahrheitswerte, oft: {0, 1}.
IN = {1, 2, 3, 4, ...} ist die Menge der natürlichen Zahlen,
$\mathbf{IN}_0$ = {0, 1, 2, 3, ...} die Menge der natürlichen Zahlen mit der 0,
Z = { ..., -2, -1, 0, 1, 2, 3, ... } ist die Menge der ganzen Zahlen,
A sei die Menge der Tastaturzeichen zuzüglich einiger sog. Steuerzeichen (vgl. später: ASCII-Code, EBCDIC-Code).
Q = {n/m | n ganze Zahl, m natürliche Zahl, n und m teilerfremd} ist die Menge der rationalen Zahlen.
IR bezeichnet die Menge der reellen Zahlen (Vervollständigung der rationalen Zahlen, so dass jeder Punkt der Zahlengeraden als Grenzwert einer konvergierenden ("Cauchy"-) Folge erfasst wird, z.B. die Kreiszahl π oder die Wurzel aus 2).
C ist die Menge der komplexen Zahlen; formal kommt "i" (= die Wurzel aus -1) zu **IR** hinzu; siehe Mathematikvorlesungen.

Aufbau der Zahlen in der Mathematik:

IN = {1, 2, 3, 4, ...}
↓ füge neutrales Element für "+" hinzu
$\mathbf{IN}_0$= {0, 1, 2, 3, ...}.
↓ füge inverse Elemente für "+" hinzu
Z = { ..., -2, -1, 0, 1, 2, 3, ... }
↓ füge inverse Elemente für "·" hinzu
Q = {z/n | z∈**Z**, n∈**IN**, teilerfr.}
↓ füge alle Grenzwerte hinzu
IR reelle Zahlen
↓ füge "i" (= die Lösung von x^2=-1) hinzu
C komplexe Zahlen

Elementare Datentypen 2.4.1:
Wenn eine Variable nur Werte aus der Menge M besitzen darf, so geben wir ihr in einer Deklaration folgenden Datentyp:

Menge	Datentyp
IB	Boolean oder logical
$\mathbf{IN}_0$	Natural oder Cardinal
Z	Integer
IR	Real oder Float
A	Character

$\mathbf{IN}_0$ und **IN** fasst man meist als "Unterbereich" der ganzen Zahlen auf. $\mathbf{IN}_0 = \{z \in \mathbf{Z} \mid z \geq 0\}$ bzw. $\mathbf{IN} = \{z \in \mathbf{Z} \mid z > 0\}$.

C führt man als Paar zweier reeller Zahlen ein und definiert für diese Paare die komplexen Operatoren Addition, Subtraktion, Multiplikation, Division usw.

Eine Sonderrolle spielen die rationalen Zahlen **Q**; denn eigentlich beschreibt der Typ "real" in der Praxis nicht die ganze Menge der reellen Zahlen, sondern nur eine gewisse Teilmenge der rationalen Zahlen. Wir werden dies unter dem Stichwort "Zahlendarstellungen" noch in diesem Abschnitt untersuchen.

Beispiel 2.4.2: ggT (oder gcd)
größter gemeinsamer Teiler zweier natürlicher Zahlen
(im Englischen: gcd = greatest common divisor)

Sei $n \in \mathbb{N}_0$ eine natürliche Zahl. Sei
$T(n) = \{d \in \mathbb{N}_0 \mid d \text{ teilt } n, \text{ d.h., es gibt ein } c \in \mathbb{N}_0 \text{ mit } c \cdot d = n\}$
die Menge der Teiler von n.

Wegen $1 \in T(n)$ und $n \in T(n)$ ist T(n) niemals die leere Menge.
Für $n>0$ gilt: Wenn $d \in T(n)$ ist, so ist $d \leq n$.
Speziell gilt: $T(0) = \mathbb{N}_0$.

Bilde zu zwei Zahlen $a, b \in \mathbb{N}_0$ den Durchschnitt $T(a) \cap T(b)$.
Diese Menge ist nicht leer, da sie mindestens die Zahl 1 enthält. Das maximale Element in diesem Durchschnitt heißt der größte gemeinsame Teiler von a und b, bezeichnet als ggT(a,b). Außer für a=b=0 ist er stets eindeutig bestimmt.

Für $a, b, d \in \mathbb{N}_0$ mit $a \neq 0$ oder $b \neq 0$ gilt also ggT(a,b) = d genau dann, wenn d sowohl a als auch b teilt und wenn für alle natürlichen Zahlen d', die sowohl a als auch b teilen, stets $d' \leq d$ gilt.

Zu a und b kann man ggT(a,b) daher berechnen, indem man alle Teiler von a und alle Teiler von b berechnet und hieraus die maximale Zahl ermittelt, die in beiden Teilermengen vorkommt.

Um die Teilbarkeit zu testen, verwendet man die Operation "modulo" (= Rest bei der ganzzahligen Division). Für zwei Zahlen x und y mit $y \neq 0$ gilt:
y ist genau dann ein Teiler von x, wenn (x mod y) = 0 ist.

2.4.3 Algoritmus

Setze eine Variable ggt anfangs auf den Wert 1.
Sei "min" das Minimum der Zahlen a und b,
dann prüfe für i = 2, 3, 4, ..., min,
ob i ein Teiler von b und ein Teiler von a ist;
trifft dies zu, dann setze jeweils ggt := i.

Am Ende besitzt ggt den Wert ggT(a,b), da bei dem aufsteigenden Testen das größte i, das beide Zahlen teilt, in der Variable ggt gespeichert wird.

Wir schreiben diesen Algorithmus "ggT_elementar" mit unseren Sprachelementen formal korrekt auf. Wir verwenden für die Variablen hier große Anfangsbuchstaben.


```
program ggT_elementar is
declare A, B, I, ggt, Min_a_b: Natural;
begin read (A); read (B);
      if B ≤ A then Min_a_b := B else Min_a_b := A fi;
      ggt := 1;
      for I := 2 to Min_a_b do
            if (A mod I = 0) and (B mod I = 0) then ggt:=I fi
      od;
      write (ggt)
end
```

Formulierung in Ada:

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure  ggT_elementar  is
A, B, I, Ggt, Min_a_b: Natural;
begin Get (A); Get (B);
        if B ≤ A then Min_a_b := B; else Min_a_b := A; end if;
        Ggt := 1;
        for I in 2 .. Min_a_b loop
            if (A mod I = 0) and (B mod I = 0) then Ggt:=I; end if;
        end loop;
        Put (Ggt);
end;
```


Im Falle a=b=0 gibt dieser Algorithmus jedoch den Wert 1 aus. Schreiben Sie den Algorithmus so um, dass er genau in diesem Fall den Wert 0 ausgibt.

Wie lange läuft dieser Algorithmus?
Offenbar wird die for-Schleife genau (Min_a_b - 1) Mal durchlaufen. Für große Zahlen a und b kann man diesen "elementaren ggT-Algorithmus" daher nicht verwenden.

Man erhält ein schneller arbeitendes Verfahren, *wenn man die Eigenschaften ausnutzt*, die der ggT(a,b) besitzt.

2.4.4. Eigenschaften des ggT:

(1)	ggT(a,b) = ggT(b,a)	für alle a,b∈ $\mathbb{N}_0$, "Symmetrie",
(2)	ggT(a,0) = ggT(a,a) = a	für alle a∈ $\mathbb{N}$,
(3)	ggT(a,b) = ggT(a-b, b) ggT(a,b) = ggT(a+b, b)	für alle a,b∈ $\mathbb{N}_0$ mit a≥b, für alle a,b∈ $\mathbb{N}_0$,
(4)	ggT(a,b) = ggT(b, a mod b)	für alle a,b∈ $\mathbb{N}_0$ mit a≥b.

Beweise: zu (1) und (2): Diese ersten beiden Zeilen folgen direkt aus der Definition und der Tatsache T(0) = $\mathbb{N}_0$.

zu (3): Wenn eine Zahl t sowohl a als auch b teilt, dann gibt es zwei Zahlen c_1 und c_2 mit $c_1 \cdot t = a$ und $c_2 \cdot t = b$. Hieraus folgt $a-b = c_1 \cdot t - c_2 \cdot t = (c_1-c_2) \cdot t$, d.h., t teilt auch a-b. Folglich teilt der ggT(a,b) auch die Zahl a-b (für $c_1-c_2 \geq 0$, d.h., für a≥b). Offenbar teilt t stets auch a+b (ersetze einfach "-" durch "+").

Eigenschaften des ggT:

(1)	ggT(a,b) = ggT(b,a)	für alle a,b∈ $\mathbb{N}_0$, "Symmetrie",
(2)	ggT(a,0) = ggT(a,a) = a	für alle a∈ $\mathbb{N}$,
(3)	ggT(a,b) = ggT(a-b, b) ggT(a,b) = ggT(a+b, b)	für alle a,b∈ $\mathbb{N}_0$ mit a≥b, für alle a,b∈ $\mathbb{N}_0$,
(4)	ggT(a,b) = ggT(b, a mod b)	für alle a,b∈ $\mathbb{N}_0$ mit a≥b.

Beweis zu (3), Fortsetzung:
Es sei d = ggT(a,b). Wenn d' = ggT(a-b, b) ist, so teilt d' die Zahl b und nach dem soeben Bewiesenen ist d' auch ein Teiler von (a-b)+b = a. Weil d der ggT(a,b) ist, muss daher d'≤d sein. Andererseits ist d (wie jeder Teiler von a und b) auch Teiler von b und a-b, woraus d≤d' folgt, da d' der ggT(a,a-b) ist. Aus beiden Ungleichungen folgt d'=d, d.h., ggT(a,b) = ggT(a-b, b). Analog folgert man ggT(a,b) = ggT(a+b, b).

Eigenschaften des ggT:

(1) ggT(a,b) = ggT(b,a) für alle a,b∈ $\mathbb{N}_0$, "Symmetrie",

(2) ggT(a,0) = ggT(a,a) = a für alle a∈ $\mathbb{N}$,

(3) ggT(a,b) = ggT(a-b, b) für alle a,b∈ $\mathbb{N}_0$ mit a≥b,
ggT(a,b) = ggT(a+b, b) für alle a,b∈ $\mathbb{N}_0$,

(4) ggT(a,b) = ggT(b, a mod b) für alle a,b∈ $\mathbb{N}_0$ mit a≥b.

Beweis zu (4): (hier: die Aussagen gelten auch für b = 0)
Iteriere die Gleichung (3), woraus (4) mit (1) unmittelbar folgt:
ggT(a,b) = ggT(a-b, b) für alle a,b∈ $\mathbb{N}_0$ mit a≥b,
ggT(a-b,b) = ggT(a-2·b, b) für alle a,b∈ $\mathbb{N}_0$ mit a-b≥b,
ggT(a-2·b, b) = ggT(a-3·b, b) für alle a,b∈ $\mathbb{N}_0$ mit a-2·b≥b
usw., bis man a-(k-1)·b≥b>a-k·b für k = a div b erreicht, also:
ggT(a-(a div b - 1)·b, b) = ggT(a-(a div b)·b, b) = ggT(a mod b, b)
für alle a,b∈ $\mathbb{N}_0$ mit a≥b>0.

Aus Eigenschaft (4) erhalten wir sofort einen Algorithmus zur Berechnung des ggT, wobei Eigenschaft (2) als Terminierungsbedingung dient. Halb umgangssprachliche Formulierung:

```
read(A); read(B);      -- falls A=B=0 ist, das Verfahren abbrechen
if A < B then "vertausche die Werte von A und B" fi;
while B ≠ 0 do
        nächster Wert von B wird A mod B;
        nächster Wert von A wird der Wert des alten B
od;
write(A)
```

Dieser Algorithmus endet, da wegen B > A mod B der Wert von B in jedem Schleifendurchlauf kleiner wird. Irgendwann wird daher B gleich 0 und die while-Schleife bricht ab.

Um zwei Werte auszutauschen oder einen "alten Wert" zuzuweisen, benötigt man Variablen zur Zwischenspeicherung.

Man kann "vertausche die Werte von A und B" nicht einfach durch A:=B; B:=A realisieren, da dann zwar A den Wert von B erhält, aber anschließend auch B diesen Wert bekommt, da der alte Wert von A ja bereits überschrieben wurde.

Wir verwenden also eine weitere Variable H, die im Programm sonst nicht benötigt wird, und setzen:

H := A; A := B; B := H

So gehen wir auch im Schleifenrumpf (= in der Anweisung zwischen do und od) vor.


```
program euklid1 is
declare A, B, H: natural;
begin read (A); read (B);
  if (A > 0) or (B > 0) then
     if A < B then H := A; A := B; B := H fi;
     while B ≠ 0 do
          H := A mod B; A := B; B:= H od
  fi;
  write (A)
end
```

Hinweis: Genau dann, wenn beide Eingabewerte 0 sind, wird der Wert 0 ausgegeben.

(Ende des Beispiels 2.4.2 ggT)

Festlegung 2.4.5: *Anordnung der elementaren Wertebereiche.*

In den gängigen Programmiersprachen sind alle elementaren Datentypen total angeordnet, d.h., für je zwei verschiedene Elemente a und b gilt entweder a < b oder b < a.

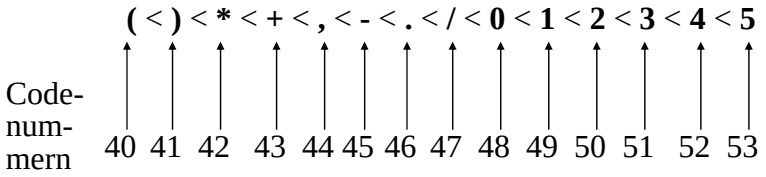
Bei Boolean setzt man false < true, bei ganzen und reellen Zahlen nutzt man die natürliche Anordnung auf der Zahlengeraden und bei Alphabetzeichen nimmt man die Anordnung ihrer Codierung. Wir legen daher fest, dass die Werte unserer fünf elementaren Datentypen (gemäß 2.4.1) in der genannten Weise angeordnet sind.

Die Alphabetzeichen **A** sind wie in 1.8.4 angegeben angeordnet.

2.4.6: ASCII-Code (ASCII = Abkürzung für "American Standard Code for Information Interchange" aus den 1950er Jahren).

Der Code in seiner alten Form umfasste $2^7 = 128$ Zeichen, von denen die 32 Zeichen für Zwecke der Steuerung im Computer verwendet wurden, 84 Zeichen waren fest mit Tastaturzeichen belegt und die verbleibenden 12 Zeichen konnten national genutzt werden (z.B. für Spezialzeichen wie ä, ö, ü, Ä, Ö, Ü, ß, [,]).

Die Anordnung der Alphabetzeichen erfolgte in der Reihenfolge der Codenummern jedes Zeichens. Beispielsweise gilt



ASCII-Code, erste Hälfte (nach ISO/IEC 646, vgl. DIN 66003)

Nr.	binär	Zeichen	Nr.	binär	Zeichen	Nr.	binär	Zeichen	Nr.	binär	Zeichen
0	0000000	NUL	16	0010000	DLE	32	0100000	Zwi.r	48	0110000	0
1	0000001	SOH	17	0010001	DC1	33	0100001	!	49	0110001	1
2	0000010	STX	18	0010010	DC2	34	0100010	"	50	0110010	2
3	0000011	ETX	19	0010011	DC3	35	0100011	#	51	0110011	3
4	0000100	EOT	20	0010100	DC4	36	0100100	$	52	0110100	4
5	0000101	ENQ	21	0010101	NAK	37	0100101	%	53	0110101	5
6	0000110	ACK	22	0010110	SYN	38	0100110	&	54	0110110	6
7	0000111	BEL	23	0010111	ETB	39	0100111	'	55	0110111	7
8	0001000	BS	24	0011000	CAN	40	0101000	(	56	0111000	8
9	0001001	HT	25	0011001	EM	41	0101001	)	57	0111001	9
10	0001010	LF	26	0011010	SUB	42	0101010	*	58	0111010	:
11	0001011	VT	27	0011011	ESC	43	0101011	+	59	0111011	;
12	0001100	FF	28	0011100	FS	44	0101100	,	60	0111100	<
13	0001101	CR	29	0011101	GS	45	0101101	-	61	0111101	=
14	0001110	SO	30	0011110	RS	46	0101110	.	62	0111110	>
15	0001111	SI	31	0011111	US	47	0101111	/	63	0111111	?

ASCII-Code, zweite Hälfte (nach ISO/IEC 646, vgl. DIN 66003)

Nr.	binär	Zeichen	Nr.	binär	Zeichen	Nr.	binär	Zeichen	Nr.	binär	Zeichen
64	1000000	@	80	1010000	P	96	1100000	`	112	1110000	p
65	1000001	A	81	1010001	Q	97	1100001	a	113	1110001	q
66	1000010	B	82	1010010	R	98	1100010	b	114	1110010	r
67	1000011	C	83	1010011	S	99	1100011	c	115	1110011	s
68	1000100	D	84	1010100	T	100	1100100	d	116	1110100	t
69	1000101	E	85	1010101	U	101	1100101	e	117	1110101	u
70	1000110	F	86	1010110	V	102	1100110	f	118	1110110	v
71	1000111	G	87	1010111	W	103	1100111	g	119	1110111	w
72	1001000	H	88	1011000	X	104	1101000	h	120	1111000	x
73	1001001	I	89	1011001	Y	105	1101001	i	121	1111001	y
74	1001010	J	90	1011010	Z	106	1101010	j	122	1111010	z
75	1001011	K	91	1011011	[	107	1101011	k	123	1111011	{
76	1001100	L	92	1011100	\	108	1101100	l	124	1111100	|
77	1001101	M	93	1011101	]	109	1101101	m	125	1111101	}
78	1001110	N	94	1011110	^	110	1101110	n	126	1111110	~
79	1001111	O	95	1011111	_	111	1101111	o	127	1111111	DEL

Meist schreibt man diesen Code als zweidimensionale Tabelle, siehe 1.8.4.

2.4.7: Codes mit 8 Binärstellen: EBCDIC-Code, Latin1.

Hat man eine Binärstelle mehr zur Verfügung, so lassen sich $2^8 = 256$ Zeichen darstellen. Der "extended binary coded decimal interchange code" EBCDIC verwendet ein Byte = 8 Binärstellen, von denen 64 Zeichen für die Steuerung reserviert sind. Die restlichen 192 Zeichen reichen aus, um alle westeuropäischen Schriftsprachen abzudecken (und damit auch alle in Amerika und Australien). Der Code "Latin1" ist eine Erweiterung der Codenummern 64 bis 127 des ASCII-Codes um 128 Zeichen des Dänischen, Spanischen, Tschechischen usw. Er kann leicht im EBCDIC beschrieben werden.

Zur genauen Definition und zu Erweiterungen des ASCII-Codes nach ISO 8859 siehe Literatur oder Vorlesungen der Technischen Informatik. In Ada kann mit den Funktionen Val und Pos zwischen Zeichen und Codenummern hin und hergeschaltet werden.

Es fehlen noch arabische, kaukasische, chinesische, afrikanische usw. Schriftzeichen sowie die vielen Symbole und Sonderzeichen der verschiedenen Wissenschaften, insbesondere der Mathematik und der Naturwissenschaften.

Bisher lag der Bedarf weltweit bei etwa 30.000 Zeichen. Um Platz für Erweiterungen zu haben, wurde der "Unicode" definiert, der aus 16 Binärstellen besteht, somit $2^{16} = 65536$ Zeichen beschreiben kann und alle bisher benutzten Zeichen einschl. alter Hieroglyphen umfasst (siehe ISO-Standard 10646). Dessen erste 256 Zeichen enthalten den Latin1-Code. Der Unicode besitzt noch Lücken, die erst künftig ausgefüllt werden.

Fazit: Man kann die Menge der Zeichen anordnen, speziell lassen sich die in Programmen verwendeten Tastaturzeichen anordnen, wobei deren Anordnung in Westeuropa und den USA wie im ASCII-Code erfolgt.

Die Darstellung der natürlichen Zahlen ($\mathbb{N}_0$).

Natürliche Zahlen werden in der Regel durch ein Stellenwertsystem beschrieben (erfunden in Indien im 7. Jahrhundert).

Definition 2.4.8: (vgl. 1.7.6)
Ein Stellenwertsystem ist ein Tripel $S = (b, Z, \beta)$ mit
(1) b ist eine ganze Zahl (die "Basis") mit $|b| \geq 2$,
(2) Z ist eine $|b|$-elementige Menge (die Menge der Ziffern),
(3) $\beta: Z \rightarrow \{0, 1, ..., |b|-1\} \subset \mathbb{N}_0$ ordnet jeder Ziffer umkehrbar eindeutig eine Zahl zwischen 0 und $|b|-1$ zu.

Eine Zahl wird beschrieben durch eine Folge von Ziffern aus Z ohne führende Nullen (außer der Null selbst). Dabei ergibt sich der Wert einer Ziffer z aus der Zahl $\beta(z)$ und aus der *Stelle*, an der die Ziffer in der Ziffernfolge steht (daher der Name "Stellenwertsystem"). In der Praxis ist $Z \subset \mathbb{A}$.
Beachten Sie: b darf negativ sein, auch wenn in der Praxis nur positive Zahlen verwendet werden.

Die Zahldarstellung $z = z_{n-1}z_{n-2} \ldots z_1z_0$ mit $n>0$ und $z_i \in Z$ für $i=0,1,...,n-1$ ohne führende Nullen (außer der Null selbst) bezeichnet die folgende Zahl $\phi(z)$

$$\phi(z) = \sum_{i=0}^{n-1} \beta(z_i) \cdot b^i$$

Hinweis:
Man muss genau zwischen der "Ziffer" $z \in Z$ und "zugehöriger Zahl" $\beta(z) \in \mathbb{Z}$ unterscheiden, auch wenn in der Praxis die Ziffer und die Zahl durch das gleiche Zeichen hingeschrieben werden. Auf den folgenden Folien werden wir daher die Ziffern mit fetten blauen Zeichen und die Zahlen mit normalen Zeichen darstellen. **4** ist also eine Ziffer und 4 die Zahl vier. Später werden wir diese Unterscheidung nicht mehr treffen.

Ist b = 10, so spricht man von einem Dezimalsystem.

Sei also (b, Z, β) ein Dezimalsystem, d.h., sei (b, Z, β)
mit b = 10, Z = {**0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**} und
β: Z → {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} ⊂ $\mathbb{N}_0$ mit
β(**0**) = 0, β(**1**) = 1, β(**2**) = 2, β(**3**) = 3, β(**4**) = 4,
β(**5**) = 5, β(**6**) = 6, β(**7**) = 7, β(**8**) = 8, β(**9**) = 9
ein Stellenwertsystem zur Basis 10.

Der Wert ϕ(z) der Zahldarstellung z = **37** ist daher
ϕ(**37**) = β(**3**)·10^1 + β(**7**)·10^0
= 3·10^1 + 7·10^0 = 30 + 7 = 37.

Der Wert ϕ(z) der Zahldarstellung z = **27188** lautet
ϕ(**27188**) = β(**2**)·10^4 + β(**7**)·10^3 +β(**1**)·10^2 +β(**8**)·10^1 +β(**8**)·10^0
= 2·10^4 + 7·10^3 +1·10^2 +8·10^1 +8·10^0 = 27188.

Ist b = 2, so spricht man von einem Dual- oder Binärsystem.

Sei (b, Z, β) ein Dualsystem, d.h., sei (b, Z, β) mit b = 2,
Z = {***0***, ***1***} und β: Z → {0, 1} ⊂ $\mathbb{N}_0$ mit β(***0***) = 0, β(***1***) = 1.
ein Stellenwertsystem zur Basis 2.

Der Wert ϕ(z) der Zahldarstellung z = ***101*** ist daher
ϕ(***101***) = β(***1***)·2^2 + β(***0***)·2^1 + β(***1***)·2^0
= 1·2^2 + 0·2^1 + 1·2^0 = 4 + 1 = 5.

Der Wert ϕ(z) der Zahldarstellung z = ***100101*** lautet
ϕ(***100101***)
= β(***1***)·2^5 + β(***0***)·2^4 + β(***0***)·2^3 + β(***1***)·2^2 +β(***0***)·2^1 +β(***1***)·2^0
= 32 + 4 +1 = 37.

Ist b = 8, so spricht man von einem Oktalsystem.

Sei (b, Z, β) ein Oktalsystem, d.h., sei (b, Z, β)
mit b = 8, Z = {**0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**}
und β: Z → {0, 1, 2, 3, 4, 5, 6, 7} ⊂ $\mathbb{N}_0$ mit
β(**0**) = 0, β(**1**) = 1, β(**2**) = 2, β(**3**) = 3,
β(**4**) = 4, β(**5**) = 5, β(**6**) = 6, β(**7**) = 7
ein Stellenwertsystem zur Basis 8.

Der Wert ϕ(z) der Zahldarstellung z = **45** ist daher
ϕ(**45**) = β(**4**)·8^1 + β(**5**)·8^0 = 4·8 + 5 = 37.

Der Wert ϕ(z) der Zahldarstellung z = **65064** lautet
ϕ(**65064**) = β(**6**)·8^4 + β(**5**)·8^3 + β(**0**)·8^2 +β(**6**)·8^1 + β(**4**)·8^0
= 6·4096 + 5·512 + 6·8 + 4 = 27188.

Ist b = 16, so spricht man von einem Hexadezimalsystem.

Sei also (b, Z, β) ein Hexadezimalsystem, d.h., sei (b, Z, β)
mit b = 16, Z = {**0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **A**, **B**, **C**, **D**, **E**, **F**}
β: Z → {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15} ⊂ $\mathbb{N}_0$
mit
β(**0**) = 0, β(**1**) = 1, β(**2**) = 2, β(**3**) = 3, β(**4**) = 4, β(**5**) = 5,
β(**6**) = 6, β(**7**) = 7, β(**8**) = 8, β(**9**) = 9, β(**A**) = 10, β(**B**) = 11,
β(**C**) = 12, β(**D**) = 13, β(**E**) = 14, β(**F**) = 15
ein Stellenwertsystem zur Basis 16.

Der Wert ϕ(z) der Zahldarstellung z = **25** ist daher
ϕ(**25**) = β(**2**)·16^1 + β(**5**)·16^0 = 2·16 + 5 = 37.

Der Wert ϕ(z) der Zahldarstellung z = **6A34** lautet
ϕ(**6A34**) = β(**6**)·16^3 + β(**A**)·16^2 + β(**3**)·16^1 + β(**4**)·16^0
= 6·4096 + 10·256 + 3·16 + 4 = 27188.

Hinweis 1: In der Regel benutzt man keine gesonderten Zeichen für Z, sondern nimmt die Ziffern von 0 bis |b|-1 (ab 10 fährt man mit den Buchstaben A, B, ... fort, siehe Hexadezimaldarstellung).

Hinweis 2: Als Basis verwendet man vor allem 10, 2 und 16. Kann es zu Verwechselungen kommen, welche Basis gemeint ist, so schließt man die Zahldarstellung in runde Klammern ein und setzt die Basis b tiefgestellt dahinter:

$(2101)_{\mathbf{3}}$ = "2101 zur Basis 3" = "2·27 + 1·9 + 1" = $(64)_{\mathbf{10}}$

$(2101)_{\mathbf{4}}$ = "2101 zur Basis 4" = "2·64 + 1·16 + 1" = $(145)_{\mathbf{10}}$

$(2101)_{\mathbf{5}}$ = "2101 zur Basis 5" = "2·125 + 1·25 + 1" = $(276)_{\mathbf{10}}$

Hinweis 3: Zu erwähnen sind noch die Darstellungen nach dem "chinesischen Restklassensatz". Wenn m paarweise teilerfremde natürliche Zahlen $q_1, q_2, ..., q_m$ mit $p = q_1 \cdot q_2 \cdot ... \cdot q_m$ gegeben sind, dann lässt sich jede Zahl a mit $0 \le a \le p-1$ *eindeutig* durch das m-Tupel $(r_1, r_2, ..., r_m)$ darstellen, wobei r_i der Rest von a bei der Division durch q_i ist (für $i=1, 2, ..., m$), also $r_i = a \bmod q_i$. Da man mit den Resten wie mit Zahlen (zyklisch) rechnen kann (siehe Anfang von 1.5) könnte dies bei umfangreichen computerinternen Berechnungen viel Zeit sparen:

Man überträgt die Eingaben in die m-Tupel, rechnet parallel auf allen Komponenten unabhängig voneinander viele Operationen in relativ kurzer Zeit durch und ermittelt am Ende aus den m-Tupeln die zugehörigen Ergebnisse. Die Decodierung aus den m-Tupeln zurück zu Zahlen kostet m Multiplikationen und m-1 Additionen, siehe Abschnitt 14.1.3. Zur Illustration folgt hier ein

Beispiel: m = 3, q_1 = **7**, q_2 = **11**, q_3 = **13**, p = 1001.

Die Zahl 400 wird dann durch

(400 mod 7, 400 mod 11, 400 mod 13) = (1,4,10)

repräsentiert; analog werden 10 und 40 durch (3,10,10) bzw. (5,7,1) dargestellt.

Dann ist **10+40** die durch

((3+5) mod 7, (10+7) mod 11, (10+1) mod 13) = (1,6,11)

dargestellte Zahl (nämlich 50) und **10·40** die durch

((3·5) mod 7, (10·7) mod 11, (10·1) mod 13) = (1,4,10)

dargestellte Zahl (nämlich 400, s.o.) und **40-10** die durch

((5-3) mod 7, (7-10) mod 11, (1-10) mod 13) = (2,8,4)

dargestellte Zahl (nämlich 30).

Der schnelleren Ausführung gewisser Operationen stehen auch Nachteile gegenüber, vor allem sind die natürliche Anordnung "<" der Zahlen und ein "Überlauf" (also das Verlassen des Zahlenbereichs von 0 bis p-1) nicht mehr direkt erkennbar.

Hinweis 4: Das Umschalten zwischen den Zahldarstellungen ist einfach. Wenn z eine Zahl und b eine Basis ist, so ist z mod b die letzte Ziffer von z zur Basis b und z div b die Zahl, die man aus z durch Wegstreichen der letzten Ziffer (zur Basis b) erhält. Dieses Vorgehen muss man nur iterieren und erhält die Ziffern von z zur Basis b in der Reihenfolge "von hinten nach vorne".

Beispiel: z sei die Zahl fünfhundertdreiundzwanzig, z = $(523)_{10}$. z soll zur Basis 7 dargestellt werden.

523 mod 7 = 5, 523 div 7 = 74,
74 mod 7 = 4, 74 div 7 = 10,
10 mod 7 = 3, 10 div 7 = 1,
1 mod 7 = 1, 1 div 7 = 0.

Also lautet die Darstellung von $(523)_{10}$ zur Basis 7: $(1345)_7$.

Übung: Schreiben Sie diese Umwandlung als Ada-Funktion.

Die Darstellung der ganzen Zahlen (**Z**).

2.4.9: *Möglichkeit 1:* Darstellung durch Betrag und Vorzeichen, d.h., wie bei den natürlichen Zahlen zuzüglich eines Vorzeichens "+" oder "-". Auf diese Weise lassen sich mit n Binärstellen die Zahlen von -2^{n-1} bis 2^{n-1} darstellen, wobei die Zahl Null zwei Darstellungen "+0" und "-0" erhält.

Möglichkeit 2: Im Falle der Binärdarstellung (Basis = 2) kann man die erste Stelle der Zahl zur Bildung der negativen Zahlen verwenden, indem man 2^{n-1} abzieht, sofern die erste Stelle eine "1" ist. Der Vorteil dieser zunächst eigenartig wirkenden Darstellung ist, dass die Addition und die Subtraktion mit dem gleichen Algorithmus durchgeführt werden können, während Möglichkeit 1 zwei Verfahren und Anpassungsoperationen benötigt, und dass die Null nur eine Darstellung besitzt. Der Nachteil ist eine Asymmetrie der dargestellten Zahlen.

Möglichkeit 2, Fortsetzung: Der Wert $\phi_2(z)$ einer n-stelligen binären Zahldarstellung $z = z_{n-1}z_{n-2} \ldots z_1z_0$ mit $n>0$ und $z_i \in \{0,1\}$ für i=0,1,...,n-1 ist also definiert als

$$\phi_2(z) = -\beta(z_{n-1})\cdot 2^{n-1} + \sum_{i=0}^{n-2} \beta(z_i)\cdot 2^i$$

2.4.10: Die Darstellung ϕ_2 heißt Zwei-Komplementdarstellung und wird oft für die Darstellung von Zahlen in den Registern und Speicherzellen von Computern verwendet.

Hierdurch werden die Zahlen von -2^{n-1} bis $2^{n-1}-1$ beschrieben. Ob eine Zahl negativ ist, erkennt man eindeutig an der ersten Stelle.

Beispiele zur Zwei-Komplementdarstellung:

n = 8, darstellbar sind die Zahlen von -128 bis +127.

0 1 1 1 1 1 1 1 ist die Zahl 127,

1 0 0 0 0 0 0 0 ist die Zahl -128,

1 1 1 1 1 1 1 1 ist die Zahl -1,

0 1 0 1 0 1 0 1 ist die Zahl 85,

1 0 1 0 1 0 1 0 ist die Zahl -86,

1 0 1 0 1 0 1 1 ist die Zahl -85.

Zu einer Zahl z erhält man -z, indem man jede Binärstelle der Darstellung von z komplementiert (also 0 durch 1 und 1 durch 0 ersetzt) und anschließend eine 1 addiert.

Hinweis 2.4.10 a:

Die folgende Darstellung $\phi_1(z)$

$$\phi_1(z) = -\beta(z_{n-1})\cdot(2^{n-1}-1) + \sum_{i=0}^{n-2} \beta(z_i)\cdot 2^i$$

heißt Eins-Komplementdarstellung und wird manchmal an Stelle der Zwei-Komplementdarstellung verwendet. Es gilt:

$\phi_1(z) = \phi_2(z)$ für $0 < z < (2^{n-1}-1)$ und
$\phi_1(z) = \phi_2(z-1)$ für $(-2^{n-1}+1) < z < 0$.
Die Zahl 0 wird hier sowohl durch die Binärdarstellung 000...0 als auch durch 111...1 dargestellt.
Untersuchen Sie: Wie läuft die Addition ab?

2.4.11: *Möglichkeit 3:* Verwende eine negative Zahl als Basis in einem Stellenwertsystem gemäß 2.4.8. *Hierdurch lassen sich alle ganzen Zahlen eindeutig darstellen*, sofern b negativ und $|b| \geq 2$ ist.

Beispiel: Stellenwertsystem zur Basis -2. Seien wie bisher $Z = \{\mathbf{0}, \mathbf{1}\}$ und $\beta(\mathbf{0}) = 0$, $\beta(\mathbf{1}) = 1$. Die ersten Zahldarstellungen:

z	ϕ(z)	z	ϕ(z)	z	ϕ(z)
0	0	**110**	2	**1100**	-4
1	1	**111**	3	**1101**	-3
10	-2	**1000**	-8	**1110**	-6
11	-1	**1001**	-7	**1111**	-5
100	4	**1010**	-10	**10000**	16
101	5	**1011**	-9	**10001**	17

In einem Stellenwertsystem mit negativer Basis gelten ungewöhnliche Regeln; zum Beispiel gilt im System zur Basis -2:

1 + **1** = **110**, **1** + **11** = **0**, **10** + **11** = **1101**,

11 + **11** = **10**, **111** + **111** = **11010**,

100 - **111** = **1** usw.

Wir arbeiten im Dezimalsystem, wobei der Übergang zum Dualsystem keine großen Schwierigkeiten bereitet, da die Operationen +, -, * und div sehr ähnlich bleiben. Geht man dagegen zu einer negativen Basis über, so kann man ebenfalls Algorithmen zur Durchführung dieser Operationen angeben, diese sind aber ungewohnt. Doch sie könnten sich als günstiger für künftige Computer herausstellen!? Daher muss man möglichst viele Darstellungen für elementare Datenbereiche kennen, untersuchen, erproben usw.

2.4.12: Die Darstellung der rationalen und der reellen Zahlen (**Q** und **IR**).

Rationale und reelle Zahlen werden meist als Dezimalzahlen mit (Dezimal-) Punkt und Nachkommastellen geschrieben. So ist 1/8 = 0.125 oder 2/3 = 0.666666... oder $\sqrt{2}$ = 1.414241... .

Die Darstellung zu einer Basis geschieht ebenso wie bei den natürlichen Zahlen, nur dass man nun auch negative Exponenten zulässt:

$z = z_{n-1}z_{n-2} \ldots z_1z_0 . z_{-1}z_{-2} \ldots z_{-k}$ .... bezeichnet also die Zahl

$$\phi(z) = \sum_{i=-\infty}^{n-1} \beta(z_i) \cdot b^i \qquad \text{(vgl. 2.4.8)}$$

Beispiel: $(101.1001)_2 = 1 \cdot 2^2 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-4} = (5.5625)_{10}$.

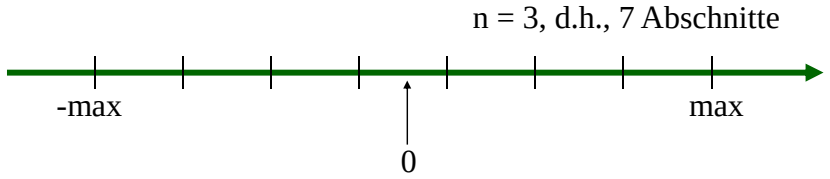
Darstellung in der Programmierung:

Zahlen werden in der Regel im Rechner mit 32, 64 oder 128 Binärstellen beschrieben, was für die meisten Anwendungen ausreicht. Hiermit kann man aber nur 2^{32} bzw. 2^{64} bzw. 2^{128} verschiedene Zahlen erfassen. Jedes endliche Intervall in **Q** und **IR** ist jedoch unendlich groß und fast alle hierin liegenden Zahlen lassen sich nicht durch eine beschränkte Zahl von Ziffern beschreiben. Folglich lassen sich fast alle rationalen und reellen Zahlen mit einer beschränkten Zahl an Stellen nur annähern.

Wie kann man ein rationales oder reelles Intervall von -max bis +max mit 2^n Zahlenwerten möglichst gut annähern (n=32, 64 oder 128)? Im Folgenden sei max die größte Zahl, die man darstellen möchte.

Wir betrachten einige naheliegende Möglichkeiten.

Fall 1a: gleichmäßige Aufteilung des Intervalls [-max, max].
Dieses Intervall wird in (2^n-1) Abschnitte der Länge $2\cdot max/(2^n-1)$ unterteilt:

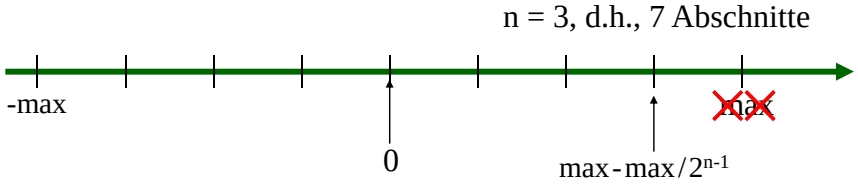


Die 2^n darstellbaren Zahlen sind dann:
$-max + i\cdot 2\cdot max/(2^n-1)$ für $i = 0, 1, 2, ..., 2^n-1$.

Nachteil: Die Zahl 0 ist nicht exakt darstellbar. Dieser Nachteil ist in der Praxis nicht akzeptabel.

Daher verzichtet man auf die Darstellung der Zahl max und endet bei $max - 2\cdot max/2^n$.

Fall 1b: gleichmäßige Aufteilung des Intervalls [-max, max).
Das Intervall von -max bis $max - 2\cdot max/2^n$ wird in 2^n-1 Abschnitte der Länge $2\cdot max/2^n = max/2^{n-1}$ unterteilt:



Die 2^n darstellbaren Zahlen sind dann:
$-max + i\cdot 2\cdot max/2^n$ für $i = 0, 1, 2, ..., 2^n-1$.

Hierbei wird die Null stets exakt dargestellt.

Der Vorteil ist, dass man die Darstellungen für die ganzen Zahlen auch für die Annäherung der reellen Zahlen verwenden kann; denn die Zahl
$-max + i\cdot 2\cdot max/2^n = (i-2^{n-1})\cdot max/2^{n-1} = j\cdot(max/2^{n-1})$
wird durch die ganze Zahl j eindeutig bestimmt; dabei läuft j von -2^{n-1} bis $+2^{n-1}-1$ (vgl. Zwei-Komplementdarstellung 2.4.10).

$max/2^{n-1}$ heißt Schrittweite. Sie ist kleinste Zahl in diesem System, um die sich zwei Zahlen unterscheiden.

Im Falle $max = 2^{n-1}$ erhält man die Schrittweite 1 und somit die ganzen Zahlen von -2^{n-1} bis $+2^{n-1}-1$.

In der Praxis wählt man max stets als Zweierpotenz, sodass $(max/2^{n-1}) = 1/2^d$ ist. Dadurch erhält jede darstellbare Zahl im Binärsystem die Form

<Vorzeichen> <Folge von n-1-d Nullen und Einsen> **.** <Folge von d Nullen und Einsen>

Man kann dann die reellen Zahlen als ganze Zahlen schreiben, wobei sich aber an einer festen Stelle (zwischen der d-t und der (d+1)-letzten Position) der Binärpunkt, also der Beginn der Nachkommastellen befindet. Dies führt zur

2.4.13: Festpunktdarstellung

Reelle Zahlen werden hierbei wie ganze Zahlen dargestellt, wobei ein gedachter Binärpunkt die Nachkommastellen bezeichnet. (Man kann auch eine andere Basis statt 2 nehmen.)

Die Zahl der Nachkommastellen d muss bei der Deklaration angegeben werden, z.B.:

```
declare Y: real binary fixed (d)      -- Darstellung mit Binärziffern
declare X: real decimal fixed (d);    -- Darstellung mit Dezimalziffern
```

In Ada verwendet man hierfür das Schlüsselwort delta (siehe 1.8.6) und gibt hiermit exakt den Wert $\max / 2^{n-1}$ und zusätzlich das Intervall durch seine Grenzen an. Jede Programmiersprache hat eigene Darstellungsmöglichkeiten.

Hypothetisches Beispiel:

declare U,V: real binary fixed (3);

```
U := 1011001;        -- U erhält die Zahl  11.125
V := 10101;          -- V erhält die Zahl  2.625
U := U + V;          -- U besitzt nun die Zahl 13.75
                        (binär:  1101.110)
```

Für Zahlen in U und V sind also die drei letzten Stellen Nachkommastellen.

Der Vorteil dieser Festpunktdarstellung ist, dass man wie mit ganzen Zahlen rechnen kann: Die Zahlen stellt man im Zwei-Komplement dar und die Operationen +, -, * lassen sich einfach im Computer realisieren. Anwendungen liegen z.B. im Bankenbereich, wo mit zwei Nachkommastellen gerechnet wird.

Dieses Vorgehen, das gesamte Intervall gleichmäßig mit 2^n Zahlen zu überstreichen, hat aber den Nachteil, dass der Bereich um die Zahl 0, in dem fast immer wesentlich genauer als in anderen Bereichen gerechnet werden muss, genau so grob aufgeteilt wird wie die weniger interessanten Bereiche in der Nähe der Intervallgrenzen. Man verwendet daher in der natur- und ingenieurwissenschaftlichen Praxis eine andere Darstellung für die rationalen bzw. die reellen Zahlen.

2.4.14 Fall 2: Gleitpunktdarstellung

Reelle Zahlen $z \neq 0$ werden hierbei in folgender Form dargestellt $z = m \cdot b^e$ mit $m \in \mathbb{R}$, $1/b \leq |m| < 1$, $b, e \in \mathbb{Z}$, $b \geq 2$.

Im Falle z = 0 setzen wir m = 0. In diesem Fall ist die Darstellung nicht eindeutig, da nun ein beliebiger Exponent e verwendet werden kann. (Dies stört aber im Weiteren nicht.)

m heißt Mantisse, e heißt Exponent von z bzgl. der Basis b.

Wir müssen uns davon überzeugen, dass die Definition der Gleitpunktdarstellung "sinnvoll" ist. Von einer solchen Darstellung werden wir verlangen, dass es sie immer gibt und dass sie für jede Basis und jede reelle Zahl eindeutig ist.

Den Fall z = 0 haben wir bereits betrachtet ($z=0 \Leftrightarrow m=0$).

Hilfssatz 2.4.15: Existenz und Eindeutigkeit

> Es sei $b \geq 2$ eine natürliche Zahl. Zu jeder reellen Zahl $z \neq 0$ gibt es genau ein $m \in \mathbb{R}$ mit $1/b \leq |m| < 1$ und genau ein $e \in \mathbb{Z}$ mit $z = m \cdot b^e$.

Beweis: Wir zeigen zunächst die Eindeutigkeit der Darstellung: Wenn für zwei von Null verschiedene Zahlen $z_1 = m_1 \cdot b^{e_1}$ und $z_2 = m_2 \cdot b^{e_2}$ gilt $z_1 = z_2$ (o.B.d.A. sei $e_1 \leq e_2$), so muss $m_1 \cdot b^{e_1} - m_2 \cdot b^{e_2} = (m_1 \cdot b^{e_1} - m_2 \cdot b^{e_2-e_1+e_1}) = (m_1 - m_2 \cdot b^{e_2-e_1}) \cdot b^{e_1} = 0$ sein. Wegen $b^{e_1} \neq 0$ muss $m_1 = m_2 \cdot b^{e_2-e_1}$ gelten.

Wäre nun $e_1 < e_2$, also $b^{e_2-e_1} \geq b$, so würde $1/b \leq |m_1| < 1$, aber $1 \leq |m_2| \cdot b^{e_2-e_1}$ gelten (wegen $1/b \leq |m_2|$). Dann können aber m_1 und $m_2 \cdot b^{e_2-e_1}$ nicht gleich sein.

Folglich muss $e_1 = e_2$ sein, woraus sofort $m_1 = m_2$ folgt. Also gilt $z_1 = z_2$ und die Darstellung ist folglich eindeutig.

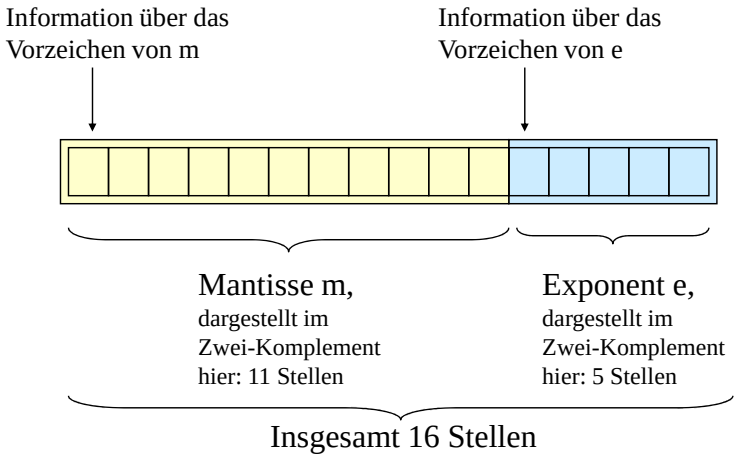
Fortsetzung des Beweises:

Wir müssen noch die Existenz einer solchen Darstellung zeigen. Betrachte |z|. Wegen $z \neq 0$ ist $|z| > 0$.

Falls $|z| \geq 1$, so bestimme die größte negative ganze Zahl e', für die $|z| \cdot b^{e'} < 1$ ist. (D.h., dividiere |z| ständig durch b, bis der Wert kleiner als 1 geworden ist.) Setze dann $m := z \cdot b^{e'}$. Mit z und $b^{e'}$ ist auch m eine reelle Zahl, und da e' maximal gewählt wurde, muss $1/b \leq |m| = |z| \cdot b^{e'} < 1$ gelten. Folglich existiert die Gleitpunktdarstellung $z = m \cdot b^{-e'}$ in diesem Fall.

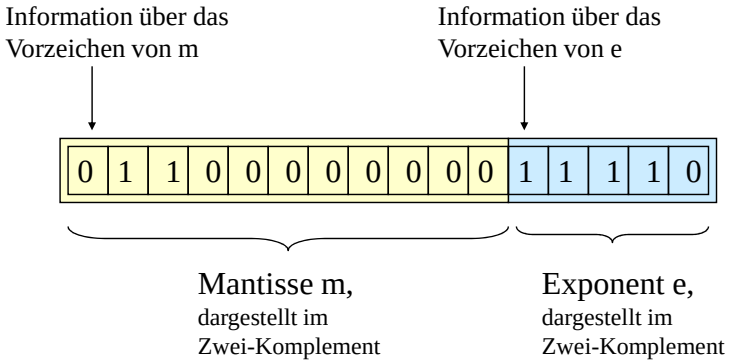
Falls $|z| < 1$ ist, so bestimme die kleinste natürliche Zahl $e' \geq 0$, für die $|z| \cdot b^{e'} < 1$, aber $|z| \cdot b^{e'+1} \geq 1$ ist. Setze $m := z \cdot b^{e'}$. Dann ist m eine reelle Zahl, und da e' minimal gewählt wurde, muss $1/b \leq |m| < 1$ gelten. Folglich existiert auch in diesem Fall die Gleitpunktdarstellung $z = m \cdot b^{e}$ (mit e := -e'). ■

Beispiel: Betrachte n = 16, m ist 11-stellig, e ist 5-stellig.



Beispiel: Betrachte n = 16, m ist 11-stellig, e ist 5-stellig.

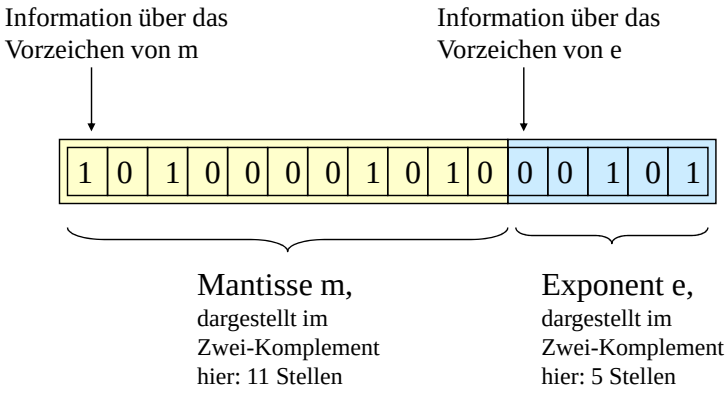
Sei b = 2 und $z = 0{,}1875 = 1/16 + 1/8 = (0{,}0011)_2$. Dann ist $m = (0.11)_2 = 0.75$ und $2^e = 1/4$, also e = -2. Gleitpunktdarstellung:



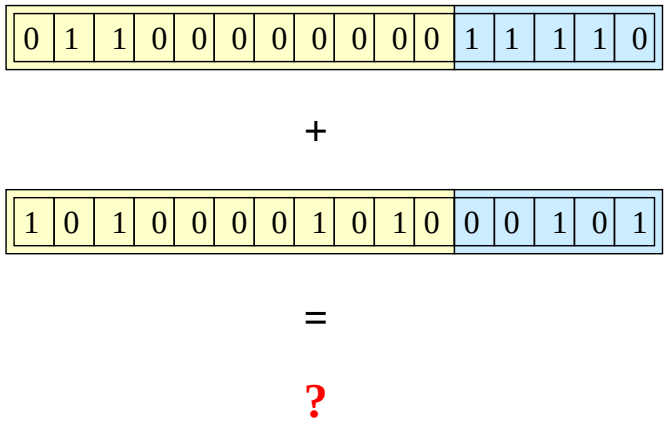
Proberechnen: $(0110...)_2 = 0.5+0.25 = 0.75$ ⇒ $z = 0.75 \cdot 2^{-2} = 0.1875$
Zwei-Komplement für e: -16+8+4+2 = -2

Beispiel: Betrachte n = 16, m ist 11-stellig, e ist 5-stellig, b=2.

Sei $z = -23.6875 = -(23 + 1/2 + 1/8 + 1/16) = -(10111.1011)_2$.
Dann ist $m = -(0.101111011)_2$ und $2^e = 32$, also e = 5. Das Zwei-Komplement zu $-(0.1011110110)_2$ liefert 1010001010 .



Was ist nun die Summe der beiden Zahlen: 0.1875 + (- 23.6875) ?



2.4.16: Addition zweier positiver reeller Zahlen

Betrachte zwei positive Zahlen z_1 und z_2 und ihre Gleitpunktdarstellungen $z_1 = m_1 \cdot b^{e_1}$ und $z_2 = m_2 \cdot b^{e_2}$.

Falls die Exponenten e_1 und e_2 verschieden sind, so wähle als Zahl z_2 die Zahl, deren Exponent der kleinere ist. Es gilt also $e_1 \geq e_2$ und folglich $m_2 \cdot b^{e_2-e_1} \leq m_2 < 1$.

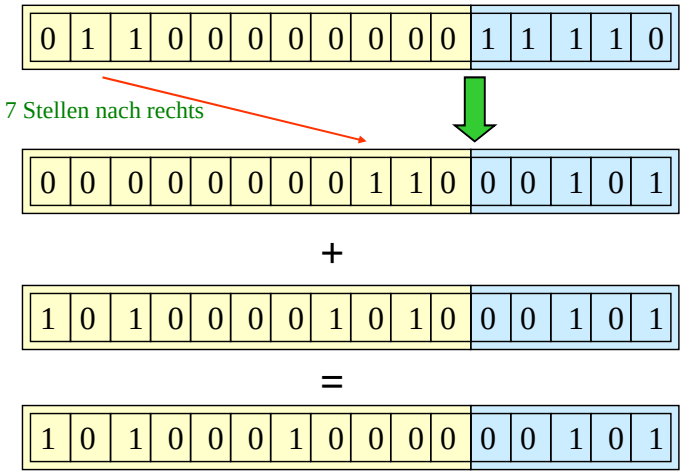
$$\begin{aligned} z_1 + z_2 &= m_1 \cdot b^{e_1} + m_2 \cdot b^{e_2} \\ &= m_1 \cdot b^{e_1} + m_2 \cdot b^{e_2-e_1+e_1} \\ &= (m_1 + m_2 \cdot b^{e_2-e_1}) \cdot b^{e_1} \end{aligned}$$

Also führt man die Addition folgendermaßen durch:

Verschiebe die Mantisse der kleineren Zahl um e_1-e_2 Stellen nach rechts (schiebe hierbei vorne Nullen nach und lasse die rechts rausgeschobenen Ziffern weg),
addiere die Mantissen: $m = m_1+m_2$,
falls $m > 1$ ist, verschiebe m um eine Stelle nach rechts (schiebe vorne eine Null nach und lass die rechts rausgeschobene Ziffer weg) und erhöhe e_1 um 1,
das Resultat der Addition ist die Zahl $m \cdot b^{e_1}$, d.h., die Zahl, deren Mantisse m und deren Exponent e_1 ist.

Liegen negative Zahlen vor, so muss man dieses Verfahren modifizieren.
Die Subtraktion lässt sich in ähnlicher Weise behandeln.
Beispiele siehe Übungen.

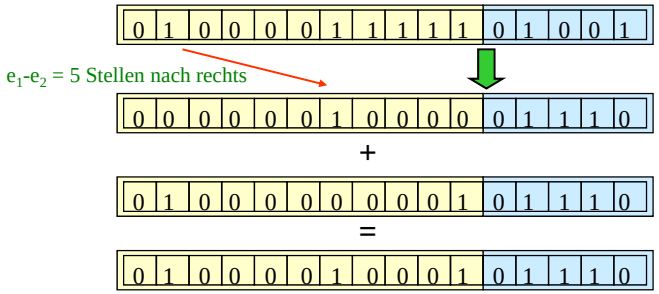
Summe der beiden Zahlen: (- 23.6875) + 0.1875:
$e_1 = 5$, $e_2 = -2$, also Mantisse um 7 Stellen nach rechts:



Dies ist die Zahl $(-1+1/4+1/64)*2^5 = -32+8+0.5 = -23.5$

Im allgemeinen Fall lässt sich eine Zahl nicht exakt darstellen oder bei der Rechts-Verschiebung verschwinden Stellen der Mantisse. Zahlen werden also immer nur angenähert im Rechner gespeichert. *Beispiel* 8207.3 + 271.5 = 8478.8:

$(1/2+1/1024)*2^{14} = 8208$, $e_1 = 14$. (Die Zahl 8207,3 muss durch 8208 angenähert werden, da ohne Vorzeichen nur 10 Mantissen-Stellen vorhanden sind.)
$(1/2+1/64+1/128+1/256+1/512+1/1024)*2^9 = 271.5$, $e_2 = 9$.



Dies ist die Zahl $(1/2+1/64+1/1024)*2^{14} = 8192+256+16 = 8464$.

Fazit: Reelle Zahlen können nur angenähert dargestellt werden. Rationale Zahlen werden nicht gesondert betrachtet, sondern werden entweder als reelle Zahlen aufgefasst oder müssen als eigener Datentyp Rational definiert werden (siehe Kap. 1.6).

Die Darstellung einer reellen Zahl r durch die am nächsten bei r liegende Zahl $m \cdot b^e$ bezeichnet man als Rundung. Die hierbei auftretende Differenz $|r - m \cdot b^e|$ nennt man Rundungsfehler.

Die Bezeichnung "Rundungsfehler" verwendet man auch für die Fehler, die im weiteren Verlauf von Berechnungen auftreten und die sich zu großen Zahlen aufschaukeln können.

Beispiel: Basis 2, Gesamtstellenzahl n = 7,
Mantissenlänge (mit Vorzeichen) = 4, Exponentenlänge = 3.

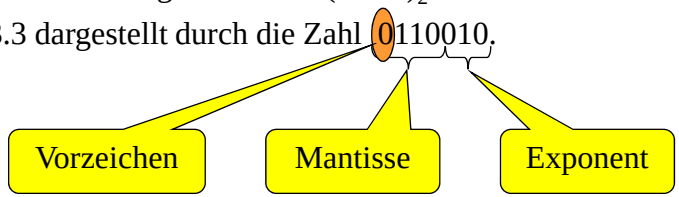
Betrachte die reellen Zahlen 3.3 und 0.07 .

Darstellung von $3.3 = 0.825 * 2^2$; $3.3 = (11.0100110011001...)_2$

Der Exponent ist somit 2.

Angenäherte Darstellung von 0.825: $(0.110)_2 = 0.75$.

Also wird 3.3 dargestellt durch die Zahl 0110010.



Beispiel: Basis 2, Gesamtstellenzahl n = 7,
Mantissenlänge (mit Vorzeichen) = 4, Exponentenlänge = 3.

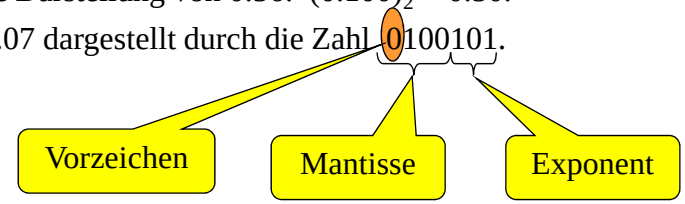
Betrachte die reellen Zahlen 3.3 und 0.07.

Darstellung von $0.07 = 0.56 * 2^{-3}$; $0.07 = (0.0001000111101...)_2$

Der Exponent ist somit $-3 = -(011)_2$; Zwei-Komplement: 101.

Angenäherte Darstellung von 0.56: $(0.100)_2 = 0.50$.

Also wird 0.07 dargestellt durch die Zahl 0100101.



Beispiel: Basis 2, Gesamtstellenzahl n = 7,
Mantissenlänge (mit Vorzeichen) = 4, Exponentenlänge = 3.

Betrachte die reellen Zahlen 3.3 und 0.07 .

Addiere nun 0110010 und 0100101 .

Die kleinere Zahl ist 0100101 mit dem Exponenten -3; die größere ist 0110010 mit dem Exponenten 2.

Schiebe daher 0100 um 2-(-3) = 5 Stellen nach rechts und schiebe vorne Nullen nach. Ergebnis: 0000101.

Addiere nun die Mantissen. Dies ergibt die Mantisse 0110. Überlauf links durch eine 1 liegt nicht vor.

Ergebnis ist also 0110010, d.h. die Zahl 3.0.

(Folglich wird bei dieser Darstellung 3.3 + 0.07 = 3.0)

Daher: Vorsicht beim Rechnen mit reellen Zahlen!

2.4.17: Die **Rundungsfehler** sinken mit der Anzahl der Stellen n und sie wachsen mit der Anzahl der Operationen, die während einer Berechnung angewendet werden.

Die besondere Warnung: Bei der Division durch betragsmäßig kleine Zahlen entstehen oft in nur wenigen Schritten schon sehr große Rundungsfehler.

Wer reelle Zahlen benutzt, sollte daher stets mit einer möglichst großen Stellenzahl n arbeiten. Die meisten Programmier-sprachen erlauben es, die Standarddarstellungen (meist n=32) auf n=64 oder n=128 usw. zu erhöhen. Beispiele siehe Vorlesungen über Mathematik, Numerik, Simulation usw.

2.4.18 Selbstdefinierter elementarer Datentyp

Wir vereinbaren nun noch, dass man einen neuen Datentyp im Deklarationsteil eines Programms mit Hilfe des Schlüssel-wortes "type" einführen darf in der Form:

type <Name des Datentyps> is (<Liste der Elemente>)

Die Reihenfolge in der Liste gibt zugleich die Anordnung der Elemente an.

Beispiele:

type Wochentage is (Mo, Di, Mi, Do, Fr, Sa, So);

type Farbe is (weiß, gelb, grün, rot, blau, schwarz);

type erste_zehn_Primzahlen is (2,3,5,7,11,13,17,19,23,29)

Welche Konstruktoren gibt es, um aus elementaren Datentypen kompliziertere Strukturen zu erhalten?

2.4.19: Hier folgt man den mathematischen Strukturen:
Unterbereiche, Intervalle (z.B. Einschränkung auf a..b),
kartesische Produkte (Datensätze, "record"),
n-faches Produkt einer Menge (Feld, Vektor, "array"),
(disjunkte) *Vereinigung* von Mengen (varianter record, union),
Potenzmenge ("set of "),
Funktionen zwischen Mengen (function, procedure),
Graphen, Relationen (realisiert durch Zeiger, pointer, "reference").

Wir betrachten hier nur kurz Unterbereiche und Felder. Mit den anderen Datentypen beschäftigen wir uns in Kapitel 3.

2.4.20 Unterbereiche

Wenn M eine angeordnete Menge ist und a und b zwei Elemente aus M sind, dann ist

a .. b = {x ∈ M | a ≤ x ≤ b}

der Unterbereich von a bis b der Menge M.

Gilt M ≠ ∅ und a ≤ b, so ist a .. b ≠ ∅.
Im Falle a > b ist a .. b = ∅.

Unterbereiche von Zahlen sind Intervalle, z.B.
17 .. 35 oder -23 .. -4 oder -23.6875 .. 0.1875 .

In den meisten Programmiersprachen sind nur Unterbereiche zulässig, die endlich sind; insbesondere erlaubt man meist keine Unterbereiche reeller Zahlen.

Zur Definition verwenden wir wieder das Schlüsselwort type.

Wenn man eine Menge selbst definiert, so wird die Reihenfolge in der Definition als Anordnung dieser Menge aufgefasst (siehe 2.3.18).

Vereinbare erneut folgende Menge:
type Wochentage is (Mo, Di, Mi, Do, Fr, Sa, So) ,
dann gilt Mo < Di < Mi < Do < Fr < Sa < So.

Unterbereiche hiervon können sein
type Arbeitstage is Mo .. Fr oder
type Ausgehtage is Fr .. Sa.

Ein anderes Beispiel lautet type Großbuchstaben is 'A' .. 'Z'
als Unterbereich des Datentyps Character.
(In Ada verwendet man subtype, wenn nur der Wertebereich eingeschränkt werden soll; man verwendet "type", wenn man einen neuen Typ einführen möchte. Vgl. 1.9.1.)

2.4.21 Felder

Sehr häufig benötigt man n Variablen der gleichen Art, auf die mit einem Index zugegriffen werden kann, vgl. Abschnitt 1.9.2. Diese fasst man unter einem Namen zu einem "Feld" (oder Vektor) zusammen.

Zur Angabe solcher Felder benötigt man den Datentyp, den jede Variable besitzen soll, und einen Bereich für den Index. Man verwendet das Schlüsselwort "array", um ein Feld zu bezeichnen, gibt dann den Indexbereich an (meist als Unterbereich) und fügt daran den gemeinsamen Datentyp der einzelnen Komponenten an, abgetrennt durch "of". Darstellung in einer Deklaration:

declare X: array (<Datentyp des Index>) of <Datentyp>.

Beispiel: Wir übernehmen den selbstdefinierten Datentyp:
type Wochentage is (Mo, Di, Mi, Do, Fr, Sa, So)

Hierzu bilden wir den Unterbereich:

type Arbeitstage is Mo .. Fr

und dann folgendes Feld aus fünf Elementen

type Arbeitsbeginn is array (Arbeitstage) of 0 .. 23

Variablen dieses Typs werden deklariert durch

declare X: Arbeitsbeginn;

Auf ihre Komponenten wird mittels X(i) zugegriffen. Dabei durchläuft i den Wertebereich des Datentyps Arbeitstage, d.h. i ∈ {Mo, Di, Mi, Do, Fr}.

Hinweis: In Ada wird der Index-Datentyp ebenfalls in runde Klammern eingeschlossen; oft verwendet man in Sprachen auch eckige Klammern bei Unterbereichen und bei Feldern. Runde Klammern erinnern daran, dass ein Feld eigentlich eine "Funktions-Tabelle" ist.

2.4.22 Skalarprodukt im $\mathbb{R}^n$

Im n-dimensionalen Vektorraum $\mathbb{R}^n$ werden Punkte durch ihren Koordinaten beschrieben, also durch den Vektor $(x_1,x_2,...,x_n)$. Der Abstand vom Nullpunkt ist dann die Wurzel aus dem Skalarprodukt mit sich selbst $x_1^2 + x_2^2 + ... + x_n^2$.

Allgemein ist das Skalarprodukt zweier Vektoren $(x_1,x_2,...,x_n)$ und $(y_1,y_2,...,y_n)$ definiert als $x_1 \cdot y_1 + x_2 \cdot y_2 + ... + x_n \cdot y_n$. Folgendes Programm berechnet dieses Skalarprodukt.

Wir nehmen an, die Zahl n, der Vektor x und der Vektor y stehen in dieser Reihenfolge in der Eingabe. Wir lesen diese Werte ein, berechnen dann das Skalarprodukt und geben es aus.


```
program skalarprodukt1 is

declare i, n: natural; s: real;
    x, y: array (1..n) of real;
begin  read(n);
    if n > 0 then
        for i := 1 to n do read (X(i)) od;
        for i := 1 to n do read (Y(i)) od;
        s := 0.0;
        for i := 1 to n do  s := s + X(i) * Y(i)  od;
        write(s)
    fi
end
```


Hinweis: Grundsätzlich strebt man in der Programmierung an, dass alle Werte, die an einer Stelle des Programms verwendet werden, bekannt sind, wenn man sich dort befindet.

Das Programm skalarprodukt1 erfüllt diese Forderung nicht, weil an der Stelle " x, y: array (1..n) of real; " der Wert von n noch nicht bekannt ist; denn er wird erst später eingelesen.

Daher lässt sich das Programm skalarprodukt1 nicht direkt in jede Programmiersprache übertragen. Meist muss ein *Block-konzept* vorhanden sein, das wir bereits in Abschnitt 1.11.5 vorgestellt haben. Obiges Programm ist also nicht korrekt, da zum Zeitpunkt der Deklaration von X und Y der Wert von n noch unbekannt ist.

In der Feld-Deklaration

array (<Datentyp des Index>) of <Datentyp>

darf der <Datentyp> der Komponenten erneut eine Felddeklaration sein:

```
array (<Datentyp des 1.Index>) of
            array (<Datentyp des 2.Index>) of <Datentyp>
```

Dies kürzt man ab, z.B. durch

array (<Datentyp des 1.Index, Datentyp des 2.Index>) of <Datentyp>

bzw. man schreibt bei Unterbereichen anstelle von

array (3 .. 25) of array (-4 .. 7) of real

die Deklaration:

array (3 .. 25, -4 .. 7) of real

und nennt dies ein zwei-dimensionales Feld. Wiederholt man dies, so lassen sich d-dimensionale Felder deklarieren, siehe 1.9.2. Man kann die Grenzen auch noch offen lassen, vgl. 1.9.3.

Abschnitt 2.5 soll Ihnen folgende Inhalte vermitteln:

Zu jedem Programm π gehört eine Abbildung f_π: E $\rightarrow$ A. Hierbei ist E die Menge der zulässigen Eingabedaten und A die Menge der Ausgabedaten des Programms. f_π heißt die von π realisierte Abbildung.

Zur Menge aller korrekten Programme gehört die Menge der von diesen Programmen realisierten Abbildungen $\wp$. Diese Menge beschreibt alles, was unsere Programme leisten können. In diesem Abschnitt lernen Sie einige Eigenschaften dieser Menge.

Bei diesen Präzisierungen lernen Sie zugleich die zugehörigen Formalismen kennen, insbesondere die Begriffe abzählbar und aufzählbar. Definiert werden die "Menge der Folgen" über einer Menge M (freies Monoid M*) und dort definierte Operationen ($\subseteq$, $\neq$, $\cup$, $\cap$, Komplement, Konkatenation $\circ$).

2.5 Realisierte Abbildungen

Programme verändern Daten, d.h., sie bilden Eingabedaten auf Ausgabedaten ab. Genauer: Sei π ein korrekt aufgebautes Programm und seien E und A die Mengen seiner Eingabedaten bzw. der Ausgabedaten. Dann ordnet π jeder Eingabe e die zugehörige Ausgabe a zu, sofern π bei Eingabe von e nach endlich vielen Schritten anhält; anderenfalls wird e die Ausgabe "undefiniert" zugeordnet.

Definition 2.5.1: Die von einem Programm π mit den Eingabe- und Ausgabemengen E und A realisierte (partielle) Abbildung f_π: E $\rightarrow$ A ist definiert durch

$$f_\pi(e) = \begin{cases} a, & \text{sofern } \pi \text{ bei Eingabe von e nach endlich vielen Schritten anhält und hierbei die Ausgabe a liefert,} \\ \perp \ (=\text{"undefiniert"}) & \text{sonst.} \end{cases}$$

Beispiel (= Beispiel 2.4.2):

```
program fakultaet1 is
declare ergebnis, i, n: natural;
begin   read (n);  ergebnis := 1;
        for i := 1 to n do ergebnis := ergebnis*i od;
        write (ergebnis)
end
```

Die Eingabemenge E ist hier $\mathbb{N}_0$ (n ist vom Typ "Natural"), die Ausgabemenge A ist ebenfalls $\mathbb{N}_0$. Das Programm "fakultaet1" terminiert für jede Eingabe und berechnet die Fakultät der Eingabe. Also lautet die realisierte Abbildung f_1: $\mathbb{N}_0 \rightarrow \mathbb{N}_0$

$$f_1(n) = n! = \begin{cases} 1 \cdot 2 \cdot \ldots \cdot n, & \text{für } n > 0, \\ 1, & \text{für } n = 0. \end{cases}$$

Beispiel ($\mathbb{A}$ ist die Menge der Zeichen, siehe 2.4.5 bis 2.4.7):

```
program q is
declare a1, a2, a3: character;
begin   read(a1); read(a2); read(a3);
        if a1 = a2 then a2 := a3 fi;
        a3 := a1; write(a1); write(a2); write(a3)
end
```

Die Eingabemenge E ist hier $\mathbb{A}^3$ (die Variablen a1, a2 und a3 sind vom Typ "character"), die Ausgabemenge A ist hier ebenfalls $\mathbb{A}^3$. Das Programm "q" terminiert für jede Eingabe und gibt stets drei Zeichen aus, von denen das erste und dritte übereinstimmen. Die realisierte Abbildung f_q: $\mathbb{A}^3 \rightarrow \mathbb{A}^3$ lautet:

$$f_q(x,y,z) = \begin{cases} (x,z,x), & \text{falls } x = y, \\ (x,y,x), & \text{falls } x \neq y. \end{cases}$$

Definition 2.5.2: Menge der berechenbaren Funktionen

$\wp_{E,A} = \{f: E \to A \mid$ Es gibt ein Programm π mit $f_\pi = f\}$

heißt die Menge der von Programmen realisierbaren oder berechenbaren Abbildungen (oder Funktionen) von E nach A.

Man beachte, dass man die Mengen E und A nicht beliebig wählen kann. Denn es dürfen nur solche Mengen benutzt werden, die als Eingabe- oder Ausgabemengen von Programmen zulässig sind.

Wie kann man diese Aussage präzisieren? Genau welche Mengen sind erlaubt? Wir müssen dies formalisieren. Das Formalisieren ist eine der wichtigsten Fähigkeiten, die Sie in dieser Vorlesung erlernen sollen.

"Formalisieren" bedeutet, Sachverhalte in eine präzise formale Schreibweise zu übertragen, die keine Mehrdeutigkeiten erlaubt, aber den Sachverhalt korrekt beschreibt; meist ist dies eine mathematische Schreibweise.

Im vorliegenden Fall muss man vor allem die zulässigen Ein- und Ausgaben präzisieren. Dies sind Mengen, die aus Folgen von Elementen aus den elementaren Wertebereichen bestehen, wobei die reellen Zahlen durch die Menge Ψ der (binär oder dezimal dargestellten) Zahlen mit endlich vielen Nachkommastellen ersetzt werden müssen.

Was ist die Menge von "Folgen von Elementen"? Dies präzisieren wir auf den nächsten Folien als "das freie Monoid über einer Menge". Was sind beim Programmieren die "Elemente"? Hierfür werden wir eine Menge $\mathbb{D}$ einführen.

2.5.3: Menge aller elementarer Daten in Programmen:

$\mathbb{D} = \mathbb{B} \cup \mathbb{A} \cup \mathbb{Z} \cup \Psi$ mit

$\Psi = \{ z \mid$ Es gibt natürliche Zahlen $k \geq 1$ und $n \geq 1$ und eine Basis $b \geq 2$, so dass $z = z_{n-1}z_{n-2} \ldots z_1z_0 . z_{-1}z_{-2} \ldots z_{-k}$ oder $z = -z_{n-1}z_{n-2} \ldots z_1z_0 . z_{-1}z_{-2} \ldots z_{-k}$ gilt und jedes z_i eine der Ziffern von 0 bis b-1 ist$\}$.

Alle Elemente in $\mathbb{D}$ sind genau voneinander zu unterscheiden, weil die Elemente der Mengen $\mathbb{B}$, $\mathbb{A}$, $\mathbb{Z}$ und Ψ paarweise verschieden dargestellt werden. Ist z.B. die ganze Zahl "zwei" gemeint, so wird sie als 2 (oder binär als 10) notiert, ist die reelle Zahl "zwei" gemeint, so stellen wir sie durch 2.0 dar. Ist dagegen die Ziffer "zwei" (also als Zeichen) gemeint, so schreiben wir '2'. (Grundsätzlich schließen wir einzelne Zeichen, um sie von Bezeichnern unterscheiden zu können, in Apostrophe ein.)

Die Menge

$\mathbb{D} = \{\text{false}, \text{true}\} \cup \mathbb{A} \cup \mathbb{Z} \cup \Psi$ mit

$\Psi = \{ z \mid$ Es gibt natürliche Zahlen $k \geq 1$ und $n \geq 1$ und eine Basis $b \geq 2$, so dass $z = z_{n-1}z_{n-2} \ldots z_1z_0 . z_{-1}z_{-2} \ldots z_{-k}$ oder $z = -z_{n-1}z_{n-2} \ldots z_1z_0 . z_{-1}z_{-2} \ldots z_{-k}$ gilt und jedes z_i eine der Ziffern von 0 bis b-1 ist$\}$.

ist eine abzählbar unendliche Menge.

(**Warum?** Können Sie eine Abzählung, also eine bijektive Abbildung zu den natürlichen Zahlen angeben? Beachten Sie: Für alle z ist die Länge der Darstellung endlich, aber nicht beschränkt.)

Die Eingabe in ein Programm ist nun nichts anderes als eine Folge von Elementen aus $\mathbb{D}$. Wir definieren die "Menge der Folgen".

Definition 2.5.4: Es sei M eine Menge. Die Menge aller endlichen Folgen von Elementen aus M

$M^* = \{a_1 a_2 \ldots a_n \mid n \geq 0 \text{ und } a_i \in M \text{ für } i = 1, 2, \ldots, n\}$

heißt Menge der Folgen oder Wortmenge oder Menge der Wörter oder freies Monoid über M. Die Elemente von M* heißen Wörter oder Folgen.

Für ein Wort $u = u_1 u_2 \ldots u_n$ heißt $n = |u|$ die Länge des Wortes. Das Wort der Länge 0 heißt leeres Wort und wird mit ε bezeichnet. (ε ist das Wort, für das n=0 in der obigen Definition von M* gilt.)

Auf M* ist die Operation "Konkatenation" (hier im Sinne von Hintereinanderschreiben) definiert:
Für zwei Wörter $u = u_1 u_2 \ldots u_n$ und $v = v_1 v_2 \ldots v_m$ ist
$u\,v = u_1 u_2 \ldots u_n v_1 v_2 \ldots v_m$.

Diese Operation ist assoziativ, d.h. für alle u, v, w ∈ M* gilt:
$u\,(v\,w) = (u\,v)\,w$.

Sie besitzt ε als Einheit, d.h., $u\,\varepsilon = \varepsilon\,u = u$ für alle u ∈ M*.
Offenbar gilt $|u\,v| = |u| + |v|$.

Speziell sei u^k (*u hoch k*) die k-fache Konkatenation eines Wortes u mit sich. Man kann dies auch induktiv definieren:
$u^0 = \varepsilon$ und $u^{k+1} = u\,u^k$ für alle $k \geq 0$.

D*, die Menge der Wörter über **D**, ist ebenfalls eine abzählbare Menge. (Warum? Beweisen Sie dies! Vgl. Anhang zu diesem Kapitel.)

In ein Programm werden Elemente vom Datentyp Boolean, Integer, Real oder Character eingegeben, also Elemente aus **D**. Ebenso werden solche Elemente aus **D** vom Programm ausgegeben. *Jedes* Programm π realisiert somit eine (partielle) Abbildung f_π: **D*** → **D***.

Kein Programm π schöpft hierbei ganz **D*** aus, vielmehr sind der Definitionsbereich von f_π und das Bild unter f_π meist eine der "üblichen Mengen", also etwa f_π: **A*** → **A*** oder f_π: $\mathbf{Z}^3 \to \mathbf{Z}^2$ usw.
Allgemein lässt sich dann die Menge aller von Programmen realisierbarer Abbildungen wie folgt definieren:

Definition 2.5.5:

$\wp = \{f \mid$ Es gibt ein Programm π mit $f = f_\pi$: **D*** → **D***$\}$

ist die Menge der von Programmen realisierbaren Abbildungen. Man nennt diese Menge auch die
Menge der von Programmen berechenbaren Funktionen oder die Menge der partiell rekursiven Funktionen (den Begriff "rekursive Funktion" verwendet man vor allem dann, wenn die Vor- und Nachbereiche Zahlenmengen sind).

Die Mengen aus Definition 2.5.2
$\wp_{E,A} = \{f: E \to A \mid$ Es gibt ein Programm π mit $f_\pi = f\}$
bilden Teilmengen von $\wp$, da alle Eingabe- und Ausgabemengen E und A Teilmengen von **D*** sind.

Die Menge $\wp$ gilt als eine der "großen Entdeckungen" des 20. Jahrhunderts. Sie bildet die größte Menge dessen, was man mit Programmen beschreiben und realisieren, bzw. was man mit Algorithmen erreichen kann. Sie hat interessante Eigenschaften.

Insbesondere ist die Menge $\wp$ abgeschlossen gegen gewisse Operationen. So gilt beispielsweise: Wenn $f, g \in \wp$ sind, dann ist auch die Hintereinanderausführung $f \circ g \in \wp$ [$f \circ g(x) := f(g(x))$ für alle x]. Wenn $f \in \wp$ ist, dann ist auch dessen "Iterierte" $f^i = \underbrace{f \circ f \circ f \circ \ldots \circ f}_{\text{i-mal}} \in \wp$.

Wenn $f, g \in \wp$ und b ein Boolescher Ausdruck sind, dann ist auch $\text{cond}(b,f,g) \in \wp$ mit $\text{cond}(b,f,g)(x) =$ if b(x) then f(x) else g(x) fi für alle Argumente x.

Der Rest von 2.5 kann übersprungen werden.

Die Menge **D*** gilt für alle Programme. Jedes Programm π hat jedoch genau eine (durch die Datentypen der Variablen in den Eingabeanweisungen) definierte Menge E_π seiner Eingabedaten und ebenso genau eine (durch den Datentyp der Variablen oder Ausdrücke in den Ausgabeanweisungen) definierte Menge von Ausgabedaten A_π; beide Mengen E_π und A_π sind Teilmengen von **D*** und nur mittels Vereinigungen und Hintereinanderschreiben aus den Mengen **IB**, **Z**, **Ψ** und **A** aufgebaut (beachte $\mathbb{N}_0 \subset \mathbb{Z}$).

Folglich gibt es zu jedem Programm π eine Eingabemenge E_π, die genau die Menge aller Folgen von Eingabedaten definiert, die von dem Programm π korrekt vollständig eingelesen werden. Wir sagen, das Programm π terminiert immer, wenn π für alle Eingabedaten aus E_π nach endlich vielen Schritten anhält (ggf. mit Fehlerabbruch).

Beispiel: Folgendes Programm ist immer terminierend:

```
program h is
declare  n, i: Integer;  a1, a2: character;
begin  read (a1);
       i := 1;
       if a1 = 'J'  then read (n); a2 := 'A'
                    else read (a2); n := 2 fi;
       while n > 0 do  i := i+i; n := n-1 od;
       write (a1); write (a2); write (i)
end
```

Die Eingabemenge $E_h \subset$ **D*** dieses Programms h lautet
$E_h =$ **AZ** $\cup$ **AA**,
d.h., korrekte Eingaben bestehen aus einem Zeichen gefolgt von einer ganzen Zahl oder aus zwei aufeinanderfolgenden Zeichen.

Analog lautet die Ausgabemenge A_h dieses Programms
$A_h =$ **AAZ**.

Beispiel, Fortsetzung:

Die Eingabemenge E_h ist zu unterscheiden von der Menge derjenigen Eingaben, für die das Programm ohne Abbruch arbeitet. Gibt man in obiges Programm beispielsweise 'J' 'J' ein, so bricht das Programm mit einem Fehler ab, da es nach Einlesen des Zeichens 'J' versucht, eine Zahl einzulesen, aber in der Eingabe statt dessen ein Zeichen vorfindet. Die Menge, für die h ohne Fehlerabbruch arbeitet, lautet:

$K_h = \{xy \mid x = \text{'J'} \in \mathbf{A} \text{ und } y \in \mathbf{Z}\} \cup \{xy \mid x,y \in \mathbf{A} \text{ und } x \neq \text{'J'}\}$
$= \{\text{'J'}\}\,\mathbf{Z} \cup (\mathbf{A} \setminus \{\text{'J'}\})\,\mathbf{A}$.

Diese Menge der fehlerfrei bearbeiteten Eingaben K_h lässt sich im Allgemeinen nicht exakt bestimmen (siehe später; diese Menge ist generell "nicht entscheidbar").

Geben Sie die realisierte Funktion f_h an! ■

Definition 2.5.6:

$\Re = \{f \mid \text{Es gibt ein Programm } \pi, \text{ das immer terminiert, mit } f = f_\pi\}$

ist die Menge der von stets anhaltenden Programmen realisierbaren Abbildungen. Man nennt diese Menge auch die Menge der von Programmen total berechenbaren Funktionen oder die Menge der total rekursiven Funktionen.

Die in $\Re$ liegenden Abbildungen sind total in dem Sinne, dass sie für jede Eingabe zu einem Ende kommen entweder durch einen Fehlerabbruch, weil die Datentypen bei der Eingabe nicht stimmen oder ein Fehler bei der Bearbeitung geschieht (z.B. Division durch 0), oder durch Erreichen des Endes des Programms.

Hinweis:

Diese Formalisierung der "realisierten Abbildung" erfolgte in den 1930er Jahren. Mit den ersten Arbeiten wurde auch gezeigt, dass es nicht-aufzählbare Mengen gibt.

Wir haben uns viel Zeit genommen, um die Formalisierung durchzuführen. Dieser Schritt ist für jede Übertragung eines Problems bzw. seiner Lösungsverfahren in ein Programm wichtig und ist insbesondere von der konkreten Programmiersprache unabhängig. Wir werden dieses Vorgehen noch öfter in dieser Veranstaltung anwenden und einüben.

Abschnitt 2.6 soll Ihnen folgende Inhalte vermitteln:

Ein Programm ist eine Folge von Zeichen, die gewissen Regeln gehorchen müssen. Eine Menge von Zeichenfolgen (Wörtern), die nach bestimmten Regeln aufgebaut sind, nennt man (formale) Sprache. Programme bilden daher eine Sprache, nämlich die "Programmiersprache".

Sie lernen die für "Sprachen" und einige ihrer Operationen erforderlichen Definitionen und ein paar typische Beispiele.

2.6 (Künstliche) Sprachen

Wir haben Algorithmen mit Hilfe von Programmen beschrieben. Jetzt wollen wir den zulässigen Aufbau von Programmen mit Hilfe eines Regelsystems beschreiben.

Programme bestehen aus einer Folge von Zeichen. Diese Zeichen sind:

(1) Die Elemente von **A**, die auf der Tastatur zu finden sind.

(2) Besondere Wörter ("Schlüsselwörter" der Programme) und zwar bisher die Elemente der folgenden Menge

SW = {procedure, is, begin, end, declare, if, then, else, fi, while, do, od, for, to, downto, repeat, until, read, write, natural, Boolean, integer, real, character, array, of, skip, halt, true, false, not, and, or, div, mod}.

(Man erkennt sie u.a. daran, dass sie unterstrichen werden.)

Ein Programm ist eine Zeichenfolge, wobei die Zeichen aus dem "Alphabet" $\mathbf{A} \cup SW$ stammen.

Ein Programm ist also ein Wort aus $(\mathbf{A} \cup SW)^*$.

Eine Programmiersprache L ist daher eine Teilmenge des freien Monoids, also $L \subseteq (\mathbf{A} \cup SW)^*$.

Man muss nun genau festlegen, welche Zeichenfolge ein Programm ist und welche nicht. Meist wird dies durch "Regeln" festgelegt, die wir in Abschnitt 1.10 bereits betrachtet haben.

Wir erläutern hier an dem Beispiel "Bezeichner" mehrere Möglichkeiten der präzisen Definition.

Als einführendes Beispiel untersuchen wir einen bekannten Fall, nämlich den Aufbau von "Bezeichnern". Bezeichner sind Zeichenketten über Buchstaben, Ziffern und dem Unterstrich "_". Seien also

$\Phi_B = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$
die Menge der Buchstaben und
$\Phi_Z = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ die Menge der Ziffern und
$\Phi = \Phi_B \cup \Phi_Z \cup \{\_\}$ die Menge der 63 Zeichen,
die für einen Bezeichner verwendet werden dürfen.

Definition 2.6.1 (umgangssprachlich): Ein Bezeichner ist eine Folge von Elementen aus Φ, die mit einem Buchstaben (d.h. mit einem Element aus Φ_B) beginnt und dann endlich viele Zeichen aus Φ besitzen darf.

Es folgt eine formalere Definition der Bezeichner.

2.6.2: *Induktive Definition:*

(1) Jedes Element aus Φ_B ist ein Bezeichner.

(2) Wenn w ein Bezeichner ist und $a \in \Phi$, dann ist auch w a ein Bezeichner.

(3) Nur Zeichenfolgen, die ausschließlich mit den Regeln (1) und (2) aufgebaut wurden, sind Bezeichner.

Zum Beispiel ist H z 4 ein Bezeichner.

2.6.3: In unserem einfachen Fall kann man die Menge der Bezeichner Bez auch *direkt* angeben:

$Bez = \{w \in \Phi^* \mid w = bv,\ b \in \Phi_B \text{ und } w \in \Phi^*\} = \Phi_B \Phi^*$.

Man kann die Menge der Bezeichner auch erzeugen lassen:

2.6.4: Es sei β ein neues Zeichen. Betrachte folgende *Regeln:*

$\beta \to \gamma$ für jedes $\gamma \in \Phi_B$,

$\beta \to \beta\,\alpha$ für jedes $\alpha \in \Phi$.

Eine Herleitung (oder "Ableitung") beginnt mit dem Zeichen β In jedem Schritt darf man ein Zeichen, das links von "$\to$" in der Regel steht, durch die davon rechts stehende Zeichenfolge ersetzen. (In unserem Fall haben nur das eine Zeichen β, das ersetzt werden darf.) Man leitet solange ab, bis das Zeichen β nicht mehr auftritt. (BNF: $\beta ::= \beta\,\alpha \mid \gamma$ für alle $\gamma \in \Phi_B$ und $\alpha \in \Phi$.)

Bez ist dann die Menge der Zeichenfolgen, die man aus β in endlich vielen Schritten herleiten (oder "ableiten") kann.

Zum Beispiel kann man $H\,z\,4$ wie folgt ableiten:

Beginne mit β und wende die Regel $\beta \to \beta\,\alpha$ speziell für $\alpha = 4$ an:

$\beta \to \beta\,4$ Wende die Regel erneut für $\alpha = z$ an:

$\beta \to \beta\,4 \to \beta\,z\,4$ Wende nun die Regel $\beta \to \gamma$ mit $\gamma = H$ an:

$\beta \to \beta\,4 \to \beta\,z\,4 \to H\,z\,4$

Da man $H\,z\,4$ auf diese Weise ableiten konnte, ist diese Zeichenfolge also ein Bezeichner.

Allgemein ist
Bez = $\{w \in \Phi^* \mid$ Es gibt eine Ableitung von β nach $w\}$.

Auf die Regeln und die Herleitung von Zeichenfolgen aus speziellen Zeichen, die im Laufe einer Herleitung verschwinden müssen, gehen wir in Abschnitt 2.7 ein.

Wir halten fest: Die uns interessierenden Mengen ("Sprachen") bestehen aus Zeichenfolgen, die über einer festen Menge (z.B. Φ oder $\mathbf{A} \cup SW$) gebildet werden. Wir definieren daher:

Definition 2.6.5: Eine endliche Menge $A = \{a_1, a_2, ..., a_n\}$, deren Elemente linear angeordnet sind ($a_1 < a_2 < ... < a_n$), heißt ein (endliches) Alphabet.
Jede Menge von Zeichenfolgen über A bezeichnen wir als Sprache über dem Alphabet A (engl.: "language").
D.h.: L ist eine Sprache über A genau dann, wenn $L \subseteq A^*$.

2.6.6 Operationen: Obige Definition hat den Vorteil, dass wir alle Operationen, die wir für Mengen und für Wortmengen kennen, auch für Sprachen nutzen können. Zum Beispiel:

Vereinigung *von Sprachen:* Wenn L_1 und L_2 Sprachen über dem gleichen Alphabet A sind, dann ist auch $L_1 \cup L_2$ Sprache über A.

Durchschnitt *von Sprachen:* Wenn L_1 und L_2 Sprachen über dem gleichen Alphabet A sind, dann ist auch $L_1 \cap L_2$ Sprache über A.

Komplement *von Sprachen:* Wenn L eine Sprache über A ist, dann ist auch das Komplement $A^* \backslash L = \overline{L}$ eine Sprache über A.

Konkatenation *("Verkettung") von Sprachen:* Sind L_1 und L_2 Sprachen über A, dann ist auch ihre Konkatenation $L_1 \circ L_2 = \{u\,v \mid u \in L_1, v \in L_2\}$ eine Sprache über A.
Man schreibt wie bei Wörtern kurz $L_1 L_2$ anstelle von $L_1 \circ L_2$.

2.6.6 Operationen (Fortsetzung)

Iteration *von Sprachen:* Wenn L eine Sprache über A ist, dann ist auch $L \circ L = LL$, also die Konkatenation von L mit sich, eine Sprache über A. Seien $L^0 = \{\varepsilon\}$ und L^i die i-fache Konkatenation von L mit sich, dann sei L^* die Vereinigung aller dieser Mengen L^i für $i \geq 0$ (bzw. L^+ für $i > 0$). *Formale Definition:*

$L^0 = \{\varepsilon\}$ *Beachte:* $L\{\varepsilon\} = L$, also $L\,L^0 = L^1 = L$.

$L^{i+1} = L\,L^i$ für alle $i \geq 0$

Die Iterierte von L oder *das von L erzeugte Untermonoid in A**

$L^* = \bigcup_{i \geq 0} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup L^4 \ldots.$

$L^+ = \bigcup_{i \geq 1} L^i = L^1 \cup L^2 \cup L^3 \cup L^4 \ldots.$ *Die von L erzeugte Unterhalbgruppe in A**

Es gilt: $L^* = L^+ \cup \{\varepsilon\}$.

Beispiele: Sei A = {0, 1} das zugrunde liegende Alphabet.

Sei K = {0}, dann ist $K^2 = KK = \{00\}$, $K^3 = \{000\}$ usw.,

$K^* = K^0 \cup K^1 \cup K^2 \cup K^3 \cup ... = \{\varepsilon, 0, 00, 000, ...\}$
$= \{0^i \mid i \geq 0\}$ und

$K^+ = \{0, 00, 000, ...\} = \{0^i \mid i > 0\}$.

Sei L = {ε, 0, 01}, dann ist $L^2 = \{\varepsilon, 0, 00, 01, 001, 010, 0101\}$,

$L^3 = \{\varepsilon, 0, 00, 01, 000, 001, 010, 0001, 0010, 0100, 0101,$
$00101, 01001, 01010, 010101\}$ usw.,

$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup ... =$?

Das lässt sich nicht mehr einfach hinschreiben! Jedes Wort aus L* muss man als Folge von Wörtern aus L darstellen können.

Gehört zum Beispiel 0100101000010 zu L*?

Ja, wegen der Zerlegung 01 0 01 01 0 0 0 01 0 ∈ L*

Beispiel 2.6.7: Alphabet A = {0, 1, 2}. Sei L die Menge aller Wörter über A, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Skizze, wie solche Wörter aussehen:

$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10}\ a_{11}\ a_{12}\ a_{13}\ a_{14}\ a_{15}\ a_{16}\ a_{17}$ ..... a_n =

.....0.....0.....2.....

Hier dürfen nur Nullen stehen

Hier dürfen nur Zweien stehen

Beispiel 2.6.7: Alphabet A = {0, 1, 2}. Sei L die Menge aller Wörter über A, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Skizze, wie solche Wörter aussehen:

$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10}\ a_{11}\ a_{12}\ a_{13}\ a_{14}\ a_{15}\ a_{16}\ a_{17}$ ..... a_n =

.....0.....0.....2.....

Hier dürfen nur Nullen stehen

Hier dürfen nur Zweien stehen

Beispiel 2.6.7: Alphabet A = {0, 1, 2}. Sei L die Menge aller Wörter über A, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Skizze, wie solche Wörter aussehen:

$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10}\ a_{11}\ a_{12}\ a_{13}\ a_{14}\ a_{15}\ a_{16}\ a_{17}$ ..... a_n =

.....0.....0.....2.....2.....

Hier dürfen nur Nullen stehen

Hier dürfen nur Zweien stehen

Folglich können diese Wörter nur die Form
000...00111....11222.....2 = $0^i1^j2^k$ mit i, j, k ≥ 0 besitzen.

Beispiel 2.6.7: Alphabet A = {0, 1, 2}. Sei L die Menge aller Wörter über A, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Es gilt also $L = \{0^i1^j2^k \mid i, j, k \geq 0\} \subset A^*$.

Mit Hilfe unserer Operationen kann man L auch wie folgt beschreiben:

$L = \{0\}^* \{1\}^* \{2\}^*$.

Wie sehen Regeln zur Ableitung der Wörter, die zu L gehören, aus?

Dies ist einfach. Finden Sie die Lösung selbst!
Wir geben eine Lösung auf der folgenden Folie an.

Beispiel 2.6.7: Alphabet A = {0, 1, 2}. Sei L die Menge aller Wörter über A, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Wir verwenden vier neue Zeichen B, C, R und S.
Die Regeln lauten (ε ist das leere Wort):

$S \to RBC$
$R \to \varepsilon$ $\quad R \to 0R$
$B \to \varepsilon$ $\quad B \to 1B$
$C \to \varepsilon$ $\quad C \to 2C$

Beispiel für die Ableitung des Wortes 0112:

$S \to RBC \to 0RBC \to 0BC \to 01BC \to 011BC$
$\to 011C \to 0112C \to 0112$

Beispiel 2.6.7: Alphabet A = {0, 1, 2}. Sei L die Menge aller Wörter über A, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Wir hätten auch folgende Regeln verwenden können, um genau die gleiche Menge L abzuleiten (R entfällt hierbei):

$S \to 0S$ $\quad S \to B$ $\quad B \to 1B$
$B \to C$ $\quad C \to 2C$ $\quad C \to \varepsilon$

Beispiel für die Ableitung des Wortes 0112 mit diesen Regeln:

$S \to 0S \to 0B \to 01B \to 011B \to 011C \to 0112C \to 0112$

Beispiel 2.6.7: Alphabet A = {0, 1, 2}. Sei L die Menge aller Wörter über A, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Es gilt $L = \{0^i1^j2^k \mid i, j, k \geq 0\} \subset A^*$.

Möchte man zusätzlich, dass die Zahl der Nullen und der Zweien gleich sein soll, so erhält man die Sprache
$L' = \{0^i1^j2^k \mid i, j, k \geq 0 \text{ mit } i = k\} \subset A^*$.

Aufgabe: Versuchen Sie, Regeln für die Ableitung genau der Wörter aus L' zu finden.

Weitere Beispiele in den Übungen.

Abschnitt 2.7 soll Ihnen folgende Inhalte vermitteln:

Um Sprachen, insbesondere Programmiersprachen beschreiben zu können, lernen Sie kontextfreie Grammatiken kennen, mit denen die zulässigen Wörter der Sprache aus einem Startsymbol abgeleitet werden. Sie werden vor allem mit dem Begriff der Ableitung vertraut gemacht.

Als Beispiel lernen Sie, wie man Bezeichner und einfache arithmetische Ausdrücke mit Grammatiken exakt beschreibt. Zugleich wird der Begriff der Eindeutigkeit herausgearbeitet.

Schließlich wird der allgemeine Grammatikbegriff vorgestellt. Am Ende des Abschnitts sollten Sie in der Lage sein, zu einfachen Sprachen eine Grammatik anzugeben.

2.7 Grammatiken

Definition 2.7.1: Eine kontextfreie Grammatik ist ein Viertupel $G = (V, \Sigma, P, S)$ mit

(1) V ist eine nicht-leere endliche Menge (die Menge der Nichtterminalzeichen oder Variablen),

(2) Σ ist eine nicht-leere endliche Menge (die Menge der Terminalzeichen) mit $V \cap \Sigma = \emptyset$,

(3) $S \in V$ ist ein Nichtterminalzeichen (das Startsymbol),

(4) $P \subset V \times (V \cup \Sigma)^*$ ist eine endliche Menge (die Menge der Regeln oder Produktionen).

(Englisch: contextfree grammar.)

Beispiel: Betrachte die Grammatik $G_1 = (V_1, \Sigma_1, P_1, S_1)$ mit $V_1 = \{S_1\}$, $\Sigma_1 = \{0, 1\}$, $P_1 = \{(S_1, 1), (S_1, S_1 0), (S_1, S_1 1)\}$.

Die Idee ist, ausgehend von dem Startsymbol S_1 schrittweise alle Nichtterminalzeichen (hier ist dies nur das Zeichen S_1) durch ihre rechten Seiten in P_1 zu ersetzen. Alle Wörter, die man auf diese Weise erhält und die kein Nichtterminalzeichen mehr enthalten, bilden die Sprache, die von dieser Grammatik erzeugt wird. Bei der Ersetzung ("Ableitung" oder "Herleitung") hat man Freiheiten, da man sich willkürlich für eine Regel entscheiden kann, sofern P_1 mehrere Regeln mit der gleichen linken Seite besitzt.

Betrachte unser Beispiel: Da S_1 die linke Seite von drei Regeln ist, hat man in jedem Schritt mehrere Auswahlmöglichkeiten.

Grammatik $G_1 = (V_1, \Sigma_1, P_1, S_1)$ mit
$V_1 = \{S_1\}$, $\Sigma_1 = \{0, 1\}$, $P_1 = \{(S_1, 1), (S_1, S_1 0), (S_1, S_1 1)\}$.

S_1 ersetzen durch 1 (fertig, denn das Wort 1 enthält kein Nichtterminalzeichen mehr). Andere Möglichkeiten:

S_1 ersetzen durch $S_1 0$, hierin S_1 wieder ersetzen durch $S_1 0$, so dass $S_1 00$ entsteht; hierin S_1 ersetzen durch 1; Ergebnis ist das Wort 100 (fertig, da 100 kein Nichtterminalzeichen enthält).

S_1 ersetzen durch $S_1 1$, hierin S_1 ersetzen durch $S_1 0$, so dass $S_1 01$ entsteht; hierin S_1 ersetzen durch $S_1 1$, wobei $S_1 101$ entsteht; hierin S_1 ersetzen durch 1; Ergebnis ist das Wort 1101 (fertig, da dieses Wort kein Nichtterminalzeichen mehr enthält).

Welche Wörter entstehen auf diese Weise aus S_1? (Schreiben Sie diese Menge hin.) ■

Um die Ersetzung der linken Seite A durch die rechte Seite w eines Paares $(A,w) \in P$ deutlicher hervor zu heben, schreibt man statt (A,w) im Allgemeinen $A \to w$.

$P_1 = \{(S_1, 1), (S_1, S_1 0), (S_1, S_1 1)\}$ schreibt man also folgendermaßen:

$$P_1 = \{S_1 \to 1,\ S_1 \to S_1 0,\ S_1 \to S_1 1\}.$$

Wir müssen nun den Ableitungsprozess genau definieren. Ein Wort xAy darf man durch das Wort xwy in einem Schritt ersetzen, sofern $A \to w$ eine Regel ist. Wiederholt man dies endlich oft (auch "keinmal"), so erhält man eine Ableitung.

Definition 2.7.2: Gegeben sei eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$. Die Regelmenge P definiert auf der Menge $(V \cup \Sigma)^*$ die "Ableitungsrelationen" $\Rightarrow$ und $\Rightarrow^*$:

(1) Es gilt $u \Rightarrow v$ genau dann, wenn man die Wörter u und v in der Form $u = xAy$, $v = xwy$ mit $x, y \in (V \cup \Sigma)^*$ und $(A,w) \in P$ schreiben kann. Man sagt: v ist aus u in einem Schritt ableitbar oder v lässt sich aus u in einem Schritt ableiten.

(2) Es gilt $u \Rightarrow^* v$ genau dann, wenn entweder $u = v$ ist oder wenn es Wörter $z_0, z_1, ..., z_k \in (V \cup \Sigma)^*$ für ein $k \geq 1$ gibt mit $u = z_0$, $v = z_k$, $z_i \Rightarrow z_{i+1}$ für alle $i = 0, 1, ..., k-1$. Man sagt dann, v ist aus u herleitbar oder ableitbar. (Die Zahl k heißt Länge der Ableitung.)

Hinweis: $\Rightarrow^*$ ist der so genannte "reflexive und transitive Abschluss" von $\Rightarrow$. (Englisch: Ableitung = derivation.)

Definition 2.7.3:
Die von einer kontextfreien Grammatik $G = (V, \Sigma, P, S)$ erzeugte Sprache (engl.: generated language) ist die Menge
$L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \} \subseteq \Sigma^*$.
Eine Sprache $L \subseteq \Sigma^*$ heißt kontextfreie Sprache, wenn es eine kontextfreie Grammatik G mit $L = L(G)$ gibt.

Die Nichtterminalzeichen dienen also dazu, dass der Ableitungsprozess durchgeführt werden kann; zur erzeugten Sprache zählen dagegen nur die Wörter, die ausschließlich aus Terminalzeichen bestehen.

Wir werden im Abschnitt 2.8 sehen, wie man Programmiersprachen mit Hilfe von kontextfreien Grammatiken erzeugt. Hier betrachten wir "Bezeichner" und "Ausdrücke".

Beispiel 2.7.4: Betrachte erneut die Grammatik
$G_1 = (V_1, \Sigma_1, P_1, S_1)$ mit $V_1 = \{S_1\}$, $\Sigma_1 = \{0, 1\}$ und
$P_1 = \{S_1 \to 1,\ S_1 \to S_1 0,\ S_1 \to S_1 1\}$.

Aus dem Startsymbol S_1 kann man mit der zweiten und der dritten Regel $S_1 0$ und $S_1 1$ ableiten, hieraus mit den gleichen Regeln $S_1 00$, $S_1 10$, $S_1 01$ und $S_1 11$, hieraus mit den gleichen Regeln $S_1 000$, $S_1 100$, $S_1 010$, $S_1 110$, $S_1 001$, $S_1 101$, $S_1 011$ und $S_1 111$ usw., also alle Wörter der Form $S_1 x$ für ein beliebiges Wort $x \in \Sigma^*$. Um das Nichtterminalzeichen zu entfernen, muss irgendwann die erste Regel verwendet werden, welche hieraus das Wort 1x für ein beliebiges Wort $x \in \Sigma^*$ herleitet.

Da andere Wörter nicht herleitbar sind, gilt:
$L(G_1) = \{1x \mid x \in \Sigma^*\} = \{1\}\Sigma^*$. ■

Beispiel 2.7.5: Menge der Bezeichner, vgl. Definition 2.6.1 und Regeln 2.6.4.

Erinnerung: Seien $\Phi_Z = \{0,1,2,3,4,5,6,7,8,9\}$,

$\Phi_B = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U,$
$V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u,$
$v, w, x, y, z\}$,

$\Phi = \Phi_B \cup \Phi_Z \cup \{\_\}$, dann ist die Menge der Bezeichner

Bez $= \{w \in \Phi^* \mid w = bv,\ b \in \Phi_B \text{ und } v \in \Phi^*\} = \Phi_B \Phi^*$.

Definiere die kontextfreie Grammatik $G_2 = (V_2, \Sigma_2, P_2, S_2)$ mit
$V_2 = \{S_2, S_2', S_2''\}$ und $\Sigma_2 = \Phi$.
P_2 bestehe aus folgenden 66 Regeln:

P_2 =

$\{ S_2' \to A,\ S_2' \to B,\ S_2' \to C,\ S_2' \to D,\ S_2' \to E,\ S_2' \to F,\ S_2' \to G,$
$S_2' \to H,\ S_2' \to I,\ S_2' \to J,\ S_2' \to K,\ S_2' \to L,\ S_2' \to M,\ S_2' \to N,$
$S_2' \to O,\ S_2' \to P,\ S_2' \to Q,\ S_2' \to R,\ S_2' \to S,\ S_2' \to T,\ S_2' \to U,$
$S_2' \to V,\ S_2' \to W,\ S_2' \to X,\ S_2' \to Y,\ S_2' \to Z,\ S_2' \to a,\ S_2' \to b,$
$S_2' \to c,\ S_2' \to d,\ S_2' \to e,\ S_2' \to f,\ S_2' \to g,\ S_2' \to h,\ S_2' \to i,$
$S_2' \to j,\ S_2' \to k,\ S_2' \to l,\ S_2' \to m,\ S_2' \to n,\ S_2' \to o,\ S_2' \to p,$
$S_2' \to q,\ S_2' \to r,\ S_2' \to s,\ S_2' \to t,\ S_2' \to u,\ S_2' \to v,\ S_2' \to w,$
$S_2' \to x,\ S_2' \to y,\ S_2' \to z,$
$S_2'' \to 0,\ S_2'' \to 1,\ S_2'' \to 2,\ S_2'' \to 3,\ S_2'' \to 4,\ S_2'' \to 5,$
$S_2'' \to 6,\ S_2'' \to 7,\ S_2'' \to 8,\ S_2'' \to 9,\ S_2'' \to \_$
$S_2 \to S_2',\quad S_2 \to S_2 S_2',\quad S_2 \to S_2 S_2'' \}$

Es gilt dann:

Aus S_2' sind in einem Schritt genau alle Buchstaben ableitbar, d.h.: $S_2' \Rightarrow y$ genau dann, wenn $y \in \Phi_B$.
Aus S_2'' sind in einem Schritt genau alle Ziffern und der Unterstrich ableitbar, d.h.: $S_2' \Rightarrow y$ genau dann, wenn $y \in \Phi_Z \cup \{\_\}$.

Mit den letzten beiden Regeln sind aus S_2 genau alle Wörter der Form $S_2 x$ mit $x \in \{S_2', S_2''\}^*$ ableitbar.

Aus S_2 sind mit den letzten drei Regeln genau alle Wörter der Form $S_2' x$ mit $x \in \{S_2', S_2''\}^*$ ableitbar.

Da aus S_2' nur Buchstaben und aus S_2'' nur die Ziffern und der Unterstrich ableitbar sind, so sind folglich aus S_2 genau alle Wörter der Form zx mit $z \in \Phi_B$ und $x \in \Phi^*$ ableitbar, d.h.:
$L(G_2) = \{zx \mid z \in \Phi_B \text{ und } x \in \Phi^*\} = \Phi_B \Phi^* =$ Bez. ■

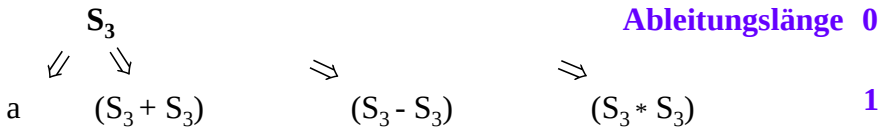
Beispiel 2.7.6: Einfache arithmetische Ausdrücke

Definiere die kontextfreie Grammatik $G_3 = (V_3, \Sigma_3, P_3, S_3)$ mit
$V_3 = \{S_3\}$ und $\Sigma_3 = \{$ **(**, **)**, **a**, **+**, **-**, ***** $\}$.
P_3 bestehe aus folgenden vier Regeln:

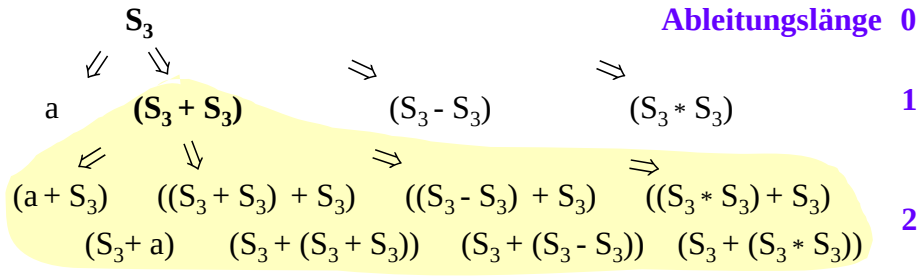
$S_3 \to \mathbf{a},\quad S_3 \to \mathbf{(}S_3 \mathbf{+} S_3\mathbf{)},\quad S_3 \to \mathbf{(}S_3 \mathbf{-} S_3\mathbf{)},\quad S_3 \to \mathbf{(}S_3 \mathbf{*} S_3\mathbf{)}$

Die Sprache $L(G_3)$ lässt sich nun nicht mehr in einfacher Form angeben. Man erkennt jedoch, dass sich alle *vollständig geklammerten arithmetischen Ausdrücke, die nur den Operanden "a" enthalten*, aus S_3 ableiten lassen, z.B.:

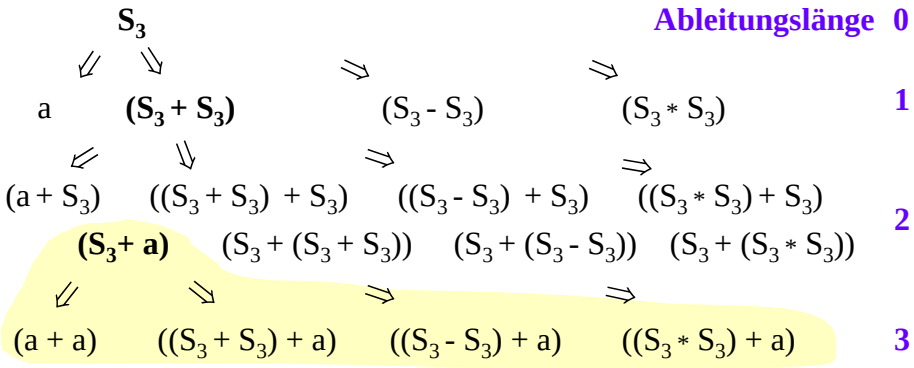
$S_3 \Rightarrow \mathbf{a}$, $\quad S_3 \Rightarrow \mathbf{(}S_3 \mathbf{+} S_3\mathbf{)} \Rightarrow \mathbf{(a + }S_3\mathbf{)} \Rightarrow \mathbf{(a + a)}$,
$S_3 \Rightarrow \mathbf{(}S_3 \mathbf{+} S_3\mathbf{)} \Rightarrow \mathbf{(a + }S_3\mathbf{)} \Rightarrow \mathbf{(a + (}S_3 \mathbf{*} S_3\mathbf{))} \Rightarrow \mathbf{(a + (}S_3 \mathbf{*} \mathbf{a))}$
$\Rightarrow \mathbf{(a + ((}S_3 \mathbf{-} S_3\mathbf{) *} \mathbf{a))} \Rightarrow \mathbf{(a + ((a -} S_3\mathbf{) *} \mathbf{a))} \Rightarrow \mathbf{(a + ((a - a) *} \mathbf{a))}$.



Regeln: $S_3 \to a$, $S_3 \to (S_3 + S_3)$, $S_3 \to (S_3 - S_3)$, $S_3 \to (S_3 * S_3)$



Regeln: $S_3 \to a$, $S_3 \to (S_3 + S_3)$, $S_3 \to (S_3 - S_3)$, $S_3 \to (S_3 * S_3)$



Überprüfen Sie die Ableitung von Wörtern aus $(V_3 \cup \Sigma_3)^*$,

4 verschiedene Wörter werden aus S_3 in genau einem Schritt abgleitet,

24 verschiedene Wörter werden aus S_3 in genau 2 Schritten abgleitet,

230 verschiedene Wörter werden aus S_3 in genau 3 Schritten abgleitet,

mehr als 1000 verschiedene Wörter werden aus S_3 in 4 Schritten abgeleitet.

Viele Wörter lassen sich auf verschiedene Art herleiten (z.B. ?).

Betrachten Sie zum Beispiel das Wort **(a + (a - a))**.

Ableitung 1:

$S_3 \Rightarrow$ **$(S_3 + S_3) \Rightarrow (a + S_3) \Rightarrow (a + (S_3 - S_3)) \Rightarrow (a + (a - S_3))$**

$\Rightarrow$ **$(a + (a - a))$** .

Ableitung 2:

$S_3 \Rightarrow$ **$(S_3 + S_3) \Rightarrow (S_3 + (S_3 - S_3)) \Rightarrow (S_3 + (S_3 - a)) \Rightarrow (S_3 + (a - a))$**

$\Rightarrow$ **$(a + (a - a))$** .

In Ableitung 1 wurde immer das am weitesten links im Wort stehende Nichtterminalzeichen ersetzt; in Ableitung 2 ist es das am weitesten rechts stehende. Bei Ableitungen hat man also Wahlfreiheiten. (Würde man aber die beiden Ableitungen wirklich als "wesentlich verschieden" auffassen? Wir werden dies präzisieren und anschließend sehen, dass dann diese beiden speziellen Ableitungen als gleich angesehen werden, vgl. 2.7.10.)

■

Aus diesem Beispiel erkennen wir:

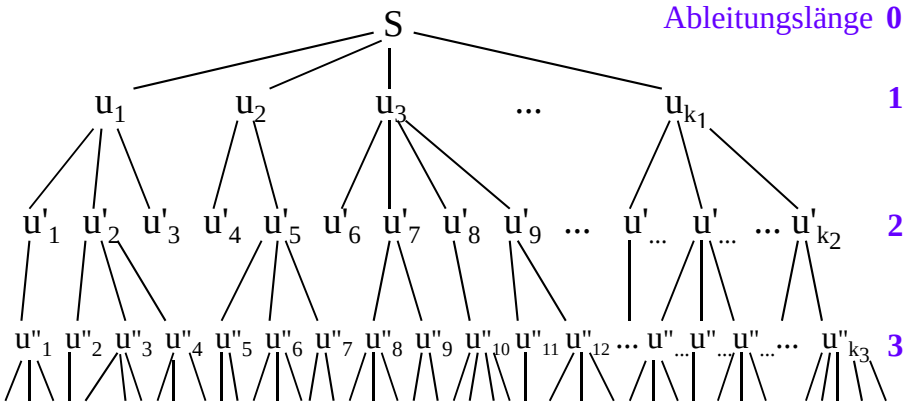
(1.) Es lassen sich alle Ableitungen aus dem Startsymbol S einer kontextfreien Grammatik entsprechend ihrer Länge grafisch gut als sog. "Baum" veranschaulichen.

(2.) Hierin ist jede einzelne Ableitung eines Wortes als ein Pfad enthalten, bei dem sich in jedem Schritt die Länge der Ableitung um eins erhöht. Zum Beispiel $S_3 \Rightarrow (S_3 + S_3) \Rightarrow (S_3 + \mathbf{a}) \Rightarrow ...$ (mit der Ableitungslängen 0, 1, 2, ...). Auch jede einzelne Ableitung lässt sich als ein Baum veranschaulichen (2.7.9).

Beide Aspekte werden wir nun genauer beleuchten und insbesondere die Darstellung als Baum herausarbeiten.

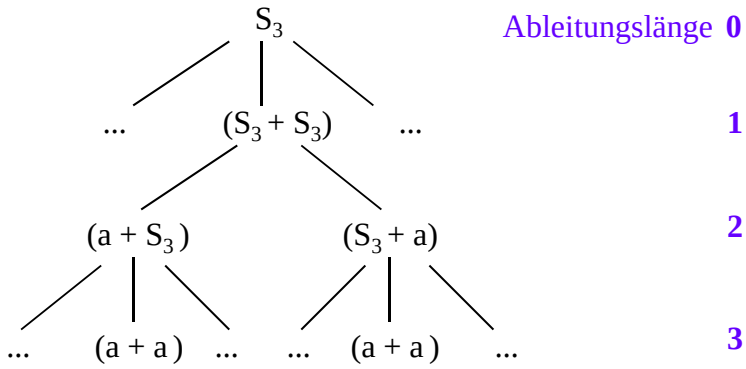
Aspekt (1.): Aus Beispiel 2.7.6 erkennen wir folgendes.

Alle Ableitungen aus dem Startsymbol S einer kontextfreien Grammatik, d.h., die Herleitung aller möglicher Wörter, lassen sich gleichzeitig in folgender Form schreiben:

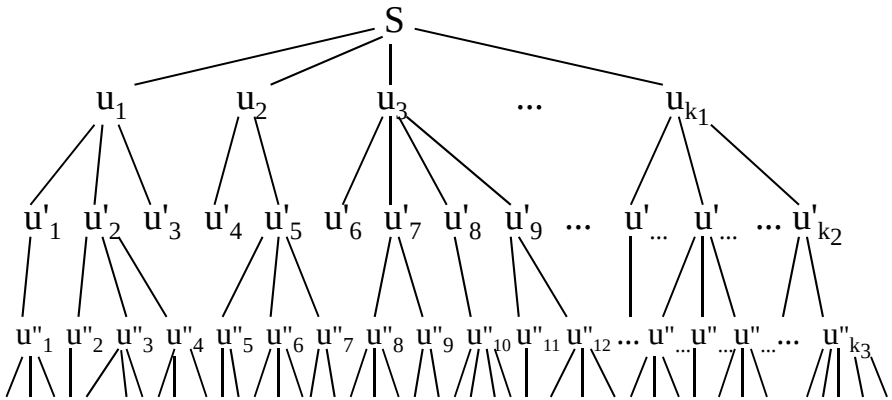


Die Menge der hergeleiteten Wörter, die nur Terminalzeichen enthalten, bilden hierbei die erzeugte Sprache $L(G_3)$.

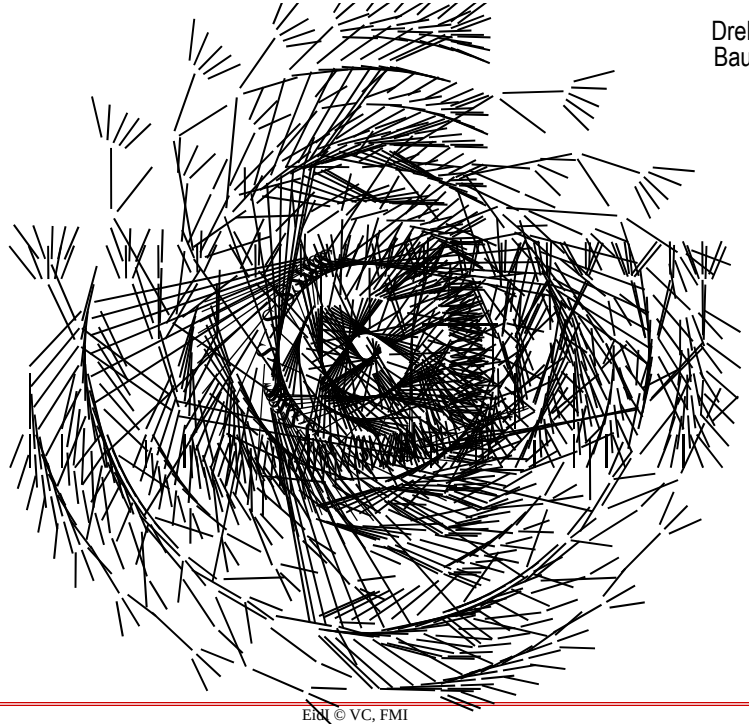
In dieser Menge aller Ableitungen können Wörter mehrfach (sogar unendlich oft) vorkommen. Zum Beispiel in obiger Grammatik G_3:

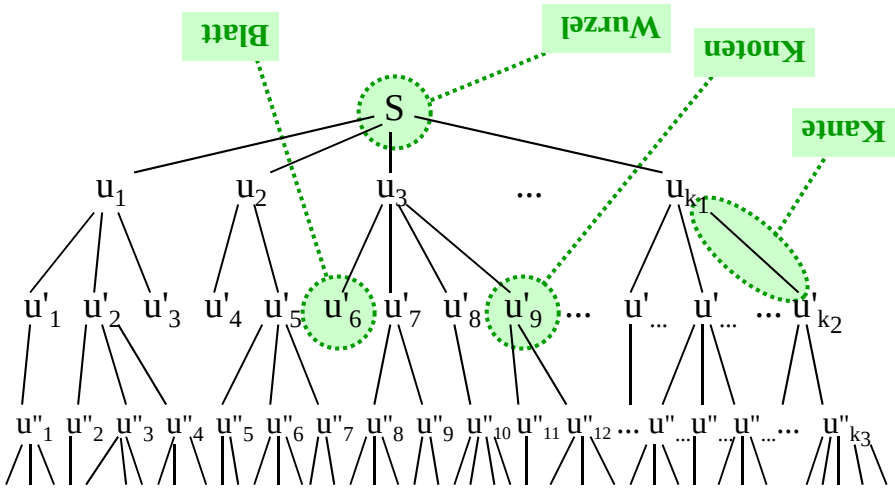


Bemerkung: Solch ein Gebilde, bei dem von jedem Punkt aus mehrere Linien abgehen, nirgends aber Linien zusammenlaufen können, nennt man einen Baum (engl.: tree). Das Gebilde sieht ja auch baumartig aus:



Allerdings wächst dieser Baum nach unten statt nach oben.





Einige Bezeichnungen bei Bäumen (hier umgedreht dargestellt):

Bezeichnungen 2.7.7 bei "Bäumen":

Bei den Linien spricht auch von Zweigen, meist aber von Kanten (engl.: edges);

eine Folge aneinander nach unten sich anschließender Kanten nennt man einen Pfad (oder einen Weg, engl.: path),

die Stellen, an denen die Wörter stehen und von denen Kanten ausgehen, heißen Knoten (engl.: nodes);

ein Knoten, von dem keine Zweige mehr weiterführen, heißt Blatt (engl.: leaf);

der Knoten, an dem alle Pfade beginnen, heißt die Wurzel des Baums (engl.: root).

Feststellung 2.7.8: Alle Ableitungen einer kontextfreien Grammatik zusammen bilden einen (i.A. unendlich großen) Baum.

In der Wurzel des Baumes steht das Startsymbol S.

Jeder Ableitung

$$u = z_0 \Rightarrow z_1 \Rightarrow z_2 \Rightarrow z_{3i} \Rightarrow \ldots \Rightarrow z_{k-1} \Rightarrow z_k = v$$

entspricht in diesem Baum ein Pfad (= eine Folge von Kanten) beginnend an einem Knoten, in dem u steht, und endend an einem Knoten, in dem v steht.

Knoten, in denen Wörter aus Terminalzeichen stehen, bilden stets Blätter; alle diese Wörter bilden die erzeugte Sprache L(G).

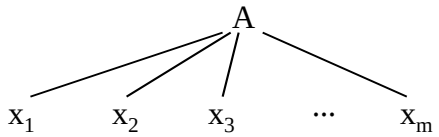
Die Ableitungen von Wörtern der erzeugten Sprache beginnen (als Pfad) stets in der Wurzel S und enden bei dem jeweiligen Wort in einem Blatt.

Aspekt (2.): Die Ableitung eines speziellen Wortes aus dem Startsymbol einer Grammatik haben wir wie folgt dargestellt:
$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (a + S_3) \Rightarrow$ **(a + a)**

Diese Ableitung kann man umstellen, indem man zuerst das zweite S_3 durch a ersetzt und danach erst das erste S_3:
$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (S_3 + a) \Rightarrow$ **(a + a)**
Dies ist im Wesentlichen *die gleiche Ableitung*.

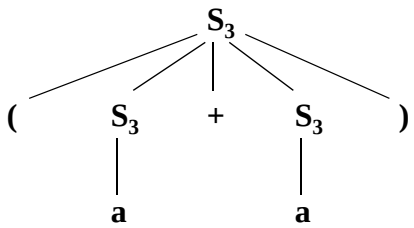
Um dies präzise zu definieren, schreiben wir auch die einzelnen Ableitungen baumartig auf, wobei in jedem Knoten genau ein Terminal- bzw. Nichtterminalzeichen steht.

Genauer: Eine Regel $A \rightarrow x_1 x_2 x_3 \ldots x_m$ mit $x_i \in (V \cup \Sigma)$ notieren wir in der Form



Die Regel $A \rightarrow \varepsilon$ (mit dem leeren Wort ε auf der rechten Seite) schreiben wir in der Form:

Die Ableitung $S_3 \Rightarrow (S_3 + S_3) \Rightarrow (S_3 + a) \Rightarrow (a + a)$ wird dann durch folgenden Baum dargestellt:



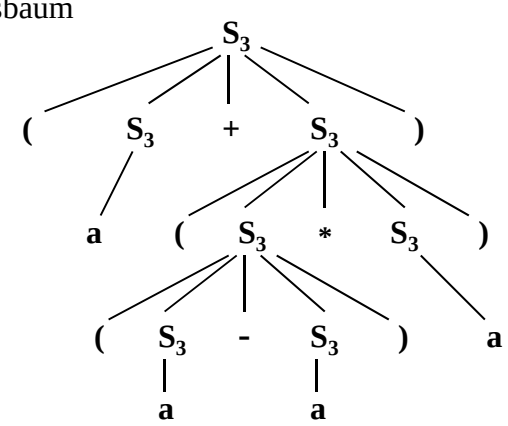
2.7.9:
Dieses Gebilde heißt Ableitungsbaum (engl.: derivation tree) zu obiger Ableitung für das Wort $(a + a)$ in der Grammatik G_3.

Dieser Baum ist zugleich der Ableitungsbaum zur Ableitung
$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (a + S_3) \Rightarrow (a + a)$

Als weiteres Beispiel betrachten wir (aus 2.7.6) die Ableitung

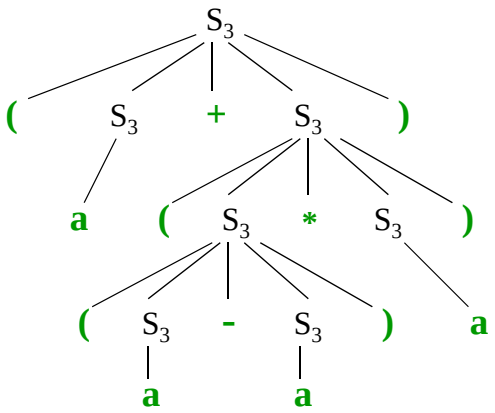
$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (a + S_3) \Rightarrow (a + (S_3 * S_3)) \Rightarrow (a + (S_3 * a))$
$\Rightarrow (a + ((S_3 - S_3) * a)) \Rightarrow (a + ((a - S_3) * a)) \Rightarrow (a + ((a - a) * a))$.

Ableitungsbaum hierzu:



Als weiteres Beispiel betrachten wir (aus 2.7.6) die Ableitung

$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (a + S_3) \Rightarrow (a + (S_3 * S_3)) \Rightarrow (a + (S_3 * a))$
$\Rightarrow (a + ((S_3 - S_3) * a)) \Rightarrow (a + ((a - S_3) * a)) \Rightarrow$ **(a + ((a - a) * a))** .



Geht man von links nach rechts die Blätter durch, so erhält man das abgeleitete Wort.

Diesen Ableitungsbaum besitzen auch andere Ableitungen, z.B.:

$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (S_3 + (S_3 * S_3)) \Rightarrow (S_3 + (S_3 * a))$
$\Rightarrow (S_3 + ((S_3 - S_3) * a)) \Rightarrow (S_3 + ((a - S_3) * a))$
$\Rightarrow (S_3 + ((a - a) * a)) \Rightarrow (a + ((a - a) * a))$

oder:

$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (S_3 + (S_3 * S_3)) \Rightarrow (a + (S_3 * S_3))$
$\Rightarrow (a + ((S_3 - S_3) * S_3)) \Rightarrow (a + ((a - S_3) * S_3))$
$\Rightarrow (a + ((a - S_3) * a)) \Rightarrow (a + ((a - a) * a))$.

2.7.10: *Ableitungen, die den gleichen Ableitungsbaum besitzen, sehen wir als gleich an.* Sie unterscheiden sich nur in der Reihenfolge, in der Regeln auf Nichtterminalzeichen angewendet werden, die an voneinander unabhängigen Stellen im Wort stehen.

Das Wort, das sich aus einem Ableitungsbaum ergibt, wenn man die Blätter von links nach rechts durchläuft, heißt das mit diesem Ableitungsbaum abgeleitete Wort.

Definition 2.7.11: Es sei G = (V, Σ, P, S) eine kontextfreie Grammatik und w ∈ L(G) ein Wort der erzeugten Sprache.

(1) w heißt eindeutig (bzgl. G), wenn es nur genau einen Ableitungsbaum gibt, dessen abgeleitetes Wort w ist.
(2) Gibt es mindestens zwei verschiedene Ableitungsbäume für w, so heißt w mehrdeutig (bzgl. G).
(3) G heißt eindeutig, wenn alle Wörter w ∈ L(G) eindeutig sind.
(4) G heißt mehrdeutig, wenn mindestens ein Wort w ∈ L(G) mehrdeutig ist.

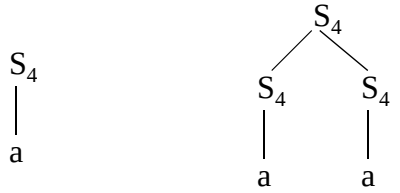
(Englisch: mehrdeutig = ambiguous, eindeutig = unambiguous.)

Beispiel 2.7.12: Wir untersuchen die folgende kontextfreie Grammatik $G_4 = (V_4, \Sigma_4, P_4, S_4)$ mit $V_4 = \{S_4\}$, $\Sigma_4 = \{a\}$ und $P_4 = \{S_4 \rightarrow S_4S_4,\ S_4 \rightarrow a\}$.

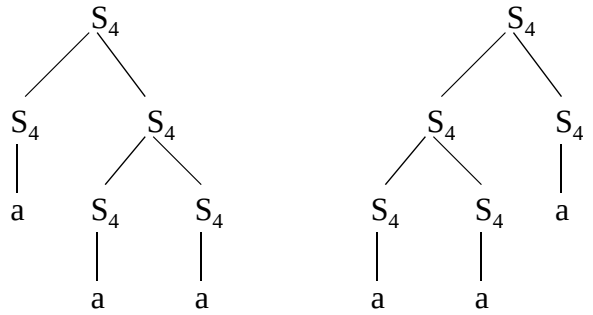
Offenbar kann man mit der ersten Regel aus S_4 jede nicht-leere Folge des Zeichens S_4 erzeugen; mit der zweiten Regel lässt sich hieraus dann jedes nicht-leere Wort über Σ_4 ableiten, d.h.,

$L(G) = \Sigma_4^+ = \{a^n \mid n \geq 1\}$.

Die Wörter a und aa sind eindeutig bzgl. G_4. Sie besitzen nur die Ableitungsbäume:



Jedes Wort über Σ_4, das mindestens die Länge 3 besitzt, ist mehrdeutig. aaa besitzt z. B. die beiden verschiedenen (!) Ableitungsbäume



Beachten Sie: Damit zwei Bäume gleich sind, muss sich der eine Baum in der Ebene deckungsgleich auf den andern schieben lassen. Spiegelungen sind nicht erlaubt! ■

Hinweis zu natürlichen Sprachen: Dort sind Grammatiken gut bekannt. Ein Ausschnitt aus der deutschen Grammatik (Nichtterminalzeichen sind die in < ... > eingeschlossenen Zeichenfolgen) könnte lauten:

<Satz> → <Subjekt> <Prädikat> <Objekt>
<Subjekt> → <Artikel im Nominativ> <Substantiv im Nominativ>
<Objekt> → <Artikel im Dativ> <Substantiv im Dativ>
<Objekt> → <Artikel im Akkusativ> <Substantiv im Akkusativ>

<Artikel im Nominativ> → das	<Artikel im Nominativ> → die
<Artikel im Dativ> → dem	<Artikel im Dativ> → der
<Artikel im Akkusativ> → den	<Artikel im Akkusativ> → die
<Substantiv im Nominativ> → Kind	<Substantiv im Nominativ> → Luft
<Substantiv im Dativ> → Fernrohr	<Substantiv im Dativ> → Fernrohr
<Substantiv im Akkusativ> → Mann	
<Prädikat> → <Verb, 3.Person,Präsenz>	
<Verb, 3.Person,Präsenz> → sieht	<Verb, 3.Person,Präsenz> → lernt
<Verb, 3.Person,Präsenz> → hört	<Verb, 3.Person,Präsenz> → rennt

.....

Aus dem Startsymbol <Satz> kann man z.B. folgendes "Wort" ableiten:

<Satz> ⇒ <Subjekt> <Prädikat> <Objekt>
⇒ <Artikel im Nominativ> <Substantiv im Nominativ> <Prädikat> <Objekt>
⇒ das <Substantiv im Nominativ> <Prädikat> <Objekt>
⇒ das Kind <Prädikat> <Objekt> ⇒ ... ⇒ das Kind sieht den Mann

Leiten Sie weitere Wörter ab. Stellen Sie weitere Regeln auf. Weisen Sie Mehrdeutigkeiten nach wie "Das Kind sieht den Mann mit dem Fernrohr." ... usw. usw.

Die Anwendung kontextfreier Grammatiken auf Programmiersprachen, erfolgt im nächsten Abschnitt. Hier wollen wir abschließend den allgemeinen Grammatikbegriff vorstellen.

Die bisher eingeführten Grammatiken heißen "kontextfrei", weil die Ableitung eines Nichtterminalzeichens "ohne Beachtung des Kontexts" erfolgt, d.h., eine Regel $A \to w$ kann auf das Nichtterminalzeichen A in einem Wort angewendet werden, ohne die links und rechts von A stehenden Zeichen zu beachten.

Darf man eine Regel $A \to w$ aber nur anwenden, wenn links von A das (Teil-) Wort x und rechts das (Teil-) Wort y stehen, dann würde man die Regel $xAy \to xwy$ verwenden müssen. Dies führt zu den "kontextsensitiven" Grammatiken. Wir verallgemeinern diesen Gedanken noch einmal und erhalten:

Definition 2.7.13:

G = (V, Σ, P, S) heißt (Chomsky-) Grammatik genau dann, wenn:

(1) V ist eine nicht-leere endliche Menge (die Menge der Nichtterminalzeichen oder Variablen),

(2) Σ ist eine nicht-leere endliche Menge (die Menge der Terminalzeichen),

(3) $S \in V$ ist ein Nichtterminalzeichen (das Startsymbol),

(4) $P \subset V^+ \times (V \cup \Sigma)^*$ ist eine endliche Menge (die Menge der Regeln oder Produktionen).

Wir haben an Definition 2.7.1 also nur eine Kleinigkeit geändert, indem wir in (4) die Menge V durch V^+ (siehe 2.6.6) ersetzt haben. Die Begriffe "Ableitung" und "erzeugte Sprache" können unverändert aus 2.7.2 und 2.7.3 übernommen werden (bei der Ableitung ist nur $p \to q$ statt $A \to w$ zu schreiben).

Definition 2.7.14: Gegeben sei eine Grammatik $G = (V, \Sigma, P, S)$. Die Regelmenge P definiert auf der Menge $(V \cup \Sigma)^*$ die "Ableitungsrelationen" $\Rightarrow$ und $\Rightarrow^*$:

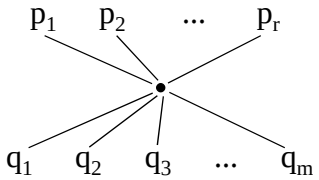
(1) Es gilt $u \Rightarrow v$ genau dann, wenn man die Wörter u und v in der Form $u = xpy$, $v = xqy$ mit $x, y \in (V \cup \Sigma)^*$ und $p \to q \in P$ schreiben kann. Man sagt: v ist aus u in einem Schritt ableitbar oder v lässt sich aus u in einem Schritt ableiten.

(2) Es gilt $u \Rightarrow^* v$ genau dann, wenn entweder $u = v$ ist oder wenn es Wörter $z_0, z_1, ..., z_k \in (V \cup \Sigma)^*$ für ein $k \geq 1$ gibt mit $u = z_0$, $v = z_k$, $z_i \Rightarrow z_{i+1}$ für alle $i = 0, 1, ..., k-1$. Man sagt dann, v ist aus u herleitbar oder ableitbar. (Die Zahl k heißt Länge der Ableitung.)

Hinweis: $\Rightarrow^*$ ist der so genannte "reflexive und transitive Abschluss" von $\Rightarrow$. (Englisch: Ableitung = derivation.)

Definition 2.7.15:
Die von einer Grammatik $G = (V, \Sigma, P, S)$ erzeugte Sprache (engl.: generated language) ist die Menge
$L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \} \subseteq \Sigma^*$.

Ableitungen kann man nun nicht mehr als Baum darstellen, vielmehr bilden sie ein "Netz", das aus Gebilden der Form



für eine Regel $p_1 p_2 p_3 ... p_r \to q_1 q_2 q_3 ... q_m$ mit $p_i \in V$, $q_j \in (V \cup \Sigma)$, $r \geq 1$, $m \geq 0$, aufgebaut wird. Wir betrachten ein Beispiel.

Beispiel 2.7.16: Gegeben sei folgende Grammatik $G_5 = (V_5, \Sigma_5, P_5, S_5)$ mit $V_5 = \{S_5, A, B, C\}$, $\Sigma_5 = \{a, b\}$ und

$P_5 = \{S_5 \to A S_5 B,\ S_5 \to CC,\ AC \to BA,\ CB \to BC,$
$BAB \to C,\ AB \to a,\ CC \to b\}$.

Um die "Arbeitsweise" der Grammatik zu verstehen, versucht man zunächst, einige Wörter herzuleiten:

$S_5 \Rightarrow CC \Rightarrow b$

$S_5 \Rightarrow A S_5 B \Rightarrow ACCB \Rightarrow BACB \Rightarrow BABC \Rightarrow CC \Rightarrow b$

Hinweis: Dies sind offensichtlich zwei wesentlich verschiedene Ableitungen, d.h., die Grammatik G_5 ist mehrdeutig. Man kann jetzt aber nicht mehr mit Bäumen argumentieren, sondern man muss "Ableitungsnetze" betrachten.

Zunächst stellen wir die einzelnen Regeln dar:

$S_5 \to A S_5 B$

$S_5 \to CC$

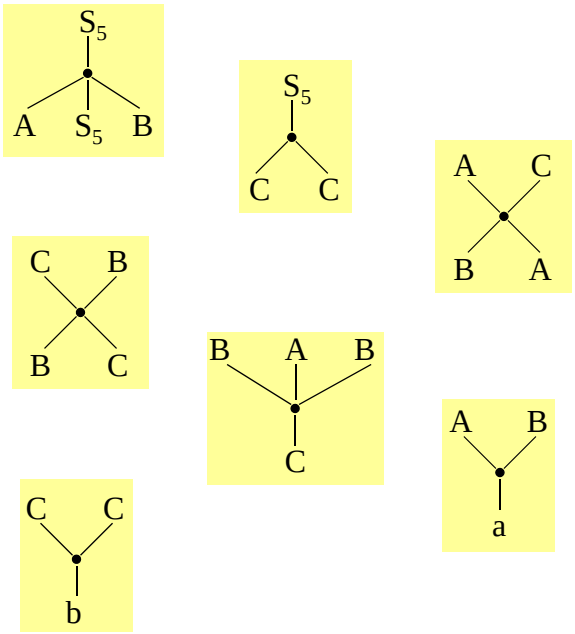
$AC \to BA$

$CB \to BC$

$BAB \to C$

$AB \to a$

$CC \to b$



Nun kleben wir die einzelnen Regeln zu Ableitungen zusammen:

$S_5 \to AS_5B$

$S_5 \to CC$

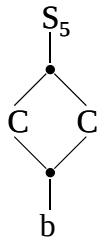
$AC \to BA$

$CB \to BC$

$BAB \to C$

$AB \to a$

$CC \to b$



Nun kleben wir die einzelnen Regeln zu Ableitungen zusammen:

$S_5 \to AS_5B$

$S_5 \to CC$

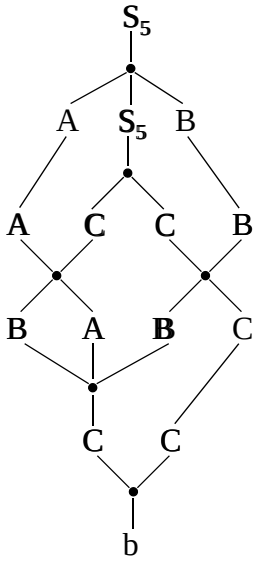
$AC \to BA$

$CB \to BC$

$BAB \to C$

$AB \to a$

$CC \to b$



S_5

$\Rightarrow AS_5B$

$\Rightarrow ACCB$

$\Rightarrow BACB$

$\Rightarrow BABC$

$\Rightarrow CC$

$\Rightarrow b$

Suche nach weiteren Ableitungen:

$S_5 \Rightarrow AS_5B \Rightarrow ACCB \Rightarrow ACBC \Rightarrow ABCC \Rightarrow aCC \Rightarrow ab$

$S_5 \Rightarrow AS_5B \Rightarrow AAS_5BB \Rightarrow AACCBB \Rightarrow ABACBB$
$\Rightarrow ABABCB \Rightarrow ACCB \Rightarrow ACBC \Rightarrow ABCC \Rightarrow aCC \Rightarrow ab$

Es gibt noch viele weitere Ableitungen, aber sie alle erzeugen eines der beiden Wörter b oder ab, d. h., $L(G_5) = \{b, ab\}$. ■

Stimmt das wirklich? Wie kann man so etwas beweisen? Wie kann man die Menge L(G) für eine Grammatik bestimmen bzw. wie kann man zu einem Wort w und einer Grammatik feststellen, ob diese Grammatik das Wort erzeugt oder nicht (sog. Wortproblem)?

Hinweis: In Theorievorlesungen wird beweisen, dass dieses Problem (wie das Halteproblem!) algorithmisch unlösbar ist.

BNF und kontextfreie Grammatiken sind im Wesentlichen das Gleiche. Wir listen den Formalismus hier auf, ohne auf Details noch einmal einzugehen.

Definition 2.7.17: Backus-Naur-Form
Eine BNF ist ein Viertupel (V, Σ, P, S) mit

(1) V ist eine nicht-leere endliche Menge (die Menge der Nichtterminalzeichen); alle Elemente sind von der Form < Zeichenkette > (in "Zeichenkette" treten '<' und '>' nicht auf),

(2) Σ ist eine nicht-leere endliche Menge (die Menge der Terminalzeichen) mit $V \cap \Sigma = \varnothing$ und $| \notin \Sigma$,

(3) $S \in V$ ist ein Nichtterminalzeichen (das Startsymbol),

(4) $P \subset V\,\{::=\}\,(V \cup \Sigma)^*\,(\{|\}\,(V \cup \Sigma)^*\,)^*$ ist eine endliche Menge (die Menge der Regeln oder Produktionen).

(Zu Beispielen siehe Kapitel 1.10.)

Umwandlung einer kontextfreien Grammatik in die BNF:
Gegeben sei eine kontextfreie Grammatik (V', Σ, P', S').

1. Sortiere die Regeln nach dem links stehenden Nichtterminalzeichen.
2. Ersetze jedes Nichtterminalzeichen A' ∈ V' durch einen aussagekräftigen Namen A der Form <Zeichenkette>, wobei die Zeichen '<' und '>' in dieser "Zeichenkette" nicht auftreten dürfen. Dies ergibt die neue Menge der Nichtterminalzeichen V mit dem Start-symbol S (anstelle S'). Führe diese Ersetzung für alle Regeln durch.
3. Wenn $A \to u_1, A \to u_2, ..., A \to u_k$ alle Regeln im neuen P' mit A als linker Seite sind, dann ersetze diese k Regeln durch
   $A ::= u_1 \mid u_2 \mid ... \mid u_k$.
   So entsteht die Regelmenge P der BNF.

Beispiel 2.7.17.a: Arithmetische Ausdrücke (hier als eindeutige (!) Grammatik mit korrekter Punkt-vor-Strich-Priorität).
Gegeben ist die kontextfreie Grammatik (V, Σ, P, E) mit

V = {B, E, F, T, Z}, Σ = {a, z, (,), +, -, *, /} und den Regeln

P = { $B \to a$, $Z \to z$, $F \to (E)$, $E \to T$, $E \to T + E$, $E \to T - E$,
$T \to F$, $T \to F * T$, $T \to F / T$, $F \to B$, $F \to Z$ }

Schritt 1: Sortiere alle Regeln nach dem links stehenden Nichtterminalzeichen:

$E \to T$, $E \to T + E$, $E \to T - E$,
$T \to F$, $T \to F * T$, $T \to F / T$,
$F \to B$, $F \to Z$, $F \to (E)$,
$B \to a$,
$Z \to z$

Schritt 2: Ersetze die Nichtterminalzeichen durch aussagekräftige Namen in spitzen Klammern:

<Ausdruck> → <Term>, <Ausdruck> → <Term> + <Ausdruck>,
<Ausdruck> → <Term> - <Ausdruck>
<Term> → <Faktor>, <Term> → <Faktor> * <Term>,
<Term> → <Faktor> / <Term>
<Faktor> → <Bezeichner>, <Faktor> → <Zahl>,
<Faktor> → (<Ausdruck>),
<Bezeichner> → a, < Zahl> → z

Schritt 3: Fasse die rechten Seiten mit gleichem linken Nichtterminalzeichen getrennt durch "|" zusammen und ersetze "→" durch "::=" .

<Ausdruck> ::= <Term> | <Term> + <Ausdruck> | <Term> - <Ausdruck>
<Term> ::= <Faktor> | <Faktor> * <Term> | <Faktor> / <Term>
<Faktor> ::= <Bezeichner> | <Faktor> ::= <Zahl> | (<Ausdruck>)
<Bezeichner> ::= a
<Zahl> ::= z

Der Rest von 2.7 kann übersprungen werden.

Definition 2.7.18: Gegeben sei (siehe 2.7.13) eine Grammatik G = (V, Σ, P, S). Die Grammatik G heißt

(1) vom Typ 1 oder kontextsensitiv genau dann, wenn alle Regeln von der Form $xAy \to xwy$ sind mit $A \in V$, $x, y \in V^*$ und $w \in (V \cup \Sigma)^+$ (beachte: $w \neq \varepsilon$).

(2) vom Typ 2 oder kontextfrei genau dann, wenn alle Regeln von der Form $A \to w$ sind mit $A \in V$ und $w \in (V \cup \Sigma)^*$ ($w = \varepsilon$ ist hier erlaubt).

(3) vom Typ 3 oder rechtslinear genau dann, wenn alle Regeln von der Form $A \to uB$ oder $A \to u$ sind mit A, $B \in V$ und $u \in \Sigma^*$.

(4) linkslinear genau dann, wenn alle Regeln von der Form $A \to Bu$ oder $A \to u$ sind mit A, $B \in V$ und $u \in \Sigma^*$.

Definition 2.7.19: Eine Sprache heißt "xxx", wenn es eine "xxx" Grammatik G mit L = L(G) gibt für "xxx" ∈ {kontextsensitiv, kontextfrei, rechtslinear, linkslinear}.

Die Einteilung in Grammatiken vom Typ 0 (= keine Einschränkung), Typ 1 (kontextsensitiv), Typ 2 (kontextfrei) und Typ 3 (rechtslinear) stammt aus der Originalarbeit von Noam Chomsky 1959 und hat sich bis heute gehalten. Es gibt aber mittlerweile sehr viele weitere Grammatikklassen.

Hinweise: Die Sprache der Bezeichner Bez und die Sprache der korrekten Zahldarstellungen sind rechtslinear. Die Sprache der korrekt geklammerten arithmetischen oder Booleschen Ausdrücke ist kontextfrei, aber nicht rechtslinear.

Einige Aussagen (ohne Beweis):

Fügt man zu einer "xxx" Sprache eine endliche Sprache hinzu oder zieht man eine endliche Sprache ab, so bleibt das Ergebnis eine "xxx" Sprache.

Eine Sprache genau dann rechtslinear, wenn sie linkslinear ist.

Die Sprache
$\{a^n b^n \mid n \geq 0\}$ ist kontextfrei, aber nicht rechtslinear.

Die Sprache
$\{a^n b^n a^n \mid n \geq 0\}$ ist kontextsensitiv, aber nicht kontextfrei.

Es gibt Sprachen vom Typ 0, die nicht kontextsensitiv sind (leider lassen sie sich nicht mit einer kurzen Formel beschreiben).

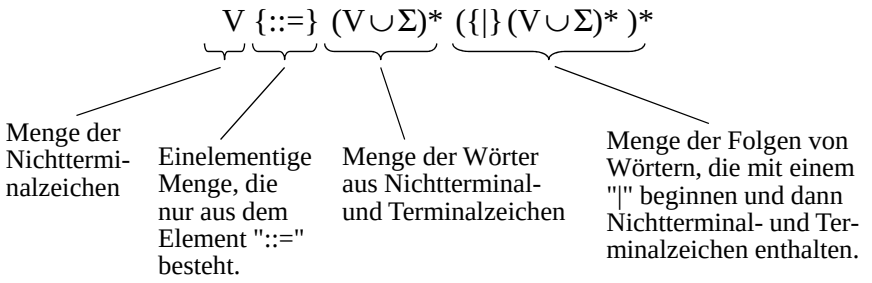
Anwendung dieser Grammatiken:

Alle in der Praxis verwendeten Programmiersprachen sind kontextsensitive Sprachen; sie lassen sich als kontextfreie Sprachen mit zusätzlichen Einschränkungen beschreiben.

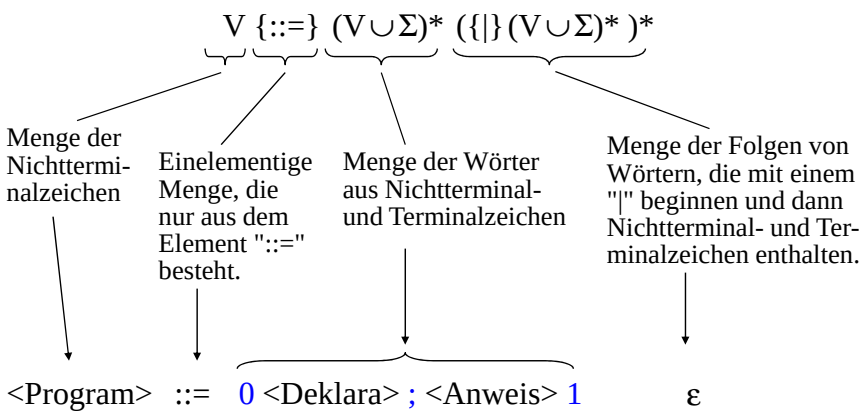
Dies werden wir später erläutern. Zuvor wird noch eine grafische Schreibweisen für kontextfreie Grammatiken eingeführt, nämlich die Syntaxdiagramme.

2.7.20: Zum Formalismus der BNF: Welche Elemente sind in der Menge $V\,\{::=\}\,(V \cup \Sigma)^*\,(\{|\}\,(V \cup \Sigma)^*\,)^*$?

Dies ist die Konkatenation von vier Mengen, wobei die dritte und die vierte Menge beliebige Konkatenationen enthalten können. Betrachten wir diese Menge genauer:

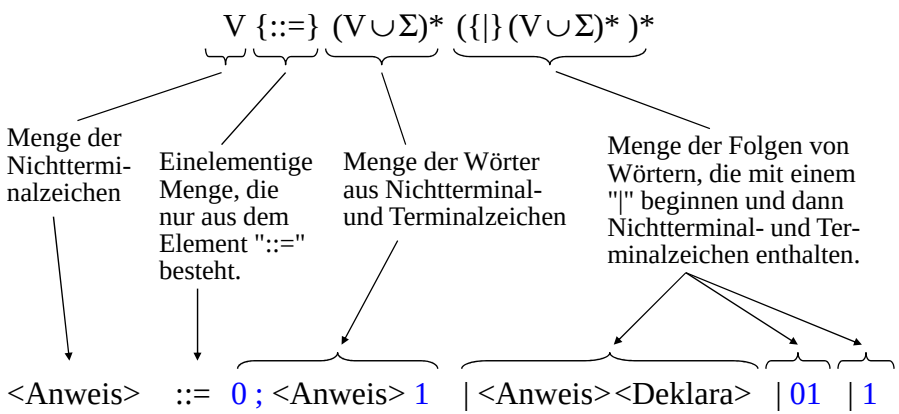


Beispielwort, das in dieser Menge liegt, für
V = {<Program>, <Deklara>, <Anweis>} und Σ = {0, 1, ;} :



ein Zeichen *::=* *Wort der Länge 5 über V∪Σ* *leeres Wort*

Noch ein Beispielwort, das in dieser Menge liegt, für
V = {<Program>, <Deklara>, <Anweis>} und Σ = {0, 1, ;}



ein Zeichen *::=* *Wort der Länge 4* *3 Wörter, die jeweils mit "|" beginnen*

Folglich gilt mit den obigen Mengen V und Σ:

<Program>::=0<Deklara>;<Anweis>1
∈ V {::=} (V∪Σ)* ({|}(V∪Σ)*)*

und

<Anweis>::=0;<Anweis>1|<Anweis><Deklara>| 01| 1
∈ V {::=} (V∪Σ)* ({|}(V∪Σ)*)*

Umgekehrt kann man mit dieser Anleitung auch eine BNF in eine "normale" kontextfreie Grammatik zurück verwandeln.

Der Ableitungsbegriff der BNF ist daher der gleiche wie der für kontextfreie Grammatiken. Somit sind die Ableitungsrelationen $\Rightarrow$ und $\Rightarrow^*$ sowie die erzeugte Sprache L(G) auch für die BNF definiert (vgl. 2.7.2 und 2.7.3).

Die BNF wird, wie bereits definiert, um einige hilfreiche Abkürzungen zur EBNF erweitert (Schlüsselwörter hervorheben, eckige und geschweifte Klammern).

Beispiel 2.7.21: Darstellung zur Basis b für $2 \le b \le 16$ in Ada. Diese Zahlen werden in der Form
2 # 10010 # oder 8 # 60175 # oder 16 # A2F03 #
dargestellt, wobei zwischen den #-Zeichen mindestens eine Ziffer stehen muss und nur die Basen von 2 bis 16 zugelassen sind. Ansatz, um dies mit einer EBNF zu beschreiben:

< Darstellung mit Basis > ::= <Basis> # <Ziffernfolge> #

Doch dieser Ansatz wird nicht erfolgreich sein, da zu jeder Basis eine andere Menge von Ziffern gehört. Wir gehen daher den aufwendigen Weg, zu jeder Zahl von 2 bis 16 die Regeln neu aufzuschreiben, wobei sich eine offensichtliche Regelmäßigkeit ergibt.
Abkürzend schreiben wir <DarBa2> für die <Darstellung zur Basis 2> und analog für die anderen Basen sowie <DarBa> für das Startsymbol <Darstellung mit Basis>.

<Ziffern2> ::= 0 | 1
<DarBa2> ::= 2 # <Ziffern2> {<Ziffern2>} #
<Ziffern3> ::= <Ziffern2> | 2
<DarBa3> ::= 3 # <Ziffern3> {<Ziffern3>} #
<Ziffern4> ::= <Ziffern3> | 3
<DarBa4> ::= 4 # <Ziffern4> {<Ziffern4>} #
... *usw.* ...
<Ziffern11> ::= <Ziffern10> | A
<DarBa11> ::= 11 # <Ziffern11> {<Ziffern11>} #
... *usw* ...
<Ziffern16> ::= <Ziffern15> | F
<DarBa16> ::= 16 # <Ziffern16> {<Ziffern16>} #

<DarBa> ::= <DarBa2> | <DarBa3> | <DarBa4> | <DarBa5> |
<DarBa6> | <DarBa7> | <DarBa8> | <DarBa9> |
<DarBa10> | <DarBa11> | <DarBa12> | <DarBa13> |
<DarBa14> | <DarBa15> | <DarBa16>

■

Überlegung 2.7.22

Diese Schreibweise ist ziemlich aufwendig. Lässt sie sich irgendwie vereinfachen?

Wie müsste man die EBNF erweitern, damit man etwas k Mal wiederholen kann statt beliebig oft? (Hier ist k = 15.)

Wir machen im Folgenden hierzu einen Vorschlag. Er stellt eine rein gedankliche Überlegung dar, deren Ergebnis nicht in der erweiterten BNF benutzt wird. Dies soll Sie anregen, die verwendeten Formalismen auf deren Nützlichkeit hin zu hinterfragen.

noch: Überlegung 2.7.22:

Es ist wenig sinnvoll, fünfzehn Mal im Wesentlichen das Gleiche immer wieder hinzuschreiben. Könnte man ein neues Sprachelement zur EBNF hinzufügen, welches "für k von 2 bis 16" diese Wiederholungen vornimmt?!

In Ada-ähnlicher Notation ***könnte*** man das Sprachelement

for k in 2..16 loop <Folge von Regeln > end loop

einführen. In den Regeln wollen wir diese Variable k benutzen, aber wir können nicht einfach "k" dort verwenden, da nicht klar ist, ob die Variable oder das Zeichen gemeint ist. Wir schließen k daher in ein Sonderzeichen (z. B. $) ein und erlauben, k oder allgemeiner $"Ausdruck, in dem k vorkommt"$ in den Regeln zu verwenden. Dies wird am Beispiel der Darstellungen zu einer Basis klar.

Die Zahldarstellungen zur Basis $2 \le b \le 16$ kann man dann folgendermaßen aufschreiben, wobei wir die einzelnen Ziffern von 0 bis b-1 hier in der Form (0), (1), (2), ..., (b-1) notieren:

```
<Ziffern2>  ::=  (0) | (1)
<DarBa2> ::= 2 # <Ziffern2> {<Ziffern2>} #
<DarBa>  ::=  <DarBa2>

for k in 3 .. 16 loop
<Ziffern$k$>  ::=  <Ziffern$k-1$> | ($k-1$)
<DarBa$k$> ::= $k$ # <Ziffern$k$> {<Ziffern$k$>} #
<DarBa> ::= <DarBa> | <DarBa$k$>
end loop
```

Die Zeile <DarBa> ::= <DarBa> | <DarBak> soll hier bedeuten: Es wird das alte <DarBa> um <DarBak> erweitert und das Ergebnis heißt wiederum <DarBa>.

Hätte man das Symbol "nichts" oder "leere Menge" ∅ zur Verfügung, so könnte man auch folgende etwas einfachere Definition verwenden:

```
<Ziffern1>  ::=  (0)
<DarBa>  ::=  ∅

for k in 2 .. 16 loop
<Ziffern$k$>  ::=  <Ziffern$k-1$> | ($k-1$)
<DarBa$k$> ::= $k$ # <Ziffern$k$> {<Ziffern$k$>} #
<DarBa> ::= <DarBa> | <DarBa$k$>
end loop
```

Problem: Was wurde nun eigentlich definiert? Nur <DarBa> oder auch <Ziffern3>, <Ziffern4>, <DarBa3>, <DarBa4> usw.? (Doch dies lässt sicher präzise definieren.)

Warum gibt es solche Erweiterungen nicht in der Praxis?

Zunächst ist anzunehmen, dass es sehr viele solcher Erweiterungen der EBNF gibt, die von einzelnen Labors oder Firmen auf ihre Bedürfnisse zugeschnitten sind.

Das Problem ist die Bedeutung der Sprachelemente, insbesondere Eindeutigkeit und Verständlichkeit:

- Bleiben die zwischenzeitlich aufgebauten Nichtterminal-zeichen erhalten?
- Wenn solche for-Konstrukte geschachtelt ineinander und dort in Alternativen vorkommen, ist dann das, was entsteht, stets eindeutig bestimmt?
- Und wenn man dies alles exakt definieren kann, ist der Formalismus dann noch leicht erlernbar?

Ende der Überlegung 2.7.22 ■

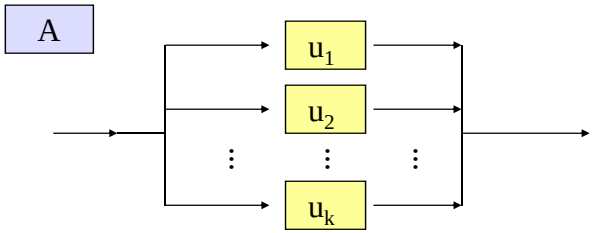
2.8 Syntaxdiagramme

2.8.1: Grafische Darstellung von EBNF-Regeln. Für jede linke Seite (also für jedes Nichtterminalzeichen) legen wir ein eigenes Diagramm an, das mit dem Nichtterminal bezeichnet wird.

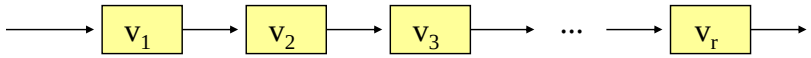
Betrachte eine solche Regel mit dem Nichtterminalzeichen A:

$A ::= u_1 \mid u_2 \mid ... \mid u_k$

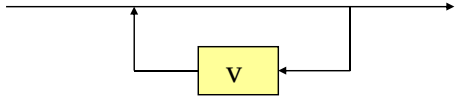
Dann zeichne hierzu das Diagramm



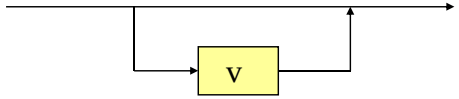
Ist ein u_i von der Form $v_1 v_2 ... v_r$ (Konkatenation von Zeichen), so ersetze → u_i → durch:



Ist ein u_i oder v_j von der Form { v } , so ersetze es durch



Ist ein u_i oder v_j von der Form [v] , so ersetze es durch



Zum Schluss ersetzt man bei allen Terminalzeichen und Schlüsselwörtern die rechteckigen Umrandungen durch Kreise.

2.8.2: Dieses Vorgehen liefert schließlich für jede Grammatik bzw. EBNF eine grafische Darstellung, das sog. Syntaxdiagramm.

Um Wörter aus einem Nichtterminalzeichen A zu erzeugen, durchläuft man das zu A gehörende Syntaxdiagramm vom Eingangspfeil bis zum Ausgangspfeil. Hier gibt es i. A. viele Wege. Für den gewählten Weg sammelt man die hierbei besuchten Terminalzeichen (= die in Kreisen stehenden Zeichen) in der Durchlaufreihenfolge auf. Alle diese Zeichenfolgen bilden dann die Menge der von A erzeugten Wörter. Eine genauere Formulierung und Beispiele folgen.

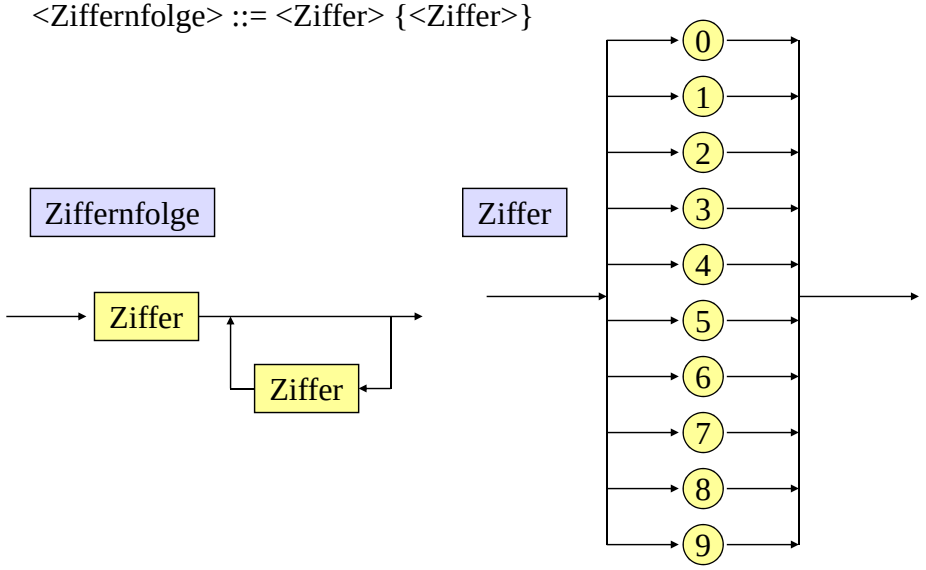
2.8.3: Präzisere Beschreibung der Bedeutung von Syntaxdiagrammen: Gegeben sind Syntaxdiagramme und eine anfangs leere Ausgabe.

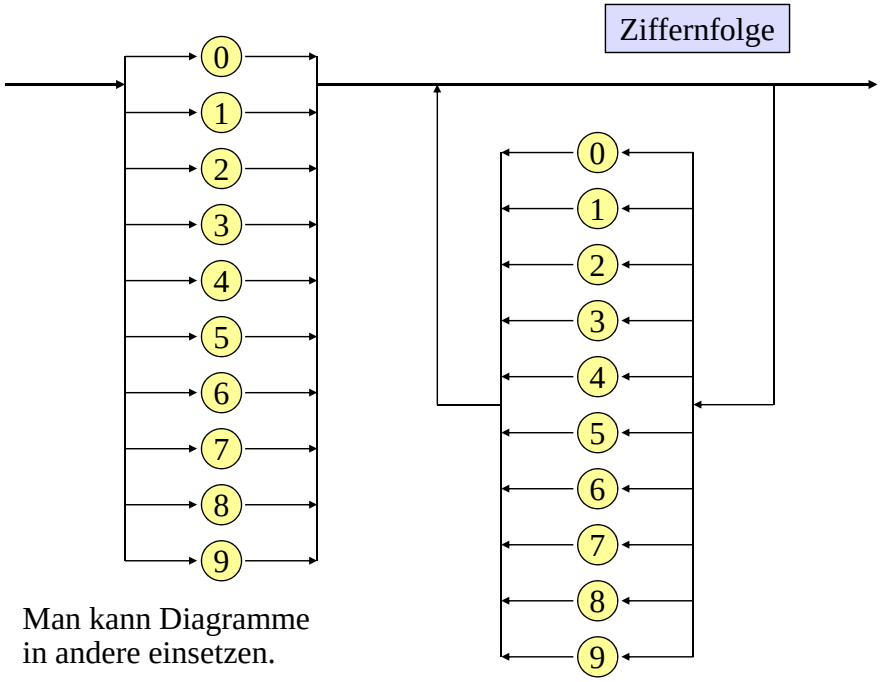
Ziel ist es, das Diagramm, das zum Startsymbol gehört, in Richtung der Pfeile vom Eingangspfeil bis zum Ausgangspfeil zu durchlaufen. Trifft man hierbei auf Zeichen in Kreisen, so schreibt man diese in der Durchlaufreihenfolge hinter die bereits vorhandene Ausgabe. Trifft man auf ein Rechteck mit dem Nichtterminalzeichen A, so klebt man das zu A gehörende Diagramm an dieser Stelle ein und durchläuft das neu entstandene Diagramm weiter. (Gibt es kein zu A gehörendes Diagramm, so bricht man ergebnislos ab.)

Alle möglichen Ausgaben, die zu einem vollständigen Durchlauf vom Eingangspfeil bis zum Ausgangspfeil des Startsymbol-Diagramms gehören, bilden die erzeugte Sprache.

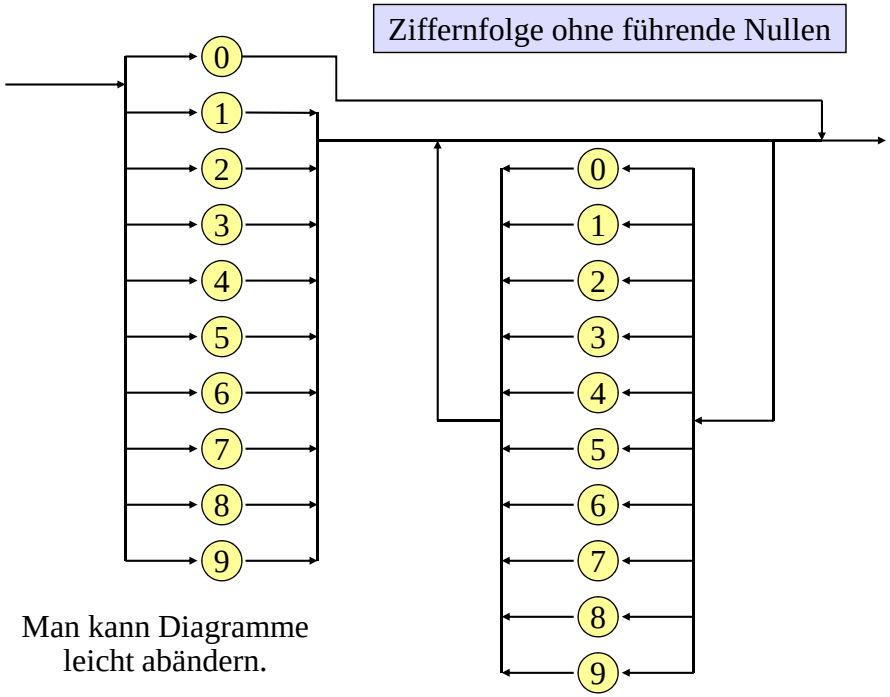
Beispiel 2.8.4: <Ziffer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Ziffernfolge> ::= <Ziffer> {<Ziffer>}





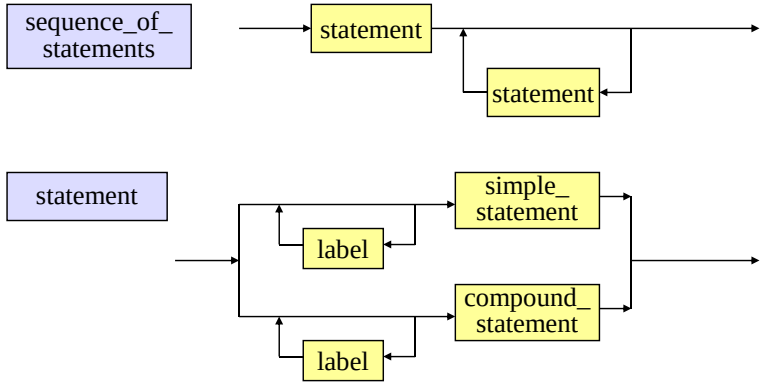
Man kann Diagramme in andere einsetzen.



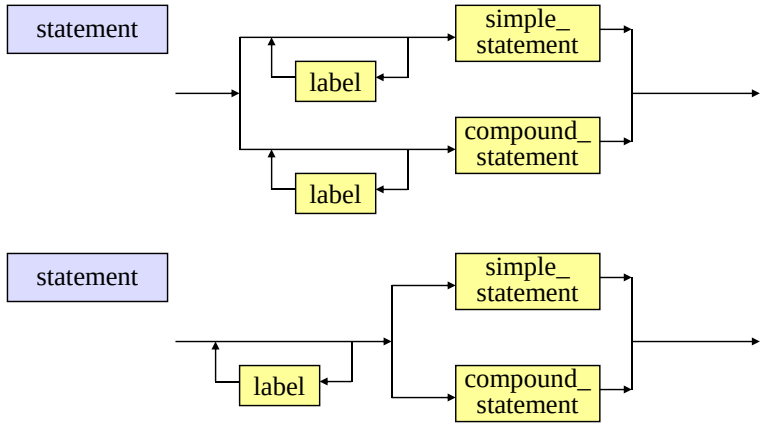
Man kann Diagramme leicht abändern.

Beispiel 2.8.5: Nochmals: Ada-Anweisungen

<sequence_of_statements> ::= <statement> {<statement>}
<statement> ::= {<label>} <simple_statement> |
{<label>} <compound_statement>



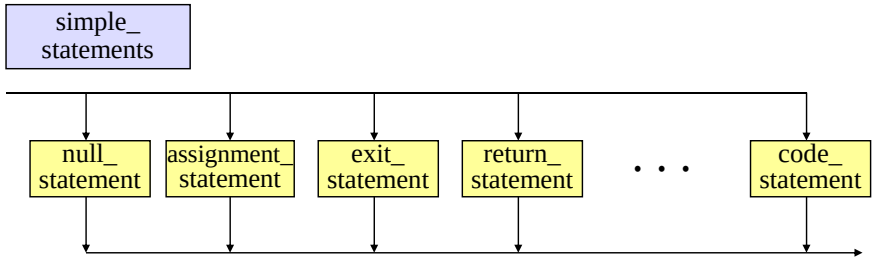
<statement> ::= {<label>} <simple_statement> |
{<label>} <compound_statement>



Es gibt oft Vereinfachungen bei der grafischen Darstellung.

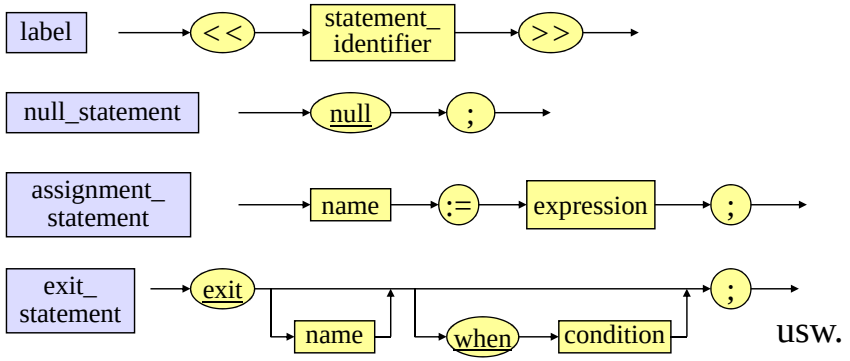
Beispiel 2.8.6: Nochmals: Ada-Anweisungen

<simple_statement> ::= <null_statement> |
<assignment_statement> | <exit_statement> |
<simple_return_statement> | <procedure_call_statement> |
<entry_call_statement> | <goto_statement> |
<requeue_statement > | <delay_statement> |
<abort_statement> | <raise_statement> | <code_statement>



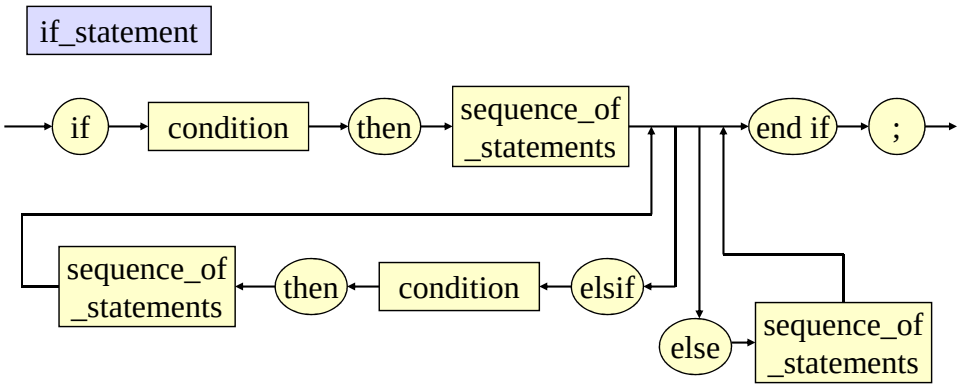
Beispiel 2.8.7: Nochmals: Ada-Anweisungen

label ::= "<<" *label*_statement_identifier ">>"
statement_identifier ::= direct_name
null_statement ::= 'null' ";"
assignment_statement ::= *variable*_name ":=" expression ";"
exit_statement ::= 'exit' [*loop*_name] ['when' condition] ";"
goto_statement ::= 'goto' *label*_name ";"



Beispiel 2.8.8: Alternative in Ada

if_statement ::= 'if' condition 'then' sequence_of_statements
{'elsif' condition 'then' sequence_of_statements}
['else' sequence_of_statements] 'end if' ";"



2.9 Sprachen zur Beschreibung von Sprachen

2.9.1 Aspekte von Sprachen. Eine Sprache, mit der man eine andere Sprache beschreiben kann, nennt man Metasprache. Dabei muss man zwischen mehreren "Ebenen" unterscheiden:

Ebene "Syntax": Hier geht um den korrekten Aufbau der Wörter bzw. Sätze einer Sprache. Der oft zitierte Satz

"Der Tisch ist ein Säugetier."

ist syntaktisch (oder grammatikalisch) korrekt, völlig unabhängig davon, ob er inhaltlich zutrifft.

Kritischer ist dies bei einem Satz wie

"Nachts ist es kälter als draußen."

da es möglicherweise von der deutschen Grammatik her nicht zulässig ist, eine Temperaturangabe mit einer Ortsangabe zu vergleichen. Dennoch klingt der Satz syntaktisch korrekt.

Die formal richtige Anordnung der Sprachelemente lässt sich insbesondere durch Grammatiken oder gleichwertige Kalküle festlegen. Dies wurde in der Informatik in den letzten vierzig Jahren gut untersucht und hier liegen zugleich umfangreiche Erfahrungen aus der Praxis vor.

Will man die Syntax von Programmiersprachen oder anderen Sprachen im mathematisch-technisch-naturwissenschaftlichen Bereich beschreiben, so verwendet man spezielle kontextfreie Grammatiken, die folgende Eigenschaft besitzen: Ist der Aufbau der Wörter bzw. Sätze einer Sprache hierin formuliert, so kann man automatisch einen Algorithmus (einen sog. "Parser") erzeugen lassen, der das Wortproblem in relativ kurzer Zeit löst, d.h., der zu *jedem* Wort über dem Terminalalphabet feststellt, ob es zur Sprache gehört oder nicht, und zwar in einer Zeitspanne, die proportional zur Länge des Wortes ist.

Ebene "Semantik": Die Semantik ordnet jedem Wort bzw. Satz einer Sprache seine Bedeutung zu. Die Bedeutung kann recht unterschiedlich sein: Die Bedeutung von Sätzen der natürlichen Sprachen ist meist eine Handlung oder eine Information; bei mathematisch-ingenieurwissenschaftlichen Aussagen geht es meist um deren inhaltliche Korrektheit, "logische Gültigkeit" und Widerspruchsfreiheit; bei Programmiersprachen weist die Semantik jedem Programm seine realisierte Abbildung zu.

Eine genaue Semantik ist unverzichtbar für die Sicherheit von Software. Zum einen muss das Programm korrekt sein, also genau das durchführen, was (in einer sog. "Spezifikation") vorgegeben ist, zum anderen muss es zuverlässig arbeiten, also z. B. gegenüber Veränderungen der Semantik, bedingt durch eine neue Umgebung, stabil sein.

Mit der Semantik ist der Begriff der Programm-"Äquivalenz" verbunden: Zwei Wörter einer Sprache heißen äquivalent, wenn sie die gleiche Bedeutung besitzen. In der Praxis möchte man oft ein Programm durch ein anderes ersetzen, das das Gleiche leistet, aber in irgendeiner Hinsicht besser ist. Der Benutzer soll allerdings hiervon nichts mitbekommen.

Die beiden Programmstücke (declare X, Y, i, A, B, C: natural)

(1) i:=0; while i < X do X:=X+Y; Y:=X+Y; i:=i+1 od

(2) A:=1; B:=1; for i:=2 to 2*X do C:=A+B, A:=B; B:=C od;
if X>0 then C:=(B-A)*X+A*Y; Y:=A*X+B*Y; X:=C fi

sind äquivalent *(wirklich?)*, wenn man nur die Variablen X und Y betrachtet. In der Semantik untersucht man u.A., wie man Äquivalenzen beweisen oder wie kann man aus dem einen das andere Programm automatisch erzeugen lassen kann.

Ebene "Pragmatik": Die Pragmatik untersucht und beschreibt die Beziehungen zwischen der Sprache und deren Benutzern (Menschen, Maschinen) bzw. der jeweiligen Umwelt. Dies können einfache Fragen der Auswirkungen von Wörtern bzw. Sätzen sein, aber auch verwickelte Zusammenhänge zwischen Interessen, die man mit der Benutzung einer Sprache verfolgt.

Die Pragmatik führt aus dem Bereich der über Alphabeten aufgebauten Sprachen hinaus ("welche Wirkung soll ein Satz erzielen?"), wirkt aber auch in sie von außen hinein, indem sie z.B. die Begründung für Sprachelemente oder Darstellungen von Daten ist ("wir führen arrays ein, weil man hiermit die in der Technik verwendeten Vektoren wiedergeben kann").

Der Pragmatik wurde in der Informatik bisher wenig Aufmerksamkeit gewidmet; dies ändert sich aber zurzeit.

Fast jede natürliche Sprache (Deutsch, Chinesisch, Latein, Englisch usw.) kann als Metasprache verwendet werden, z.B., indem wir mit dieser Sprache eine andere natürliche Sprache beschreiben und erlernen (Fremdsprachenunterricht).

Natürliche Sprachen besitzen die Eigenschaft, *sich selbst* beschreiben zu können (Muttersprachunterricht einschl. Grammatikkunde). Diese Eigenschaft finden wir bei vielen künstlichen Sprachen auch: Oft lässt sich in einer Sprache die Sprache selbst beschreiben; insbesondere kann man hiermit die Sprache erweitern, indem man die Syntax und die Bedeutung der zusätzlichen Sprachelemente in der Sprache selbst formuliert. Natürliche Sprachen und Programmiersprachen können sich auf diese Weise ständig weiterentwickeln und neue Gebiete der Beschreibung und Bearbeitung erschließen.

Wir demonstrieren diese "Selbstbeschreibungsfähigkeit" an einem Beispiel, indem wir kontextfreie Grammatiken mit Hilfe der EBNF erzeugen (die EBNF ist eine kontextfreie Grammatik, siehe Abschnitt 2.7).

Hierzu müssen wir Grammatiken als Wörter einer Sprache aufschreiben. Die Grammatik G_1 aus Abschnitt 2.7

$G_1 = (V_1, \Sigma_1, P_1, S_1)$ mit $V_1 = \{S_1\}$,
$\Sigma_1 = \{0, 1\}$, $P_1 = \{(S_1, 1), (S_1, S_1 0), (S_1, S_1 1)\}$

kann man als folgendes Wort der Länge 25

$S_1;0,1;(S_1,1)(S_1,S_1 0)(S_1,S_1 1);S_1 \in A^*$

über dem Alphabet A = { '**(**' , '**)**' , '**,**' , '**;**' , '**0**' , '**1**' , '**S_1**' }
auffassen.

Auf diese Weise erzeugen wir nun (kontextfreie) Grammatiken aus einer EBNF.

Beispiel 2.9.2: Folgende EBNF (V_G, Σ_G, P_G, <Grammatik>) umfasst die Sprache aller kontextfreien Grammatiken, deren Terminalzeichen alle von der Form Tz und deren Nichtterminalzeichen alle von der Form Nz sind mit z∈ {1}{0,1}*:

V_G = {<Ntz>, <Tz>, <P>, <rSeite>, <Grammatik>},

Σ_G = { '**(**' , '**)**' , '**,**' , '**;**' , '**0**' , '**1**' , '**N**' , '**T**' },

Regeln:

<Grammatik> ::=
<Ntz> {'**,**' <Ntz>} '**;**' <Tz> {'**,**' <Tz>} '**;**' <P> '**;**' <Ntz>

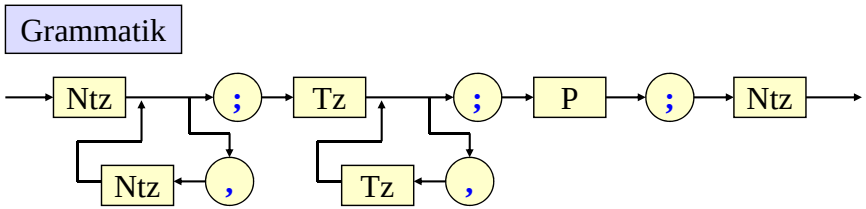
<Ntz> ::= '**N**' '**1**' {'**0**' | '**1**'}

<Tz> ::= '**T**' '**1**' {'**0**' | '**1**'}

<P> ::= { '**(**' <Ntz> '**,**' <rSeite> '**)**' }

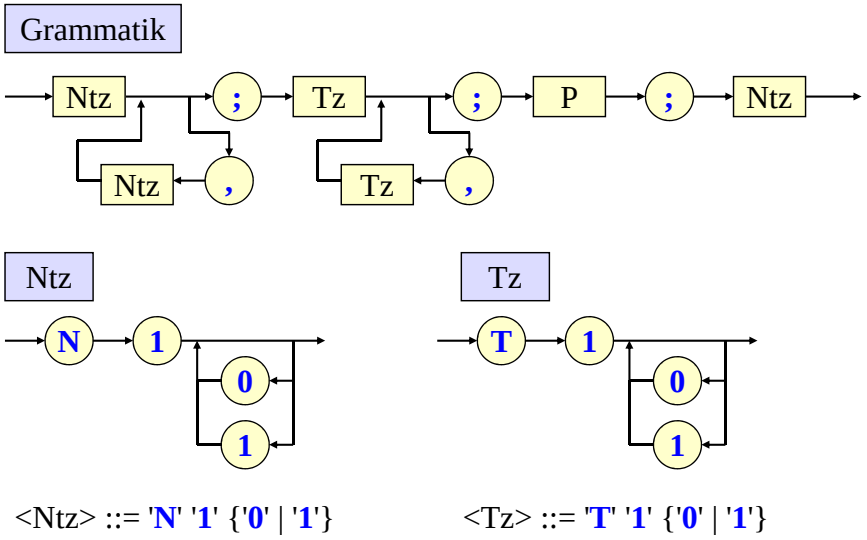
<rSeite> ::= { <Ntz> | <Tz> }

Syntaxdiagramme hierzu zeichnen:



<Grammatik> ::=
<Ntz> {'**,**' <Ntz>} '**;**' <Tz> {'**,**' <Tz>} '**;**' <P> '**;**' <Ntz>

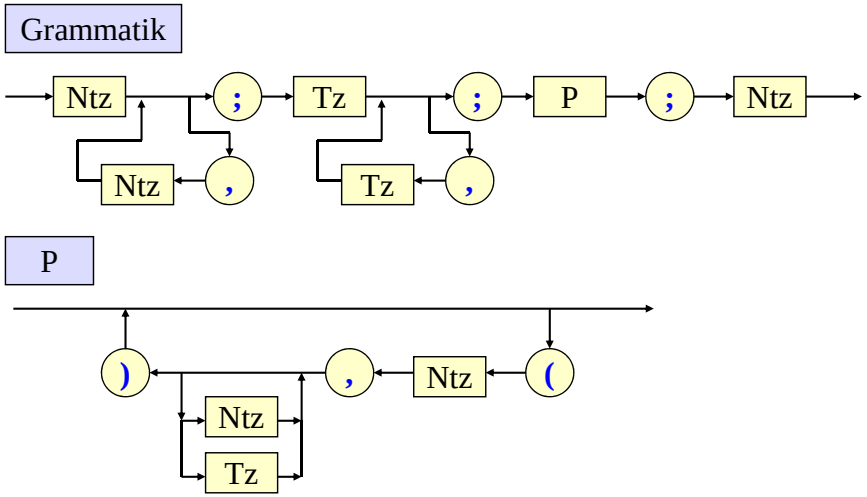
Syntaxdiagramme hierzu zeichnen:



<Ntz> ::= '**N**' '**1**' {'**0**' | '**1**'}

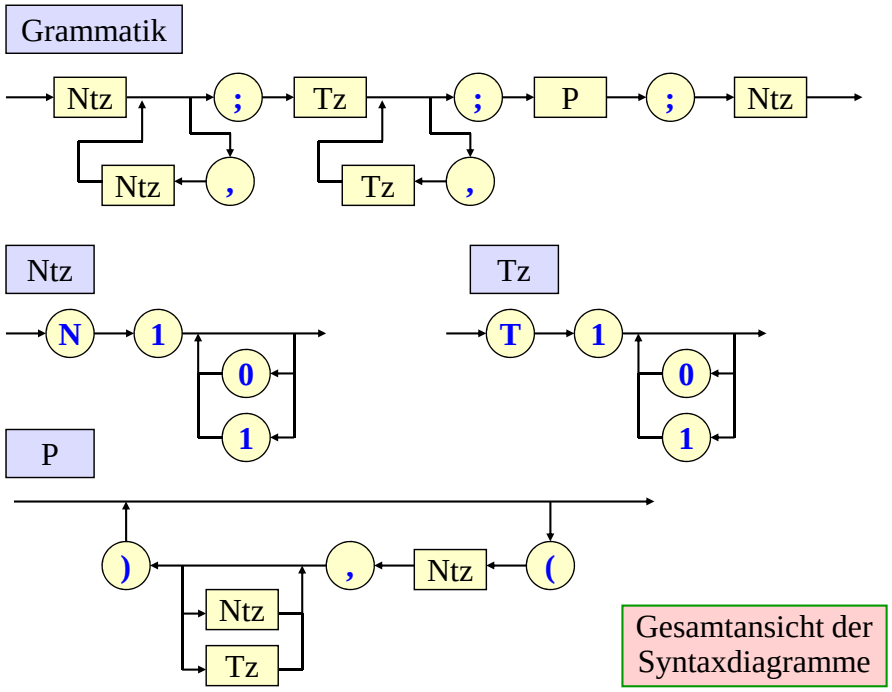
<Tz> ::= '**T**' '**1**' {'**0**' | '**1**'}

Syntaxdiagramme hierzu: (<rSeite> kann man leicht einsparen)



<P> ::= {'**(**' <Ntz> '**,**' <rSeite> '**)**'}

<rSeite> ::= { <Ntz> | <Tz> }



Nun erzeugen wir ein Wort u mit dieser EBNF, z.B.:

N1,N11;T10,T101;(N1,N1N11)(N1,T10)(N11,T101);N1 $\in \Sigma_G^*$

Zugehörige Grammatik: $G_u = (V_u, \Sigma_u, P_u, S_u)$ mit $V_u = \{$**N1**, **N11**$\}$, $\Sigma_u = \{$**T10**, **T101**$\}$ und $P_u = \{$**N1**$\rightarrow$**N1N11**, **N1**$\rightarrow$**T10**, **N11**$\rightarrow$**T101**$\}$. Die erzeugte Sprache lautet $L(G_u) = \{$**T10**$\}\{$**T101**$\}^*$.

Indem man die Zeichen nach irgendeiner Vorschrift umcodiert (z.B.: **N1** $\leftrightarrow$ S, **N11** $\leftrightarrow$ Y, **T10** $\leftrightarrow$ a, **T101** $\leftrightarrow$ b), gewinnt man aus G_u eine gleichwertige Grammatik G, die bis auf Umbenennung von Zeichen genau dem oben abgeleiteten Wort u entspricht:

$G = (V, \Sigma, P, S)$ mit $V = \{S, Y\}$, $\Sigma = \{a, b\}$ und $P = \{ S \rightarrow SY,\ S \rightarrow a,\ Y \rightarrow b \}$. Die erzeugte Sprache lautet $L(G) = \{a\}\{b\}^* = \{a\,b^k \mid k \geq 0\}$.

2.9.3 Hinweise

Hinweis 1: Beliebige Grammatiken lassen sich auf die gleiche Weise beschreiben; man muss nur als "linke Seite" der Regeln eine nicht-leere Folge von Nichtterminalzeichen zulassen.

Hinweis 2: Manches lässt sich nicht mit einer EBNF darstellen, nämlich alle "Kontext bezogenen" Bedingungen. Insbesondere:

- Alle Nichtterminalzeichen müssen paarweise verschieden sein.
- Alle Terminalzeichen müssen paarweise verschieden sein.
- Alle Zeichen, die in den Regeln vorkommen, müssen in den Auflistungen der Terminal- bzw. Nichtterminalzeichen vorkommen.

Hinweis 3: Über die Eindeutigkeit der dargestellten Grammatik kann man hier keine Aussage machen. Sie muss i. A. für jede Grammatik einzeln bewiesen werden.

Hinweis 4: Bei der Definition von Programmiersprachen setzt man bei der Semantik gerne ein schrittweises Vorgehen ein: Zuerst definiert man eine sehr kleine Sprache, wie wir es z. B. mit den Forderungen 2.1.5 (A1) bis (A9) (ohne A8b, c, d; in A7 reicht die einseitige Alternative) getan haben. Ist die Bedeutung dieser "Kern"-Sprache bekannt, so erweitert man sie und führt die Bedeutung der neuen Sprachelemente auf die der Kernsprache zurück. Z.B. kann man auf diese Weise die for- und die repeat-Schleife, die zweiseitige Fallunterscheidung, eine exit-Anweisung usw. schrittweise hinzunehmen und durch äquivalente Programmstücke beschreiben.

Im Übersetzerbau ist dieses Vorgehen als Bootstrapping bekannt, wobei man immer mächtigere Übersetzer einer Sprache in eine andere erhält. Am Ende kann man sogar einen optimierenden Übersetzer in der Sprache selbst schreiben und von einem "schlechten" Übersetzer realisieren lassen.

2.9.4: Bemerkungen zur Definition von Programmiersprachen

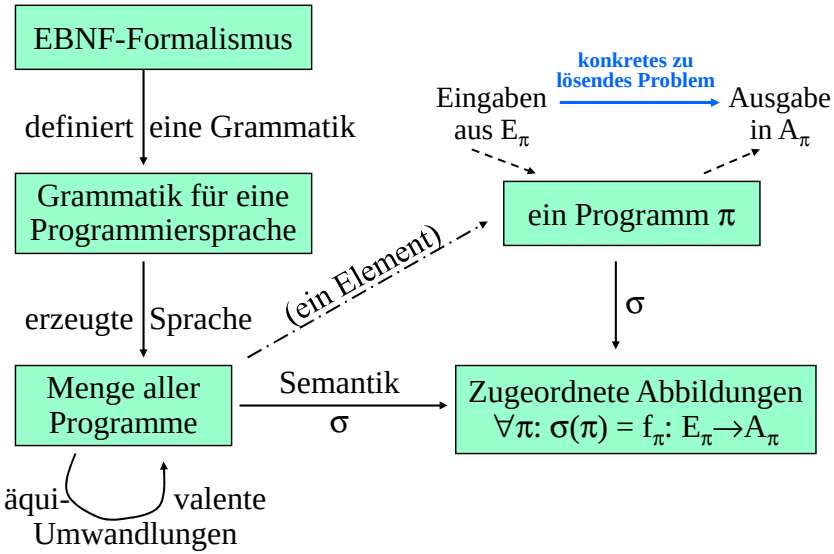
Die Syntax wird durch eine kontextfreie Grammatik bzw. eine EBNF definiert, der man Zusatzbedingungen umgangssprachlich hinzufügt (vgl. Hinweis 2). Oft lässt sich aus der EBNF ein Parser automatisch erzeugen.

Die Semantik wird meist anhand vieler Beispiele erläutert. Es gibt jedoch auch formale Methoden, um einem Programm die realisierte Abbildung zuzuordnen (Stichwörter: denotationale Semantik, axiomatische Semantik, vgl. späteres Kapitel 7).

Bei der Definition geht man meist schrittweise vor, wobei man die Sprache ständig um neue Sprachelemente anreichert. (Dies entspricht gerade dem "Bootstrapping".)

Die Eindeutigkeit der EBNF einer Programmiersprache muss getrennt nachgewiesen werden (diese Eigenschaft ist "unentscheidbar").

Skizze zu Syntax und Semantik:



2.10 Historische Anmerkungen

Zum Begriff Algorithmus: Hiermit verbindet man das griechische Wort arithmós (Zahl), vor allem aber den arabisch-persischen Mathematiker Mohamad Ibn Musa Al-Chwarismi; dieser lebte von 780 bis ca. 850 n.Chr., stammte aus der Region südöstlich des Kaspischen Meeres, arbeitete am Hof des Kalifen von Bagdad und hat neben anderen das "Kitab al-jabr w'al-muqabala" (das Buch über die "Regeln der Wiedereinsetzung und Reduktion") geschrieben; es behandelt die Lösungen von linearen und quadratischen Gleichungen; im Titel tritt das hier mit "Wiedereinsetzung" übersetzte Wort "algebra" erstmals in der Mathematik auf. In der lateinischen Fassung wird der Autor mit "Algorithmi" bezeichnet. Hieraus entstand das Wort Algorithmus als Begriff für "exaktes Rechenverfahren".

Algorithmen sind als mathematische Lösungsverfahren recht alt. Der Euklidische Algorithmus zur Berechnung des größten gemeinsamen Teilers natürlicher Zahlen

while b≠0 do r := a mod b, a := b; b := r od

stammt aus der Zeit um 300 v. Chr. Das Newtonsche Verfahren zur Berechnung einer einfachen Nullstelle von stetig differenzierbaren Funktionen f (wähle einen Anfangswert für x und die Genauigkeit δ geeignet)

while |f(x)| > δ do x := x - f(x) / f '(x) od

wird seit ca. 1670 verwendet. Das Gaußsche Eliminationsverfahren zur Lösung linearer Gleichungssysteme wird seit Anfang des 19. Jahrhunderts eingesetzt.

Die formale Definition für "Algorithmus " erfolgte 1936 unabhängig voneinander in drei Arbeiten, und zwar über den Lambda-Kalkül (dieser λ-Kalkül bildet die Grundlage der funktionalen Programmierung) von Alanzo *Church* (1903-1995), über μ-rekursive Funktionen (dies sind einfache Programme mit den Konstruktoren ";", "if-then" und "while") von Steven Cole *Kleene* (1909-1994) und über eine auf Zeichen arbeitende Maschine (die Turingmaschine) von Alan Mathison *Turing* (1912-1954).

Ab nun wurde es möglich, Aussagen über Algorithmen herzuleiten und "Unmöglichkeitsbeweise" wie den Satz 2.3.2 exakt zu führen. In der Folgezeit wurden weitere Kalküle als gleichwertig zu den drei oben genannten Darstellungen für Algorithmen nachgewiesen.

Zur Darstellung der Grundbereiche: Für Boolean und character sind sie recht alt (vor dem griechischen Alphabet plus Satzzeichen gab es bereits bei den Phöniziern und bei den Ägyptern vor 1300 v. Chr. ein buchstabenorientiertes Alphabet). Der ASCII-Code wurde Anfang der 1950er Jahre fixiert.

Das dezimale Stellenwertsystem für natürliche Zahlen hat sich ab dem 7. Jahrhundert in Indien entwickelt und gelangte durch die Araber bis Ende des 12. Jahrhunderts nach Europa. Das mathematische Standardwerk "liber abbaci", das "Buch der Rechenkunst" von Leonardo Pisano (genannt "Fibonacci") aus dem Jahre 1202, verwendete und verbreitete dieses System im Westen. Hier traten erstmals negative Zahlen auf. Somit gibt es bereits seit 800 Jahren die uns geläufige Darstellung für den Datentyp Integer.

Gottfried Wilhelm Leibniz beschreibt 1697 das Binärsystem, das vieles vereinfacht. Aber erst nach 1930 werden diese Ideen technisch für den Bau von Rechenmaschinen umgesetzt.

Erweitert man das Dezimalsystem auf Brüche, so erhält man die Darstellung des Datentyps Real. Diese Erweiterung nahm erstmals Al-Kasi, Direktor der Sternwarte von Samarkant, 1427 vor.

Die Darstellung im Zweierkomplement erfolgte mit der Entwicklung digitaler Rechenautomaten in den 1940er Jahren.

Zur Programmierung: Der englische Mathematiker *Charles Babbage* (1792-1871) entwirft ab 1838 die "Analytical Engine", eine modern anmutende Rechenmaschine mit Steuerwerk, Rechenwerk und Programmspeicher. Seine Assistentin ist *Ada Augusta Countess of Lovelace* (1815-1852, Tochter von Lord Byron), die in Ergänzung eines von ihr übersetzten Artikels die Verwendung von Programmen anregt und hierfür erste Programme schreibt. So wurde sie zur "ersten Programmiererin". Alle diese Arbeiten geraten jedoch in Vergessenheit und werden erst nach dem Bau der ersten Computer wieder entdeckt.

Die "moderne" Programmierung beginnt ab 1940 mit Folgen von Maschinenbefehlen, zwischen 1956 und 1961 mit Programmen in den Programmiersprachen FORTRAN, ALGOL, LISP, APL und COBOL. Das Programmieren wird ein Kerngebiet der neuen Wissenschaft "Informatik".

Zu Grammatiken und BNF: Eine wesentliche Arbeit hierzu stammt 1959 von Noam Chomsky. Hierin sind die Typ 0, 1, 2 und 3 Grammatiken und Sprachen und deren diverse Eigenschaften beschrieben. Es folgen viele Arbeiten, vor allem über kontextfreie Grammatiken. Die Backus-Naur-Form BNF entstand im Rahmen der Entwicklung der Programmiersprache ALGOL ab 1957; sie ist nach ihren wesentlichen Erfindern, dem Amerikaner J. Backus und dem Dänen P. Naur, benannt. Die Syntax der meisten heutigen Programmiersprachen wird in EBNF formuliert. Die Syntaxdiagramme wurden als Veranschaulichung von BNF und kontextfreien Grammatiken in den 1960er Jahren vorgeschlagen.

2.11 Übungsaufgaben

Aufgabe 1: *Division*
Beschreiben Sie die Division zweier natürlicher Zahlen; das Ergebnis sei eine Gleitpunktzahl beschränkter Länge.

i) Verwenden Sie eine iterierte Subtraktion.
ii) Beschreiben Sie den zeichenweise arbeitenden Algorithmus, den Sie aus der Schule kennen.
iii) Nehmen Sie an, die Zahl a besitzt n Stellen, die Zahl b besitzt m Stellen und das Ergebnis möge genau k Stellen besitzen. Wie viele Schritte führt der Algorithmus ii) dann durch?

Aufgabe 2: *Zahldarstellungen natürlicher Zahlen*
a) Schreiben Sie die Zahlen 5, 26, 144, 5040 jeweils zur Basis 2, 3, 7, 16, -2 und -3.
b) Schreiben Sie diese vier Zahlen als Viertupel (r_1, r_2, r_3, r_4) wobei r_i die Reste bzgl. der Zahlen $q_1 = 3$, $q_2 = 11$, $q_3 = 17$ und $q_4 = 26$ entsprechend Hinweis 3 nach 2.4.8 sind. Multiplizieren Sie 5 und 26 sowie 144 und 5040 in dieser Reste-Darstellung und prüfen Sie Ihre Resultate.

Aufgabe 3: *Zahldarstellungen rationaler Zahlen*
a) Schreiben Sie die Zahlen 3.4, 169.52 und -888.88 zur Basis 2 und 12 mit 9 Nachkommastellen (vgl. 2.4.12) sowie in der Gleitpunktdarstellung mit Mantissenlänge 11 und Exponentenlänge 5 (vgl. 2.4.14 bis 16).
b) Schreiben Sie -8.86 und -0.14 zur Basis 2 mit dem Zwei-Komplement und addieren Sie sie in dieser Darstellung.

Aufgabe 4: *Datentypdeklarationen*
Beschreiben Sie folgende Daten mit Hilfe der in Kapitel 1 vorgestellten Datentypen und Konstruktoren:
a) Liste der Nummern der Großbuchstaben im ASCII-Code.
b) Menge der Vornamen aller Freunde und Freundinnen.
c) Zuordnung der Geburtsjahre zu jedem/r Freund/Freundin.
d) Datentyp für die Wochentage und Datentyp für fünf selbst zu definierende Wetterlagen (kühl, regnerisch, sonnig, ...).
e) Zuordnung von Wochentag und Wetterlage zu der bei dieser Situation geeigneten Kleidung.
f) Jede Teilmenge einer angeordneten Menge M mit n Elementen lässt sich durch einen n-stelligen Booleschen Vektor beschreiben, dessen i-te Komponente genau dann true ist, wenn das i-te Element in der Teilmenge liegt. Definieren Sie auf diese Weise den Datentyp "Menge der Teilmengen von M".

Aufgabe 5: *Operationen auf Sprachen (vgl. 2.6.6)*
Es sei A die zweielementige Menge $A = \{a, b\}$.
(1) Bilde zur Menge $L = \{a, bb\}$ die Mengen LL und LLL.
(2) Skizzieren Sie ein Verfahren, das zu einem Wort $w \in A^*$ feststellt, ob $w \in L^*$ ist oder nicht.
(3) Sei $J = \{ab\}$. Beschreiben Sie die Mengen J^* und $A^* \backslash J^*$.
(4) Sei $K = \{b, ab\}$. Beschreiben Sie die Menge $(K \cup L)^*$.
(5) Beschreiben Sie die Menge $\{(abb)^n \mid n > 0\}$ mit Hilfe der Sprachen J und K und der Sprachoperationen Vereinigung, Differenz, Durchschnitt, Konkatenation und/oder Iteration.
(6) Sei $H = \{ba, bb\}$. Geben Sie einen Algorithmus für das Wortproblem der Menge $(H \cup J)^*$ an, d. h., geben Sie ein Verfahren an, das zu jedem Wort $w \in A^*$ feststellt, ob $w \in (H \cup J)^*$ gilt oder nicht.

Aufgabe 6: *Erzeugte kontextfreie Sprachen*
Es seien $V = \{S, A\}$ und $\Sigma = \{0, 1\}$. Welche Sprachen L_i werden von folgenden kontextfreien Grammatiken $G_i = (V, \Sigma, P_i, S)$ erzeugt mit
$P_1 = \{ S \to 0, S \to 1 \}$
$P_2 = \{ S \to 1, S \to S, S \to 0, A \to 0 \}$
$P_3 = \{ S \to 0S, S \to 00 \}$
$P_4 = \{ S \to S1, S \to 00, S \to 1, A \to 1, A \to AS \}$
$P_5 = \{ S \to 0A0, S \to 1, A \to 0S0 \}$
$P_6 = \{ S \to 0AS, S \to 0, A \to 1A1, A \to 1 \}$
$P_7 = \{ S \to AA, A \to 0A0, A \to 1 \}$
$P_8 = \{ S \to AS, A \to \varepsilon, A \to 0S1, S \to A \}$
$P_9 = \{ S \to SAS, A \to 10, A \to 0S1A, S \to A \}$

Aufgabe 7: *Eindeutige kontextfreie Grammatiken?*
Betrachten Sie folgende Grammatiken $G_i = (V, \Sigma, P_i, S)$ mit $V = \{S, A, B\}$ und $\Sigma = \{0, 1, 2, 3\}$. Welche sind eindeutig? Begründen Sie Ihr "ja" oder geben Sie ein mehrdeutiges Wort an.

$P_1 = \{ S \to 0, S \to A, A \to 0 \}$
$P_2 = \{ S \to 1, S \to S \}$
$P_3 = \{ S \to 0SS, S \to 1 \}$
$P_4 = \{ S \to S1, S \to 0, A \to AA, A \to 1, A \to AS \}$
$P_5 = \{ S \to 00B01S11, S \to 00B01S10S11, S \to 2, B \to 3 \}$
$P_6 = \{ S \to A0S, S \to A, A \to B1A, A \to B, B \to 2S2, B \to 3 \}$
$P_7 = \{ S \to AB, A \to 0A1, A \to \varepsilon, B \to 0B, B \to \varepsilon \}$
$P_8 = \{ S \to A1B, S \to B1A, A \to 0A0, A \to 1, B \to 0B, B \to 0 \}$
$P_9 = \{ S \to AS, A \to 1B, B \to 0S, B \to 1AS, S \to 0B, S \to 2 \}$

Anmerkung: P_5 hat etwas mit der ein- und zweiseitigen Fallunterscheidung, P_6 etwas mit arithmetischen Ausdrücken zu tun.

Aufgabe 8: *Syntaxdiagramme*
Geben Sie zu folgenden kontextfreien Grammatiken $G_i = (V, \Sigma, P_i, S)$ mit $V = \{S, A, B\}$ und $\Sigma = \{0, 1, 2, 3\}$ möglichst einfache Syntaxdiagramme an:

$P_1 = \{ S \to 0SS, S \to 1 \}$
$P_2 = \{ S \to AA, S \to 0, A \to AA, A \to 1, A \to AS \}$
$P_3 = \{ S \to 0B1S3, S \to 0B1S2S3, S \to 00, B \to 11 \}$
$P_4 = \{ S \to A0S, S \to A, A \to B1A, A \to B, B \to 2S2, B \to 3 \}$
$P_5 = \{ S \to ABS, A \to 0A1, A \to \varepsilon, B \to 0B, B \to \varepsilon, S \to 2 \}$
$P_6 = \{ S \to A1B, S \to B1A, A \to 0A0, A \to 1, B \to 0B, B \to 0 \}$

Aufgabe 9:
Definieren Sie die Syntax der BNF mit Hilfe der BNF.

2.12 Einschub: Was Sie in der Mathematik lernten (oder hätten gelernt haben sollen).

Eine Menge ist eine Zusammenfassung von paarweise verschiedenen Elementen zu einem Ganzen.
Eine Menge kennzeichnet man stets durch geschweifte Klammern "{" und "}".
Man kann sie beschreiben, indem man ihre Elemente auflistet, evtl. mit "...", sofern das Bildungsgesetz klar ist, z.B.:

Dezimalziffer = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
IB = {0, 1} $\triangleq$ {false, true} (Menge der Binärziffern bzw. der Booleschen Werte),
Wochentag = {Mo, Di, Mi, Do, Fr, Sa, So},
$\mathbb{N}_0$ = {0, 1, 2, 3, ...} (Menge der natürlichen Zahlen mit der 0).

noch Einschub

Oder man kann eine Menge M durch eine charakteristische Eigenschaft beschreiben nach dem Schema
M = { a | a besitzt die Eigenschaft .... }.

Beispiele:
G = {a | a ist eine Dezimalziffer und a ist als Zahl durch 2 teilbar},
T = {w | w ist ein Familienname und w kommt im Telefonbuch von Stuttgart des Jahres 2009 vor},
Prim = {p | p ist eine natürliche Zahl, p > 1 und p ist nur durch die Zahlen 1 und p teilbar},
E = {w | w ist der Name eines Ortes, dessen kürzeste Entfernung nach Stuttgart 50 km beträgt}.

noch Einschub

Wenn M eine Menge ist und a zur Menge M gehört, so sagt man, a ist ein Element von M, und schreibt hierfür: $a \in M$. Gehört a nicht zu M, so schreibt man $a \notin M$.

In einer Menge kommt kein Element mehrfach vor. Die Reihenfolge, in der die Elemente aufgeschrieben werden, spielt keine Rolle. Die drei Mengen {1, 3, 1, 2, 3, 3, 3}, {1, 2, 3} und {3, 1, 2} sind also gleich.

Gleichheit zweier Mengen: $M = N$ genau dann, wenn

(1) für jedes $a \in M$ gilt $a \in N$ und
(2) für jedes $a \in N$ gilt $a \in M$.

Wenn M nicht gleich N ist, dann ist M ungleich N, geschrieben $M \neq N$. $M \neq N$ gilt also genau dann, wenn es ein Element a gibt, das in N, aber nicht in M, oder das in M, aber nicht in N liegt.

noch Einschub

Vereinigung zweier Mengen M und N:

$M \cup N = \{ a \mid a \in M \text{ oder } a \in N\}$.

Durchschnitt zweier Mengen M und N:

$M \cap N = \{ a \mid a \in M \text{ und } a \in N\}$.

Differenz zweier Mengen M und N:

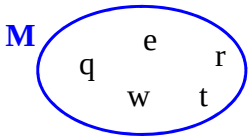
$M \setminus N = \{ a \mid a \in M \text{ und } a \notin N\}$.

Machen Sie sich diese Operationen auf Mengen an der folgenden grafischen Darstellung klar.

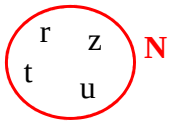
noch Einschub

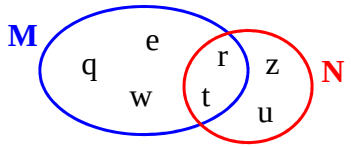
Übliche grafische Darstellung für zwei Mengen M und N:

M = {q, w, e, r, t}



N = {r, t, z, u}





noch Einschub

Definition: "Enthalten", "Teilmenge":
M heißt Teilmenge von N, im Zeichen: $M \subset N$ oder $N \supset M$, genau dann, wenn für jedes $a \in M$ gilt $a \in N$.

Ist M nicht Teilmenge von N, so schreibt man $M \not\subset N$.

M und N sind genau dann gleich, wenn M in N und N in M enthalten sind. Formel hierfür: $M = N \Leftrightarrow M \subset N$ und $N \subset M$.

Die spezielle Menge, die kein Element besitzt, nennt man die leere Menge und bezeichnet sie mit dem Symbol $\varnothing$. Sie ist Teilmenge jeder Menge.

noch Einschub

Zu grundlegenden (Zahlen-) Mengen in Mathematik und Informatik: siehe Beginn von Abschnitt 2.4: von den natürlichen Zahlen bis zu den komplexen Zahlen.

Eine Abbildung $f: M \to N$ ordnet jedem $a \in M$ höchstens ein Element $f(a) \in N$ zu. M heißt Vorbereich oder Urbildbereich (engl.: domain), N heißt Nachbereich oder Bildbereich (engl.: codomain).

Wird hierbei jedem $a \in M$ ein Element $f(a) \in N$ zugeordnet, so heißt die Abbildung total.

Falls es möglich sein kann (aber nicht muss), dass manchen $a \in M$ kein Element durch die Abbildung zugeordnet wird, so spricht man von einer partiellen Abbildung. Wird einem $a \in M$ kein Element durch die Abbildung f zugeordnet, so sagt man, f(a) ist "nicht definiert" oder "undefiniert". Insbesondere ist jede totale Abbildung auch eine partielle Abbildung.

noch Einschub

Statt "Abbildung" sagt man oft "Funktion", auch wenn dieser Begriff in der Mathematik meist auf Abbildungen beschränkt ist, deren Vor- und Nachbereiche Zahlenmengen sind.

Die Menge der Elemente
$Def(f) = \{a \in M \mid \text{es gibt ein } b \in N \text{ mit } f(a) = b\} \subset M$
heißt Definitionsbereich von f.
$f: M \to N$ ist also genau dann total, wenn $Def(f) = M$ gilt.

Die Menge der Elemente
$f(M) = \{b \in N \mid \text{es gibt ein } a \in M \text{ mit } f(a) = b\} \subset N$
heißt Bild von f oder Bild von M unter f.

Die Menge der Abbildungen von M nach N bezeichnet man mit N^M oder mit $Abb(M,N) = \{f \mid f: M \to N\} = N^M$.

(Achten Sie bei Büchern genau darauf, ob jeweils partielle oder totale Abbildungen gemeint sind!)

noch Einschub

Eine Abbildung $f: M \to N$ heißt injektiv genau dann, wenn für alle $a, b \in M$ mit $a \neq b$ gilt: $f(a) \neq f(b)$.

Eine Abbildung $f: M \to N$ heißt surjektiv genau dann, wenn es zu jedem $c \in N$ mindestens ein $a \in M$ mit $f(a) = c$ gibt.
Beachte: f ist genau dann surjektiv, wenn $f(M) = N$ gilt.

Eine Abbildung $f: M \to N$ heißt bijektiv genau dann, wenn f injektiv und surjektiv ist.

Wenn $f: M \to N$ eine bijektive Abbildung ist, dann gibt es zu jedem $c \in N$ ein eindeutig bestimmtes $a \in M$ mit $f(a) = c$. Setze daher $f^{-1}(c) = a$. Dies definiert eine bijektive Abbildung $f^{-1}: N \to M$. Diese Abbildung f^{-1} heißt die Umkehrabbildung oder die Inverse zu f. Bijektive Abbildungen besitzen also stets eine Inverse, die ebenfalls bijektiv ist.

noch Einschub

Wir führen folgende Begriffe für Mengen M ein: "endlich", "unendlich", "abzählbar", "überabzählbar", "aufzählbar" und "beschränkt". Mit #M oder |M| bezeichnen wir die Anzahl der Elemente der Menge M.

M ist endlich $\Leftrightarrow$ M ist leer oder es gibt eine natürliche Zahl n und eine Bijektion $f: M \to \{1, ..., n\}$.
(In diesem Falle ist $\#M = n \in \mathbb{N}_0$.)

M ist unendlich $\Leftrightarrow$ M ist nicht endlich.

M abzählbar (unendlich) $\Leftrightarrow$ Es gibt eine Bijektion von M zu den natürlichen Zahlen $\mathbb{N}$.

M ist höchstens abzählbar $\Leftrightarrow$
M ist endlich oder M ist abzählbar unendlich.

noch Einschub

Beispiele:
{1, 7, 12, 16, 19, 21, 22} und {Mo,Di,Mi,Do,Fr,Sa,So}
sind endliche Mengen (mit jeweils 7 Elementen).

Die Menge der geraden natürlichen Zahlen
$G = \{0, 2, 4, 6, 8, 10, 12, ...\} = \{ 2 \cdot n \mid n \in \mathbb{N}_0 \}$
ist eine abzählbar unendliche Menge. Eine Bijektion
$f: \mathbb{N} \to G$ lautet: $f(n) = 2 \cdot (n-1)$ für alle $n \in \mathbb{N}$.

Auch die Menge der Primzahlen
Prim = {p | p ist eine natürliche Zahl, p > 1 und p ist nur durch die Zahlen 1 und p teilbar}
ist abzählbar unendlich, weil es unendlich viele Primzahlen gibt (denn: gäbe es nur endlich viele Primzahlen, so müsste deren um 1 erhöhtes Produkt eine weitere Primzahl sein).

noch Einschub

Folgerung: Wenn eine Menge M höchstens abzählbar ist, dann ist auch jede Teilmenge von M höchstens abzählbar.

Beweis:
Es sei M höchstens abzählbar und $K \subseteq M$ eine Teilmenge von M. Falls K endlich ist, so ist K nach Definition höchstens abzählbar. Sei also K unendlich. Dann muss auch M als Obermenge von K unendlich sein. Weil M abzählbar ist, gibt es eine Bijektion $f: \mathbb{N} \to M$. Wir bezeichnen mit $m_n := f(n)$ das n-te Element von M und können M in der Form $M = \{m_1, m_2, m_3, ...\}$ schreiben. Unter diesen Elementen m_i müssen auch die Elemente von K vorkommen. Setze g(1) = erstes Element von K, das in der Reihenfolge $m_1, m_2, m_3, ...$ vorkommt, g(2) = zweites Element von K, das in der Reihenfolge $m_1, m_2, m_3, ...$ vorkommt usw. Dies definiert eine bijektive Abbildung $g: \mathbb{N} \to K$, d.h., auch K ist abzählbar. *(Dieser Beweis ist nicht konstruktiv!)*

noch Einschub

M ist überabzählbar ⇔ M ist nicht "höchstens abzählbar".

M ist aufzählbar ⇔ M ist leer oder es gibt einen Algorithmus, der jeder natürlichen Zahl n ein Element der Menge M zuordnet, und zu jedem Element von M gibt es mindestens eine hierdurch zugeordnete natürliche Zahl.

Andere Formulierung:
M ist genau dann aufzählbar, wenn entweder M eine endliche Menge ist oder wenn es eine bijektive Abbildung $f: \mathbb{N} \to M$ gibt, die berechenbar ist (d.h., es gibt ein Programm, so dass dieses Programm zu jeder natürlichen Zahl n das n-te Element von M nach endlich vielen Schritten ermittelt).
Damit eine abzählbar unendliche Menge M aufzählbar ist, verlangt man also zusätzlich, dass es mindestens eine Bijektion $f: \mathbb{N} \to M$ gibt, die man mit einem Programm berechnen kann.

noch Einschub

Folgerung:
Jede aufzählbare Menge ist höchstens abzählbar.

Die Umkehrung gilt jedoch nicht. Ein Beispiel ist das "Komplement des Halteproblems", z. B. die Menge aller Texte, die zu Algorithmen gehören, die bei Eingabe ihres eigenen Textes nicht anhalten.
(Anderer Beweis: Siehe am Ende dieses Abschnitts.)
Hieraus kann man folgern: Eine aufzählbare Menge besitzt in der Regel Teilmengen, die nicht aufzählbar sind.
(Es gilt sogar: Jede unendliche aufzählbare Menge hat unendlich viele nicht-aufzählbare Teilmengen.
Hinweis: Malen Sie diesen Sachverhalt auf und versuchen Sie, ihn sich auf diese Weise zu veranschaulichen.)

noch Einschub

Der Begriff "beschränkt" ist ein relativer Begriff; er bezieht sich auf die jeweils betrachtete Umgebung. Hierzu muss man *vorher* eine Schranke k festlegen. *Bei abzählbaren Mengen* lautet eine solche Definition dann beispielsweise:

M ist beschränkt $\Leftrightarrow$ Es gibt eine vorab festgelegte natürliche Zahl k, so dass M nicht mehr als k Elemente besitzt.

Eine beschränkte Teilmenge von $\mathbb{N}$ ist endlich, aber aus der Eigenschaft "endlich" folgt nicht "beschränkt", da man im Allgemeinen die Schranke k nicht vorab festlegen kann.

noch Einschub: Beispiele

Die leere Menge $\emptyset$ ist endlich. Sie besitzt $\#\emptyset = 0$ Elemente.

Für jede natürliche Zahl n ist die Menge {1, 2, ..., n} endlich.
Die Menge der natürlichen Zahlen $\mathbb{N} = \{1, 2, 3, ...\}$ bzw.
$\mathbb{N}_0 = \{0, 1, 2, 3, ...\}$ und die Menge der ganzen Zahlen
$\mathbb{Z} = \{ ..., -2, -1, 0, 1, 2, 3, ... \}$ sind abzählbar unendlich und selbstverständlich aufzählbar.

Die Menge der rationalen Zahlen

$$\mathbb{Q} = \{n/m \mid n \in \mathbb{Z}, m \in \mathbb{N}, \text{ n und m teilerfremd}\}$$

ist abzählbar unendlich *und* aufzählbar (wie zeigt man dies?).

Die Menge der reellen Zahlen $\mathbb{R}$ und die Menge der komplexen Zahlen $\mathbb{C}$ sind überabzählbar (Beweis über das Cantorsche Diagonalverfahren, vgl. Mathematikvorlesungen).

noch Einschub: Beweisidee

Dass $\mathbb{Q}$ aufzählbar ist, zeigt folgende Skizze ($n \in \mathbb{Z}$, $m \in \mathbb{N}$):

m \ n	0	1	-1	2	-2	3	-3	4	-4	5	-5	....
1	$\frac{0}{1}$	$\frac{1}{1}$	$\frac{-1}{1}$	$\frac{2}{1}$	$\frac{-2}{1}$	$\frac{3}{1}$	$\frac{-3}{1}$	$\frac{4}{1}$	$\frac{-4}{1}$	$\frac{5}{1}$	$\frac{-5}{1}$	
2	$\frac{0}{2}$	$\frac{1}{2}$	$\frac{-1}{2}$	$\frac{2}{2}$	$\frac{-2}{2}$	$\frac{3}{2}$	$\frac{-3}{2}$	$\frac{4}{2}$	$\frac{-4}{2}$	$\frac{5}{2}$	$\frac{-5}{2}$	
3	$\frac{0}{3}$	$\frac{1}{3}$	$\frac{-1}{3}$	$\frac{2}{3}$	$\frac{-2}{3}$	$\frac{3}{3}$	$\frac{-3}{3}$	$\frac{4}{3}$	$\frac{-4}{3}$	$\frac{5}{3}$	$\frac{-5}{3}$	
4	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{-1}{4}$	$\frac{2}{4}$	$\frac{-2}{4}$	$\frac{3}{4}$	$\frac{-3}{4}$	$\frac{4}{4}$	$\frac{-4}{4}$	$\frac{5}{4}$	$\frac{-5}{4}$	
5	$\frac{0}{5}$	$\frac{1}{5}$	$\frac{-1}{5}$	$\frac{2}{5}$	$\frac{-2}{5}$	$\frac{3}{5}$	$\frac{-3}{5}$	$\frac{4}{5}$	$\frac{-4}{5}$	$\frac{5}{5}$	$\frac{-5}{5}$	
6	$\frac{0}{6}$	$\frac{1}{6}$	$\frac{-1}{6}$	$\frac{2}{6}$	$\frac{-2}{6}$	$\frac{3}{6}$	$\frac{-3}{6}$	$\frac{4}{6}$	$\frac{-4}{6}$	$\frac{5}{6}$	$\frac{-5}{6}$	
7	$\frac{0}{7}$	$\frac{1}{7}$	$\frac{-1}{7}$	$\frac{2}{7}$	$\frac{-2}{7}$	$\frac{3}{7}$	$\frac{-3}{7}$	$\frac{4}{7}$	$\frac{-4}{7}$	$\frac{5}{7}$	$\frac{-5}{7}$	
⋮												

Nun geht man dieses Schema diagonal von rechts oben nach links unten durch, wobei man bereits aufgetretene rationale Zahlen wegstreicht. So entsteht eine "Aufzählung" der rationalen Zahlen.

noch Einschub:

Dass $\mathbb{Q}$ aufzählbar ist, zeigt folgende Skizze ($n \in \mathbb{Z}$, $m \in \mathbb{N}$):

m \ n	0	1	-1	2	-2	3	-3	4	-4	5	-5	....
1	$\frac{0}{1}$ 1	$\frac{1}{1}$ 2	$\frac{-1}{1}$ 3	$\frac{2}{1}$ 5	$\frac{-2}{1}$ 8	$\frac{3}{1}$ 11	$\frac{-3}{1}$ 15	$\frac{4}{1}$ 20	$\frac{-4}{1}$ 25	$\frac{5}{1}$ 30	$\frac{-5}{1}$ 37	
2	~~$\frac{0}{2}$~~	$\frac{1}{2}$ 4	$\frac{-1}{2}$ 6	~~$\frac{2}{2}$~~	~~$\frac{-2}{2}$~~	$\frac{3}{2}$ 16	$\frac{-3}{2}$ 21	~~$\frac{4}{2}$~~	~~$\frac{-4}{2}$~~	$\frac{5}{2}$ 38	$\frac{-5}{2}$	
3	~~$\frac{0}{3}$~~	$\frac{1}{3}$ 7	$\frac{-1}{3}$ 9	$\frac{2}{3}$ 12	$\frac{-2}{3}$ 17	~~$\frac{3}{3}$~~	~~$\frac{-3}{3}$~~	$\frac{4}{3}$ 31	$\frac{-4}{3}$ 39	$\frac{5}{3}$	$\frac{-5}{3}$	
4	~~$\frac{0}{4}$~~	$\frac{1}{4}$ 10	$\frac{-1}{4}$ 13	~~$\frac{2}{4}$~~	~~$\frac{-2}{4}$~~	$\frac{3}{4}$ 26	$\frac{-3}{4}$ 32	~~$\frac{4}{4}$~~	~~$\frac{-4}{4}$~~	$\frac{5}{4}$	$\frac{-5}{4}$	
5	~~$\frac{0}{5}$~~	$\frac{1}{5}$ 14	$\frac{-1}{5}$ 18	$\frac{2}{5}$ 22	$\frac{-2}{5}$ 27	$\frac{3}{5}$ 33	$\frac{-3}{5}$ 40	$\frac{4}{5}$	$\frac{-4}{5}$	~~$\frac{5}{5}$~~	~~$\frac{-5}{5}$~~	
6	~~$\frac{0}{6}$~~	$\frac{1}{6}$ 19	$\frac{-1}{6}$ 23	~~$\frac{2}{6}$~~	~~$\frac{-2}{6}$~~	~~$\frac{3}{6}$~~	~~$\frac{-3}{6}$~~	~~$\frac{4}{6}$~~	~~$\frac{-4}{6}$~~	$\frac{5}{6}$	$\frac{-5}{6}$	
7	~~$\frac{0}{7}$~~	$\frac{1}{7}$ 24	$\frac{-1}{7}$ 28	$\frac{2}{7}$ 34	$\frac{-2}{7}$ 41	~~$\frac{3}{7}$~~	$\frac{-3}{7}$	$\frac{4}{7}$	$\frac{-4}{7}$	$\frac{5}{7}$	$\frac{-5}{7}$	
⋮												

Die grünen Zahlen geben den Anfang einer Bijektion zwischen $\mathbb{N}$ und den rationalen Zahlen $\mathbb{Q}$ an. Die Reihenfolge beginnt mit 0, 1, -1, 1/2, 2, -1/2, 1/3, -2, -1/3, 1/4, 3, 2/3, -1/4, 1/5, -3, 3/2, -2/3, ...

noch Einschub:

Wenn M eine aufzählbare Menge ist, dann ist stets auch die Menge M* aufzählbar (Argument analog wie bei **Q**).

Zu Sprachen siehe 2.6.5 und 2.6.6. Aus den Definitionen folgt:
Jede Teilmenge einer abzählbaren Menge ist abzählbar.
Zu jeder aufzählbaren Menge gibt es einen Algorithmus, der diese Menge aufzählt; folglich kann es nicht mehr aufzählbare Mengen (über einem festen Alphabet) geben, als es Texte für Algorithmen gibt, d. h., die Menge aller aufzählbaren Mengen muss aufzählbar und somit auch abzählbar sein.
Die Menge aller Teilmengen über einem festen Alphabet ist aber überabzählbar, folglich muss es unendlich viele abzählbare Mengen geben, die nicht aufzählbar sind.

3. Daten, ihre Strukturierung und Organisation

3. Daten und ihre Strukturierung

Hinweis zur Syntax: Der aktuelle Stand ist als Anhang P unter
http://www.adaic.org/standards/05rm/RM.pdf
zu finden. Siehe auch:
http://www.iste.uni-stuttgart.de/ps/ada-doc/rm2005/RM-TOC.html

3.1 Programmaufbau (speziell in Ada 2005)

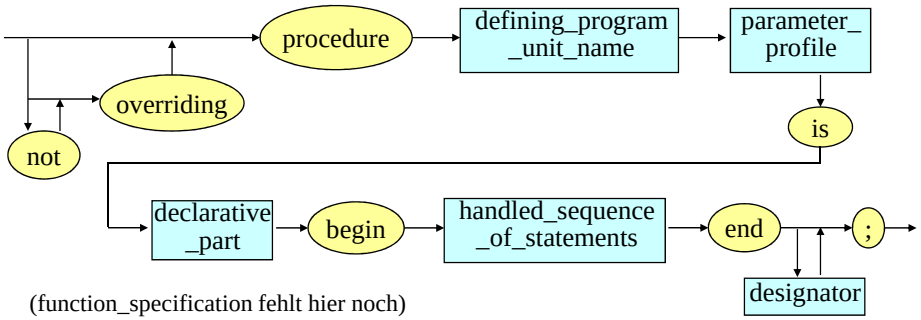
3.1.1 Syntax

subprogram_declaration ::=
[overriding_indicator] subprogram_specification ;

subprogram_specification ::=
procedure_specification | function_specification

procedure_specification ::=
procedure defining_program_unit_name parameter_profile
function_specification ::=
function defining_designator parameter_and_result_profile

subprogram_body ::= [overriding_indicator]
subprogram_specification is declarative_part
begin handled_sequence_of_statements end [designator] ;
overriding_indicator ::= [not] overriding



(function_specification fehlt hier noch)

Ein Programm bekommt einen Namen

defining_program_unit_name

Ein Name bezeichnet ein Programmobjekt. In der Regel ist ein Name ein "Bezeichner" (= identifier), in Ada 95:

identifier ::= identifier_letter {[underline] letter_or_digit}

Bezeichner wurden bereits behandelt (2.1.5, 2.6.2, 2.7.5). In Ada 2005 kann man Bezeichner noch etwas anreichern. Ein reserviertes Wort darf nicht als Bezeichner verwendet werden.

Ein Name kann aber auch ein indizierter Bezeichner sein, z. B. X(i), oder ein Operatorsymbol (siehe Ende von 1.7). In Ada 2005 lautet die Syntax für einen Namen:

name ::= direct_name | explicit_dereference | indexed_component
| slice | selected_component | attribute_reference |
type_conversion | function_call | character_literal

Der Anweisungsteil folgt nach dem Deklarationsteil in Form einer handled_sequence_of_statements . Dies ist eine Folge von Anweisungen (sequence_of_statements), der eine Ausnahmebehandlung (exception) folgen darf.

handled_sequence_of_statements ::=
sequence_of_statements
[exception exception_handler {exception_handler}]

Eine Anweisung (statement) ist eine einfache Anweisung (simple_statement) oder eine zusammengesetzte Anweisung (compound_statement), der eine oder mehrere Marken (label) vorangestellt werden dürfen. Dies hatten wir bereits in 1.11 besprochen, Syntaxbeispiele ab 2.8.5.
Den "exception-Teil" (Ausnahmebehandlung) nehmen wir zunächst nur zur Kenntnis. Er soll Abweichungen von der normalen Berechnung und Fehler abfangen.

3.1.2 Deklarationsteil: Programme beginnen nach dem "is" mit einem Deklarationsteil (declarative_part). Syntax:

declarative_part ::= {declarative_item}

declarative_item ::= basic_declarative_item | body

basic_declarative_item ::=
basic_declaration | aspect_clause | use_clause

basic_declaration ::=

type_declaration	subtype_declaration
object_declaration	number_declaration
subprogram_declaration	abstract_subprogram_declaration
package_declaration	renaming_declaration
exception_declaration	generic_declaration
generic_instantiation	null_procedure_declaration

Hier erkennt man erneut, wie umfangreich die Sprache Ada ist. Für unsere Zwecke benötigen wir zunächst nur die Typdeklaration (type_declaration) und die Einführung von Variablen (mit oder ohne Indizierung).

Die komplette object_declaration lautet:

object_declaration ::=
defining_identifier_list : [aliased] [constant]
subtype_indication [:= expression] ; |
defining_identifier_list : [aliased] [constant]
access_definition [:= expression] ; |
defining_identifier_list : [aliased] [constant]
array_type_definition [:= expression] ; |
single_task_declaration |
single_protected_declaration

Wir benötigten bisher nur den Teil, mit dem man einfache und indizierte Konstanten und Variablen vereinbart:

object_declaration ::=
    defining_identifier_list : [constant]
        subtype_indication [:= expression] ; |
    defining_identifier_list : [constant]
        array_type_definition [:= expression] ;

defining_identifier_list ::=
  defining_identifier {, defining_identifier}

defining_identifier ::= identifier

subtype_indication ist für uns zunächst nur der Bezeichner für einen Datentyp. Die array_definition wurde bereits behandelt. "body" bezeichnet den Implementierungsteil einer vereinbarten Programmeinheit (Prozedur, Task, Paket usw.).

3.2 Lexikalische Einheiten (in Ada)

3.2.1 Zeichensatz

Wir wählen exemplarisch die Sprache Ada. Andere Sprachen sind ähnlich aufgebaut. Zunächst zum *Zeichensatz*.
Früher: Ada-95-Programme wurden im Zeichensatz Latin-1 gemäß ISO-Standard 8859 geschrieben. In 1.8.4 ist Latin-1 komplett abgedruckt. Latin-1 umfasst $2^8 = 256$ Zeichen einschließlich einiger Steuerzeichen, die sich nicht auf der Tastatur befinden. Der aus $2^7 = 128$ Zeichen bestehende ASCII-Code (siehe 2.4.6) ist hierin enthalten. Man kann durch "Val" und "Pos" auf alle Zeichen zugreifen (vgl. 1.8.1 und 1.8.4).
Heute: Ab Ada 2005 wird der "Multiple octet-coded character set" der Norm ISO/IEC 10646:2003 verwendet. Einige Zeichen werden hierbei verboten, die Bedeutung mancher Zeichen (insbesondere der grafischen Zeichen) ist abhängig von der Implementierung. (Ada stellt viele Zeichensätze bereit.)

3.2.2 Lexikalische Einheiten

Ein Programm ist eine Folge Zeichen aus dem Zeichensatz. Diese Folgen sind nicht beliebig aufgebaut, sondern sie bilden eine Folge von textuellen Einheiten, genannt "lexikalische Einheiten".

Ada kennt die folgenden lexikalischen Einheiten:
- Bezeichner zur Identifizierung von Programmobjekten,
- Literale zur Bezeichnung von Werten (numerisch, Zeichen, String),
- Begrenzer ("delimiter") mit spezieller Bedeutung, die zugleich lexikalische Einheiten voneinander trennen, und zwar:
  & ' ( ) * + , - . / : ; < = > | => .. ** := /= >= <= << >> <>
- 72 reservierte Wörter mit spezieller Bedeutung, siehe 1.4,
- Trennzeichen ("separator") zur besseren Aufbereitung (Zwischenraum, Tabulatorzeichen, Zeilenende),
- Kommentare zur besseren Verständlichkeit.

Beispiel: Betrachte folgende Programmzeile:

```
for i in 15..N loop  K1Z:=K6T+1; L:=N; end loop  -- Mache etwas
```

Begrenzer	Literale	Bezeichner	Wortsymbol	Kommentar
..	15	i	for	-- Mache etwas
:=	1	N	in	
+		K1Z	loop	
;		K6T	end	
		L		

Trennzeichen sind der Zwischenraum (außerhalb von Strings und Kommentaren), das Zeilenende und Formatierungszeichen.

■

Bezeichner = Identifikator für Programmobjekte wie Variablen, Funktionen, Typen, Marken usw.

Literal = Bezeichnung eines festen Wertes, vor allem Zahlen, Zeichen oder Zeichenketten (Text, String).

Begrenzer = Symbol zum Begrenzen lexikalischer Einheiten. Es kann aus einem oder mehreren Zeichen bestehen. Oftmals ein Operator.

Wortsymbol = Reserviertes Wort der Sprache wie for, if, and, mod usw. (oft unterstrichen oder fett oder farbig geschrieben).

Trennzeichen = Trennt lexikalische Einheiten voneinander, meist aus Gründen der Lesbarkeit.

Kommentar = Zeichenfolge vom Doppelstrich -- bis zum Zeilenende. Dient der Erläuterung und Lesbarkeit. Es sollte stets ein längerer Kommentar am Anfang stehen, und sie sollten zur Erläuterung ins Programm eingetreut sein.

comment ::= --{non_end_of_line_character}
numeric_literal ::= decimal_literal | based_literal
decimal_literal ::= numeral [.numeral] [exponent]
numeral ::= digit {[underline] digit}
exponent ::= E [+] numeral | E - numeral
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
based_literal ::=
    base # based_numeral [.based_numeral] # [exponent]
base ::= numeral
based_numeral ::= extended_digit {[underline] extended_digit}
extended_digit ::= digit | A | B | C | D | E | F
character_literal ::= 'graphic_character'
string_literal ::= "{string_element}"
string_element ::= "" | non_quotation_mark_graphic_character

In Bezeichnern wird *in Ada* nicht zwischen großen und kleinen Buchstaben unterschieden! A1g und a1G sind hier also die gleichen Bezeichner.

3.2.3 Wortsymbole

auch "reservierte Wörter" oder "Schlüsselwörter" genannt.

Diese Wörter haben eine feste Bedeutung in der jeweiligen Programmiersprache. Sie dürfen nicht als Bezeichner verwendet werden!

In Ada 95 waren es 69 Wörter, in Ada 2005 kamen die drei Wörter "interface", "overriding" und "synchronized" hinzu, siehe 1.4. In der Sprache Java gibt es 50 reservierte Wörter, in Delphi sind es 62.

Es gibt/gab Programmiersprachen, bei denen die reservierten Wörter umdefiniert oder als Bezeichner benutzt werden können (Standardbeispiel ist PL/I). Dies dient aber nicht der Lesbarkeit und Durchschaubarkeit von Programmen.

Gewisse Wörter haben eine vordefinierte Bedeutung, können aber vom Benutzer anderweitig verwendet werden. Beispiele in Ada sind Integer, storage_error, Float, Boolean, True usw. Die in der Ada-Umgebung vordefinierte Bedeutung geht dann innerhalb der jeweiligen Programmeinheit verloren.

Beachte: In Ada darf man auch nicht AND oder aNd als Bezeichner verwenden, da die Groß- und Kleinschreibung hier (im Gegensatz zu den meisten anderen Programmiersprachen) nicht unterschieden wird.

Hinweis: Manche reservierte Wörter haben in einer anderen Umgebung eine andere Bedeutung, z. B. bedeutet or in einem Booleschen Ausdruck das logische "Oder", während es in einer select-Anweisung eine nichtdeterministische Auswahl bezeichnet.

3.3 Überblick Datenstrukturen - Kontrollstrukturen

Datenstrukturen und Kontrollstrukturen stehen in einer engen Beziehung zueinander. *Jede Datenstruktur sollte nur mit der zu ihr gehörenden Kontrollstruktur bearbeitet werden.*

Hat man umgangssprachlich eine Lösungsidee oder einen Lösungsalgorithmus aufgeschrieben, so formalisiere man zunächst die Datenbereiche und formuliere anschließend, wie das Lösungsverfahren hierauf arbeiten soll.

Datenstrukturen, die in der Praxis häufiger auftreten, werden in den heutigen Programmiersprachen in Form von Moduln (Pakete, Klassen, ...), die in "Bibliotheken" abgelegt sind, mitgeliefert. Ein Großteil der detaillierten Programmierung entfällt daher für die Informatiker(innen), die sich zunehmend darauf konzentrieren, Probleme mit Hilfe *vorhandener* Strukturen zu beschreiben und zu lösen.

Bezeichnung	Mathem.Symbol	Datenstruktur	Kontrollstruktur	in Ada
Aufzählung einer Menge	$\{m_1, m_2, ..., m_k\}$	Aufzählungstyp	mit Succ und Pred	Succ, Pred und andere Attribute
Zushängende Teilmenge	[a..b]	range ..	for i:=a by d to b	for i in a..b
kartesisches Produkt	$M_1 \times M_2 \times ... \times M_n$	record	;	;
kartesisches Prod. mit sich	M^n	array <Index> of <Kompon.>	for i:=1 to n	for i in 1..n
disjunkte Vereinigung	$M_1 \cup M_2 \cup ... \cup M_n$	record mit case	case when ...	case ... when ... => ...
Folgen, Wortmenge	M^*	seq of ... oder Listen	while	while
Bäume, Graphen	Term, Ausdruck, Graph	BinBaum, AVL, Graph usw.	Rekursion (selten while)	Rekursion (oder while)
Potenzmenge	2^M	set of ...	for mit Eigenschaften	simulieren mit arrays, Listen usw.
Menge der Abbildungen	$Abb_p(M,N)$, N^M	function ... result procedure	Aufruf, Applikation	Aufruf, Zeiger auf Prozeduren
Menge der Namen	Hierarchien $\{M_1, M_2, ..., M_n\}$	Referenzen, Zeiger	new und dereferenzieren	new und .all (evtl. aliased)

Zusammenspiel von Daten- und Kontrollstrukturen

Welche Konstruktoren gibt es nun, um aus elementaren Datentypen zusammengesetzte Strukturen zu erhalten?

Mathematische Operatoren, um Bereiche zusammenzusetzen, sind (wir kopieren hier aus der Folie 2.4.19):

Unterbereiche, Intervalle (z.B. Einschränkung auf a..b),

kartesische Produkte (Datensätze, "record"),

n-faches Produkt einer Menge (Feld, Vektor, "array"),

(disjunkte) *Vereinigung* von Mengen (varianter record, union),

Potenzmenge ("set of "),

Funktionen zwischen Mengen (function, procedure),

Graphen, Relationen (realisiert durch Zeiger, pointer, "reference").

Diese Konstruktoren, die zugehörigen Kontrollstrukturen und ihre programmiersprachliche Beschreibung im Überblick:

3.4 Zeiger (access-Datentypen)

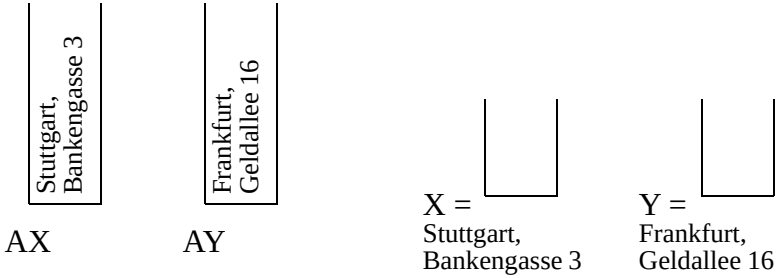
3.4.1 Begriffsbestimmung: Statt die Werte von Variablen direkt zu manipulieren, kann man auch mit Verweisen ("Zeiger" oder Pointer auf die Variablen) arbeiten. Man deklariert dann den Datentyp "Verweis auf ..." oder "Zeiger auf ...".

In Ada schreibt man: access <Datentyp>.

Dies sieht zunächst etwas gekünstelt aus, ist aber ein gängiges Konzept aus dem Alltag: Die gesuchte Information liegt nicht unmittelbar vor, sondern befindet sich an der Stelle ... oder an der Adresse ... .

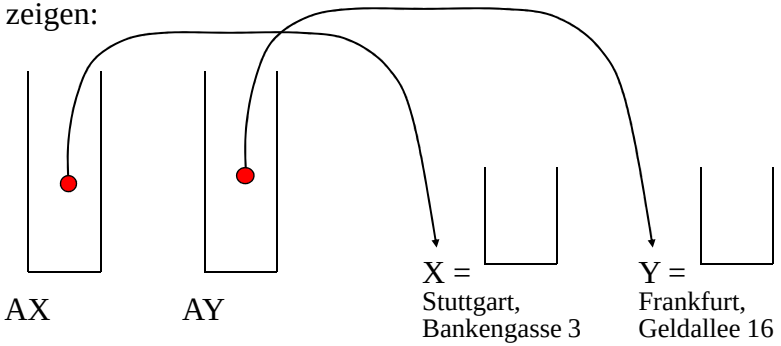
Solche Verweise auf die Stellen, wo sich die eigentlichen Daten befinden, sind zum Beispiel Adressbücher, Standortangaben von Büchern in einer Bibliothek, Verweise in diesem Skript der Art "siehe Kapitel ...." oder "siehe Lehrbuch ..., Seite ...", Hinweise auf Paragraphen im Rechtswesen usw.

Umgangssprachliches Beispiel: Addiere die Euro-Barbestände zweier Bankfilialen; die eine befindet sich in Stuttgart in der Bankengasse 3 und die andere in Frankfurt, Geldallee 16.



Mit AX+AY meinen wir dann, dass die beiden Variablen zu addieren seien, die sich an den Adressen befinden, die in AX und in AY gespeichert sind.

Umgangssprachliches Beispiel (Fortsetzung): Dieser Sachverhalt wird grafisch durch Zeiger ausgedrückt, die auf Variablen zeigen:



Mit AX+AY meinen wir also: Folge dem Zeiger, der von AX, und dem Zeiger, der von AY ausgeht, und addiere die Werte, die sich in den referenzierten Variablen befinden. ■

3.4.2 Externe und interne Namen

Jede bisher betrachtete Variable besitzt einen im Programm verwendeten "externen" Namen (meist ein Bezeichner, eventuell mit Zusätzen). Sie muss sich aber zugleich irgendwo im Speicher befinden. Die Stelle, ab der die Variable (bzw. ihr Wert) im Speicher steht, nennt man den "internen Namen" oder die Adresse der Variablen.

Der (externe) Name der Variablen ist dann nichts anderes als die symbolische Bezeichnung für die (Speicher-) Adresse. Diese Auffassung erklärt auch, warum verschiedene Variablen innerhalb eines Blocks verschiedene Namen besitzen müssen: Zu jedem Block gehört ein eigener Speicherbereich und in jedem Speicherbereich ist jedem Bezeichner genau eine Adresse zugeordnet.

Der Wert einer Zeiger-Variablen, die auf eine andere Variable verweist, ist (logisch betrachtet) der Name und gleichzeitig (physisch betrachtet) die Adresse der referenzierten Variablen.

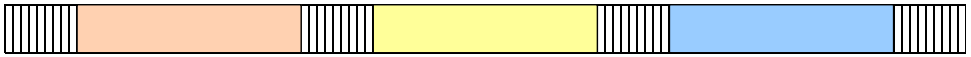
In Ada darf einer "normalen" Zeigervariablen nur ein interner Name zugewiesen werden, also eine Variable, die man nicht mit einem externen Namen im Programm ansprechen kann, sondern die man nur über Zeigervariablen erreichen kann. Mit "allgemeinen" Zeigervariablen kann man auch auf Variablen des Kellerspeichers zugreifen (die zugehörigen reservierten Wörter lauten in Ada: all und aliased).

Zusätzlich trennt Ada die Variablen, die durch Deklarationen (also durch die Vereinbarung eines externen Namens) definiert werden, sorgfältig von denen, auf die nur verwiesen werden kann. Die ersteren stehen im Kellerspeicher, letztere in der "Halde".

3.4.3 Speichereinheiten: Im Berechnungsmodell für die Sprache Ada gibt es also drei große Speichereinheiten:

- Der Speicherbereich für das Programm.
- Der ("Keller"-) Speicher für die deklarierten Variablen. Dieser Speicher unterliegt der Blockstruktur, die zugleich die Lebensdauer und den Sichtbarkeitsbereich festlegt.
- Die "Halde" (in Ada "storage pool" genannt) für dynamisch erzeugte Variablen oder Speicherbereiche, die keinen externen Namen, sondern nur noch interne Namen, d.h., nach außen nicht-sichtbare Adressen besitzen.

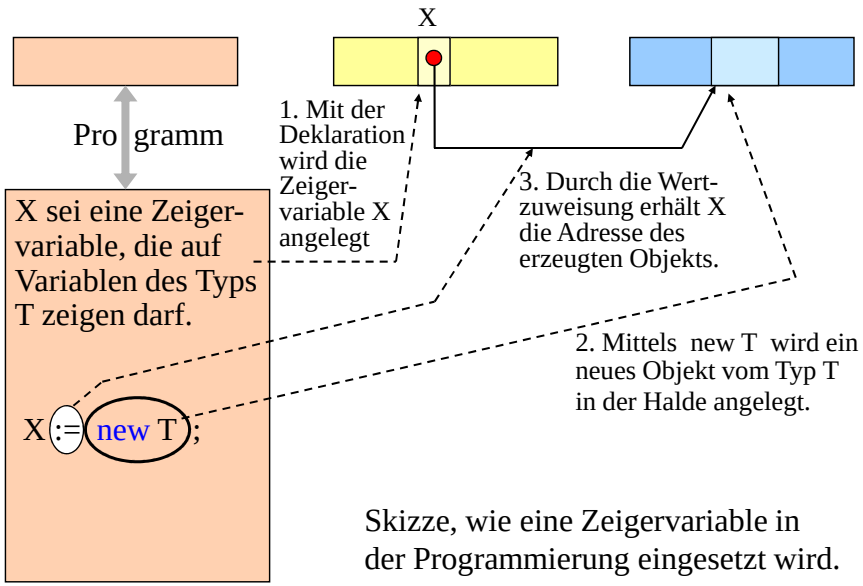
Gesamter Speicher



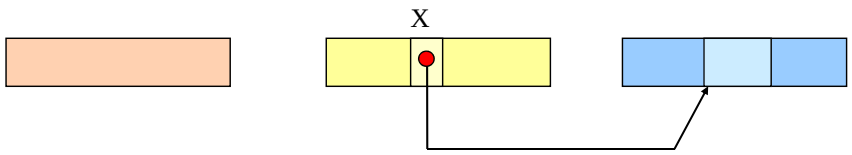
Programmspeicher, wird beim Ablauf des Programms nicht verändert.

Kellerspeicher: Pulsiert mit dem Wechsel zwischen Blöcken und Prozeduren.

Halde: Wo Platz ist, werden referenzierte Daten abgelegt. Wird dauernd aktualisiert.



Skizze, wie eine Zeigervariable in der Programmierung eingesetzt wird.



X := new T ;

Ergebnis der Deklaration und dieser Wertzuweisung.

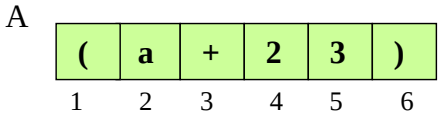
3.4.4 Verwendung von Zeigertypen

Mit Zeigern lassen sich vor allem Daten, deren Struktur sich ständig ändert, gut beschreiben. Solche durch Zeiger zusammengehaltene Daten heißen "dynamische Datenstrukturen".

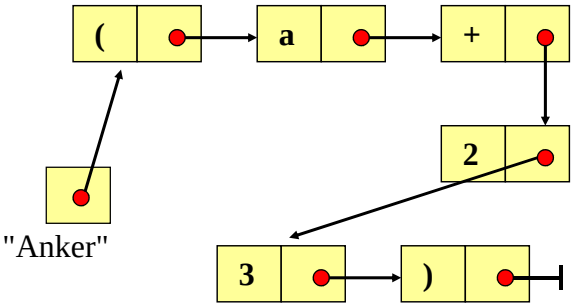
Die einzelnen Teile solcher Strukturen brauchen keinen externen Namen zu besitzen. Man muss nur wissen, wie man zum nächsten Element oder zu einer Menge von benachbarten Elementen gelangt. Statt des externen Namens genügt der "Verweis" (Referenz oder Zeiger) auf den internen Namen. Von außen muss nur mindestens ein Zeiger auf mindestens ein Element der Struktur (z.B. das "Anfangselement") bekannt sein.

Wir beginnen mit der Darstellung eines arithmetischen Ausdrucks und erläutern Listen und ihre Operationen.

Beispiel: Array-Darstellung A: array (1..6) of Character

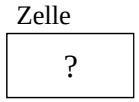


Andere Darstellung (anschaulich):

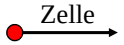


Der Anker nennt den Ort, wo das erste Element '(' steht, dort steht dann, wo das zweite Element 'a' abgelegt ist, usw.

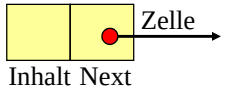
Darstellung in Ada mittels "access":




```
type Zelle;
```




```
type Ref_Zelle is access Zelle;
```




```
type Zelle is record
                Inhalt: Character;
                Next: Ref_Zelle;
end record;
```

Prinzip: In einigen Programmiersprachen wie Ada müssen *alle* Bestandteile einer Deklaration *zuvor* definiert werden. Daher muss eine "Vor-Information" type Zelle gegeben werden, die auf die kommende Präzisierung hinweist. Dies haben wir bereits bei rekursiven Funktionen, siehe 1.7 viertes Beispiel, kennen gelernt. In anderen Programmiersprachen verlangt man dies in der Regel nicht.


```
type Zelle;       -- Vorab-Information, dass später "Zelle" definiert wird
type Ref_Zelle is access Zelle;
type Zelle is  record
          Inhalt: Character;
          Next: Ref_Zelle;
end record;
```

Z: Zelle; Anker: Ref_Zelle;

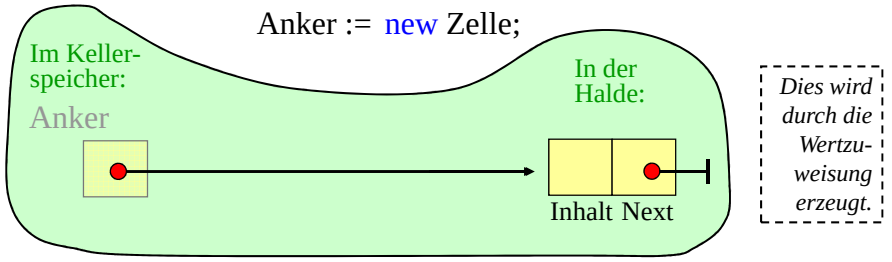
Hierdurch werden zwei Variablen im Kellerspeicher angelegt. In Ada werden Zeigervariablen stets mit null (="kein Verweis") initialisiert:

⊣ bedeutet "kein Verweis" (null)

Will man eine Variable in der Halde anlegen, auf die nur verwiesen werden soll, so verwendet man das Schlüsselwort new mit Angabe des zu erzeugenden Datentyps (new ist ein "Generator für Speicherplatz", engl. "allocator").

Anker: Ref_Zelle; ... Anker := new Zelle;

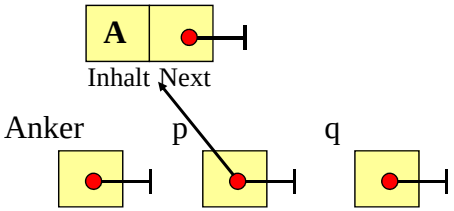
Veranschaulichung:



Nun können wir eine beliebig lange Kette von Verweisen bilden:

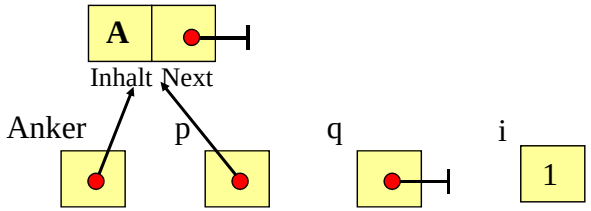
```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.Inhalt := 'A'; p.Next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.Inhalt := Character'Val (65+i);
        p.Next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



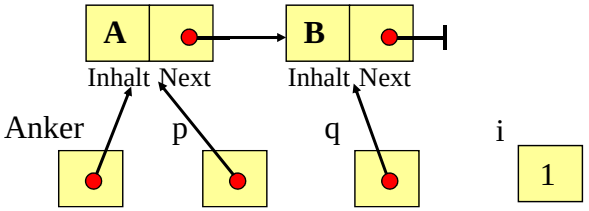

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.Inhalt := 'A'; p.Next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.Inhalt := Character'Val (65+i);
        p.Next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



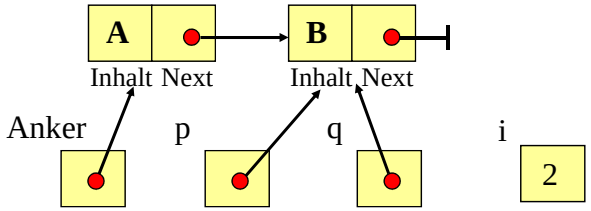

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.Inhalt := 'A'; p.Next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.Inhalt := Character'Val (65+i);
        p.Next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



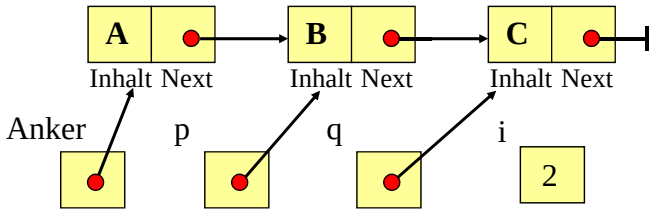

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.Inhalt := 'A'; p.Next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.Inhalt := Character'Val (65+i);
        p.Next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



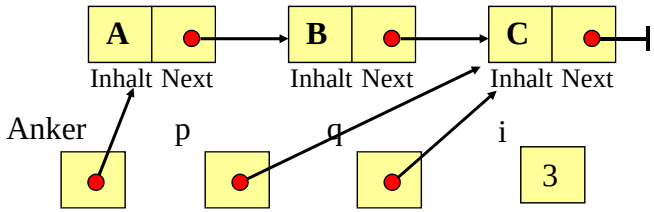
```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.Inhalt := 'A'; p.Next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.Inhalt := Character'Val (65+i);
        p.Next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



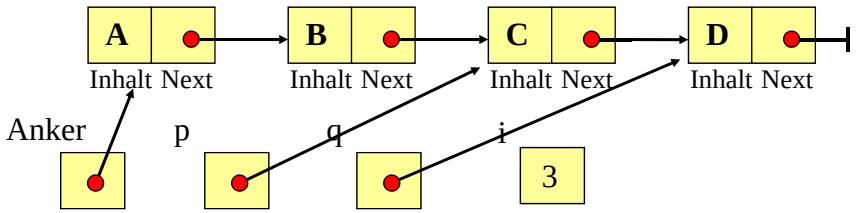

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.Inhalt := 'A'; p.Next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.Inhalt := Character'Val (65+i);
        p.Next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



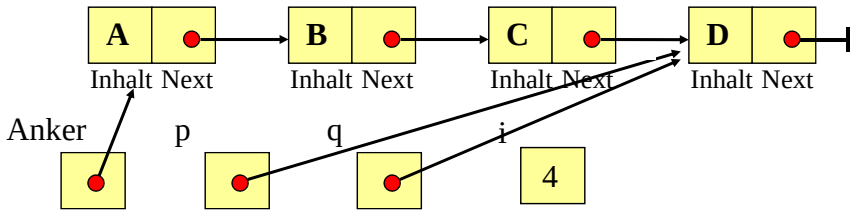

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.Inhalt := 'A'; p.Next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.Inhalt := Character'Val (65+i);
        p.Next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



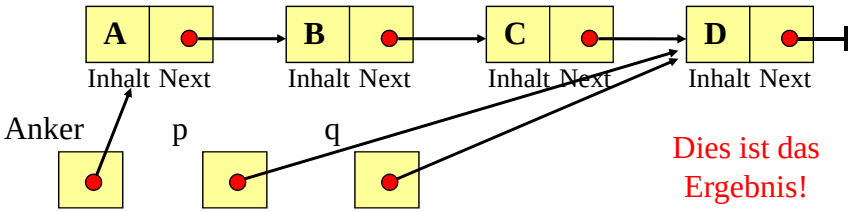

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.Inhalt := 'A'; p.Next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.Inhalt := Character'Val (65+i);
        p.Next := q; p:=q;
end loop;  ...
```

Veranschaulichung:

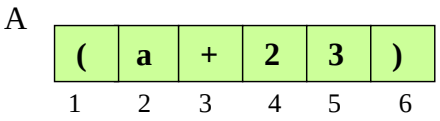


```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.Inhalt := 'A'; p.Next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.Inhalt := Character'Val (65+i);
        p.Next := q; p:=q;
end loop;  ...
```

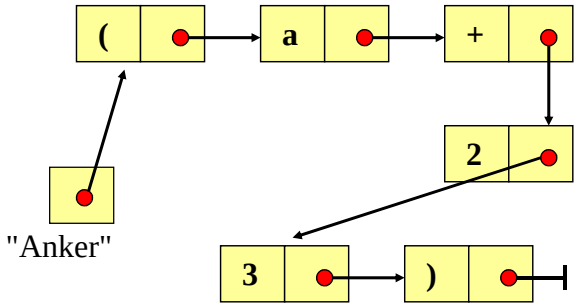
Veranschaulichung:



Zurück zu unserem Beispiel. Die Aufgabe lautet: Wir wollen anstelle des Feldes



eine Struktur der folgenden Form aufbauen:



Wir formulieren die Lösung dieser Aufgabe als Programm auf der folgenden Folie.

3.4.5: Allgemeines Programm zum Aufbau einer Verweiskette.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Char_Liste is
type Zelle;  type Ref_Zelle is access Zelle;
type Zelle is record Inhalt: Character; Next: Ref_Zelle; end record;
Anker, p, q: Ref_Zelle;  Anzahl: Natural;

begin  ...
    p := new Zelle; Get(p.Inhalt); Anker := p; Anzahl := 1;
    while not End_Of_File loop
        q:= new Zelle;
        Get(q.Inhalt); Anzahl := Anzahl+1;
        p.Next := q;  q.Next := null;    -- q.Next:=null; ist hier überflüssig
        p := q;
    end loop;
...
end;
```

Die einzelnen Zeichen der Eingabe werden nacheinander eingelesen und in eine "Verweiskette", auf deren Anfang die Variable Anker zeigt, abgelegt. Zugleich wird die Länge der Liste in "Anzahl" notiert.

3.4.6 Wertzuweisungen bei Zeigern

Es sei X eine Zeigervariable. Was bedeutet dann die Wertzuweisung X := α genau? Und wie darf "α" aussehen?
Wir klären zunächst:

Was bewirkt eine Deklaration (z.B. die Deklaration von X)?

- Sie führt ein Objekt ein und gibt ihm einen "externen" Namen.
- Sie ordnet dem Objekt einen "Typ" zu und klärt somit die Verwendungsmöglichkeiten und Beschränkungen.
- Mit der Deklaration wird Speicherplatz für das Objekt im Kellerspeicher angelegt, dessen (Anfangs-) Adresse ("interner" Name) fest mit dem externen Namen verbunden wird.

"Normale" Zeigervariablen erhalten als Werte nur die internen Namen von Variablen in der Halde. Diese werden nicht durch eine Deklaration, sondern durch "new" erzeugt.

"Allgemeine" Zeigervariablen können dagegen auf Variablen, die im Kellerspeicher stehen, verweisen (siehe 3.4.3).
Hier kann man bei der Typdefinition durch "access constant" den Variablenzugriff auf lesenden Zugriff beschränken.

Für normale Zeigervariablen ist in X := α der Ausdruck α beschränkt auf
- null,
- auf eine einzelne Zeigerkonstante oder -variable jeweils vom gleichen Typ wie X oder
- auf die Erzeugung eines neuen Objekts mittels new ... .

Wenn T ein Datentyp ist, dann ist
type Ref_T is access T
der zugehörige Zeigertyp.

G, H, L: Ref_T;
sind Deklarationen entsprechender Zeigervariablen.

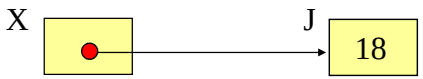
R, S: constant Ref_T;
sind Deklarationen entsprechender Zeigerkonstanten.

Konkrete Beispiele (vgl. auch 3.4.5):

```
type Rational is record Zähler: Integer; Nenner: Positive; end record;
type Bel_Vektor is array (Integer range <>) of Rational;
type Ref_Bel_Vektor is access Bel_Vektor;
X: Ref_Bel_Vektor;  Y: Ref_Bel_Vektor (1..50);
Z: constant Ref_Bel_Vektor := new Bel_Vektor (1..50 => (0,1));  ...
X := new Bel_Vektor (1..50);
Y := Z;  X := Y;  Y.all(25).Zähler := 3; Z.all(25).Nenner := 28;
```


Oft dienen die Zeiger nur als elegante Möglichkeit, die Werte der Variablen, auf die verwiesen wird, zu manipulieren.

Es sei X eine Zeigervariable, die auf die Integer-Variable mit dem (internen) Namen und dem Wert 18 zeigt.



Mit X.all wird dann die Variable J bezeichnet. Man nennt diesen Übergang vom Zeiger zur Variablen, auf die gezeigt wird, "Dereferenzierung" (= man folge dem Zeiger) und drückt dies in Ada durch ein nachgestelltes .all aus (vgl. Y.all(25)).
Der Typ dieser Variablen ist meist ein Record oder eine andere Datenstruktur, und das "all" deutet an, dass die gesamte Datenstruktur das Ergebnis der Dereferenzierung ist.

Prinzipiell muss man in Ada die Verweise explizit mittels "all" angeben. Im Falle, dass eine Zeigervariable auf ein Record verweist, darf man das "all" weglassen und direkt die Punktnotation verwenden (die Punktnotation zeigt bereits an, dass eine Derefenzierung erfolgen muss). Beispiel:

```
type Zelle;
type Ref_Zelle is access Zelle;
type Zelle is  record
                  Inhalt: Character;
                  Next: Ref_Zelle;
end record;
X: Ref_Zelle; R: Zelle := ('B', null);
X := new Zelle('S', new Zelle('E', null));
X.Next.Inhalt := 'm';          -- eigentlich:  X.all.Next.all.Inhalt
R.Inhalt := X.Inhalt;          -- eigentlich:  R.Inhalt := X.all.Inhalt
```

In Ada gibt es keine automatische Dereferenzierung (außer bei der Punktnotation). Grundsätzlich gilt *für jede Wertzuweisung*, dass

X := α

nur zulässig ist, wenn X und α vom gleichen Datentyp sind. Dies gilt auch für Zeiger. Beispiel:

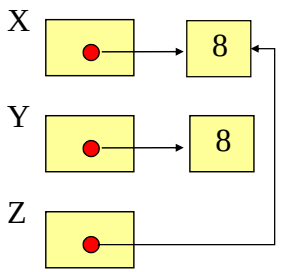
```
type Ref_Int is access Integer;
X, Y: Ref_Int;  K: Integer := 0;  ...
X := new Integer'(3);  Y := new Integer'(5);
K := X;          -- verboten, da verschiedener Datentyp
X := Y.all;      -- verboten, da verschiedener Datentyp
X := K+4;        -- verboten, da verschiedener Datentyp
K := X.all;      -- zulässig, da gleicher Datentyp
X.all := K;      -- zulässig, da gleicher Datentyp
X.all := Y.all;  -- zulässig, da gleicher Datentyp
```


3.4.7 Gleichheit bei Zeigern

Auf Zeigertypen ist stets auch die Gleichheit definiert. Wenn X und Y zwei Zeigervariablen gleichen Typs sind, so bedeutet X = Y, dass X und Y auf dieselbe Variable zeigen.

```
type Ref_Int is access Integer;
X, Y, Z: Ref_Int;
X := new Integer'(8);
Y := new Integer'(8);
Z := X;
```

Dann sind X = Z true und X = Y false. Ebenso sind X/=Y und Y/=Z true. Weiterhin ist Y.all = Z.all, da sich diese Bedingung auf den Inhalt und nicht auf die Zeiger bezieht.



3.5 Listen

3.5.1 Für die Menge M* der endlichen Folgen über einer Menge M (= Menge der Wörter oder freies Monoid über M)

$M^* = \{a_1 a_2 \ldots a_n \mid n \geq 0 \text{ und } a_i \in M \text{ für } i = 1, 2, \ldots, n\}$

kann man ein Feld

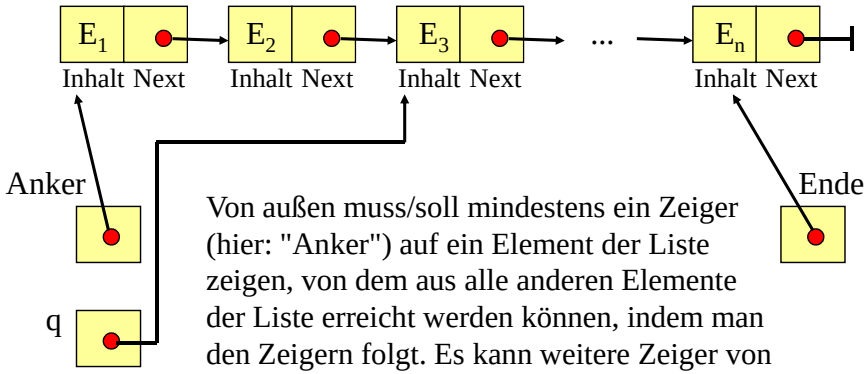
type M_Folgen is array (<Indexdatentyp>) of <Datentyp>

verwenden, allerdings ist man dann begrenzt auf den (in der Praxis beschränkt großen) Indexbereich.

Wie im Beispiel 3.4.4 veranschaulicht stellt man Folgen von Elementen meist durch eine Verweiskette dar. Eine solche Kette von Verweisen, in der die Elemente wie auf einer Perlen-schnur angeordnet sind (also keine Verzweigungen enthalten), nennt man eine (lineare) Liste.

Hinweis: n = 0 ist zulässig (leeres Wort). Hierzu gehört die "leere Liste" null (= kein Verweis).

Anschauliche Darstellung einer Liste:



Von außen muss/soll mindestens ein Zeiger (hier: "Anker") auf ein Element der Liste zeigen, von dem aus alle anderen Elemente der Liste erreicht werden können, indem man den Zeigern folgt. Es kann weitere Zeiger von außen in die Liste hinein geben (hier: Ende, q).

Hierbei sind mit "Inhalt" das jeweilige Element E_i und weitere Informationen gemeint. Next zeigt auf das folgende Element.

Erinnerung: Eine oder mehrere Mengen ("Wertebereiche") zusammen mit hierauf definierten Operationen nennt man einen (konkreten) Datentyp.

Auch eine Liste ist ein Datentyp. Seine Wertemenge ist die Menge der Folgen M* zusammen mit mindestens den Mengen **IB** und $\mathbf{IN}_0$, um Vergleiche und Anzahlen (etwa die Länge einer Liste) beschreiben zu können. Was sind die Operationen?

Zum Beispiel: Initialisierung einer Liste:
Leere Liste (d.h., kein Element, leeres Wort).
Zum Beispiel: Suchen nach einem Element:
Stelle fest, ob ein Element in der Liste vorhanden ist.
Zum Beispiel: Einfügen:
Füge ein Element in die Liste ein (aber wo??).
Wir betrachten folgende sechs Grundoperationen für Listen und formulieren sie in Ada.

3.5.2 Grundoperationen auf (linearen) Listen

Wir benötigen einen "Rahmen", in dem wir die Operationen realisieren. Dieser Rahmen ist ein Block, der die Deklaration einer Liste enthält, also den Datentyp und einen Anker. Die zu definierenden Operationen sind dann Funktionen oder Prozeduren, deren formale Parameter die Funktionalität der Operationen widerspiegeln.

Der Rahmen wird auf der nächsten Folie beschrieben. Prinzipiell lässt sich jede der hier betrachteten Operationen als Funktion schreiben, die eine Liste und Zusatzinformationen erhält und einen Wert oder ein Tupel von Werten (auch vom Typ access ...) als Ergebnis zurück gibt. Manchmal formulieren wir diese Operationen aber auch als Prozeduren, wobei die Ergebnisse den jeweiligen aktuellen Parametern zugeordnet werden.

3.5.2.0 Der Rahmen für unsere Operationen:

```
procedure Anwendung is
type Item is ...              -- Datentyp der Elemente der Liste
type Zelle;  type Ref_Zelle is access Zelle;
type Zelle is record Inhalt: Item; Next: Ref_Zelle; end record;
Anker: Ref_Zelle;             -- Verweis auf die Liste
Anzahl: Natural;              -- Länge der Liste
function ....                 -- hier sind die Operationen einzufügen
procedure ....                -- hier sind die Operationen einzufügen
...                           -- sonstige Deklarationen
begin  ...                    -- Rumpf der Prozedur "Anwendung"
end Anwendung;
```

Hier fügen wir die im Folgenden formulierten Funktionen und Prozeduren ein.

3.5.2.1 Erste Operation: Initialisieren als leere Liste.

```
function Empty return Ref_Zelle is
begin  return null;  end Empty;
```

3.5.2.2 Zweite Operation: Prüfen, ob eine Liste leer ist.

```
function Isempty (A: Ref_Zelle) return Boolean is
begin  return (A = null);  end Isempty;
```

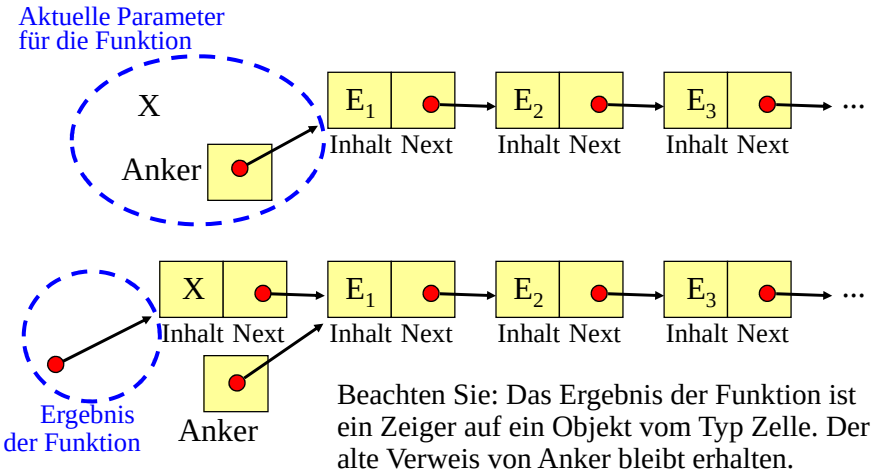
3.5.2.3 Dritte Operation: Erstes Element der Liste.

```
function First (A: Ref_Zelle) return Item is
begin  if not Isempty (A) then return A.Inhalt;
       else raise <Ausnahmebehandlung ... >; end First;
```

Man folgt hier dem Zeiger "A", trifft auf eine "Variable", die eine Komponente "Inhalt" besitzt und ermittelt deren Wert.

3.5.2.4 Vierte Operation: Hinzufügen eines Elements am Anfang.

```
function In_Front (E: Item; A: Ref_Zelle) return Ref_Zelle is
begin  return new Zelle'(E, A);  end In_Front;
```



Beachten Sie: Das Ergebnis der Funktion ist ein Zeiger auf ein Objekt vom Typ Zelle. Der alte Verweis von Anker bleibt erhalten.

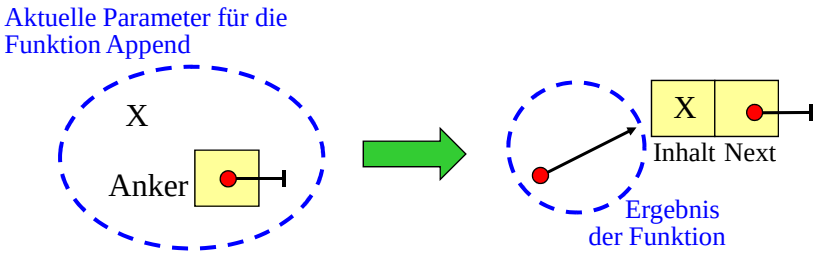
3.5.2.5 Fünfte Operation: Hinzufügen eines Elements am Ende.

```
function Append (E: Item; A: Ref_Zelle) return Ref_Zelle is
p, q: Ref_Zelle;
begin          -- p und q laufen bis ans Ende durch die Liste,
               -- wobei p auf das jeweils betrachtete und q auf
               -- das unmittelbar vor p stehende Element zeigt.
               -- Für p=null zeigt q auf das letzte Elemente der Liste.
   p := A; q := null;
   while p /= null loop q := p; p := p.Next; end loop;
   if  q = null then q := new Zelle'(E, null); return q;
      else  q.Next := new Zelle'(E, null); return A; end if;
end Append;
```

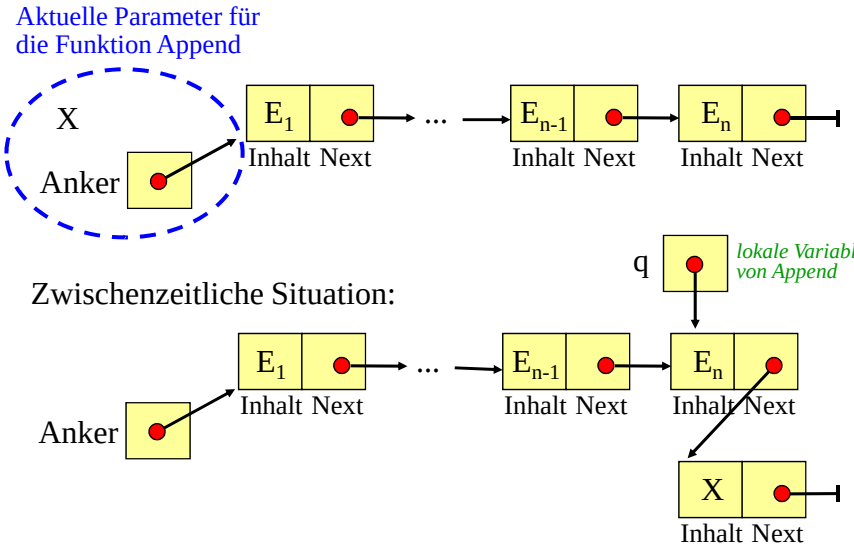
Aufruf dieser Funktion z.B. durch (X sei vom Typ Item)
Anker := Append (X, Anker);

Append (X, Anker) liefert einen Zeiger auf ein Objekt vom Typ Zelle.

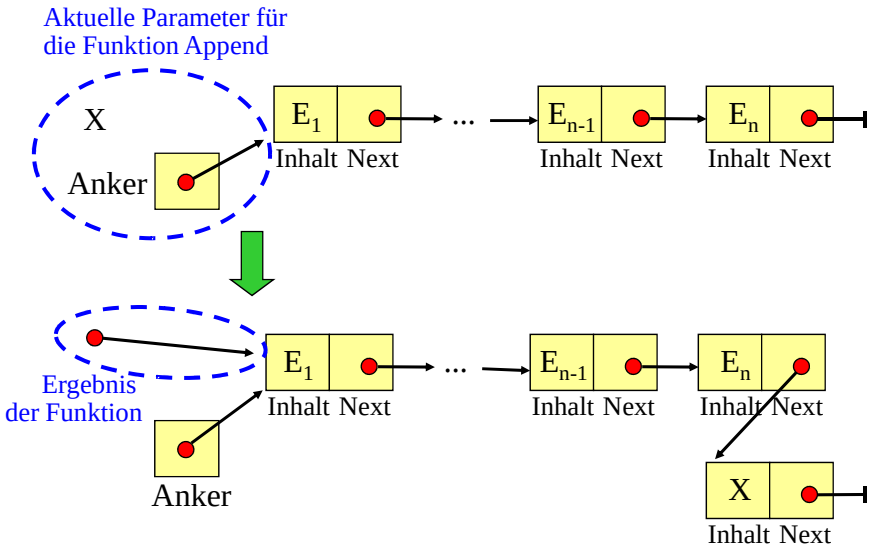
Fall 1: Anker ist der leere Zeiger.



Fall 2: Anker ist nicht leer.



Fall 2: Anker ist nicht leer, Ergebnis:



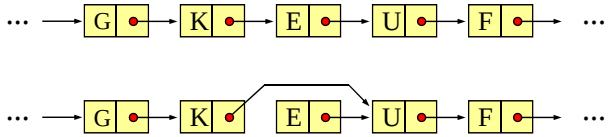
3.5.2.6 Sechste Operation: Löschen des ersten Auftretens eines Elements E aus der Liste.

Idee: Man durchläuft die Liste von vorne nach hinten.

Fall 1: Hierbei trifft man auf das Element E. Dann entferne man dieses Listenelement. Beachte: *Man löscht den Zugriff* und nicht das Listenelement! Man setzt also den Zeiger des Vorgängers auf den Nachfolger, siehe Skizze.

Fall 2: Das Element E ist in der Liste nicht enthalten. Dann bleibt die Liste unverändert.

Skizze zu Fall 1: Entferne E erfolgt durch "Ausklinken":



Löschen des ersten Auftretens eines Elements aus der Liste.

```
function Delete (E: Item; A: Ref_Zelle) return Ref_Zelle is
p, q: Ref_Zelle;
begin                -- p und q laufen bis ans Ende durch die Liste,
                     -- wobei p auf das jeweils betrachtete und q auf
                     -- das unmittelbar vor p stehende Element zeigt.
   p := A; q := null;
   while p /= null and then p.Inhalt /= E loop
                q := p;  p := p.Next;  end loop;
   if p /= null then      -- E wurde gefunden, p zeigt dorthin, q davor
      if q = null  then  return null;   -- die Liste besteht nur aus E
      else  q.Next := p.Next; return A;  end if;       -- ausklinken
   else  return A;        -- E wurde nicht gefunden
   end if;
end Delete;
```


Diese Funktion Delete gibt keine Auskunft darüber, ob das Element in der Liste vorkam oder nicht. Man kann eine Prozedur schreiben, die die Liste als globale Variable auffasst und eine Boolesche Variable auf True setzt, falls das Element einmal gelöscht wurde, anderenfalls False.

Dies sollten Sie selbst programmieren.

Wir lösen das allgemeinere Problem: Lösche *alle* Auftreten des Elements E und informiere zugleich darüber, wie viele Löschungen vorgenommen wurden.

Hierfür muss man im Wesentlichen nur das Ausklinken in die while-Schleife verlagern und bei jedem Ausklinken einen Zähler Z weiterzählen.

Sechste Operation (erweitert):
Löschen aller Auftreten eines Elements aus der Liste mit Anzahlangabe der Löschungen.

Idee: Man durchläuft die Liste von vorne bis ans Ende.

Jedes Mal, wenn das Element E gefunden wird, klinken wir das zugehörige Objekt vom Typ Zelle aus der Liste aus und zählen den Zähler Z weiter.

Wir schreiben hierfür eine Prozedur, die neben dem zu löschenden Element E den Anker der Liste als in-out-Variable übergeben bekommt sowie eine out-Variable für den Zähler Z.

Entfernung aller Vorkommen eines Elements aus der Liste.

```
procedure Remove1 (E: Item; A: in out Ref_Zelle; Z: out Natural) is
p, q: Ref_Zelle;
begin  Z := 0;  p := A;  q := null;
   while p /= null loop
      if p.Inhalt = E then      -- entferne dieses Element nun aus der Liste
         if  q = null  then  A := A.Next;    -- erstes Element der Liste
         else  q.Next := p.Next;  end if;    -- ausklinken
         Z := Z+1;
      end if;
      q := p;  p := p.Next;     -- zum nächsten Element der Liste gehen
   end loop;
end Remove1;
```

Vorsicht: Diese Prozedur ist falsch. Untersuchen Sie sie genau, finden Sie die Fehler und geben Sie eine korrekte Prozedur an.

Zweiter Versuch: Entfernung aller Vorkommen eines Elements.

Offenbar entsteht bei Remove1 ein Fehler, wenn das Element E zweimal nacheinander in der Liste steht. Daher Neufassung:

```
procedure Remove2 (E: Item; A: in out Ref_Zelle; Z: out Natural) is
p, q: Ref_Zelle;
begin  Z := 0;  p := A;  q := null;
   while p /= null loop
      if p.Inhalt = E then      -- entferne dieses Element nun aus der Liste
         if  q = null  then  A := A.Next;    -- erstes Element der Liste
         else  q.Next := p.Next;  end if;    -- ausklinken
         Z := Z+1;              -- in diesem Fall q nicht verändern
      else  q := p;  end if;    -- anderenfalls q weiterschalten
      p := p.Next;              -- zum nächsten Element der Liste gehen
   end loop;
end Remove2;
```

Aufgabe: Prüfen Sie, ob die Prozedur Remove2 nun korrekt ist!?

3.5.3 Varianten der linearen Listen

Eine Liste, in der jedes Element nur auf seinen Nachfolger verweist, heißt einfach verkettete Liste.

Eine Liste, bei der das letzte Element nicht auf null, sondern auf das erste Element der Liste (zurück) verweist, heißt zyklische Liste.

Eine Liste, bei der jedes Element sowohl auf den Vorgänger in der Liste als auch auf den Nachfolger verweist, heißt doppelt verkettete Liste. Beispiel für deren Deklaration:

```
type DZelle;
type Ref_DZelle is access DZelle;
type DZelle is  record
                  Inhalt: Character;
                  Vor, Nach: Ref_DZelle;
end record;
```

Aufbau einer doppelt verketteten zyklischen Liste:

```
... Anker, p, q: Ref_DZelle; Anzahl: Natural := 0; ...
if not End_Of_File then p := new DZelle;
        Get(p.Inhalt); Anker := p; Anzahl := 1;
        p.Vor := p; p.Nach := p; end if;
while not End_Of_File loop
        q:= new DZelle; Get(q.Inhalt); Anzahl := Anzahl+1;
        q.Nach := p.Nach; q.Vor := p;
        p.Nach := q; Anker.Vor := q;
        p := q;
end loop;
```

Hinweis: Man kann p.Vor:=p und Anker.Vor:=q streichen und dafür nach der Schleife Anker.Vor := p; hinzufügen.

3.5.4 Keller (Stack)

Definition: Eine lineare Liste heißt Keller oder Stapel (engl.: stack oder pushdown), wenn auf ihr genau die folgenden fünf Operationen zugelassen sind:

(1) "Empty" = Leeren der Liste.

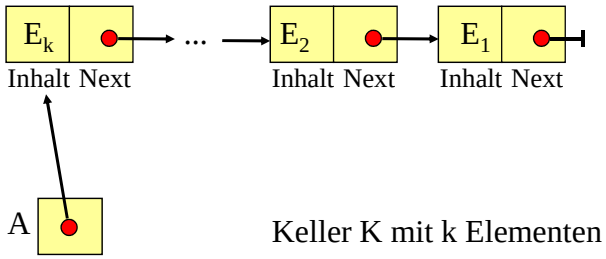
(2) "Isempty" = Abfragen auf Leerheit der Liste.

(3) "Top" = Kopieren des letzten Elements der Liste.

(4) "Push" = Hinzufügen eines Elements am Ende der Liste.

(5) "Pop" = Löschen des letzten Elements der Liste.

Die Realisierung durch Ada-Programmstücke ist einfach, wobei man die Zeiger immer zum Anfang der Liste hin orientiert.



Keller K mit k Elementen

K ist durch den Zeiger A gegeben.
Zum Beispiel ist Empty (K) gleichbedeutend mit A := null.
Top (K) ist der Wert E_k, also A.Inhalt.
Pop (K) wird durch A := A.Next realisiert, sofern K nicht leer ist (in diesem Fall eine Ausnahmebehandlung anstoßen).


```
procedure Empty (A: in out Ref_Zelle) is
begin A := null; end Empty;

function Isempty (A: Ref_Zelle) return Boolean is
begin return (A = null); end Isempty;

function Top (A: Ref_Zelle) return Item is
begin return A.Inhalt; end Top;

procedure Push (A: in out Ref_Zelle; E: Item) is
begin A := new Zelle'(E, A); end Push;

procedure Pop (A: in out Ref_Zelle) is
begin
   if Isempty (A) then Put ("Keller ist bereits leer.");
   else A := A. Next; end if;
end Pop;
```

Hinweis: In der Prozedur Pop sollte man im Falle Isempty(A) besser eine Ausnahmebehandlung "erwecken" (raise ...), siehe 3.5.2.3.

Man sagt auch, ein Keller ist eine Liste, die nach dem LIFO-Prinzip arbeitet.

LIFO = Last in, first out.
Dies bedeutet:
Die Elemente, die als letzte in den Keller eingefügt wurden, müssen als erste wieder herausgenommen werden.

Ein Beispiel ist der Ablagekorb auf dem Schreibtisch: Die Akten werden in der umgekehrten Reihenfolge, in der sie in den Ablagekorb gelegt wurden, herausgeholt und bearbeitet.

Ein anderes Beispiel ist das Verwalten der Blockstruktur: Mit jedem neuen Block wird am Ende des Speichers neuer Speicherplatz reserviert, der bei Erreichen des zugehörigen end wieder frei gegeben wird.

Beispiel: Spiegeln eines Textes

Sei K ein Zeichen-Keller, also eine lineare Liste (mit dem Inhalts-Datentyp "Item" = Character), auf der nur die fünf genannten Operationen verwendet werden dürfen. B sei eine Variable vom Typ Character. Wir lesen einen Text ein und geben ihn in rückwärtiger Reihenfolge wieder aus:

```
Empty(K);
while not End_Of_File loop
        Get(B);
        Push(K,B);
end loop;
while not Isempty(K) loop
        Put(Top(K));
        Pop(K);
end loop;
```


3.5.5 Schlange (Queue)

Definition: Eine lineare Liste heißt Schlange (engl.: queue), wenn auf ihr genau die folgenden fünf Operationen zugelassen sind (selbst realisieren!):

(1) "Empty" = Leeren der Liste.

(2) "Isempty" = Abfragen auf Leerheit der Liste.

(3) "First" = Kopieren des ersten Elements der Liste.

(4) "Enter" = Hinzufügen eines Elements am Ende der Liste.

(5) "Remove" = Löschen des ersten Elements der Liste.

Die Realisierung durch Ada-Programmstücke ist einfach. Für Schlangen verwendet man zwei Zeiger: Einer zeigt auf den Anfang und einer auf das Ende der Schlange.

Man sagt auch, eine Schlange ist eine Liste, die nach dem FIFO-Prinzip arbeitet.

FIFO = First in, first out.

Wie der Name schon sagt, setzt man Schlangen für alle Situationen ein, bei denen Elemente nacheinander aufgestellt werden und in einer Reihe warten müssen, bis sie bearbeitet werden.

Beispiele sind Simulationen von Warteschlangen, etwa vor einem Schalter, im Verkehr oder beim Lernen. Auch das Ableiten von Wörtern aus einer Grammatik kann man mit Warteschlangen beschreiben.

3.5.6 Geflechte (Graphen)

Wir müssen die Elemente nicht wie an einer Perlenschnur aufreihen (lineare Liste), sondern können Verweise auch auf beliebige Elemente des zugrunde liegenden Datentyps setzen.

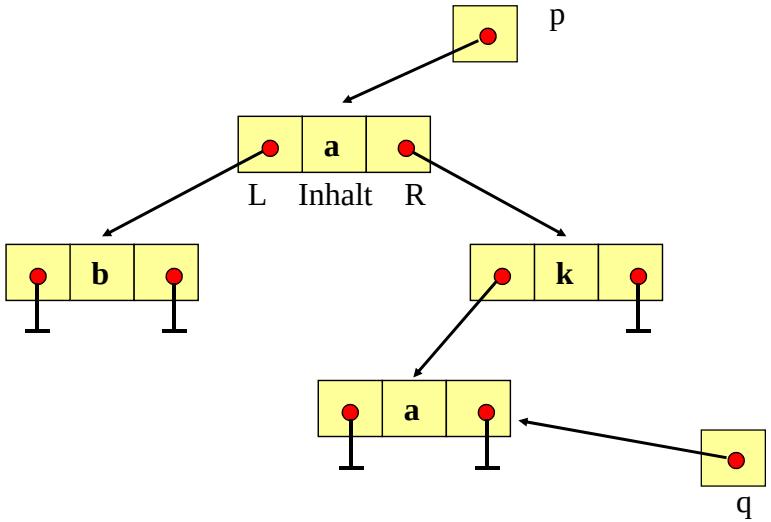
Dadurch entstehen beliebig vernetzte Gebilde. Diese werden Graphen (siehe 3.7) genannt.

Sie bestehen aus den Elementen des Datentyps ("Knoten" genannt), die durch Verweise ("Kanten" oder Pfeile genannt) miteinander verbunden sind.

Mathematisch gesehen sind dies Relationen über der Menge M des zugrunde liegenden Datentyps.

Wir werden hierzu noch viele Beispiele kennen lernen. Hier betrachten wir nur ein einfaches Beispiel (formuliert in Ada):


```
procedure Char_Liste is
type BZelle;
type Ref_BZelle is access BZelle;
type BZelle is record
        Inhalt: Character;
        L, R: Ref_BZelle;
end record;

p, q: Ref_BZelle;

begin  p := new BZelle;  p.Inhalt:='a';
       q:= new BZelle;   p.L := q;      q.Inhalt := 'b';
       q:= new BZelle;   p.R := q;      q.Inhalt := 'k';
       q:= new BZelle;   p.R.L := q;    q.Inhalt := 'a';
...
end;
```

Situation nach Ausführen des Programmstücks. p und q stehen im Kellerspeicher, die übrigen Objekte in der Halde.

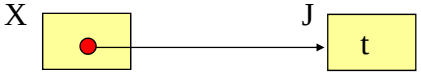
3.5.7 Zeiger auf Kellerspeicher-Objekte

Es ist grundsätzlich erlaubt, Zeigervariablen auch die internen Namen von Variablen, die im Kellerspeicher des Programms stehen, zuzuweisen. In Ada muss dies jedoch in mehrfacher Hinsicht in den Deklarationen "angekündigt" werden:

Der Datentyp der Zeigervariable X muss ein "allgemeiner" Zeigertyp auf T sein: type Ref_T is access all T; ... X: Ref_T;

und die referenzierte Variable J muss als "aliased" (Bedeutung: J ist noch über einen weiteren Zugriff erreichbar) deklariert worden sein; weiterhin muss bei der Zuweisung des Namens einer Variablen das "Access"-Attribut verwendet werden:

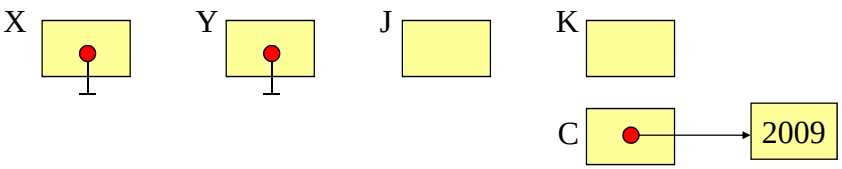
```
J: aliased T := t;
...; X := J'Access;
```



Wir geben hierzu nur ein Beispiel an:

Beispiel:

```
type Ref_AInteger is access all Integer;
X, Y: Ref_AInteger;
J, K: aliased Integer;
C: constant aliased Ref_AInteger := new Integer'(2009);
```

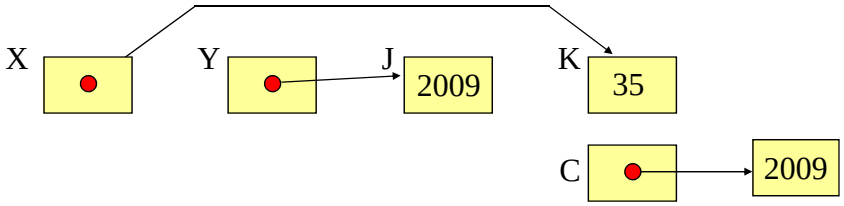


Die Folge von Wertzuweisungen
X := K'Access; Y := J'Access;
K := 35; J := C.all;
bewirkt anschließend folgendes:

Die fünf Objekte X, Y, J, K und C liegen im Kellerspeicher. 2009 liegt in der Halde.

Beispiel:

```
type Ref_AInteger is access all Integer;
X, Y: Ref_AInteger;
J, K: aliased Integer;
C: constant aliased Ref_AInteger := new Integer'(2009);
```



X := K'Access; Y := J'Access; K := 35; J := C.all;

Beachte: C.all := 2010; ist zulässig. Denn nur der Verweis, aber nicht der Inhalt dessen, worauf gezeigt wird, ist konstant.

Hinweis 1: aliased
Das Schlüsselwort "aliased" (gesprochen ['eɪlɪəsd]) erfüllt zwei Aufgaben: Zum einen warnt es die Programmierer, dass der Inhalt dieser Variablen über einen zweiten Zugriffsweg verändert werden kann, zum anderen darf der Compiler auf solche Variablen nicht die üblichen Optimierungen anwenden.

Hinweis 2: Orthogonalität (freie Kombinierbarkeit von Konzepten)
Zeiger auf Zeiger sind zulässig. Man kann also deklarieren:

```
type Ref_Integer is access Integer;
type RefRef_Integer is access Ref_Integer;
type RefRefRef_Integer is access RefRef_Integer;
type RefRefRefRef_Integer is access RefRefRef_Integer;
X: RefRefRefRef_Integer; Y := RefRef_Integer;
X := new RefRefRef_Integer; ...;
X.all.all.all.all := 1;  Y := X.all.all;  Y.all.all := 5;
```


Hinweis 3: Hängende Zeiger ("dangling pointers")

Folgender Fall kann eintreten (eine beliebte Fehlersituation):

```
declare X: Ref_All_Item;
begin ...
        declare E: aliased Item;
        begin ...
                X := E'Access;
        end;
        ...; ...X...; ...
end;
```

Hier endet die Lebensdauer von E.

Hier verweist X "ins Nichts", weil E nicht mehr existiert.

Generell wird daher gefordert: Die Lebensdauer eines Objekts, auf das verwiesen wird (hier: E), muss mindestens so groß sein wie die Lebensdauer des hierauf zeigenden Objekts (hier: X). [Dies lässt sich meist schon zur Übersetzungszeit feststellen.]

3.6 Referenzkonzept

3.6.1 In jedem Programm gibt es eine spezielle Menge, nämlich die Menge aller Objekte, die in diesem Programm sichtbar oder erzeugbar sind. *Jedes Objekt erhält in der Programmierung einen Namen*, auch wenn dem Objekt im Programmtext kein Name explizit zugeordnet wurde. Z. B. erhält in Ada der Datentyp
"array (1..n) of Integer" in der Variablendeklaration
"X, Y: array (1..n) of Integer" zweimal einen (internen) Namen, auch wenn dies nicht im Programmtext geschieht.

In einem Programm ist es entscheidend, *dass alle Namen unterschieden werden können*, da man sonst die Objekte nicht auseinander halten kann. Der Name besteht hierbei aus einem "Identifikator" (Bezeichner) und einer Information über seine Umgebung (in verschiedenen Umgebungen (z.B. Programmen) können natürlich gleiche Identifikatoren verwendet werden). Er wird intern durch eine eindeutige "Adresse" dargestellt.

3.6.2 Datentyp "Namen für Objekte eines Datentyps T"

Zugrunde liegende Menge je Datentyp T: Menge der in einem Programm sichtbaren oder erzeugbaren Namen dieses Datentyps (realisiert als Adressen = interne Namen).

Nullstellige Operationen:

nil oder null ("kein Name").

Erzeuge einen neuen Namen: new <Datentyp>.

Einstellige Operation:

Dereferenzieren: Zugriff auf das, was dieser Name repräsentiert. Man bezeichnet diese Operation gern durch

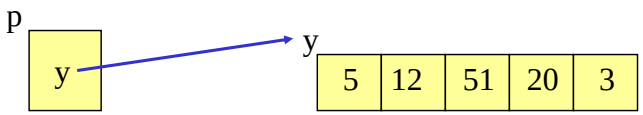
deref p (oder in PASCAL: p↑ oder in Ada: p.all).

Zweistellige Operationen:

= und ≠, also Gleichheit und Ungleichheit von Namen.

(Zu einem Namen gehört auch seine Umgebung, also z.B. der jeweilige Block oder der aktuelle Prozeduraufruf. Es dürfen daher in der Praxis nicht nur die Adressen verglichen werden, sondern zusätzlich die Umgebungsinformationen.)

Veranschaulichung über Pfeile (Zeiger, "pointer", Verweise):



3.6.3 Beispiel: Wenn man einen Vektor X aufsummiert:

```
s:= 0.0; for i in X'Range loop s := s + X(i); end loop;
```

und das Gleiche nun für zwei weitere Vektoren Y und Z tun möchte, so müsste man die Anweisung noch mal hinschreiben:

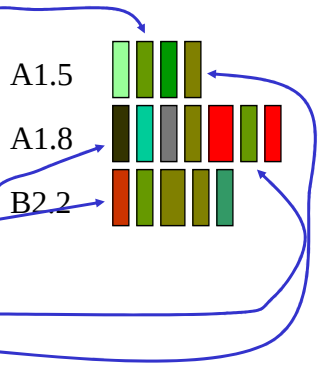
```
t:= 0.0; for i in Y'Range loop t := t + Y(i); end loop;
u:= 0.0; for i in Z'Range loop u := u + Z(i); end loop;
```

Mit Variablen p und q vom Datentyp "Menge der Namen" *könnte* man schreiben (in Ada geht dies nicht unmittelbar)

```
p: (X,Y,Z); q: (s,t,u);          -- wie Aufzählungstypen behandeln
for p in (X..Z) loop
    if p=X then q:=s; else q:=Succ(q); end if;
    q↑:= 0.0; for i in p↑'Range loop q↑ := q↑ + p↑(i); end loop;
end loop;
```

3.6.4 Beispiel: Bibliothek

Standort der Bücher:



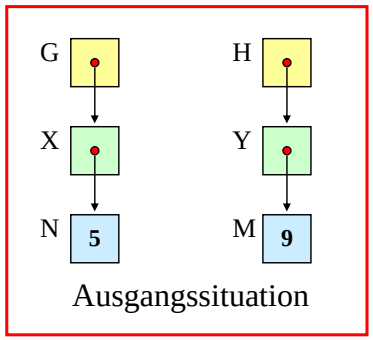
Karteikarten oder
Computereinträge

...
Kaiser, Albert, "Über die ...", B2.2
Kaiser, Karl, "Die helle ...", A1.5
Kaiser, Wolfgang, "Das ...", A1.8
Kaisers, Emma, "Über die ...", A1.5
Kaisser, Fritz, "Das alte ...", A1.8
...

Karteikarten sind meist Records, die den Datentyp "Name" oder "Adresse" enthalten.

3.6.5 Wertzuweisung. Wir betrachten:

```
type Ref_Integer is access Integer;
type RefRef_Integer is access Ref_Integer;

N, M: Integer;  X, Y: Ref_Integer;   G, H: RefRef_Integer;

begin  N:= 5; M := 9; X := N; Y := M; G := X; H := Y;
       M := N;
       X := Y;
       G := H;
       Y := N;
       N := 8;
       M := G;
       G := X;
       H := N;
end;
```



Übliche Regel zur Auswertung von **Z := α;**

(1) Werte den Ausdruck α aus. Dieser sei vom Datentyp T. (Falls α eine Variable vom Typ T_1 ist, so ist $T = T_1$.)

(2) Prüfe, ob **Z** Werte vom Datentyp T enthalten darf. Falls ja, weise **Z** den Wert des Ausdrucks α zu; fertig.

(3) Falls nein: Wenn α eine Konstante ist, dann breche die Wertzuweisung mit einer Fehlermeldung ab, anderenfalls ersetze α durch deref α und T durch den Datentyp von deref α und fahre bei (2) fort.

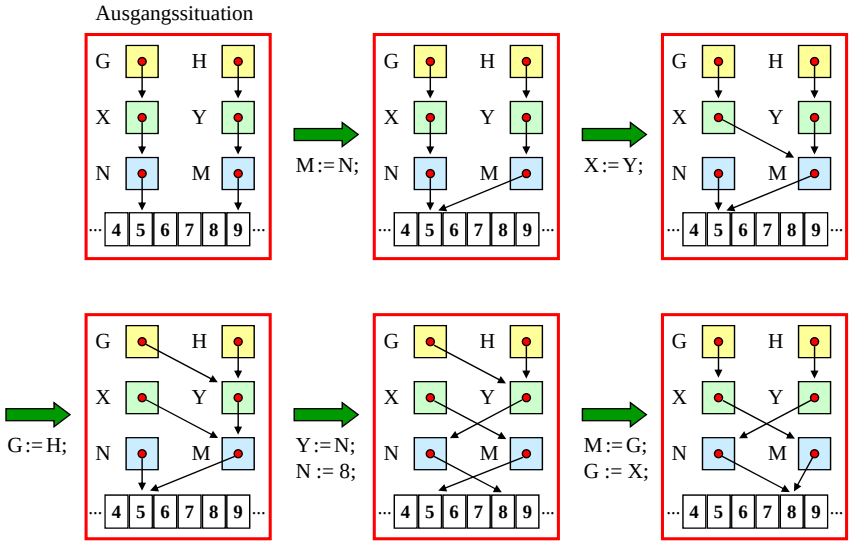
In diesem Sinne sind die obigen Wertzuweisungen
" M := N; X := Y; G := H; Y := N; N := 8; M := G; G := X; "
durchführbar, nur die Wertzuweisung
" H := N; " ist nicht zulässig.

Wir vollziehen die obigen Wertzuweisungen
" M := N; X := Y; G := H; Y := N; N := 8; M := G; G := X; "
auf der nächsten Folie schrittweise nach.

Bisher fassten wir die Konstanten als Inhalte der Variablen auf. Diese Vorstellung modifizieren wir dahingehend, dass wir auch die normalen Variablen als "Zeiger auf Konstanten" ansehen.

Die Konstanten liegen somit ebenfalls im Speicher und die Variablen enthalten nur noch den Verweis auf die Adresse, wo die Konstante sich befindet.

Wir kennzeichnen die Konstanten durch weiß unterlegte Rechtecke. Theoretisch kann man annehmen, dass sich *alle* Konstanten irgendwo im Speicher befinden. Eine Wertzuweisung N:=8; bewirkt dann nur eine Änderung des Zeigers, der anschließend auf die Adresse der Konstanten "8" weist.

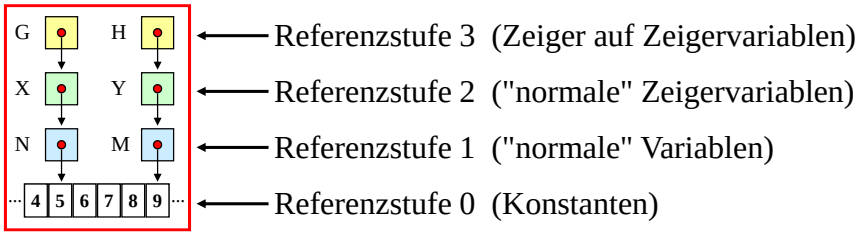


Schrittweise Ausführung der zulässigen Wertzuweisungen

3.6.6 Referenzkonzept

Jedes Objekt erhält eine "Referenzstufe". Konstanten erhalten die Referenzstufe 0, Variablen, die hierauf verweisen, die Referenzstufe 1 usw. Allgemein erhält ein Objekt, das auf Objekte der Referenzstufe k verweist, die Referenzstufe k+1.

Obiges Beispiel:



Die Referenzstufen setzt man auf Ausdrücke fort:

(1) Besteht ein Ausdruck aus genau einer Variablen oder Konstanten, so ist deren Referenzstufe zugleich die Referenzstufe des Ausdrucks.

(2) Ist ein Ausdruck von der Form f(...) mit einer Funktion f, so ist die Referenzstufe des Resultats von f zugleich die Referenzstufe des Ausdrucks.
(Speziell gilt dies auch für Operatoren. Z.B. liefert die Addition $\beta_1+\beta_2$ zweier ganzzahliger Ausdrücke eine Konstante und daher ist die Referenzstufe von $\beta_1+\beta_2$ Null.)

Nun kann man genau festlegen, wie die Wertzuweisung
Z := α; durchzuführen ist.

Ausführung der Wertzuweisung " Z := α; ". Es sei i die Referenzstufe von Z und j die Referenzstufe des Ausdrucks α.

Die Wertzuweisung Z := α; ist genau dann zulässig wenn

(1) $i \le j+1$ und

(2) Z ein Zeiger auf den Datentyp des Ausdrucks

$\alpha' = \underbrace{\text{deref deref ... deref}}_{j+1-i \text{ mal}}\ \alpha$ ist.

Beachte: Wir fassen hier "normale" Variablen als Zeiger auf Konstanten auf.

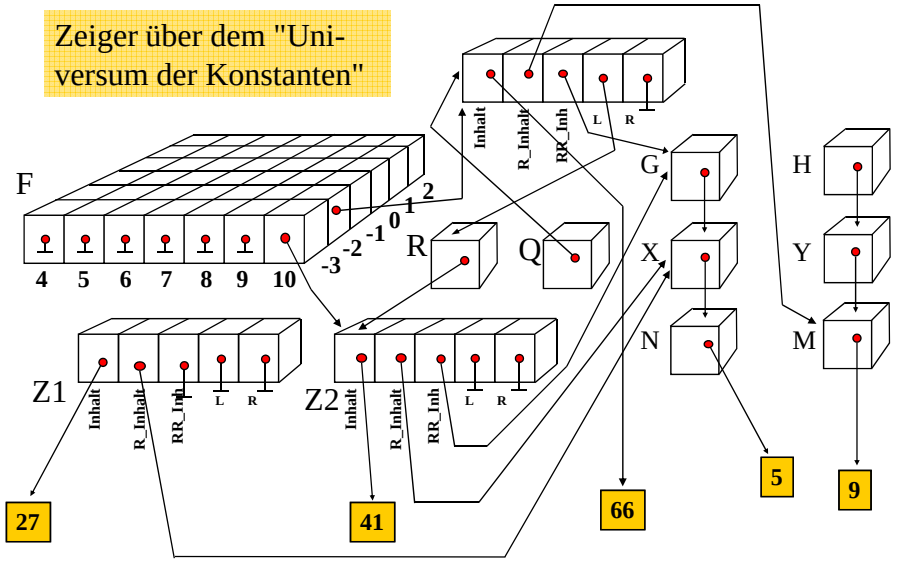
Falls die Wertzuweisung Z := α; zulässig ist, wird der Ausdruck α' ausgewertet und das Ergebnis der Variablen Z als Wert zugewiesen.

(Anmerkung: Man kann auch verlangen, dass nur im Falle i = j+1 die Wertzuweisung zulässig ist. Dann muss der Programmierer die Dereferenzierungen selbst angeben.)

3.6.7 Beispiel (in Ada muss man deref durch angehängtes all ersetzen)

```
type Ref_Integer is access Integer;
type RefRef_Integer is access Ref_Integer;
type Zelle;
type Ref_Zelle is access Zelle;
type Zelle is record
        Inhalt: Integer;
        R_Inhalt: Ref_Integer;
        RR_Inh: RefRef_Integer;
        L, R: Ref_Zelle;
end record;
type Ref_Feld is array (4..10, -3..2) of Ref_Zelle;
N, M: Integer;   X, Y: Ref_Integer;    G, H: RefRef_Integer;
Z1, Z2: Zelle := (27, X, null, null, null);    R, Q: Ref_Zelle;
F: Ref_Feld;
....
N:= 5; M := 9; X := N; Y := M; G := X; H := Y; R := Z2;
deref R.Inhalt := 41;  Z2.RR_Inh := G;  Q := new Zelle'(66, M, G, R, null);
F(10,-3) := R;  F(10,-2) := Q;
```


Veranschaulichung:



3.6.8 Paradigma: Programmieren als ständiges Umsetzen von Zeigern über der Welt aller denkbaren Konstanten.

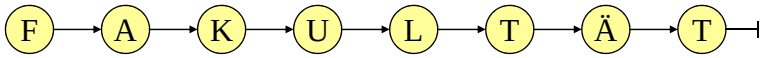
Speichert man die hierbei erreichbaren Konstanten (einschließlich der Adressen) in den Variablen oder ihren Komponenten, die zu den elementaren Datentypen (einschließlich der Zeigertypen) gehören, so erhält man die übliche Vorstellung vom "imperativen Programmieren": Befehlsartige Steuerung der Ausführung von Wertzuweisungen, durch die in jedem Schritt endlich viele Konstanten in den jeweils betrachteten Speicher-zellen ausgetauscht werden.

In der Praxis genügt die Referenzstufe 2, da man höhere Referenzstufen durch eine Liste, also durch eine Kette von Referenzen der Stufe 2 simulieren kann (siehe Listendarstellung 3.4.7).

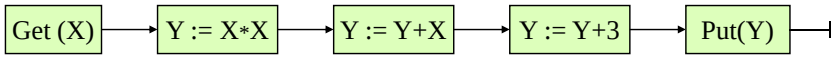
3.7 Bäume

Anachaulich: Bäume sind zusammenhängende Graphen, die sich verzweigen, aber nicht wieder zusammenlaufen.

Die einfachsten Bäume sind lineare Listen, zum Beispiel ein Text



oder die unverzweigte Hintereinanderausführung von Anweisungen:

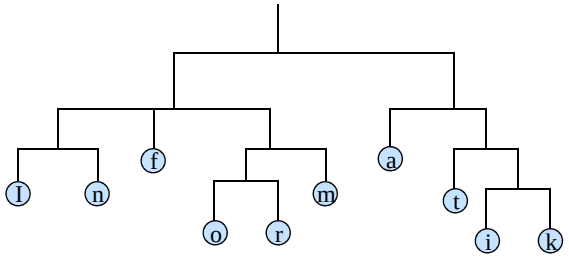


(Dies entspricht der Ausrechnung der Formel $x^2 + x + 3$.)

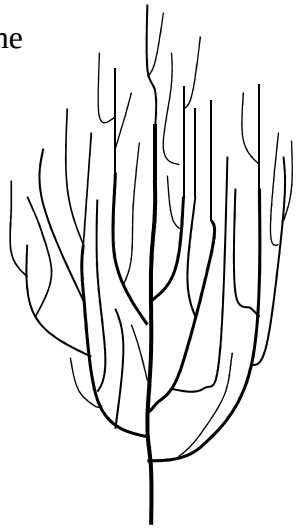
Bäume entstehen aus Listen durch Verzweigungen. Unsere Welt ist voller Beispiele, vor allem im Bereich des Planens und der Ablage von Informationen.

Anschauliche Beispiele sind:

Ein Mobile

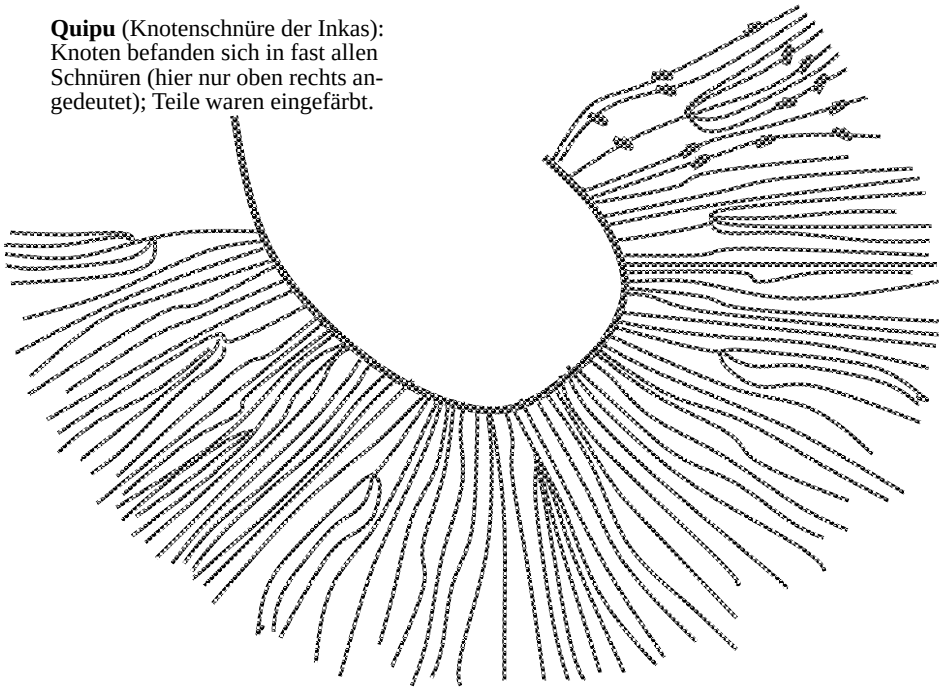


"Biologische" Bäume



und ihre Struktur

Quipu (Knotenschnüre der Inkas): Knoten befanden sich in fast allen Schnüren (hier nur oben rechts angedeutet); Teile waren eingefärbt.



Top Two Levels of the ACM Computing Classification System (1998)

A. General Literature
- A.0 GENERAL
- A.1 INTRODUCTORY AND SURVEY
- A.2 REFERENCE
- A.m MISCELLANEOUS

B. Hardware
- B.0 GENERAL
- B.1 CONTROL STRUCTURES AND MICROPROGRAMMING (D.3.2)
- B.2 ARITHMETIC AND LOGIC STRUCTURES
- B.3 MEMORY STRUCTURES
- B.4 INPUT/OUTPUT AND DATA COMMUNICATIONS
- B.5 REGISTER-TRANSFER-LEVEL IMPLEMENTATION
- B.6 LOGIC DESIGN
- B.7 INTEGRATED CIRCUITS
- B.8 PERFORMANCE AND RELIABILITY
- B.m MISCELLANEOUS

C. Computer Systems Organization
- C.0 GENERAL
- C.1 PROCESSOR ARCHITECTURES
- C.2 COMPUTER-COMMUNICATION NETWORKS
- C.3 SPECIAL-PURPOSE AND APPLI-CATION-BASED SYSTEMS (J.7)
- C.4 PERFORMANCE OF SYSTEMS
- C.5 COMPUTER SYSTEM IMPLEMENT.
- C.m MISCELLANEOUS

D. Software
- D.0 GENERAL
- D.1 PROGRAMMING TECHNIQUES (E)
- D.2 SOFTWARE ENGINEERING (K.6.3)
- D.3 PROGRAMMING LANGUAGES
- D.4 OPERATING SYSTEMS (C)
- D.m MISCELLANEOUS

E. Data
- E.0 GENERAL
- E.1 DATA STRUCTURES
- E.2 DATA STORAGE REPRESENTATIONS
- E.3 DATA ENCRYPTION
- E.4 CODING AND INFORMATION THEORY (H.1.1)
- E.5 FILES (D.4.3, F.2.2, H.2)
- E.m MISCELLANEOUS

F. Theory of Computation
- F.0 GENERAL
- F.1 COMPUTATION BY ABSTRACT DEVICES
- F.2 ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY (B.6, B.7, F.1.3)
- F.3 LOGICS AND MEANINGS OF PROGRAMS
- F.4 MATHEMATICAL LOGIC AND FORMAL LANGUAGES
- F.m MISCELLANEOUS

G. Mathematics of Computing
- G.0 GENERAL
- G.1 NUMERICAL ANALYSIS
- G.2 DISCRETE MATHEMATICS
- G.3 PROBABILITY AND STATISTICS
- G.4 MATHEMATICAL SOFTWARE
- G.m MISCELLANEOUS

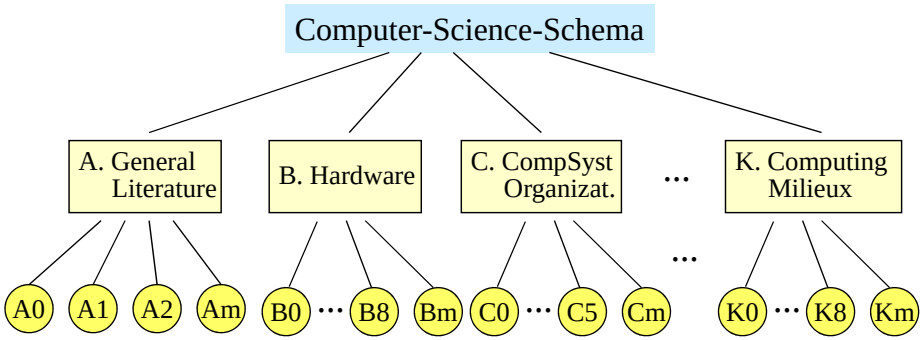
H. Information Systems
- H.0 GENERAL
- H.1 MODELS AND PRINCIPLES
- H.2 DATABASE MANAGEMENT (E.5)
- H.3 INFORMATION STORAGE AND RETRIEVAL
- H.4 INFORMATION SYSTEMS APPLICATIONS
- H.5 INFORMATION INTERFACES AND PRESENTATION (e.g., HCI) (I.7)
- H.m MISCELLANEOUS

I. Computing Methodologies
- I.0 GENERAL
- I.1 SYMBOLIC AND ALGEBRAIC MANIPULATION
- I.2 ARTIFICIAL INTELLIGENCE
- I.3 COMPUTER GRAPHICS
- I.4 IMAGE PROCESSING AND COMPUTER VISION
- I.5 PATTERN RECOGNITION
- I.6 SIMULATION AND MODELING (G.3)
- I.7 DOCUMENT AND TEXT PROCESSING (H.4, H.5)
- I.m MISCELLANEOUS

J. Computer Applications
- J.0 GENERAL
- J.1 ADMINISTRATIVE DATA PROCESSING
- J.2 PHYSICAL SCIENCES AND ENGINEERING
- J.3 LIFE AND MEDICAL SCIENCES
- J.4 SOCIAL AND BEHAVIORAL SCIENCES
- J.5 ARTS AND HUMANITIES
- J.6 COMPUTER-AIDED ENGINEERING
- J.7 COMPUTERS IN OTHER SYSTEMS (C.3)
- J.m MISCELLANEOUS

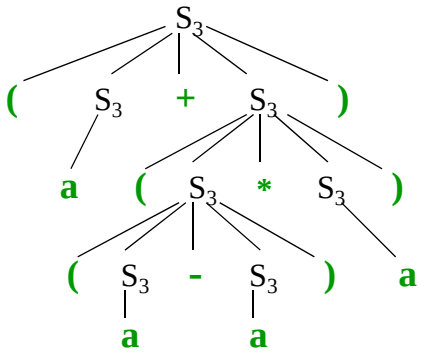
K. Computing Milieux
- K.0 GENERAL
- K.1 THE COMPUTER INDUSTRY
- K.2 HISTORY OF COMPUTING
- K.3 COMPUTERS AND EDUCATION
- K.4 COMPUTERS AND SOCIETY
- K.5 LEGAL ASPECTS OF COMPUTING
- K.6 MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS
- K.7 THE COMPUTING PROFESSION
- K.8 PERSONAL COMPUTING
- K.m MISCELLANEOUS

Ordnungsschema: ACM-Klassifikationsschema Informatik, oberste Ebene

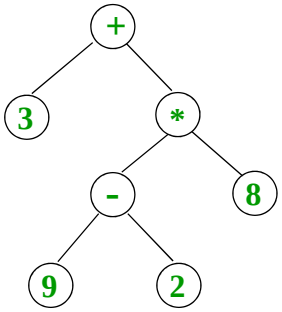


Ordnungsschema: ACM-Klassifikationsschema Informatik, oberste Ebenen.
Darstellung als Baum (das Klassifikationsschema hat mindestens 4 Ebenen).

Ableitungsbäume kontextfreier Grammatiken



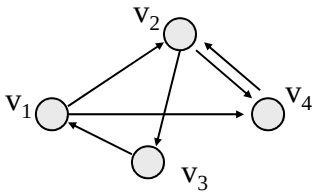
Rechenbäume zur Auswertung von Ausdrücken



Hinweis: Dieser Rechenbaum gehört zu dem links stehenden Ableitungsbaum für (a + ((a - a) * a)) mit konkreten Werten.

Definition 3.7.1: Gerichteter Graph und Weg

G = (V, E) heißt gerichteter Graph (oder Digraph) ⇔
1. V ist ein endliche nicht-leere Menge (Knoten),
2. E ⊆ V×V (Menge der Kanten).
[Digraph kommt von "directed graph" = gerichteter Graph.]

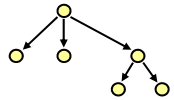


$V = \{v_1, v_2, v_3, v_4\}$

$E = \{(v_1, v_2), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_1), (v_4, v_2)\}$

Eine Folge von Knoten $(u_1, u_2, u_3, ..., u_r)$ mit $r \geq 1$ heißt (gerichteter) Weg im Graphen G, wenn $(u_i, u_{i+1}) \in E$ für $i = 1, 2, ..., r-1$ gilt. r-1 heißt die Länge des Weges.
Man sagt, der Weg $(u_1, u_2, ..., u_r)$ führt vom Knoten u_1 zum Knoten u_r.

Definition 3.7.2: Wurzel, Baum



Ein Knoten w heißt Wurzel eines Graphen G, wenn es von w zu jedem Knoten des Graphen einen Weg gibt.

Ein gerichteter Graph heißt Baum, wenn er eine Wurzel w besitzt und jeder Knoten ungleich der Wurzel genau einen Vorgänger besitzt. Das heißt, zu jedem $x \in V$, $x \neq w$ gibt es genau einen Knoten y mit $(y, x) \in E$. Dieser Knoten y heißt Vater oder direkter Vorgänger von x.

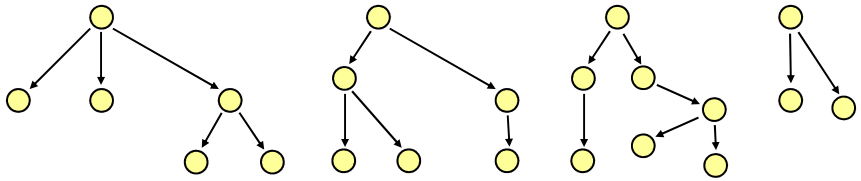
Der Knoten x heißt dann Kind oder Sohn oder direkter Nachfolger des Knotens y.

(Die Anzahl der Kinder eines Knotens nennt man auch den Verzweigungsgrad des Knotens. Aber Vorsicht, denn diese Bezeichnung bedeutet bei Bäumen und bei allgemeinen Graphen manchmal etwas Verschiedenes.)

In jedem nicht-leeren Baum muss es mindestens einen Knoten geben, der keinen direkten Nachfolger besitzt.

Ein Knoten heißt Blatt eines Baums, wenn er keinen direkten Nachfolger besitzt.

Anschaulich: Ein Graph heißt Wald, wenn er sich als disjunkte Vereinigung von Bäumen schreiben lässt.



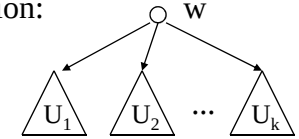
3.7.3 Rekursive Definition für Bäume

1) Die leere Menge ist ein Baum.
2) Wenn w ein Knoten und U eine endliche Menge von Bäumen sind, so ist auch $w(U)$ ein Baum.

w heißt Wurzel des Baums $w(U)$, die Elemente von U heißen Unterbäume oder Teilbäume von w im Baum $w(U)$.

Skizze: Leerer Baum:

Rekursion:



Hinweis: Ist die Menge der Bäume $U = \{U_1, ..., U_k\}$ geordnet, so spricht man von einem geordneten Baum.

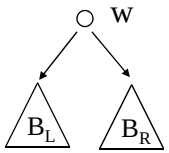
3.7.4 Binäre Bäume

1) Die leere Menge ist ein binärer Baum.
2) Wenn w ein Knoten und B_L und B_R binäre Bäume sind, dann ist auch $w(B_L, B_R)$ ein binärer Baum.

B_L heißt linker Nachfolger oder linker Unterbaum, B_R rechter Nachfolger oder rechter Unterbaum des Knotens w.
Ein binärer Baum ist ein geordneter Baum, in dem jeder Knoten, genau zwei Unterbäume hat (die leer sein können).

Skizze: Leerer Baum:

Rekursion:

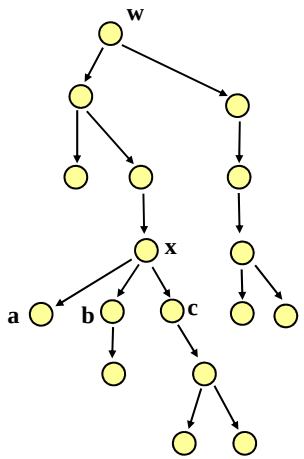


3.7.5 Ergänzungen zur Definition Baum

Hinweis: Ein Baum mit n Knoten besitzt stets n-1 Kanten.

Bezeichnung: Knoten, die den gleichen Vater im Baum haben, heißen Brüder. Knoten ($\neq$ v), die auf dem Weg von der Wurzel w zu einem Knoten v liegen, heißen Vorgänger von v.

Bezeichnung: Die Länge des längsten Wegs von der Wurzel w zu einem Blatt bezeichnet man als die Tiefe (oder Höhe) des Baums.

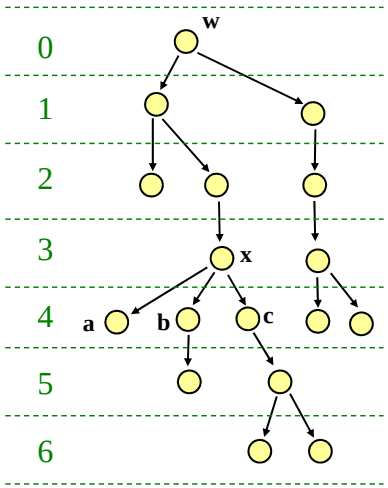


Die Knoten a, b, c sind Brüder; sie sind Kinder des Knotens x; w ist Wurzel des Baums; die Höhe des Baums ist 6.

Level der Knoten, z. B. lev(c) = 4

Der Abstand eines Knotens zur Wurzel ist in Bäumen eindeutig bestimmt. Man spricht von Schichten oder Leveln.

Level (oder Niveau) eines Knotens:
Jedem Knoten ist ebenfalls eine Höhe (sein "Level") zugeordnet, und zwar:
- die Wurzel w hat Level lev(w) = 0,
- wenn ein Knoten das Level k hat, dann haben alle seine direkten Nachfolger das Level k+1.



Dann gilt offenbar:
Die Höhe eines Baums ist das größte Level eines Knotens im Baum.

3.7.6 Durchlauf von (binären) Bäumen

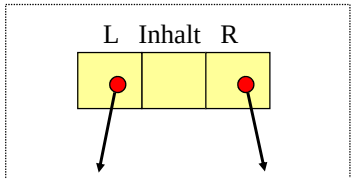
Sehr oft benötigt man den Baumdurchlauf, insbesondere den Durchlauf durch binäre Bäume.

In den Knoten können Daten stehen.

Definition: Ein binärer Baum, in dessen Knoten Werte eines geordneten Datentyps stehen (z.B. ganze Zahlen), heißt Suchbaum, wenn für jeden Knoten u gilt: Alle Inhalte von Knoten im linken Unterbaum von u sind echt kleiner als der Inhalt von u und alle Inhalte im rechten Unterbaum von u sind größer oder gleich dem Inhalt von u.

Die Datenstruktur für einen binären Baum lautet in Ada also (als Inhalt wählen wir willkürlich Integer, dann können wir diese Datenstruktur auch als Suchbaum verwenden; L und R zeigen auf den linken bzw. rechten Nachfolger):

```
type BinBaum;
type Ref_BinBaum is access BinBaum;
type BinBaum is record
        Inhalt: Integer;
        L, R: Ref_BinBaum;
end record;
```



Datenstruktur BinBaum

Der Durchlauf durch einen binären Baum erfolgt rekursiv:

```
procedure Inorder (b: Ref_BinBaum) is
begin
        if b /= null then
                Inorder (b.L);
                < bearbeite den Knoten b >;
                Inorder (b.R);
        end if;
end Inorder;
```

Man nennt diesen Durchlauf einen Inorder-Durchlauf, da der jeweilige Knoten zwischen dem Durchlauf durch seinen linken und seinen rechten Unterbaum bearbeitet wird.

Die Reihenfolgen Preorder- und Postorder-Durchlauf lauten:

```
procedure Preorder (b: Ref_BinBaum) is
begin   if b /= null then
                < bearbeite den Knoten b >;
                Preorder (b.L);
                Preorder (b.R);
        end if;
end Preorder;
```

```
procedure Postorder (b: Ref_BinBaum) is
begin   if b /= null then
                Postorder (b.L);
                Postorder (b.R);
                < bearbeite den Knoten b >;
        end if;
end Postorder;
```


Zeitaufwand: n sei die Zahl der Knoten. Jeder Knoten wird genau dreimal besucht. Der Durchlauf erfolgt somit für jeden der drei Durchläufe in 3n Schritten.

Platzaufwand: Auch hier ist die Rekursion zu beachten, die maximal so tief sein kann, wie es Knoten im Baum gibt. D.h., der zusätzliche Platzbedarf ist die Höhe des Baums; er beträgt im ungünstigsten Fall also bis zu n Speicherplätze.

Im Mittel wird man jedoch deutlich weniger Plätze benötigen. Der beste Fall liegt vor, wenn der Baum sehr gleichmäßig verzweigt ist, also die Höhe von ungefähr log(n) besitzt. In diesem Fall braucht man nur proportional zu log(n) zusätzlichen Speicherplatz.

3.7.7 Anwendung: Sortieren mit Bäumen.

Vorgehen: Lege n Zahlen a_1, ..., a_n der Reihe nach in einem binären Suchbaum ab. Lies dann den Baum inorder aus.

Programm: (es muss $n \geq 1$ gelten)

```
< "BinBaum, n und A seien hier global vereinbart" >
Anker, p, q: Ref_BinBaum;
begin ...; Anker := new Ref_BinBaum'(A(1), null, null);
    for i in 2..n loop
        p := Anker; q := null;
        while p /= null loop  q := p;
            if A(i) < p.Inhalt then p := p.L; else p := p.R; end if;
        end loop;
        if A(i) < q.Inhalt then
            q.L := new Ref_BinBaum'(A(i), null, null);
        else q.R := new Ref_BinBaum'(A(i), null, null);  end if;
    end loop;
    Inorder (Anker); ...
end;
```


Der Rumpf der Prozedur "Inorder" muss hier lauten:

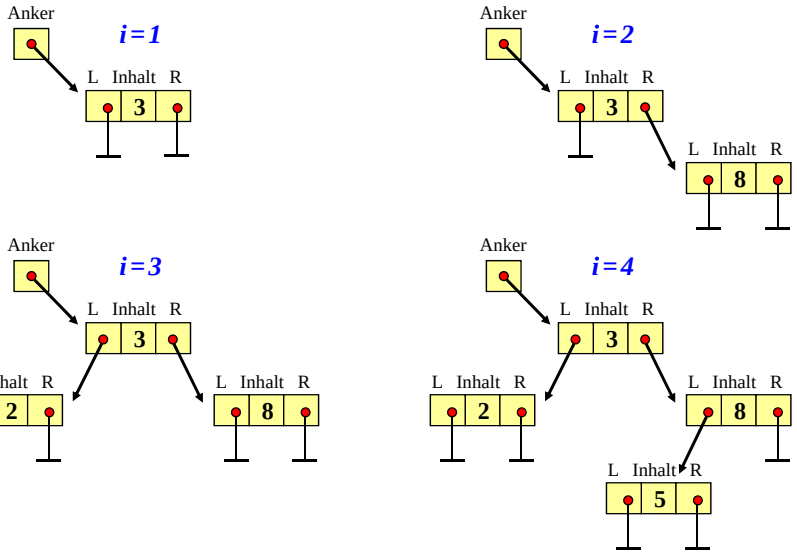
```
if b /= null then
        Inorder (b.L);
        Put (b.Inhalt);
        Inorder (b.R);
end if;
```

Hierdurch wird die sortierte Folge ausgegeben.

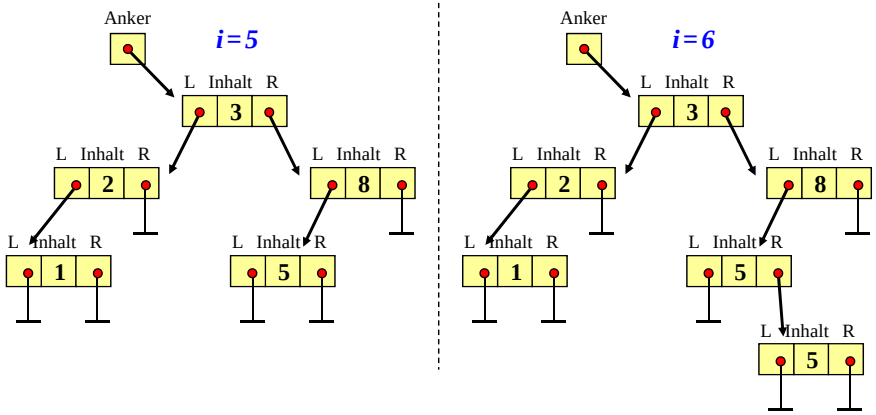
Zeitaufwand:
Überlegen Sie, wie viele Schritte dieses Verfahren benötigt. (Hier spielt die Tiefe des aufgebauten Baums eine Rolle.)

Platzbedarf:
Wie viel Speicherplatz wird für den binären Baum und wie viel für die Rekursion benötigt?

Veranschaulichung für die Zahlenfolge 3, 8, 2, 5, 1, 5:

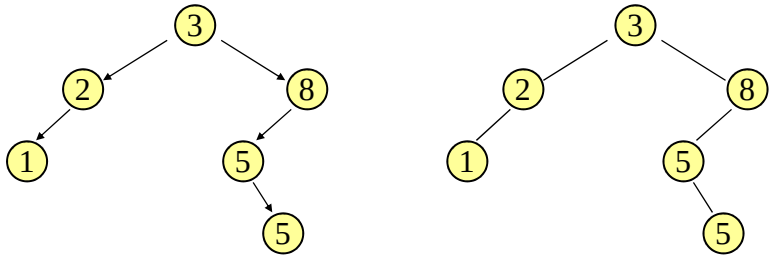


Veranschaulichung für die Zahlenfolge 3, 8, 2, 5, 1, 5:



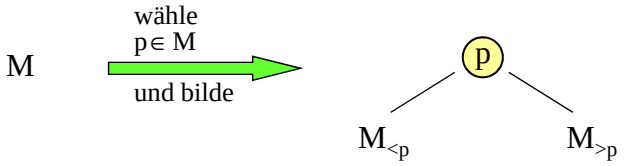
Durchläuft man den Baum, der durch das schrittweise Einsetzen der sechs Zahlen aufgebaut wurde, in Inorder-Reihenfolge, so erhält man die sortierte Folge 1, 2, 3, 5, 5, 8.

Man zeichnet diese Bäume meist ohne Pfeilspitzen:



Diesen Baum kann man sich auch anders entstanden denken: Ausgehend von der Menge M = {3, 8, 2, 5, 1, 5'} wähle man ein Element p, z.B. p = 3, aus, entferne dieses Element p und teile die Menge in zwei Mengen $M_{<p}$ und $M_{>p}$ auf mit
$M_{<p} = \{x \in M \mid x < p\} = \{1, 2\}$ und
$M_{>p} = \{x \in M \mid x > p\} = \{5, 5', 8\}$.

Da eine Menge kein Element doppelt enthält, haben wir die zweite 5 als 5' geschrieben.



Setzt man diesen Zerlegungsschritt rekursiv für $M_{<p}$ und $M_{>p}$ fort, bis diese Mengen leer sind, so erhält man einen Baum, wie er beim Baumsortieren konstruiert wurde.

Dieses Verfahren zum Sortieren (durch rekursive Anwendung des Zerlegungsschritts) liefert fast den Quicksort-Algorithmus. Quicksort unterscheidet sich nur bei gleichen Elementen hiervon, weil diese auch in den linken Unterbaum gelegt werden dürfen. Das Verfahren wird später in 7.3.3 genau erläutert.

3.8 Relationen und Graphen

Wir haben im vorigen Abschnitt Verweise eingeführt, um die Objekte, auf die verwiesen wird, ansteuern und manipulieren zu können. Man kann jedoch Zeiger auch als "Beziehung" oder "Relation" interpretieren.

Beispiel: Es sei K eine Liste aller Kreuzungen einer Stadt, dann besteht zwischen den Kreuzungen eine "Nachbarschafts"-Beziehung: Für zwei Kreuzungen k_1 und k_2 gilt $k_1 \leftrightarrow k_2$ genau dann, wenn die Kreuzungen k_1 und k_2 durch ein Straßenstück verbunden sind und auf diesem Straßenstück keine weitere Kreuzung liegt.

Wir müssen zunächst den Begriff "Relation" definieren. Danach werden wir Relationen in Form von (gerichteten oder ungerichteten) Graphen darstellen.

3.8.1 Definition "Relationen":

Es seien M, M_1, M_2, ..., M_n Mengen.

(1) Jede Teilmenge $R \subseteq M_1 \times M_2 \times ... \times M_n$ heißt n-stellige Korrespondenz (oft auch n-stellige Relation genannt).

(2) Jede Teilmenge $R \subseteq M^n$ heißt n-stellige Relation über M.

(3) Jede Teilmenge $R \subseteq M^2$ heißt zweistellige oder binäre Relation oder auch kurz Relation über M.

Hinweis: In der Literatur werden die Begriffe Korrespondenz und Relation oft vermischt und nicht auseinander gehalten. Wir werden hier nur das Wort "Relation" verwenden.

Wir kennen bereits mehrere binäre Relationen. Ordnungen (z. B. auf Zahlen oder auf Wörtern) und Ableitungen (siehe Abschnitt 2.7) sind solche zweistelligen Relationen.

Schreibweise:

Statt $(x,y) \in R$ schreibt man meist $x\,R\,y$ und sagt, x steht zu y in der Relation R.

3.8.2 Definition:

Seien M eine Menge und $R \subseteq M^2$ eine Relation.

(1) R heißt reflexiv, falls $x\,R\,x$ gilt für alle $x \in M$.

(2) R heißt irreflexiv, falls $x\,R\,x$ für kein $x \in M$ gilt.

(3) R heißt symmetrisch, falls für alle $x, y \in M$ gilt: $x\,R\,y$ genau dann, wenn $y\,R\,x$.

(4) R heißt antisymmetrisch, falls für alle $x, y \in M$ gilt: Aus $x\,R\,y$ und $y\,R\,x$ folgt stets $x = y$.

(5) R heißt transitiv, falls für alle $x, y, z \in M$ gilt: Aus $x\,R\,y$ und $y\,R\,z$ folgt stets $x\,R\,z$.

(6) R heißt alternativ, falls für alle $x, y \in M$ gilt: $x\,R\,y$ oder $y\,R\,x$ (es kann auch beides gelten).

3.8.3 Definition

(1) Eine Relation heißt Äquivalenzrelation, wenn sie reflexiv, symmetrisch und transitiv ist.

(2) Eine Relation heißt Ordnung, wenn sie reflexiv, antisymmetrisch und transitiv ist. (Man schreibt $x \leq y$ für solche Ordnungen.)

(3) Eine Relation heißt echte Ordnung, wenn sie irreflexiv, antisymmetrisch und transitiv ist. (Man schreibt $x < y$ für solche Ordnungen.)

(4) Eine Ordnung R heißt totale oder lineare Ordnung, wenn R alternativ ist. (Für je zwei Elemente x und y gilt hier also mindestens eine der Beziehungen $x\,R\,y$ oder $y\,R\,x$.)

Diese Relationen spielen eine zentrale Rolle in mathematischen Untersuchungen und in praktischen Anwendungen. Insbesondere mit Ordnungen werden wir uns noch intensiv befassen (Sortieralgorithmen).

3.8.4 Veranschaulichung

Es sei $M = \{m_1, m_2, ..., m_n, ...\}$ eine Menge und $R \subseteq M^2$ eine Relation. Man stellt die Relation R meist grafisch dar, indem man die Elemente von M hinschreibt (oft in einen Kreis eingeschlossen) und einen Pfeil (sog. "Kante") genau dann von x nach y zieht, wenn $x\,R\,y$ gilt, d.h., wenn $(x,y) \in R$ gilt.

Beispiel Teilbarkeitsrelation. Es sei $M = \mathbb{N}$ und $TB = \{(x,y) \mid x \text{ teilt } y \text{ und } x \neq y \text{ und } x \neq 1\} \subset \mathbb{N} \times \mathbb{N}$.

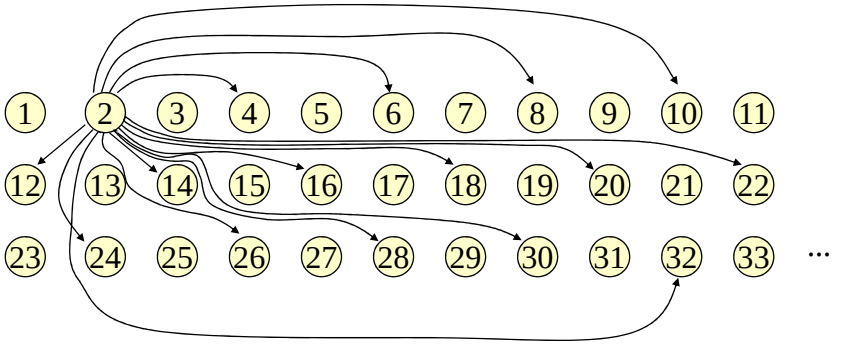
Zunächst schreiben wir die Elemente der Menge $\mathbb{N}$ (auszugsweise von 1 bis 33) hin, dann tragen wir die Relation TB ein:

1 2 3 4 5 6 7 8 9 10 11

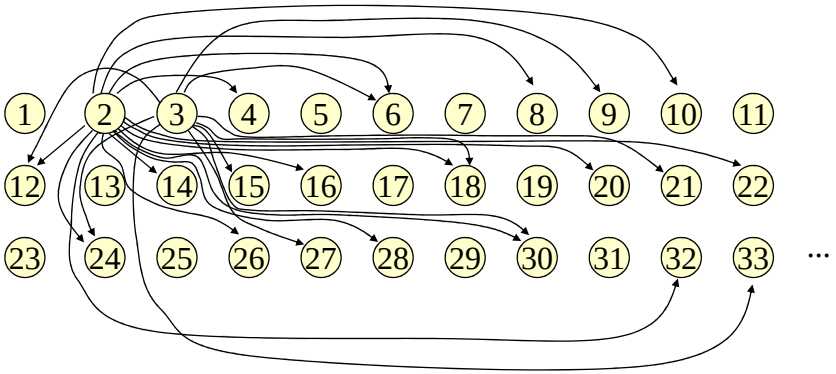
12 13 14 15 16 17 18 19 20 21 22

23 24 25 26 27 28 29 30 31 32 33 ...

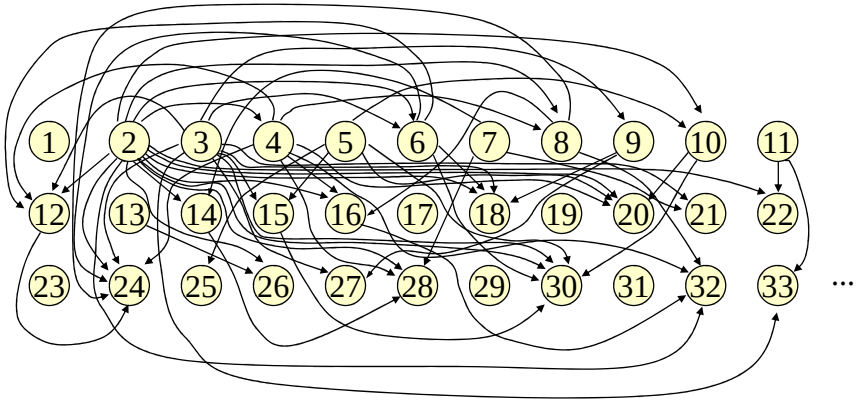
TB bezeichnet die Relation "echte Teilbarkeit". Für die Zahl 1 ist daher nichts zu tun. Die Zahl 2 muss man verbinden mit 4, 6, 8, 10, ... :



Die Zahl 3 muss man verbinden mit 6, 9, 12, ...:



Die Zahl 4 muss man verbinden mit 8, 12, 16, ..., die Zahl 5 mit 10, 15, 20, ... usw. Schließlich erhält man:



Dies ist nur ein Ausschnitt aus der unendlich großen Relation.

Überzeugen Sie sich, dass TB eine echte Ordnung, aber keine lineare Ordnung ist.

Indem man für alle Zahlen x die Paare (x, x) zu TB hinzunimmt, erhält man eine Ordnung (Reflexivität).

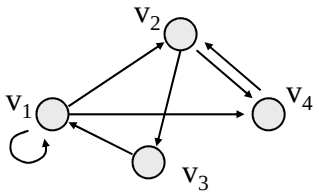
Man denke sich die gesamte Relation TB aufgezeichnet. Dann gilt: Eine Zahl ist genau dann eine Primzahl, wenn auf sie kein Pfeil zeigt, von ihr aber Pfeile ausgehen. Auf dieser Beobachtung beruht das "Sieb des Eratosthenes", siehe Aufgabe 4 in den Übungsaufgaben 1.15.

3.8.5 Graphen

Graphen bestehen aus einer Knotenmenge V und einer Kantenmenge E (englisch: Knoten = **v**ertex oder node, Kante = **e**dge). Die Kantenmenge stellt eine Relation auf V dar. Die folgende Definition einschl. Beispiel kennen wir bereits aus 3.7.1.

Definition 3.8.5 a: G = (V, E) heißt gerichteter Graph (oder Digraph) $\Leftrightarrow$
1. V ist ein endliche nicht-leere Menge,
2. $E \subseteq V \times V$.

[Digraph kommt von "directed graph" = gerichteter Graph.]

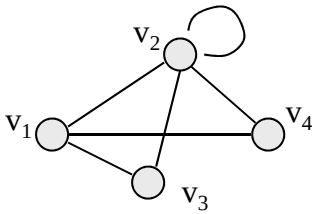


$V = \{v_1, v_2, v_3, v_4\}$

$E = \{(v_1, v_1), (v_1, v_2), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_1), (v_4, v_2)\}$

Im Falle, dass sowohl eine Kante von v nach v' als auch eine Kante von v' nach v führt, spricht man auch von einer ungerichteten Relation bzw. von einem ungerichteten Graphen. In obigem Beispiel sind (v_2, v_4) und (v_4, v_2) solche Kanten.

Definition 3.8.5 b: G = (V, E) heißt ungerichteter Graph
$\Leftrightarrow$
1. V ist ein endliche nicht-leere Menge,
2. $E \subseteq \{\{x, y\} \mid x, y \in V, x \neq y\} \cup \{\{x\} \mid x \in V\}$.



$V = \{v_1, v_2, v_3, v_4\}$

$E = \{\{v_1, v_2\}, \{v_1, v_4\}, \{v_2\}, \{v_2, v_3\}, \{v_3, v_1\}, \{v_4, v_2\}\}$

3.8.5 c:
Umschalten zwischen gerichteten und ungerichteten Graphen:

Es sei G = (V, E) ein ungerichteter Graph. Der gerichtete Graph $G_{ger} = (V, E_{ger})$ mit
$E_{ger} = \{(x, y), (y, x) \mid \{x, y\} \in E\} \cup \{(x, x) \mid \{x\} \in E\}$
heißt gerichtete Version des Graphen G.

Es sei G = (V, E) ein gerichteter Graph. Der ungerichtete Graph $G_{ung} = (V, E_{ung})$ mit
$E_{ung} = \{\{x, y\} \mid (x, y) \in E \text{ oder } (y, x) \in E\} \cup \{\{x\} \mid (x, x) \in E\}$
heißt ungerichtete Version des Graphen G.

Im ungerichteten Fall gehen wir also zu beiden Richtungen über, im gerichteten Fall ignorieren wir die Richtung.

Ein gerichteter Graph H heißt Orientierung oder Ausrichtung des ungerichteten Graphen G, wenn G die ungerichtete Version von H ist. (Zu G mit |E| Kanten gibt es $2^{|E|}$ Orientierungen H.)

Definition 3.8.5 d: Teilgraph, induzierter Teilgraph

Es sei G = (V, E) ein ungerichteter bzw. gerichteter Graph. Ein ungerichteter bzw. gerichteter Graph G' = (V', E') heißt Teilgraph von G, wenn $V' \subseteq V$ und $E' \subseteq E$ gilt.

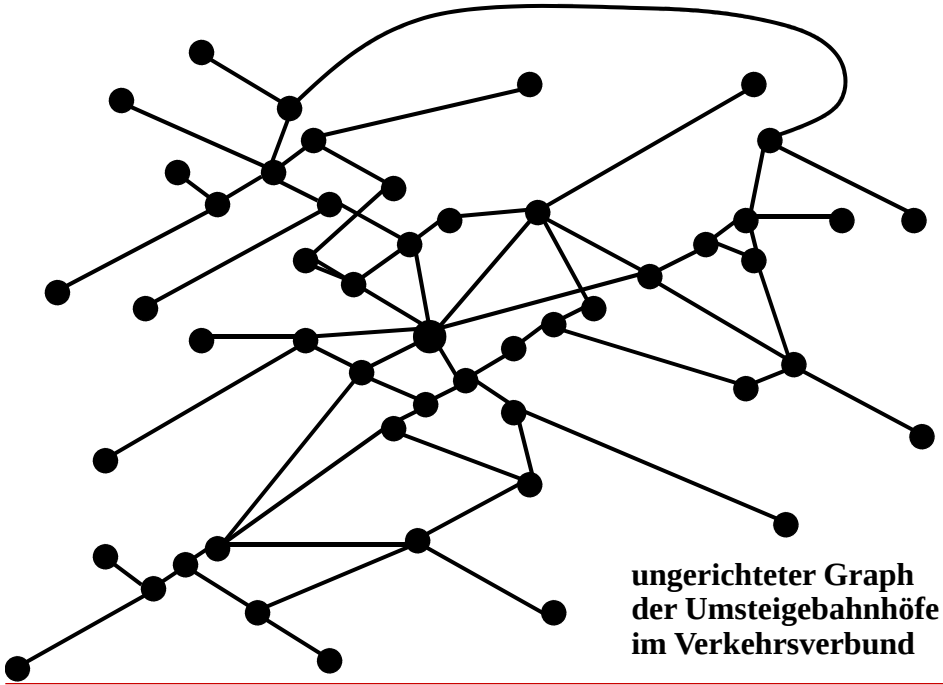
Es sei G = (V, E) ein ungerichteter bzw. gerichteter Graph und es sei $V' \subseteq V$. Der ungerichtete bzw. der gerichtete Graph G' = (V', E') heißt der von V' induzierte Teilgraph von G, wenn im gerichteten Fall

$$E' = \{\{x, y\} \mid \{x, y\} \in E \text{ und } x, y \in V'\}$$

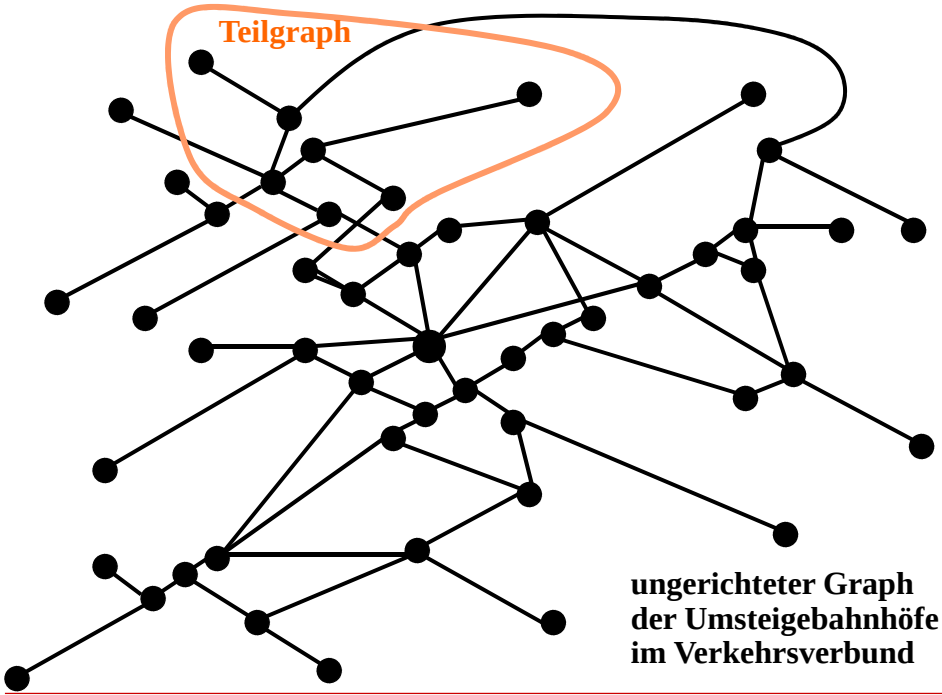
bzw. im gerichteten Fall

$$E' = \{(x, y) \mid (x, y) \in E \text{ und } x, y \in V'\}$$

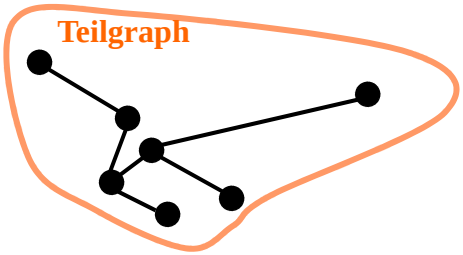
gilt.



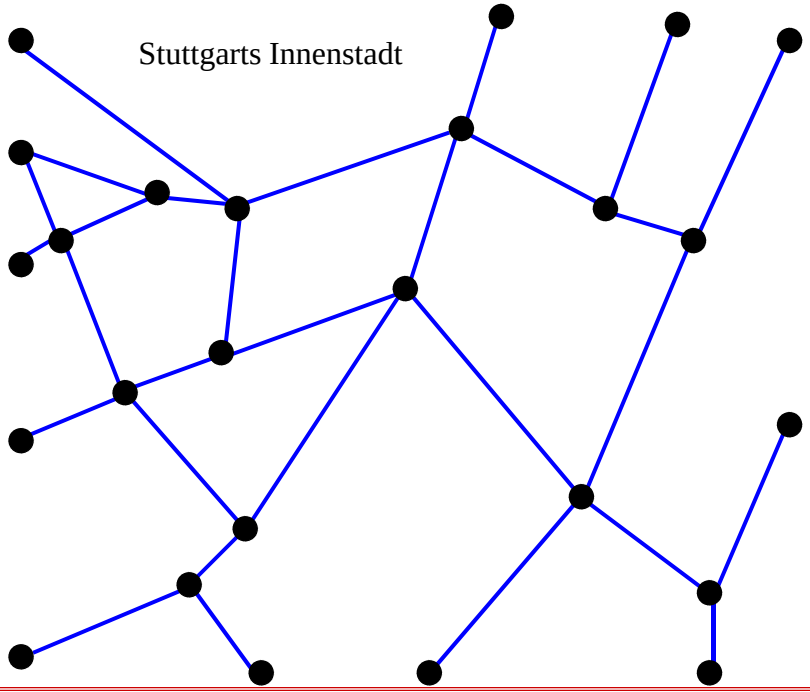
ungerichteter Graph der Umsteigebahnhöfe im Verkehrsverbund



ungerichteter Graph der Umsteigebahnhöfe im Verkehrsverbund



Machen Sie sich klar, dass ein Graph sehr sehr viele Teilgraphen besitzt. Können Sie eventuell eine Formel angeben, wie viele Teilgraphen ein Graph mit n Knoten und m Kanten hat?



Definition 3.8.5 e: (adjacent, neighbour, successor, predecessor)

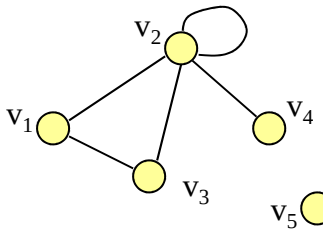
Jede Kante $\{x, y\}$ bzw. (x,y) heißt inzident zu ihren Knoten x und y. Zwei Knoten x, y mit $\{x,y\} \in E$ (ungerichteter Fall) bzw. $(x,y) \in E$ oder $(y,x) \in E$ (gerichteter Fall) heißen adjazent oder benachbart.

Ungerichteter Fall: Die Menge $N(x) = \{ y \in V \mid \{x,y\} \in E \}$ heißt die Menge der Nachbarn (oder der Nachfolger) von x.

Gerichteter Fall: $N(x) = \{ y \in V \mid (x,y) \in E$ oder $(y,x) \in E\}$ heißt die Menge der Nachbarn von x.
$S(x) = \{ y \in V \mid (x,y) \in E\}$ heißt Menge der Nachfolger von x,
$P(x) = \{ y \in V \mid (y,x) \in E\}$ heißt Menge der Vorgänger von x.

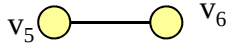
Eine Kante $\{x\}$ im ungerichteten Fall bzw. (x,x) im gerichteten Fall heißt Schlinge.

Beispiel: ungerichteter Fall

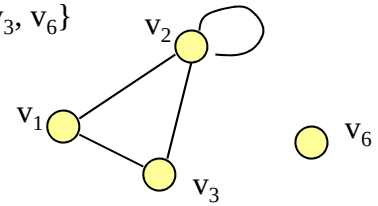


$V = \{v_1, v_2, v_3, v_4, v_5 v_6\}$

$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_1\}, \{v_2\}, \{v_4, v_2\}, \{v_5, v_6\}\}$



Der von $V' = \{v_1, v_2, v_3, v_6\}$ induzierte Teilgraph:



Definition 3.8.5 f: Grad d, Eingangs-, Ausgangsgrad, geordnet

Ungerichteter Fall: Für einen Knoten x heißt
$d(x) = |\{ y \in V \mid x \neq y, \{x,y\} \in E \}|$, falls $\{x\} \notin E$ bzw.
$d(x) = |\{ y \in V \mid x \neq y, \{x,y\} \in E \}| + 2$, falls $\{x\} \in E$
der (Knoten-) Grad von x ("degree"). Der maximale Knotengrad heißt Grad d(G) des Graphen G.

Gerichteter Fall: Für einen Knoten x heißt
$d^+(x) = |\{ y \in V \mid (x,y) \in E \}|$ der Ausgangsgrad und
$d^-(x) = |\{ y \in V \mid (y,x) \in E \}|$ der Eingangsgrad von x. Der Wert
$d(x) = d^+(x) + d^-(x)$ heißt auch der Grad von x.

Ein Graph heißt geordnet, wenn für jeden Knoten x im ungerichteten Fall die Menge der Nachbarn N(x) bzw. im gerichteten Fall die Menge der Nachfolger S(x) linear geordnet ist (d.h., die zugehörigen Kanten sind linear geordnet).

3.8.5 g Definitionen zur Darstellung von Graphen:

Graphen werden meist durch ihre Adjazenzmatrix A, durch Adjazenzlisten oder durch Inzidenzlisten dargestellt.

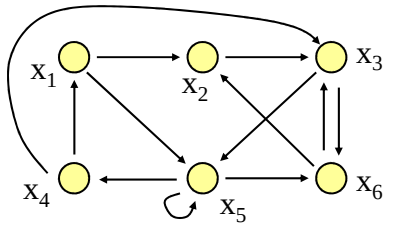
Es sei $G = (V, E)$ mit $V = \{x_1, x_2, ..., x_n\}$ ein Graph.

Ungerichteter Fall: Die Adjazenzmatrix $A = (a_{i,j})$ ist definiert durch $a_{i,j} = 1$, falls $\{x_i, x_j\} \in E$, und $a_{i,j} = 0$ sonst $(i, j = 1, ..., n)$.

Gerichteter Fall: Die Adjazenzmatrix $A = (a_{i,j})$ ist definiert durch $a_{i,j} = 1$, falls $(x_i, x_j) \in E$, und $a_{i,j} = 0$ sonst $(i, j = 1, ..., n)$.

Die *erweiterte Adjazenzmatrix* A' ist für $i \neq j$ gleich der Adjazenzmatrix, allerdings wird die Hauptdiagonale auf 1 gesetzt, d. h. $a'_{i,j} = a_{i,j}$ für $i \neq j$ und $a'_{i,i} = 1$ (für $i, j = 1, ..., n$).

Beispiel:



Adjazenzmatrix A =

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Hinweis: Die Adjazenzmatrix A zeigt an, von welchem zu welchem Knoten man "in einem Schritt" gelangen kann. Die Komponente $a^{(k)}_{i,j}$ in der Matrix A^k gibt an, wie viele verschiedene Wege der Länge k (siehe Def. 3.8.8) es vom Knoten x_i zum Knoten x_j gibt. Obiges Beispiel: Bilde $A \cdot A = A^2$:

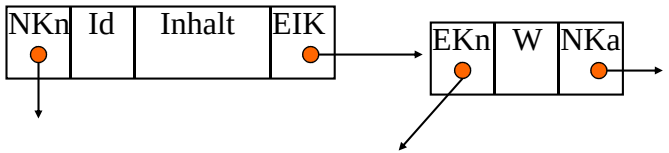
$$A^2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 2 & 1 \\ 1 & 1 & 2 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 2 & 2 \end{pmatrix}$$

Aus $a^{(2)}_{4,5} = 2$ folgt also: Es gibt zwei verschiedene Wege der Länge 2 von x_4 nach x_5 (nämlich $x_4 \to x_1 \to x_5$ und $x_4 \to x_3 \to x_5$).

3.8.6 Datentyp für Graphen: Darstellung als Adjazenzliste

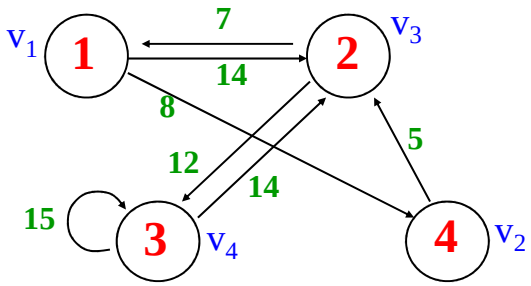
Jeder Knoten erhält einen Identifikator Id (in der Regel ist dies eine natürliche Zahl) und einen Inhalt.
Die Knoten werden in Listen zusammengefasst und besitzen daher neben Id und Inhalt noch weitere Komponenten:
- Verweis auf den **N**ächsten **K**noten in der Liste "NKn",
- Verweis auf die **E**rste **I**nzidente **K**ante "EIK".

Die von einem Knoten ausgehende Kante muss enthalten: den "Endknoten der Kante" (EKn), ihren Wert W ("weight") und einen Verweis auf die nächste Kante "NKa".



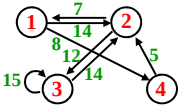
In der Praxis besitzen Graphen zusätzliche Informationen sowohl in den Knoten (z.B. geografische Angaben) als auch in den Kanten (z.B. Entfernungen, Zeitdauer).

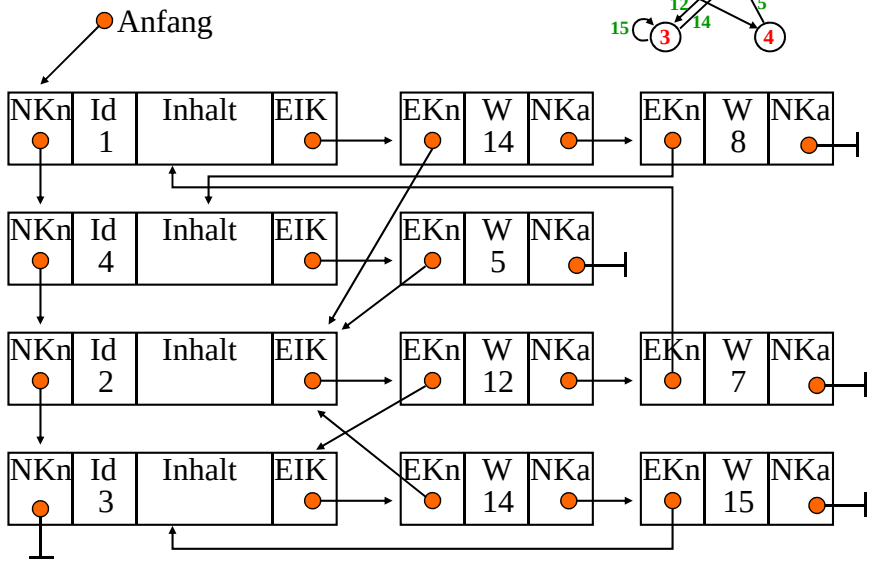
Beispiel: Gerichteter Graph mit Knotennummern und Kantenwerten:



Adjazenzliste hierzu

Zugehöriger Graph:





Die meisten Anwendungen notieren viele Informationen in den Graphen.

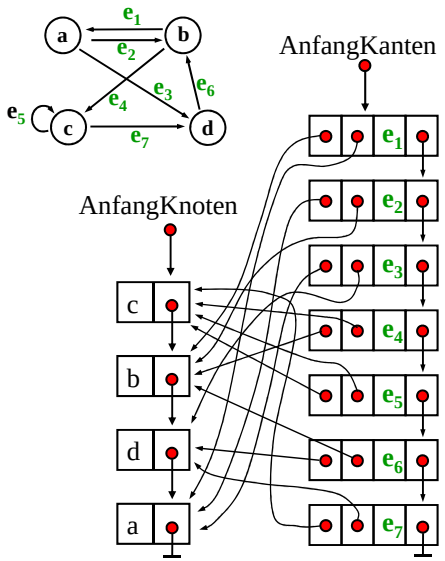
Sehr oft muss man sich zwei Zahlen merken, die Ergebnisse zuvor durchgeführten Durchläufen durch den Graphen sind, sowie einen Booleschen Wert "Besucht", der angibt, ob dieser Knoten bereits zuvor erreicht ("besucht") worden ist.

Wir fügen diese Komponenten zu den Knoten hinzu und erhalten folgende Datentypen, formuliert in Ada:


```
type Knoten; type Kante;
type NextKnoten is access Knoten;
type NextKante is access Kante;

type Knoten is record
        Id: Knotenname;
        Besucht: Boolean; zahl1, zahl2: Integer;
        Inhalt: <weitere Komponenten>;
        NKn: NextKnoten;
        EIK: NextKante;
end record;

type Kante is record
        W: <Typ des Gewichts der Kanten>;
        EKn: NextKnoten;
        NKa: NextKante;
end record;
```

Darstellung als Inzidenzliste

Hierfür bildet man eine lineare Liste der Knoten und eine lineare Liste der Kanten. Zu jeder Kante gibt man eine Referenz auf den Anfangsknoten und eine Referenz auf den Endknoten der Kante an. Die Reihenfolgen in den Listen kann man beliebig wählen.



3.8.7 Durchlauf durch Graphen

Wir wollen mit einer Prozedur GD ("Graphdurchlauf") einen Graphen durchlaufen und dabei jeden Knoten und jede Kante besuchen. Zu den Knoten werden wir häufiger gelangen, jede Kante wird aber nur genau einmal durchlaufen.
Als Darstellung wählen wir Adjazenzlisten.

In jedem Knoten benötigen wir einen Booleschen Wert "Besucht", der uns sagt, ob wir diesen Knoten bereits besucht haben oder nicht. Üblicherweise definiert man den Durchlaufalgorithmus folgendermaßen rekursiv.

Initialisierung: Setze alle Besucht-Werte auf false.
Durchlaufe dann alle Knoten entlang der Knotenliste. Bei jedem Knoten u mache man Folgendes (Prozeduraufruf GD):

Falls u als "besucht" markiert ist, so gehe zum nächsten Knoten in der Knotenliste.
Sonst: Markiere den Knoten u als "besucht".
< Bearbeite den Knoten u. >
Für alle von u ausgehenden Kanten e:
< Bearbeite die Kante e. >
Folge der Kante e zu ihrem Endknoten v
und rufe die Prozedur GD rekursiv mit v auf.

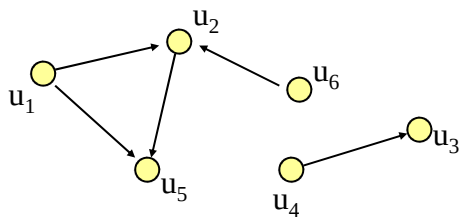
Auf diese Weise wird jeder Knoten so oft besucht, wie Kanten zu ihm führen, und einmal mehr, weil zusätzlich die Knotenliste durchlaufen wird.

Knoten und Kanten deklarieren wir wie oben (3.8.5, drei Folien zurückblättern) angegeben. Der Graph mit n Knoten und m Kanten liegt als Adjazenzliste vor. Auf die Knotenliste verweist hierbei eine NextKnoten-Variable mit dem Namen "Anfang".

Algorithmus Graphdurchlauf GD:

```
...
Anfang, p: NextKnoten;
procedure GD(u: in out NextKnoten) is
    e: NextKante;
begin  if not u.Besucht then
           u.Besucht := true; < "bearbeite den Knoten u" >;
           e := u.EIK;
           while e /= null loop  < "bearbeite die Kante e" >;
                 GD(e.EKn);  e:=e.NKa;  end loop;
       end if;
end GD;
begin  ... < "baue den Graphen auf" >; ... ;
    p := Anfang;
    while p /= null loop p.Besucht:=false; p := p.NKn; end loop;
    p := Anfang;
    while p /= null loop GD(p); p := p.NKn; end loop;  ...
end;
```

Machen Sie sich den Algorithmus an folgendem einfachen Beispiel klar. Die Reihenfolge in der Knotenliste des Graphen sei u_1, u_2, u_3, u_4, u_5, u_6 und (u_1, u_2) sei die erste von u_1 ausgehende Kante.



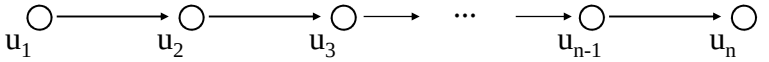
Dann werden die Knoten in der Reihenfolge
u_1, u_2, u_5, u_3, u_4, u_6 bearbeitet (entsprechend
< "bearbeite den Knoten u" > in der Prozedur GD).

Zeitaufwand: Die Prozedur GD wird n mal in der Schleife

```
while p /= null loop GD(p); p := p.NKn; end loop;
```

und m-mal in der Prozedur selbst aufgerufen. Jeder Knoten und jede Kante werden genau einmal bearbeitet. Der Zeitaufwand ist daher proportional zu (n+m).

Platzaufwand: Die rekursiven Aufrufe kosten zusätzlichen Platz im Kellerspeicher. Der ungünstigste Fall liegt vor, wenn der Graph eine Folge von Knoten ist:



In diesem Fall werden n-1 rekursive Aufrufe ineinander geschachtelt, bevor die erste Prozedur wieder verlassen wird. Da hierbei die Variablen e und v (= e.EKn) neu angelegt werden, erhalten wir einen *Zusatz-Platzbedarf* der Größe 2n.

Definition 3.8.8: ungerichtete Wege und Pfade

Gegeben sei ein ungerichteter Graph G = (V, E) mit
$E \subseteq \{ \{x,y\} \mid x,y \in V, x \neq y \} \cup \{ \{x\} \mid x \in V \}$.
Eine Folge von Knoten (u_1, u_2, u_3, ..., u_r) mit $r \geq 1$ heißt Weg im Graphen G, wenn $\{u_i, u_{i+1}\} \in E$ für i = 1, 2, ..., r-1 gilt.

r-1 heißt die Länge des Weges.

Man sagt: Der Weg (u_1, u_2, ..., u_r) führt vom Knoten u_1 zum Knoten u_r oder u_r ist von u_1 aus (über diesen Weg) erreichbar.

Zwei Knoten u und v heißen verbunden, wenn es einen Weg von u nach v gibt.

Betrachtet man dagegen die Kanten, die diesen Weg bilden, so spricht man von "Pfaden" im Graphen, d.h., wenn
(u_1, u_2, u_3, ..., u_r) ein Weg ist, so ist
$(\{u_1,u_2\}, \{u_2,u_3\}, \{u_3,u_4\}, ..., \{u_{r-1},u_r\})$
der zugehörige Pfad im Graphen.

Definition 3.8.9: Zusammenhang für ungerichtete Graphen

Ein Weg (u_1, u_2, u_3, ..., u_r) heißt doppelpunktfrei oder einfach, wenn $u_i \neq u_j$ für alle $i \neq j$ gilt.

Ein Weg heißt geschlossen, wenn $u_r = u_1$ gilt.

Ein Weg (u_1, u_2, u_3, ..., u_r) heißt Kreis oder Zyklus, wenn $r \geq 4$ ist, der Weg geschlossen ist und (u_1, u_2, u_3, ..., u_{r-1}) doppelpunktfrei ist. Ein Graph heißt zyklenfrei oder azyklisch, wenn er keine Zyklen besitzt.

Ein Graph heißt zusammenhängend, wenn jeder Knoten mit jedem Knoten verbunden ist.

Die Menge
$Z(u) = \{ v \in V \mid u \text{ und } v \text{ sind verbunden} \}$
heißt die Zusammenhangskomponente des Knotens u.
(Es gilt: Aus $v \in Z(u)$ folgt $u \in Z(v)$.)

Definition 3.8.10: gerichtete Wege und Pfade

Gegeben sei ein gerichteter Graph G = (V, E) mit $E \subseteq V \times V$. Eine Folge von Knoten $(u_1, u_2, u_3, ..., u_r)$ mit $r \geq 1$ heißt (gerichteter) Weg im Graphen G, wenn $(u_i, u_{i+1}) \in E$ für i = 1, 2, ..., r-1 gilt. r-1 heißt die Länge des Weges.

Man sagt, der Weg $(u_1, u_2, ..., u_r)$ führt vom Knoten u_1 zum Knoten u_r, oder, der Knoten u_r ist von u_1 aus über den Weg $(u_1, u_2, ..., u_r)$ erreichbar.

Betrachtet man die Kanten, die diesen Weg bilden, so spricht man von "Pfaden" im Graphen, d.h., wenn $(u_1, u_2, u_3, ..., u_r)$ ein Weg ist, so ist

$$((u_1,u_2), (u_2,u_3), (u_3,u_4), ..., (u_{r-1},u_r))$$

der zugehörige Pfad im Graphen.

Zwei Knoten u und v heißen verbunden, wenn es einen Weg von u nach v und einen Weg von v nach u gibt.

Definition 3.8.11: Zusammenhang für gerichtete Graphen

Ein Weg $(u_1, u_2, u_3, ..., u_r)$ heißt doppelpunktfrei oder einfach, wenn $u_i \neq u_j$ für alle $i \neq j$ gilt.

Ein Weg heißt geschlossen, wenn $u_r = u_1$ gilt.

Ein Weg $(u_1, u_2, u_3, ..., u_r)$ heißt Kreis oder Zyklus, wenn $r \geq 4$ ist, der Weg geschlossen ist und $(u_1, u_2, u_3, ..., u_{r-1})$ doppelpunktfrei ist. Ein Graph heißt zyklenfrei oder azyklisch, wenn er keine Zyklen besitzt.

Ein gerichteter Graph heißt stark zusammenhängend, wenn jeder Knoten mit jedem Knoten verbunden ist.

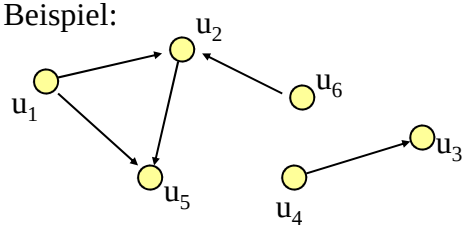
Die Menge
$Z(u) = \{ v \in V \mid u \text{ und } v \text{ sind verbunden } \}$
heißt die starke Zusammenhangskomponente des Knotens u.
Es ist wiederum klar, dass für alle $v \in Z(u)$ gilt: $u \in Z(v)$.

Definition 3.8.12: schwacher Zusammenhang

Zu einem gerichteten Graphen $G = (V, E)$ sei $G_{ung} = (V, E_{ung})$ mit $E_{ung} = \{\{x,y\} \mid (x,y) \in E \text{ oder } (y,x) \in E\} \cup \{\{x\} \mid (x,x) \in E\}$ die ungerichtete Version (siehe 3.8.5 c).

Die Menge $SwZ(u) = Z(u)$ in G_{ung} heißt die schwache Zusammenhangskomponente des Knotens u im Graphen G. (Wiederum folgt, dass für alle $v \in SwZ(u)$ gilt: $u \in SwZ(v)$.)

Beispiel:



$Z(u_1) = \{u_1\}$,
$Z(u_3) = \{u_3\}$,
$SwZ(u_1) = \{u_1, u_2, u_5, u_6\}$,
$SwZ(u_3) = \{u_3, u_4\}$.

Definition 3.8.13: transitive Hülle

Zu dem gerichteten oder ungerichteten Graphen $G = (V, E)$ heißt der gerichtete bzw. ungerichtete Graph $G_{tH} = (V, E_{tH})$ mit
$E_{tH} = \{(x,y) \mid \text{es gibt einen Weg von } x \text{ nach } y\}$
die transitive Hülle des Graphens G.

Zwei spezielle Graphen:
Ein Graph, in dem je zwei verschiedene Knoten miteinander durch eine Kante verbunden sind, heißt vollständiger Graph $\mathbf{K_n} = (V, E)$ und $E = \{\{x, y\} \mid x, y \in V, x \neq y\}$ im ungerichteten Fall bzw. $E = \{(x, y) \mid x, y \in V, x \neq y\}$ im gerichteten Fall.

Der Graph, bei dem n Knoten einen Ring bilden, heißt "Kreis" $\mathbf{C_n} = (V, E)$ mit $V = \{x_1, x_2, ..., x_n\}$ und
$E = \{\{x_1, x_2\}, \{x_2, x_3\}, \{x_3, x_4\}, ..., \{x_{n-1}, x_n\}, \{x_n, x_1\}\}$ (ungerichtet)
bzw. $E = \{(x_1, x_2), (x_2, x_3), ..., (x_{n-1}, x_n), (x_n, x_1)\}$ im gerichteten Fall.

3.8.14: *Einfache Anwendung* des Graphdurchlaufs GD

Wenn man *bei ungerichteten Graphen* zu Beginn des Aufrufs GD (siehe 3.8.7) aus der while-Schleife heraus immer eine neue Liste beginnt (sofern p.Besucht false ist), in die man alle in der Prozedur erstmals besuchten Knoten einfügt, so erhält man die Zusammenhangskomponenten.

Denn im ungerichteten Fall gibt es zu jedem Weg von u nach v auch einen Weg von v nach u, und alle Knoten, die von u aus erreichbar sind, sind bereits als "besucht" markiert, wenn man beim Durchlauf durch die Knotenliste auf sie trifft. Es verbleiben dann nur noch Knoten aus anderen Zusammenhangskomponenten.

Hinweise im Detail: Füge zum Programm für GD hinzu

```
I: Integer; Z: array (1..n) of NextKnoten;
```

Vor der letzten while-Schleife füge I := 0; ein und ersetze die letzte while-Schleife durch:

```
while  p /= null  loop
        if not p.Besucht then I:=I+1; Z(I) := null; GD(p); end if;
        p := p.NKn;
end loop;
```

und in der Prozedur fügen wir unmittelbar nach "then" ein:
Z(I) := new NextKnoten'(NKn => Z(I); Id => u.Id);

Am Ende bilden die Listen Z(1), Z(2), ... die Zusammenhangskomponenten.

Übungsaufgabe: Im gerichteten Fall muss man anders vorgehen. Überlegen Sie, wie.

4. Begriffe der Programmierung

Das Kapitel 4 stellt einige wesentliche Konzepte der Programmierung vor. Diese treten in den meisten Programmiersprachen in irgendeiner Form auf.

Jeder Abschnitt besteht aus zwei Teilen: Zunächst werden die Konzepte unabhängig von einer konkreten Sprache vorgestellt (allerdings ergibt sich eine Nähe zu Ada durch die Beispiele), danach wird kurz deren Darstellung in Ada besprochen, woran sich oft noch weitere Begriffe anschließen, die für Ada in diesem Zusammenhang von Bedeutung sind.

4.1 Blöcke, Ausnahmen, Überladen

Wiederholung: Ein Name ist in der Regel nur in einer bestimmten Umgebung eindeutig. Zum Beispiel wird es in einer Vorlesung mit 300 Hörer(inne)n meist nur höchstens einen "Paul Müller" oder eine "Rita Weber" geben, bundesweit dagegen können viele Personen diesen Namen tragen.

In einem Programm muss jeder Name eine eindeutige Bedeutung besitzen. Fügt man jedoch zwei Programme zu einem neuen zusammen, so können hierbei zwei gleiche Namen, die bisher in den beiden getrennten Programmen vorkamen, zusammentreffen und zu einer Mehrdeutigkeit führen. Man wird daher jedem Namen eine "Umgebung" zuordnen, in der er gültig ist. Diese Umgebung, also ein in sich geschlossenes Programmstück mit Deklarationen, nennt man einen "Block".

Festlegung 4.1.1 (Erinnerung, vgl. Abschnitt 1.11.5):

Ein Block ist eine in sich geschlossene, durch begin ... end geklammerte Folge von Anweisungen, die einen Deklarationsteil am Anfang (und in Ada am Ende Ausnahmebehandlungen, um Fehlersituationen abzufangen) besitzen darf.

Die im Deklarationsteil vereinbarten Bezeichnungen können nur innerhalb dieses Blocks und seiner Unterblöcke verwendet werden. Wird der Block verlassen, so sind diese Bezeichner und die durch sie bezeichneten Objekte undefiniert oder "unbekannt".

Jeder Bezeichner, der explizit im Deklarationsteil eines Blocks oder implizit als Laufvariable, Marke oder Bezeichnung einer Schleife oder eines Blocks eingeführt wird, heißt lokal zu diesem Block. Bezeichner (bzw. die durch sie bezeichneten Objekte), die in Ober-Blöcken deklariert und in Unter-Blöcken verwendet werden, heißen global in den Unterblöcken.

Wird in einem Unter-Block ein Bezeichner, der in einem Ober-Block vereinbart wurde, neu deklariert, so wird der äußere Bezeichner "ausgeblendet" und im Unter-Block kann nur das dort deklarierte Objekt unter diesem Bezeichner angesprochen werden.

Festlegung 4.1.2:
Steht ein Bezeichner X in einem Programm mit Blöcken an einer Stelle s, so bezieht sich X stets auf die Deklaration von X, die sich im Deklarationsteil des kleinsten ("innersten") Blocks befindet, der s umfasst.
Nach der Deklaration eines Bezeichners, der an einer Stelle s verwendet wird, wird daher stets von s ausgehend nach oben in der Hierarchie der Blöcke gesucht.

Hinweis (hier für Sie noch nicht ganz zu verstehen):
Diese Festlegung gilt auch für die Benutzung von Namen in Funktionen, Prozeduren, Moduln und anderen Programmeinheiten mit Deklarationen *(nach Anwendung der Kopierregel, siehe später).* Hierbei dürfen globale Variablen aber nicht "lokaler" werden, siehe Sonderfall 4 in der Kopierregel in 4.2 sowie das Beispiel auf der nächsten Folie.

Beispiel (in Ada-Schreibweise)

```
declare L: Integer := 0;
procedure P is
begin  L := L + 1;  end;
...
   declare  L: Integer := 1;
   begin
      P;
      ...
   end;
...
```

Wenn P im inneren Block aufgerufen wird, dann wird L := L+1 ausgeführt; aber mit "L" ist hier der Bezeichner gemeint, der global zur Prozedur P ist, also das mit 0 initialisierte L des äußeren Blocks.

4.1.3 Mit den Blöcken sind die Begriffe Gültigkeitsbereich (oder Sichtbarkeitsbereich) und Lebensdauer von Objekten und ihren Bezeichnern verbunden, siehe 1.11.5.

Erinnerung: Die Lebensdauer eines Bezeichners und des mit ihm verbundenen Objekts ist der Block, in dem der Bezeichner deklariert wurde. Der Bezeichner / das Objekt *lebt genau ab dieser deklarierenden Stelle* (im Falle einer Variablen bedeutet dies: Die Variable besitzt nun einen Speicherplatzbereich im lokalen Kellerspeicher), aber nur innerhalb dieses Blockes und aller Unter-Einheiten; der Bezeichner und das zugehörige Objekt "sterben", sobald dieser Block verlassen wird; außerhalb des Blocks sind sie unbekannt. Wird der Block später wieder neu betreten, so wird ein neues Objekt erzeugt (oder "geboren").

Spezialfall in Ada: Die Lebensdauer einer *Laufvariablen* ist die jeweilige Schleife. Außerhalb der Schleife ist die Laufvariable nicht existent. Die Schleife ist für die Laufvariable wie ein deklarierender Block.

Der Gültigkeitsbereich oder Sichtbarkeitsbereich eines Objekts und seines Bezeichners ist der Teil der Lebensdauer, in dem hierauf unmittelbar über den Namen zugegriffen werden kann. Nicht gültig (oder sichtbar, engl. *visible*) sind Objekt und Bezeichner in allen Untereinheiten, in denen der Bezeichner neu deklariert wird. Sie "tauchen wieder auf" (= sie werden wieder sichtbar), sobald diese Untereinheiten verlassen werden.

Beachte: Jedes Objekt lebt ab seiner Deklaration bis zum end des deklarierenden Blocks. Die allgemeine Regel, dass ein Bezeichner sichtbar sein muss, wenn man ihn verwenden will, gilt auch für den Deklarationsteil. Daher müssen wir z.B. in Ada bei access-Datentypen den Typ T, auf den verwiesen wird, zuvor spezifizieren, dann den access-Typ definieren und anschließend T selbst.

In Ada kann man lebende, aber nicht sichtbare Variablen über einen Blocknamen (am Anfang des Blocks, durch Doppelpunkt abgetrennt; Punkt-Schreibweise verwenden) ansprechen und somit sichtbar machen. Solche "verwässernden" Möglichkeiten sollte man aber nur sehr sparsam benutzen. Beispiel:

```
...
Aussen: declare X: Integer := 3;
begin ...
    declare X: Integer := 5;
    begin ...
        Aussen.X := X*X;
        ...
    end;
    ...
end;
```

Der Name Aussen.X bezeichnet also die im Block mit dem Bezeichner "Aussen" deklarierte Variable X.

4.1.4: Speicherverwaltung mit Blöcken

Die Variablen, die in Blöcken vereinbart werden, werden grundsätzlich im Kellerspeicher abgelegt. Die Anlegung der Speicherbereiche erfolgt hierbei *während der Laufzeit:* Sobald der Block betreten wird, wird der Platz für die neu vereinbarten Variablen reserviert ("Allokation" von Speicher).

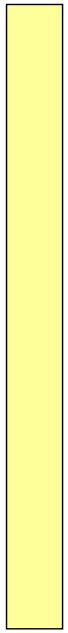
Wird der Block verlassen, so wird der Speicherplatz wieder frei gegeben. (In der Praxis wird allerdings nur der Verweis auf den letzten Kellerspeicherplatz verändert.)

Wir erläutern dies an folgendem Beispiel.

Programmzeiger

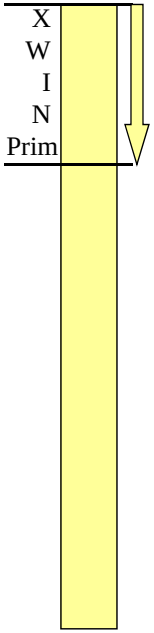
Beispiel

```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```

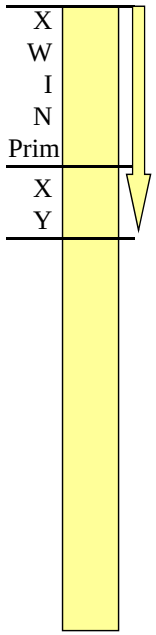


Lokaler Kellerspeicher

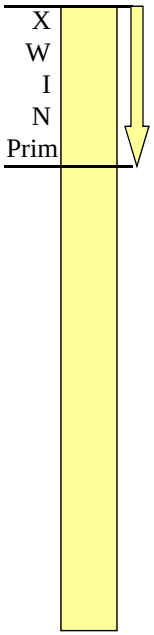

```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



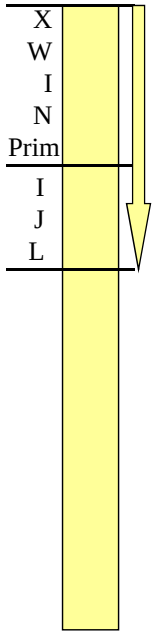

```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



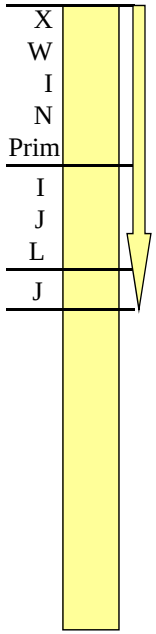

```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



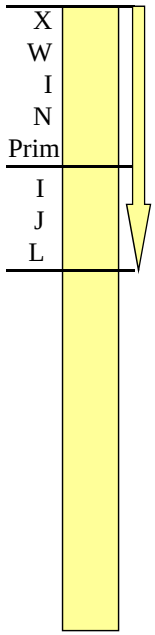
```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



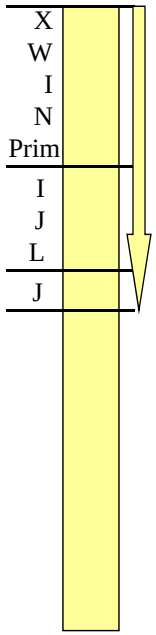

```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



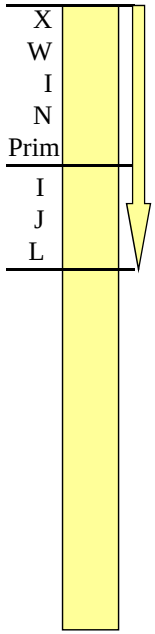

```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



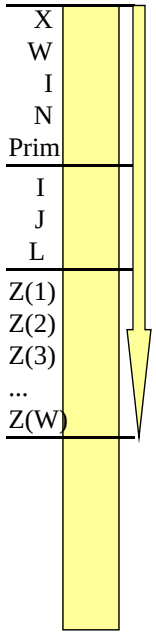

```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



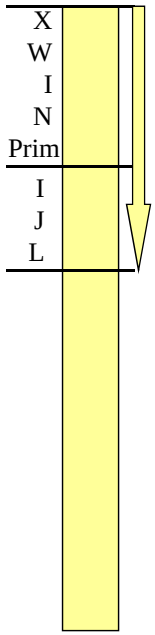
```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



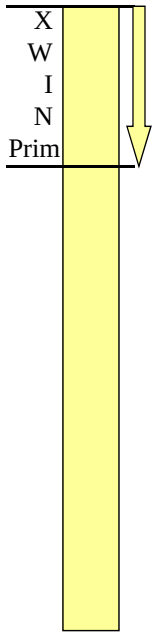

```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```

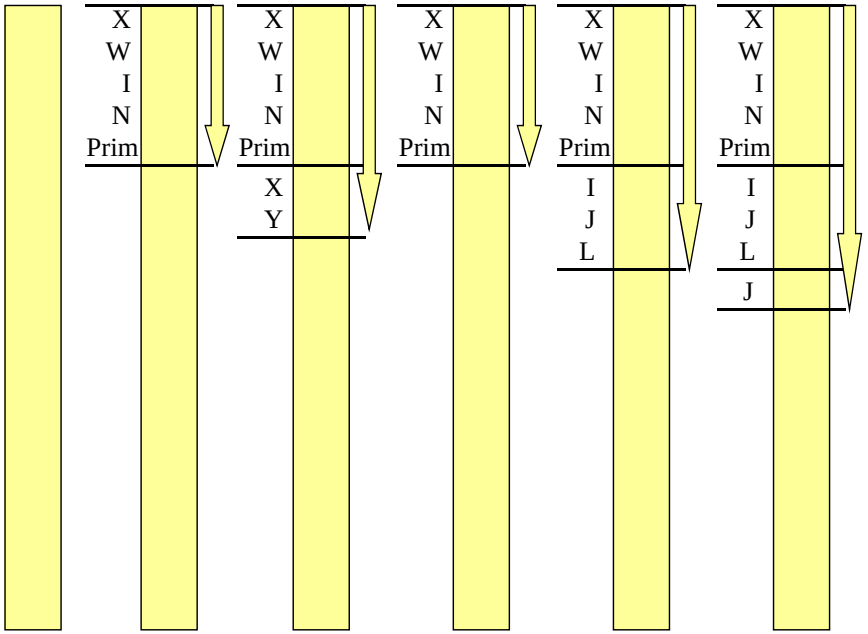



```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```

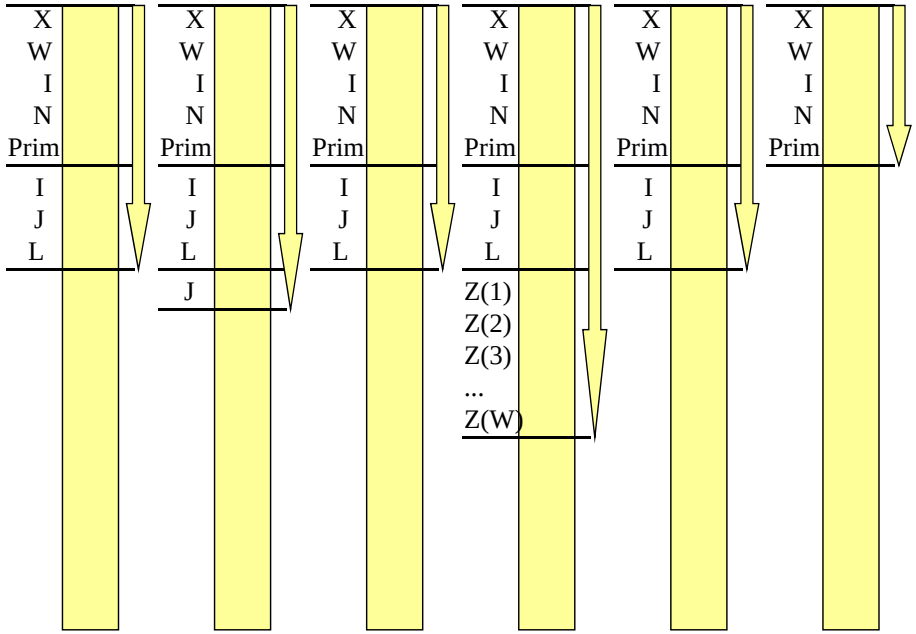



```
procedure BSP is    -- nicht auf den Inhalt achten!
X: Float; W, I, N: Integer; Prim: Boolean;
begin Get(N);
   declare X, Y: Float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: Integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: Integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of Integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```

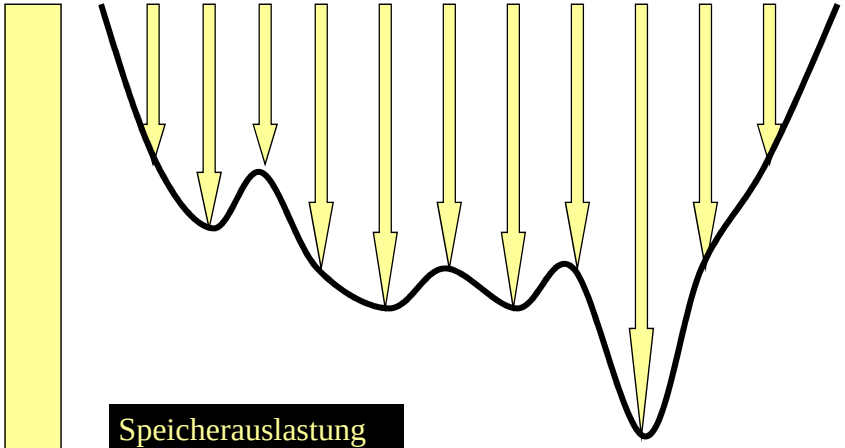




Speicherbelegung im Kellerspeicher



Speicherbelegung im Kellerspeicher



Der lokale Speicher ist ein pulsierender Kellerspeicher.
Er lässt sich durch verschachtelte Blöcke steuern.

4.1.5: Vorteile von Blöcken

1. Sie bilden kleinste in sich geschlossene Programmstücke, die man getrennt entwickeln, verstehen und optimieren kann.

2. Die benötigten Hilfsvariablen und Zwischenrechnungen sind nach Abarbeitung des Blocks verschwunden.

3. Aus Blöcken lassen sich größere Programmeinheiten zusammenstellen, ohne dass Namenskonflikte auftreten.

4. Mit Blöcken kann man Einfluss auf den kellerartigen Speicherplatz, der keiner Speicherbereinigung unterliegt, nehmen und den Speicherplatz gezielt selbst verwalten.

Hinweis: Diese Vorteile lassen sich auch mit Prozeduren und Paketen erreichen; allerdings kann bei ihnen ein zusätzlicher Verwaltungsaufwand bei der Programmausführung entstehen.

4.1.6 Bindung von Objekten an Namen/Bezeichner.

Eine Deklaration bewirkt, dass mit dem deklarierten Objekt eine Bedeutung verbunden wird. Ist das Objekt eine Variable, so wird Speicherplatz im lokalen Kellerspeicher reserviert und die Anfangsadresse dieses Speicherbereichs (= interner Name des Objekts) wird dem Bezeichner (= Name im Programm) des Objekts fest zugeordnet. Ist das Objekt eine Datentypdefinition, so wird mit dem Namen die Struktur des Typs und die später für jede Komponente benötigte Zahl an Speicherplätzen verbunden. Ist es eine Prozedur, so wird ihm der Prozedurtext oder das übersetzte Programmstück zugeordnet usw.

Diesen Vorgang des Zuordnens nennt man "Bindung". Die Größe des benötigten Speicherplatzes ergibt sich bei Variablen aus der Definition des Datentyps, bei Prozeduren aus dem Hinweis auf das Ende einer Prozedur usw.

Die Bindung zwischen dem Objekt und seinem Bezeichner besteht im gesamten Gültigkeitsbereich; dieser unterliegt der Blockstruktur.

Außerhalb des Gültigkeitsbereichs kann an den Bezeichner ein anderer interner Name bzw. eine andere Information gebunden werden.

Zum Beispiel können Operatoren solche Objekte sein. Dann liegt durch die Bindung eine Kopplung zwischen dem Bezeichner und der Operation vor. Diese Zuordnung muss nicht eindeutig sein. Standardbeispiel ist die Addition:
In A + B bezeichnet der Operator "+" je nach Kontext eine ganzzahlige, eine reelle, eine Vektor- oder eine andere Operation.

Man spricht von statischer Bindung, wenn die Bindung in einer Deklaration in einem Deklarationsteil des Programms erfolgt, die bereits zur Übersetzungszeit vorgenommen werden kann.

Kann die Bindung erst zur Laufzeit durchgeführt werden (z.B. bei dynamischen Feldern, bei Haldenobjekten oder bei lokalen Variablen in rekursiven Prozeduren), so spricht man von dynamischer Bindung.

Die Bindung nennt man polymorph, wenn mit einem Bezeichner mehrere Objekte verbunden werden. (Beispiel: obiger Operator "+". Die eindeutige Zuordnung ist dann nur durch Betrachten des Kontextes möglich. Man sagt dann auch, der Bezeichner sei "überladen". Bei Schachtelungen und Rekursionen definiert jede Programmiersprache genau, welches Objekt zu jedem Zeitpunkt gemeint ist. Bei Vererbungen geht man die Vererbungshierarchie zurück und verwendet in dieser Reihenfolge das erste definierende Vorkommen des Bezeichners. In typisierten Sprachen können Variablen auch Objekte aus abgeleiteten Klassen zugewiesen werden. Usw.)

4.1.7 Überladen von Bezeichnern

Die Mehrfachdeklaration eines Bezeichners bezeichnet man als "Überladen" (overloading).

Ein Bezeichner heißt überladen, wenn er mehrere verschiedene Bedeutungen besitzt (d.h., ihm sind mindestens zwei verschiedene Objekte zuordnet). Allerdings muss an jeder Stelle des Programms aus dem Kontext eindeutig hervorgehen, welche Bedeutung hier gemeint ist.

In der Praxis muss festgelegt sein, ob die eindeutige Zuordnung des Bezeichners zu einer der Deklarationen bereits zur Übersetzungszeit (also "statisch") möglich sein muss oder ob dies erst zur Laufzeit ("dynamisch") erfolgen soll.

In Ada ist Überladen zulässig. Zum Zeitpunkt der Abarbeitung muss die Eindeutigkeit gegeben sein.
Ada unterscheidet zwischen Bezeichnern, die überladen werden können, und solchen, für die dies nicht erlaubt ist. Überladen werden dürfen in Ada Bezeichner für Literale in Aufzählungstypen (siehe 1.8.1), Funktionen, Operatoren und Unterprogramme (siehe 1.7, Details siehe unten). Nicht überladen werden dürfen Bezeichner für Datenobjekte und implizite Angaben (Sprungziele, Namen von Blöcken usw.).
Auch wenn man Typen ableitet, erfolgt eine Überladung; z.B.:

```
type Euro is new Float delta 0.01 range 0.0 .. Float'Last;
type Cent is new Euro range 0.0 .. 0.99;
```

In diesem Fall sind die Literale für die Typen Float und Euro und ebenso die zulässigen Operatoren überladen, da diese für den abgeleiteten Typ (Euro bzw. Cent) übernommen werden.

Ein wichtiger Anwendungsfall ist in Ada das Überladen von Operatoren und Unterprogrammnamen. Das Überladen erfolgt oft indirekt, indem man entsprechende Unterprogramme mittels "with" aus anderen (Bibliotheks-) Programmeinheiten importiert.

Das Überladen eines Operators, eines Funktions- oder eines Unterprogramm-Namens ist in Ada erlaubt, sofern sich die diversen Deklarationen
- in der Reihenfolge der Parametertypen,
- in mindestens einem Parametertyp oder
- im Ergebnistyp

unterscheiden. Dann kann man den passenden Operator bzw. Unterprogramm-Namen eindeutig auffinden.

Beispiel: Erlaubt sind im gleichen Deklarationsteil:

```
function "+" (X, Y: Vektor) return Vektor is
Summe: Vektor;
begin   for J in Y'Range loop Summe (J) := X(J) + Y(J); end loop;
        return Summe;
end "+";
function "+" (X: Float; Y: Vektor) return Vektor is
Summe: Vektor;
begin   for J in Y'Range loop Summe (J) := X + Y(J); end loop;
        return Summe;
end "+";
function "+" (X, Y: Vektor) return Float is
Summe: Float := 0.0;
begin   for J in Y'Range
                loop  Summe := Summe + X(J) + Y(J); end loop;
        return Summe;
end "+";
```


Es seien:
A, B, C: Float; D, E, F: Vektor;
Mit den obigen drei Operatoren sind im weiteren Verlauf die folgenden vier Zuweisungen erlaubt, die fünfte dagegen nicht.

```
C := A+B;           -- vordefinierte Float-Addition
A := (D+E) + A;     -- links: dritter Operator und dann
                    -- rechtes "+" = Float-Addition
F := A+D;           -- zweiter Operator
E := F+D;           -- erster Operator
D := (E+B) + F;     -- nicht definiert, da "Vektor + Float"
                    -- nicht deklariert wurde.

D := (B + E) + F;   -- Dies ist dagegen zulässig: erst den
                    -- zweiten, dann den ersten Operator.
```

Beispiel: Das Skalarprodukt schreibt man meist als

```
function "*" (X, Y: Vektor) return Float is
SP: Float := 0.0;
begin   for J in X'Range loop
              SP := SP + X(J) * Y(J);  end loop;
        return SP;
end "*";
```

Beachten Sie: Die Operatorsymbole bzw. Unterprogramm-Namen unterliegen der bereits festgelegten Lebensdauer und der Sichtbarkeit. Siehe nächste Folie; dort sind die beiden Deklarationen für "*" in den geschachtelten Blöcken möglich, innerhalb des gleichen Deklarationsteil aber nicht erlaubt, weil man sie durch den Kontext (z.B. die Typen der Parameter) nicht unterscheiden könnte.

äußerer Block

```
declare  Z: Float; D, E: Vektor;
function "*" (X, Y: Vektor) return Float is
SP: Float := 0.0 ;
begin  for J in Y'Range loop  SP := SP + X(J) * Y(J);  end loop ;
       return SP ;
end "*"; ...
begin ...
    Z := D*E;   ...
    declare                                          innerer Block
    function "*" (X, Y: Vektor) return Float is
    S: Float := 0.0 ;
    begin for J in Y'Range loop  S := S + X(J) + Y(J);  end loop ;
        return S ;
    end "*"; ...
    begin  ...
        Z := D*E;  ...
    end; ...
    ...
end;
```

Im inneren Block wird der Operator "*" des äußeren Blocks umdefiniert.

Spezialfall: Zusammengefasste Daten (Aggregate). In Ada können auch Aggregate überladen werden; im konkreten Fall muss der Programmierer selbst für die Eindeutigkeit sorgen.

```
type Vektor1 is array(1..4) of Integer;
type Vektor2 is array ('A'..'D') of Integer;
procedure P (X: Vektor1) is ...
procedure P (Y: Vektor2) is ...
...
P((1,2,3,4)); ...
```

Hier ist die Bezeichnung des Feld-Aggregats (1,2,3,4) "überladen" und im vorliegenden Fall kann die "richtige" Prozedur P nicht gefunden werden. Dann muss man wie üblich eine Typqualifikation durchführen, z.B.:
P(Vektor2'(1,2,3,4));

Warnung: Das Überladen kann zu schwer erkennbaren Fehlern führen, insbesondere wenn hierbei ein Operator oder ein Unterprogrammname in geschachtelten Blöcken mehrfach umdefiniert wird.

Man gebe sich in solchen Fällen einige Regeln, etwa:

- Einheitliche Stelligkeit von Operatorsymbolen, z.B.: Deklariere ein Operatorsymbol, das üblicherweise zwei Operanden hat, nicht zu einem dreistelligen Operator um.
- Benenne die einzelnen Operatordeklarationen im Kommentarteil eindeutig und notiere im Kommentarteil hinter den Zuweisungen, welcher Operator hier gemeint ist.

4.1.8 Ausnahmen (exceptions): In der Syntaxdefinition von Blöcken tritt *in Ada* weiterhin die Ausnahmebehandlung (exception_handler) auf. Wir betrachten auch diese genauer.

Die zugehörigen Deklarationen der Namen "exception_name" erfolgt in der "exception_declaration" im Deklarationsteil des Blocks:

exception_declaration ::=
    defining_identifier_list : exception ;

defining_identifier_list ::=
    defining_identifier { , defining_identifier}

defining_identifier ::= identifier

exception_handler ::=
    when [choice_parameter_specification :]
    exception_choice {| exception_choice} =>
    sequence_of_statements

choice_parameter_specification ::= defining_identifier

exception_choice ::= *exception*_name | others

Die Ausnahmebehandlung am Ende eines Blocks hat also in der Regel die Form

```
exception
   when <Name des Fehlers>  =>  <Anweisungsfolge>
   when <Name des Fehlers>  =>  <Anweisungsfolge>
   .....
   when <Name des Fehlers>  =>  <Anweisungsfolge>
   when others  =>  <Anweisungsfolge>
```


Wer Programmeinheiten definiert, legt dort meist auch die Fehlermöglichkeiten fest. Diese werden mittels
raise < Ausnahme-Bezeichner> aufgerufen ("erweckt").

Wenn zum Beispiel von einem leeren Keller gelesen werden soll (Top(Empty) ist undefiniert), so kann man dies abfangen und bei der Definition von "Top" eine Ausnahme

```
    Lesefehler: exception
```

deklarieren und diese dann mittels "raise"

```
    if  Isempty(...)  then  raise Lesefehler;
    else  <"arbeite mit Top(Keller) weiter">  end if;
```

auslösen. Dort, wo fehlerhafte Kellerzugriffe erfolgen können, legt man dann eine Ausnahmebehandlung der Form

```
    exception
    when Lesefehler =>  Put("Zugriff auf leeren Keller");
                        < weitere Anweisungen >;
    when others => Put("Unbekannter Fehler"); ...
```

fest, um die Fehler *kontrolliert* abzufangen.

Beispiel

```
type Vektor is array (Integer range <>) of Float;
Bereichsunterschied: exception;
function Skalarprodukt (X, Y: Vektor) return Float is
SP: Float := 0.0;
begin
   if (X'First /= Y'First) or (X'Last /= Y'Last)
      then  raise Bereichsunterschied;
   else
      for J in X'Range loop
         SP := SP + X(J) * Y(J);  end loop;
      return SP;
   end if;
end Skalarprodukt;
```

Einfügen in einen größeren Zusammenhang:

```
Fehler: Boolean := False;  ...   -- irgendwo in einem Oberblock
type Vektor is array (Integer range <>) of Float;
Bereichsunterschied: exception; ...
function Skalarprodukt (X, Y: Vektor) return Float is
begin ... raise Bereichsunterschied; ... end Skalarprodukt; ...
begin ...; S := Skalarprodukt; ...
   < in diesem Block möge die Ausnahmebehandlung stehen >
   exception
   when Bereichsunterschied => S := 0;
        Put("Fehler bei ... "); Fehler := True;
end; (A)
```

Wird in der Funktion Skalarprodukt die Ausnahme Bereichsunterschied erweckt, so wird die Funktion beendet, es werden die Anweisungen S:=0; Put("...") und Fehler:=True ausgeführt und danach wird das Programm mit der auf den Block unmittelbar folgenden Anweisung (A) fortgesetzt.

Zum präzisen Ablauf der Ausnahme-Bearbeitung in Ada:
raise bewirkt also einen Sprung zur Ausnahmebehandlung der aktuellen Programmeinheit. Befindet man sich bereits im Block zwischen begin ... end, so wird dort nach dem Ausnahmebehandler gesucht; befindet man sich noch im Deklarationsteil, so wird zum Ausnahmebehandler des umgebenden Blocks verzweigt. Liegt dort kein passender Ausnahmebehandler vor, so wird die Blockhierarchie nach oben hin nach einem passenden Ausnahmebehandler durchsucht. (Beachten Sie, dass "others" dann immer passt!).

Hierbei kann es sein, dass die aktuelle Programmeinheit verlassen wird, z.B. kann ein Funktionsaufruf beendet werden. Dann werden die notwendigen Verwaltungsmaßnahmen ebenfalls durchgeführt. Verlässt man z.B. eine rekursive Prozedur über ein raise, so werden alle Kopien (Inkarnationen) der Prozedur korrekt zu Ende geführt und insbesondere wird der dadurch belegte Platz im lokalen Kellerspeicher wieder frei gegeben. Befindet sich der Ausnahmebehandler in einer Funktion, so muss im Programm dafür gesorgt werden, dass die Funktion nach der Ausnahmebehandlung wieder korrekt über ein return beendet wird (und nicht über das Ende "end").

4.1.9: In Ada sind vier Standardfehler voreingestellt:

Constraint_Error: Überschreitung eines Bereichs

Program_Error: Nicht ausführbare Anweisung

Storage_Error: Verfügbarer Speicherplatz überschritten

Tasking_Error: Fehler bei nebenläufiger Verarbeitung

In der Regel verwendet man diese voreingestellten Fehler und gibt am Ende eines Blocks an, was beim Auftreten dieser Fehler geschehen soll. Formuliert man dagegen keine Ausnahmebehandlung, so bricht das Programm beim Auftreten eines dieser Fehler mit einer Fehlermeldung ab.

In den vielen Ada-Programmen finden sich daher Blöcke mit Ausnahmeregeln der folgenden Form:

```
declare ... ;
begin ...
......
exception
    when Constraint_Error  =>  X := 24; Y:= ... ;
    when Storage_Error  =>  ...;
    when others  =>  ...;
end;
```

4.1.10: In der Definition von Blöcken tritt der Deklarationsteil (declarative_part) auf. Wir geben hierzu die Syntax genau an:

declarative_part ::= {declarative_item}

declarative_item ::= basic_declarative_item | body

basic_declarative_item ::=
    basic_declaration | representation_clause | use_clause

basic_declaration ::=
    type_declaration | subtype_declaration |
    object_declaration | number_declaration |
    subprogram_declaration | abstract_subprogram_declaration |
    package_declaration | renaming_declaration |
    exception_declaration | generic_declaration |
    generic_instantiation

Fortsetzung zu "declarative_part":

object_declaration ::=
    defining_identifier_list : [aliased] [constant]
        subtype_indication [:= expression] ; |
    defining_identifier_list : [aliased] [constant]
        array_type_definition [:= expression] ; |
    single_task_declaration |
    single_protected_declaration

number_declaration ::=
    defining_identifier_list : constant := static_expression;

Fortsetzung zu "declarative_part": (zur Kenntnisnahme)

representation_clause ::= attribute_definition_clause |
    enumeration_representation_clause |
    record_representation_clause | at_clause

use_clause ::= use_package_clause | use_type_clause

body ::= proper_body | body_stub

proper_body ::= subprogram_body | package_body |
    task_body | protected_body

body_stub ::= subprogram_body_stub | package_body_stub |
    task_body_stub | protected_body_stub

subprogram_body_stub ::=
    subprogram_specification is separate ;

Beachten Sie: Wir orientieren uns in diesem Abschnitt 4.2 nicht an Ada.

4.2 Prozeduren und Funktionen

Ein Programmstück kann zu einer Einheit zusammengefasst, mit Parametern versehen und an einer beliebigen Stelle, an der der zugehörige Bezeichner sichtbar ist, aufgerufen werden. Einen solchen in sich abgeschlossenen Algorithmus nennt man "Unterprogramm" oder "Prozedur"; wird der Algorithmus nur verwendet, um Werte zu berechnen, die in einen Ausdruck eingesetzt werden, so spricht man von einer Funktion.

Was ist nun die genaue Bedeutung eines Prozeduraufrufs? Im Prinzip muss man die formalen Parameter durch aktuelle Größen ersetzen und dann den Prozedurrumpf ausführen. Hier gibt es mehrere "Parameter-Übergabemechnismen", die wir im Folgenden erläutern.

4.2.1 Prozeduren: Eine Folge von Deklarationen und Anweisungen kann zu einer Programmeinheit, genannt "Prozedur" oder "Unterprogramm" (engl.: procedure, subprogram, subroutine) unter einem Namen einschließlich der formalen Parameter zusammenfassen. Diesen Namen mit aktuellen Parametern kann man dann wie eine (elementare) Anweisung im Sichtbarkeitsbereich des Namens benutzen (Prozeduraufruf, "call").

Dadurch wird jedes Verfahren zu einer elementaren Handlung in anderen Verfahren! (Hierarchiebildung, Black-Box-Methode)

Die Prozedurdeklaration beginnt mit der Spezifikation (Name der Prozedur und formale Parameter mit ihren Typen). Danach folgt der Rumpf bestehend aus dem Deklarationsteil und den Anweisungen.

Viele Beispiele siehe früher, etwa in 1.7, 3.5.2, 3.5.4, 3.8.7.

Wir haben bisher auch Prozeduren verwendet, die keine Parameter besitzen, sondern die nur mit globalen Variablen arbeiten. Dies sollte man in der Praxis vermeiden, da hierbei undurchschaubare "Seiteneffekte" auftreten können, die oft zu schwer zu entdeckenden Fehlern führen.
Ein Beispiel, wo solche globalen Variablen von einer Prozedur oder einer Funktion verändert werden, ist das "Zählen, wie oft eine Prozedur oder Funktion aufgerufen wurde":

```
Z: Integer := 0;    -- zum ggT vergleiche 1.6, 1.7.1, 1.7.3, 2.4.2.
function ggT(A, B: Natural) return Natural is
   R: Natural;  S: Natural := A;  T: Natural := B;
begin  while T /= 0 do R := S mod T; S := T; T := R od;
       Z := Z+1;
       return S;
end;
```

"Z := Z+1" bewirkt hier den Seiteneffekt, dass eine Größe verändert wird, die nicht bei den Parametern aufgelistet ist!

4.2.2 Rekursion

Im Inneren einer Funktion ist der Name der Funktion sichtbar. Man kann daher dort auch die Funktion selbst wiederum verwenden. Die (direkte oder indirekte) Verwendung einer Funktion in ihrem eigenen Rumpf nennt man Rekursion. Das gleiche gilt für Prozeduren.

Standardbeispiel Fakultätsfunktion

$$n! = \begin{cases} 1 & \text{für } n = 0; \\ n \cdot (n-1)! & \text{für } n > 0 \end{cases}$$

```
function fak(n: Natural) return Natural is
begin  if n=0 then return 1 else return n*fak(n-1) fi  end fak
```


Standardbeispiel Spiegeln eines Textes (vgl. 3.5.4)

```
procedure Spiegeln is

A: Character;

begin
    if not End_of_File then Get(A); Spiegeln; Put(A) fi
end;
```

Machen Sie sich klar, wie die Eingabefolge durch die Rekursion umgedreht wird.

Weitere rekursive Algorithmen:

ggT, Ulam-Collatz-Funktion (7.5.1), Gerade-Ungerade (1.7)

Lösung zur gerechten Erbschaft in 1.15, Aufgabe 15

Quicksort in 7.3.3 (in 3.7.7 angedeutet)

Hinweis: Jede while-Schleife lässt sich auch als rekursive Prozedur schreiben:

while B do A od ist gleichbedeutend mit dem Aufruf W, wobei die rekursive Prozedur W deklariert ist als:

```
procedure W is  begin  if B then A; W fi  end;
```

Weil sich W immer nur am Ende seiner Anweisungen rekursiv aufruft, nennt man diese Form auch "tail recursion".

4.2.3 Drei übliche Formen für die Parameterübergabe.

Wir betrachten eine leicht abgeänderte Funktion für den ggT:

```
function  euklid1(A, B: value Natural) return Natural is
declare H: Natural;
begin
  if  (A > 0) or (B > 0)  then
     if A < B then  H := A;  A := B;  B := H  fi;
     while B ≠ 0 do H := A mod B;  A := B;  B:= H  od  fi;
  return A
end
```

Die mit "value" versehenen formalen Parameter A und B werden als lokale Variablen der Funktion aufgefasst, denen beim Aufruf die Werte der aktuellen Parameter zugewiesen werden; sie dürfen im Rumpf der Funktion neue Werte erhalten. Am Ende werden aber die Werte von A und B *nicht* an die Aufrufstelle übermittelt. Solche Parameter heißen allgemein "call by value"-Parameter.

Statt Werte zu übergeben, kann man auch mit den Variablen des aufrufenden Programms arbeiten. Dies lässt sich elegant mit Variablen einer höheren Referenzstufe realisieren.

```
function  euklid1(A, B: access Natural) return Natural is
declare H: Natural;
begin
  if  (A > 0) or (B > 0)  then
     if A < B then  H := A;  A := B;  B := H  fi;
     while B ≠ 0 do  H := A mod B;  A := B;  B:= H  od  fi;
  return A
end
```

Die mit access versehenen formalen Parameter A und B sind nun Zeiger auf Variablen vom Typ Natural und erhalten die Adresse des jeweiligen aktuellen Parameters (der in diesem Fall natürlich eine Variable sein muss) zugewiesen. Beachte: In der Funktion ist A dann immer als "deref A" zu lesen usw.

Der Aufruf Z := euklid1(X,Y); bewirkt also, dass an der Aufrufstelle folgendes Programmstück ausgeführt wird:

```
declare A, B: access Natural;
begin
    A := X;  B := Y;     -- A und B erhalten die Adressen von X und Y
    declare H: Natural;
    begin
        if  (deref A > 0) or (deref B > 0)  then
            if  deref A < deref B  then  H := deref A;
                                 deref A := deref B;  deref B := H  fi;
            while (deref B) ≠ 0 do  H := (deref A) mod (deref B);
                                 deref A := deref B;  deref B:= H  od   fi;
        Z := deref A
    end
end
```

Dieser Aufruf übergibt Zeiger (Adressen) an die formalen Parameter. Die formalen Parameter sind nun lokale Variable vom Typ access ... . Die Funktion arbeitet über diese Zeiger mit den Variablen des aufrufenden Programmstücks!

Solche Parameter heißen "call by reference"-Parameter.

Man beachte, dass diese Zeiger auch in Strukturen hineinführen können. Z.B. kann man auch Z := euklid1(D(1), D(2)); schreiben, sofern D vom Typ array (1..N) of Natural ist. Dann werden die (Adress-) Zuweisungen A := D(1); B := D(2); ausgeführt. Das Gleiche gilt, wenn die aktuellen Parameter Komponenten eines Records sind.

Als dritte Möglichkeit betrachten wir, dass die formalen Parameter A und B textuell durch die aktuellen Parameter X und Y ersetzt werden ("call by name"-Parameter).


```
function euklid1(A, B: name Natural) return Natural is
declare H: Natural;
begin
  if (A > 0) or (B > 0) then
     if A < B then H := A; A := B; B := H fi;
     while B ≠ 0 do H := A mod B; A := B; B:= H od fi;
  return A
end
```

Der Aufruf Z := euklid1(X,Y); bewirkt dann an der Aufrufstelle die Ausführung des Programmstücks

```
declare H: Natural;
begin
  if (X > 0) or (Y > 0) then
     if X < Y then H := X; X := Y; Y := H fi;
     while Y ≠ 0 do H := X mod Y; X := Y; Y:= H od fi;
  Z := X
end
```


4.2.4 Bedeutung eines Funktionsaufrufs $f(\alpha_1, ..., \alpha_k)$

Wenn f eine (zuvor deklarierte) Funktion mit k formalen Parametern ist und f in der Wertzuweisung für eine Variable X

X := ... $f(\alpha_1, ..., \alpha_k)$ ...

steht, so wird die Berechnung des Ausdrucks "... $f(\alpha_1,...,\alpha_k)$ ..." unterbrochen, sobald man auf den Operanden f stößt. Zunächst wird geprüft, ob f hier ein sichtbarer Name ist, dann werden die Ausdrücke $\alpha_1, ..., \alpha_k$ (dies sind die k aktuellen Parameter) in irgendeiner Reihenfolge ausgewertet, diese Werte werden den zugehörigen formalen Parametern von f zugewiesen und der Funktionsrumpf von f wird hiermit ausgerechnet, wobei man ein Resultat b erhält. Wenn b den Ergebnistyp der Funktion besitzt, so wird $f(\alpha_1, ..., \alpha_k)$ durch diesen Wert ersetzt und der Ausdruck "... $f(\alpha_1, ..., \alpha_k)$ ..." wird weiter ausgerechnet.

4.2.5 Parameterübergaben für Prozeduren

Die Zuordnung der aktuellen Parameter an die formalen Parameter beim Prozeduraufruf ("call") bezeichnet man als Parameterübergabe. Dieser ist bei Funktionen und Prozeduren im Wesentlichen gleich. Die drei wichtigsten Mechanismen sind:

call by value: Nur die Werte werden übergeben; die formalen Parameter sind lokale Variablen der Prozedur.

call by reference: Ein Verweis auf die aktuelle Variable wird übergeben; die formalen Parameter sind Zeigervariablen.

call by name: Der formale Parameter wird textuell durch den aktuellen Parameter ersetzt (wobei keine Namen im aktuellen Parameter hierdurch lokal werden dürfen - sonst umbenennen).

Generell gilt für jede Verwendung von Unterprogrammen:

Globale Variablen dürfen bei der Parameterübergabe und den anschließenden Ersetzungsmechanismen nicht zu lokalen Variablen werden!

Dann liegt nämlich fast immer ein Fehler vor.

Das System muss in solchen Fällen die lokalen Variablen vor der Ausführung des Prozeduraufrufs umbenennen und auf diese Weise den "Namenskonflikt" beseitigen.

Beispiele zum Namenskonflikt finden Sie in 4.2.7.

Bedeutung des Prozeduraufrufs exakt beschreiben.

Kopierregel 4.2.6:

Gegeben
procedure P(X_1: pü$_1$ T_1; X_2: pü$_2$ T_2; ...; X_n: pü$_n$ T_n) is PRUMPF;

Hierbei gibt pü$_i$ ∈ {value, access, name} die Art der Parameterübergabe für den i-ten formalen Parameter an. PRUMPF ist der Rumpf der Prozedur P (ein Block).

Ein Prozeduraufruf $P(\alpha_1, \alpha_2, ..., \alpha_n)$ wird dann wie folgt ausgeführt (der aktuelle Parameter α_i ist ein Ausdruck vom Datentyp T_i des zugehörigen formalen Parameters):

Der Prozeduraufruf $P(\alpha_1, \alpha_2, ..., \alpha_n)$ wird durch folgenden Block BP ersetzt:

```
BP:
declare  X1: Typ1;  X2: Typ2; ...;  Xn: Typn;
begin
X1 := α1; X2 := α2; ...; Xn := αn;
modifizierterPRUMPF
end;
```

Hierbei gelte:

$Typ_i = T_i$, falls pü$_i$ = value,
Typ_i = access T_i , falls pü$_i$ = access,
$X_i: Typ_i$; und $X_i := \alpha_i$; entfallen, falls pü$_i$ = name ist.

Die Anweisung modifizierterPRUMPF ist stets ein Block. Er entsteht aus PRUMPF, indem

- die call-by-reference-Parameter (access) "dereferenziert" werden,
- die Namenskonflikte, die durch globale Variable in den Prozedurrümpfen beim Kopieren entstehen würden, durch Umbenennung beseitigt werden,
- die Namenskonflikte, die bei der name-Übergabe durch die aktuellen Parameter entstehen würden, durch Umbenennung beseitigt und anschließend die formalen name-Parameter durch den Text des zugehörigen aktuellen Parameters ersetzt werden.

Genauer:

1. Jeder formale Parameter X_j mit $\underline{pü_j}$ = access wird durch $\underline{deref}$ X_j ersetzt ($\underline{deref}$ = folge einmal dem Verweis).

2. Jeder formale Parameter und jeder lokale Name in PRUMPF, der gleich einem Namen ist, der in irgendeinem aktuellen Parameter α_i mit $\underline{pü_i}$ = name auftritt, wird durchgehend mit einem neuen Namen bezeichnet und

3. danach werden alle X_i mit $\underline{pü_i}$ = name durch α_i (textuell) ersetzt.

(4. Auf die globalen Variablen der Prozedurrümpfe gehen wir nicht näher ein, siehe Compilerbauvorlesungen, vgl. 4.2.7.)

Dann wird dieser Block BP ausgeführt. Sobald BP beendet ist, wird er wieder durch den Prozeduraufruf $P(\alpha_1, \alpha_2, ..., \alpha_n)$ ersetzt und das Programm setzt die Berechnung mit der danach folgenden Anweisung fort.

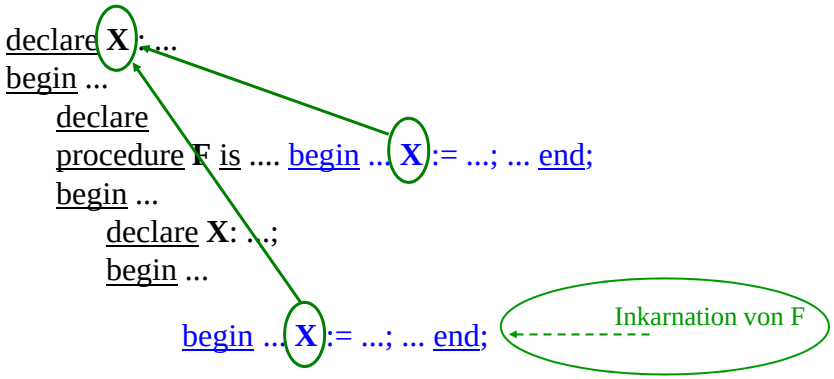
Diese (etwas modifizierte) Kopie des Prozedurrumpfs

BP:
declare X_1: Typ_1; X_2: Typ_2; ...; X_n: Typ_n;
begin
$X_1 := \alpha_1$; $X_2 := \alpha_2$; ...; $X_n := \alpha_n$;
modifizierterPRUMPF
end;

bezeichnet man als Inkarnation (oder modifizierte Kopie oder konkrete Ausprägung) der Prozedur.

4.2.7 Hinweis zum Sonderfall 4.: Globale Variable der Prozedur müssen "mitgenommen" werden. Beispiel:

```
declare X: ...
begin ...
    declare
    procedure F is .... begin ... X := ...; ... end;
    begin ...
        declare X: ...;
        begin ...
            F; ...
        end;
    ...
    end;                                   Betrachte nun
    ...                                    den Aufruf  F
end;
```

Bereits zugeordnete globale Variablen dürfen durch die Kopierregel nicht zu "lokaleren" Variablen werden !! Daher müssen Namen umbenannt werden. Wir formulieren dies hier nicht weiter aus.

Ende Beschreibung der Kopierregel ■

4.2.8 In der Vorlesung behandeln wir folgendes Beispiel:

```
procedure Test is
   I: Natural := 2;  A, H: Integer := 8;
   B: array (1..10) of Integer;
   procedure P(X: ••••• Integer) is
      H: Integer := 0;
   begin I := I+1; H := H+1; X := X+1; end;
begin  for J in 1..10 loop B(J) := J; end loop;
   P(A);
   P(H);
   P(B(I));
end;
```

Anstelle von ••••• füge "value", "access" oder "name" ein.

Welche Werte haben die Variablen jeweils am Ende der Prozedur-aufrufe? Machen Sie sich die Unterschiede zwischen den Parameter-übergaben genau klar, indem Sie die Ablaufprotokolle nachvollziehen.

4.2.9 Beispiel hier im Skript: Vertausche zwei Werte.
Wir beginnen mit: call by value.

```
program PP is
declare A, B, C, D, H: Float;

procedure vertausche (X: value Float; Y: value Float) is
declare C: Float;
begin C:=X; X:=Y; Y:=C end;

begin   ...
        ... vertausche(C, D); ...
end
```

Ersetze nun vertausche(C, D) wie oben angegeben.

vertausche(C, D); ⟷

```
declare X: Float; Y: Float;
begin
  X := C; Y := D;
  declare C: Float;
  begin C:=X; X:=Y; Y:=C end
end;
```

Ausgeführt wird also:

```
program PP is
...
begin     ...
      declare X: Float; Y: Float;
      begin  X := C; Y := D;
            declare C: Float; begin C:=X; X:=Y; Y:=C end
      end;  ...
end
```

Hier entsteht kein Namenskonflikt, weil sich die beiden Bezeichner C in verschiedenen Blöcken befinden.

Diese Ausführung bewirkt keine Veränderungen für die Variablen C und D. Zwar werden die Werte von C und D in den Variablen X und Y vertauscht, aber diese beiden Variablen sterben nach Abarbeitung des Blocks

```
declare X: ...
begin  ...  end;
```

und die Inhalte der im äußeren Block deklarierten Variablen C und D bleiben unverändert.

Es darf also keine call-by-value-Übergabe erfolgen, wenn man den Inhalt zweier Variablen auf diese Weise vertauschen will!

Wir probieren nun die call-by-reference-Übergabe aus.

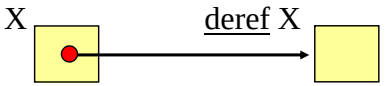
```
program PP is
declare A, B, C, D, H: Float;

procedure
vertausche (X: access Float; Y: access Float) is
declare C: Float;
begin C:=X; X:=Y; Y:=C end;

begin   ...
        ... vertausche(C,D); ...
end
```

Ersetze nun vertausche(C,D)
mittels call-by-reference.

vertausche(C,D); ⟷

```
declare  X: access Float;
         Y: access Float;
begin
   X := C; Y := D;
   declare C: Float;
   begin C:=deref X;
      deref X:=deref Y;
      deref Y:=C
   end
end;
```

deref ist ein Operator für Zeigertypen. deref X ist das Objekt, auf das X verweist:



Beim Prozeduraufruf vertausche(C, D) wird nun folgendes Programmstück ausgeführt:

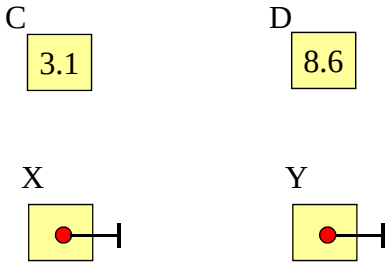
```
begin     ...
  declare X: access Float; Y: access Float;
  begin  X := C; Y := D;
    declare C: Float;
    begin C:=deref X; deref X:=deref Y; deref Y:=C end
  end;
```

Wir gehen dieses Programmstück schrittweise durch.

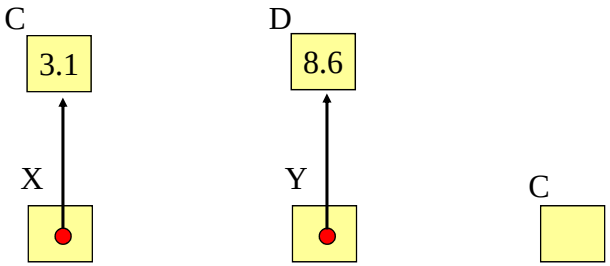
declare X: access Float; Y: access Float;

C habe den Wert 3.1
und D den Wert 8.6

→ "Programmzeiger"

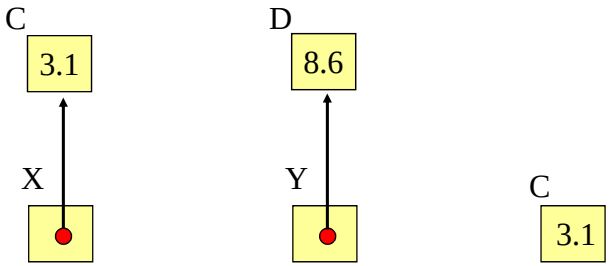


```
declare X: access Float; Y: access Float;
begin  X := C; Y := D;
   declare C: Float;
```

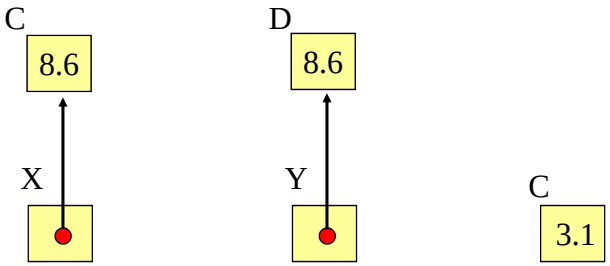


C ist lokale Variable des inneren Blocks.
Es entsteht hier kein Namenskonflikt.

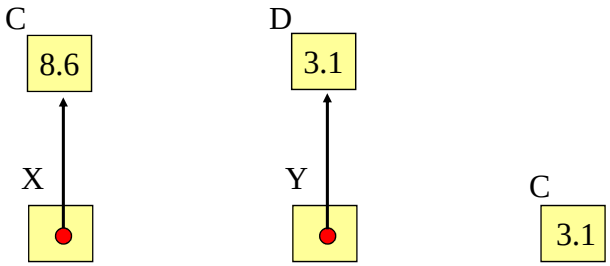

```
declare X: access Float; Y: access Float;
begin  X := C; Y := D;
   declare C: Float;
   begin C:=deref X; deref X:=deref Y; deref Y:=C end
```



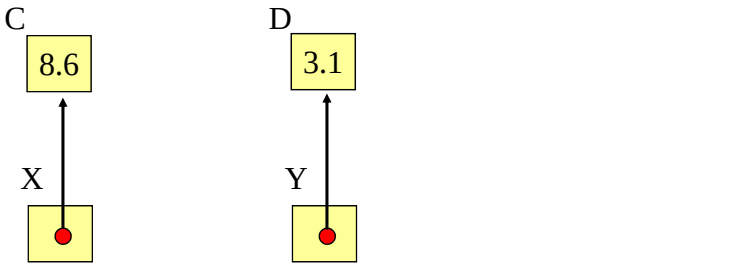

```
declare X: access Float; Y: access Float;
begin  X := C; Y := D;
   declare C: Float;
   begin C:=deref X; deref X:=deref Y; deref Y:=C end
```




```
declare X: access Float; Y: access Float;
begin  X := C; Y := D;
   declare C: Float;
   begin C:=deref X; deref X:=deref Y; deref Y:=C end
```



```
declare X: access Float; Y: access Float;
begin  X := C; Y := D;
    declare C: Float;
    begin C:=deref X; deref X:=deref Y; deref Y:=C end
end;
```




```
declare X: access Float; Y: access Float;
begin  X := C; Y := D;
    declare C: Float;
    begin C:=deref X; deref X:=deref Y; deref Y:=C end
end;
```

C hat nun den Wert 8.6
und D den Wert 3.1

call by reference leistet also das Gewünschte.

Wir untersuchen nun noch call by name.


```
program PP is
declare A, B, C, D, H: Float;

procedure
vertausche (X: name Float; Y: name Float) is
declare C: Float;
begin C:=X; X:=Y; Y:=C end;

begin   ...
        ... vertausche(C,D); ...
end
```

Ersetze vertausche(C,D) mittels call-by-name.

Aus `begin C:=X; X:=Y; Y:=C end;`
wird also `begin C:=C; C:=D; D:=C end;`

Hier liegt nun aber ein Namenskonflikt vor: Die global definierte Variable C würde durch die textuelle Ersetzung zur lokalen Variablen C:

```
declare C: Float; begin C:=C; C:=D; D:=C end;
```

Daher muss die innere Variable C umbenannt werden.
Aus `declare C: Float; begin C:=X; X:=Y; Y:=C end;`
wird also `declare C1: Float; begin C1:=X; X:=Y; Y:=C1 end;`

und erst danach erfolgt die textuelle Ersetzung, d.h.,
aus `declare C: Float; begin C:=X; X:=Y; Y:=C end;`
wird `declare C1: Float; begin C1:=C; C:=D; D:=C1 end;`

vertausche(C, D); ⟷

```
declare ;
begin
   declare C1: Float;
   begin C1:=C; C:=D; D:=C1 end
end;
```

Es ist klar, dass dieses Programmstück die korrekte Vertauschung durchführt.
call-by-name arbeitet in unserem Beispiel also auch richtig.
(Der Fall liegt anders, wenn C oder D indizierte Variablen sind, vgl. Beispiel 4.2.8.)

Ende des Beispiels ■

Die exakte Bedeutung des Prozeduraufrufs versteht man vor allem durch Beispiele, Beispiele, Beispiele. Rechnen Sie daher viele Beispiele durch. Konstruieren Sie unangenehme Fälle und rechnen Sie diese durch, indem Sie die Kopierregel anwenden.

Üblicherweise können alle Übergabemechanismen, die für Prozeduren gelten, auch in Funktionen benutzt werden, nur dass am Ende noch der berechnete Funktionswert in den Ausdruck, in dem der Funktionsaufruf steht, einzusetzen ist.

Einige Beispiele zum Üben finden Sie auf den folgenden Folien und in den Übungen. Wichtig ist aber, dass Sie (möglichst in kleinen Lerngruppen) "typische" Beispiele selber entwickeln und mit der Kopierregel nachvollziehen.

4.2.10 Übungsbeispiele

Übungsbeispiel Neu1:

```
program Neu1 is
declare X, Y: Natural;
function W(A: value Natural) return Natural is
begin
      if A<=1 then return 0  else return A + W(A-1)  fi
end W;
begin
  read (X, Y);
  for I :=1 to Y do X:=X+W(X) od;
  write (X)
end Neu1;
```

Berechnen Sie die Ausgabe zu den Eingaben 1, 2 per Hand bzw. mit dem Rechner für 6, 8 bzw. 9, 12 bzw. 10, 18.

Übungsbeispiel Neu2:

```
program Neu2 is
declare A, J, X: Integer;
function Q(A: value Integer) return Integer is
  begin  return A + X  end Q;
begin
  read (A,X);
  for J:=1 to A do X:=X+Q(A) od;
  write (X)
end Neu1;
```

Berechnen Sie die Ausgabe zu den Eingaben 1, 2 per Hand bzw. mit dem Rechner für 6, 8 bzw. 9, 12 bzw. 10, 18. Was erhält man, wenn man value durch access oder name ersetzt?

Übungsbeispiel Neu3:

```
program Neu3 is
declare A, J, X: Integer;
procedure R(A: value Integer; B: access Integer) is
declare X: Integer;
  begin  X := B+A; B := X+A; A := A+B  end R;
begin
  read (A,X);
  for J := 1 to A do R(X,A) od;
  write (A,X)
end Neu3;
```

Berechnen Sie die Ausgabe zu den Eingaben 1, 2 per Hand bzw. mit dem Rechner für 6, 8 bzw. 10, 18. Ersetzen Sie access durch name. Welche Ausgaben erhalten Sie dann?

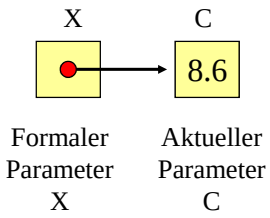
Übungsbeispiel Neu4:

```
program Neu4 is
type vektor is array (1..4) of Integer;
declare A, I, J: Integer; Y: Vektor := (1, 2, 3, 4);
procedure S(A: name Integer; B: access Vektor) is
  begin  J:=J+1;  A:=A+1;  B[A] := B[J] + A  end R;
begin  A:=2;  J:=1;  S(J,Y);  S(Y[J],Y);
  for I:= 1 to 4 do write (Y[I]) od
end Neu4;
```

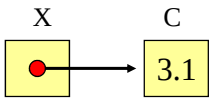
Berechnen Sie die Ausgabe zu den Eingaben 1, 2 per Hand bzw. mit dem Rechner für 6, 8 bzw. 10, 18. Ersetzen Sie access durch name. Welche Ausgaben erhalten Sie dann?

4.2.11 Alleinige Parameterübergabe value

Manche Sprachen erlauben nur call by value als Übergabemechanismus. Hiermit kann man ebenfalls Manipulationen auf den Variablen der aufrufenden Programmeinheit durchführen. Dazu übergibt man die Referenz (Adresse) einer Variablen als Wert; diese Adresse darf man dann zwar nicht mehr manipulieren, aber man kann die Objekte, auf die die Referenz verweist, abändern!



X.all := 3.1; ist nun erlaubt, da hierbei der Inhalt von X gleich bleibt! So wird der Inhalt von C tatsächlich verändert:



4.2.12 Parameterübergabe in Ada

In Ada ist das alles etwas anders!

Beachte zunächst: In Ada dürfen Spezifikation und Rumpf getrennt deklariert werden (allerdings steht die Spezifikation textuell vor dem Rumpf).

In Ada sind die drei Parameterübergaben in, out und in out vorgesehen.

Parameterübergabe "in":
Diese bedeutet, dass der formale Parameter wie eine Konstante behandelt wird: Er erhält den Wert des aktuellen Parameters und anschließend darf ihm im Rumpf kein Wert mehr zugewiesen werden. In Funktionen müssen alle formalen Parameter von dieser Form sein!

Parameterübergabe "out" (nur in Prozeduren erlaubt):
Diese bedeutet, dass der formale Parameter wie eine lokale Variable behandelt wird. Sobald die Prozedur beendet wird, wird der Wert dieser Variablen dem aktuellen Parameter zugewiesen. Der aktuelle Parameter muss folglich eine Variable sein (d.h., er muss eine Referenzstufe > 0 besitzen).

Parameterübergabe "in out" (nur in Prozeduren erlaubt):
Dieser formale Parameter wird wie ein out-Parameter als lokale Variable behandelt, deren Wert nach Beendigung des Prozeduraufrufs dem aktuellen Parameter zugewiesen wird. (Der aktuelle Parameter muss folglich ebenfalls eine Variable sein.) Zusätzlich wird zu Beginn des Prozeduraufrufs dem formalen Paranmeter der Wert des aktuellen Parameters zugewiesen.

Diese Festlegung führt jedoch dazu, dass alle aktuellen Parameter, die zu out- oder in-out-Parametern gehören, in kopierter Form vorliegen müssen, deren Werte erst am Ende der Prozedur an die aktuellen Parameter übergeben werden. Dies ist für große Datenstrukturen (z.B. Felder) viel zu aufwendig. Daher realisiert man diese Parameterübergabe meist als "call by reference", d.h., man legt Referenzvariablen an und arbeitet faktisch auf den aktuellen Parametern.

Ada legt hierbei nicht fest, ob auf den Originalen oder auf Kopien gearbeitet wird! Der Benutzer muss also so programmieren, dass beide Realisierungen zum gleichen Ergebnis führen. (In der Regel kann man davon ausgehen, dass der Ada-Compiler bei formalen Parametern, die einen skalaren Typ haben, eine lokale Variable anlegt, bei zusammengesetzten Datentypen dagegen eine lokale Variable vom access-Typ deklariert und somit ein call-by-reference implementiert wird. Aber: Man kann nicht sicher sein!)

4.3 Moduln

Ein Modul besitzt einen Zustand und Verarbeitungsvorschriften. Er ist eine in sich abgeschlossene Programmeinheit mit klar definierten Schnittstellen nach außen, die aus Konstanten, Variablen, Datentypen und Algorithmen besteht.
Moduln bilden die Bausteine, aus denen große Softwaresysteme zusammengesetzt werden. Man spricht vom "modularen" Aufbau solcher Systeme. Die Funktionalitäten der Moduln und ihre strukturelle Anordnung zusammen mit ihrem Zusammenwirken wird als Software-Architektur bezeichnet.

Hinweis zur Aussprache: Wir sagen hier "der Módul" mit Betonung auf der ersten Silbe und dem Plural "die Móduln".

Unter "das Modúl" mit Betonung auf der zweiten Silbe und dem Plural "die Modúle" verstehen wir (leicht austauschbare) physikalische Einheiten, insbesondere Hardwarebausteine.

Englisch: module.

4.3.1: Ein Modul sollte folgende Eigenschaften besitzen:

- Er bildet eine in sich abgeschlossene Einheit, die eine klar umrissene Aufgabe bearbeitet.
- Er hat eine genau definierte Schnittstelle nach außen (genannt "Spezifikation" oder "Interface"); nur die hier genannten Eigenschaften und Fähigkeiten sind nach außen hin sichtbar ("visibility"). Dies ist eine Art Benutzungsanweisung für alle, die diesen Modul einsetzen wollen.
- Seine interne Arbeitsweise oder Realisierung (Implementation) ist außen nicht bekannt. Er besitzt somit zwei Sichten (views): Die Außenansicht für den Benutzer ("partial view", dies ist vor allem die Schnittstelle) und die Innensicht des Erstellers ("full view"). Diese Innensicht bleibt "gekapselt" oder nach außen "versteckt" (Prinzip des "information hiding").
- Er ist überschaubar, gut zu testen und einfach zu warten.
- Er lässt sich in Bibliotheken aufbewahren und hierdurch leicht in beliebige Programmsysteme einbauen.

Moduln sind somit die einfachste Art, ein "Client-Server"-Modell zu realisieren:

Hierbei erstellt ein "Server" (Dienstleister, Hersteller) einen "Dienst", den er als Angebot (= Schnittstelle) nach außen bekannt macht. Wie er diese Dienstleistung tatsächlich realisiert, wird dabei nicht bekannt gegeben.

Ein "Client" (Kunde, Benutzer) benötigt eine Dienstleistung und verlässt sich darauf, dass die im Angebot eines Servers genannten Eigenschaften auch zutreffen.

Um dieses Modell in die Praxis umzusetzen, benötigt man ein Netz (z.B. das Internet), in dem ein Kunde die Angebote mit Hilfe von Suchmaschinen auffinden und sich mit Browsern anzeigen lassen kann. Der Kunde "browst" und "surft" im Internet. Der Anbieter macht durch Werbung und geeignete Suchbegriffe auf sich aufmerksam.

4.3.2: Schematischer Aufbau eines Moduls

module <Name des Moduls> is

[with ...; use ...]	Angaben, welche anderen Einheiten/Moduln verwendet werden und in welcher Weise
specification ...	Angabe der nach außen sichtbaren Datentypen, Konstanten, Variablen und "Methoden" (also Funktionen, Operatoren, Unterprogramme, Unter-Module usw.)
[implementation ...]	Weitere (nach außen nicht sichtbare) Deklarationen sowie Programme zur Implementierung der Methoden und Typen
[begin ... end]	Anweisungen zur Initialisierung, einschl. der Ausnahmebehandlungen

end module [<Name des Moduls>]

An diesem Schema erkennt man bereits:

Moduln sind die programmiersprachliche Realisierung von Datentypen. (Ein Datentyp kann riesig sein!)

Die "üblichen Moduln" realisieren hierbei konkrete Datentypen, die komplett definiert und in der Sprache ausformuliert werden.

Man kann die Moduln aber auch mit Parametern versehen und hiermit einen abstrakten Datentyp beschreiben, der durch die Angabe von Typen, Prozeduren und Daten und durch den Implementierungsteil zu einem konkreten Datentyp wird.

Gegenüberstellung mit bisherigen Datentypen (vgl. auch Kap. 5 später):

module	structure
with ... use ...	Liste der Parameter
specification	modes ... functions ....
implementation	(Erfüllung der Gesetzmäßigkeiten, laws ...)
begin ... end	(Initialisierung, Ausnahmebehandlung)
end module	end structure

4.3.3: Standardbeispiel "Stack" (für Zeichen)

Erinnerung an Abschnitt 3.5.4: Eine lineare Liste heißt Keller oder Stapel (engl.: stack oder pushdown), wenn auf ihr genau die folgenden fünf Operationen zugelassen sind:

(1) "Empty" = Leeren der Liste.
(2) "Isempty" = Abfragen auf Leerheit der Liste.
(3) "Top" = Kopieren des letzten Elements der Liste.
(4) "Push" = Hinzufügen eines Elements am Ende der Liste.
(5) "Pop" = Löschen des letzten Elements der Liste.

Hierin ist für einen "Kunden" bereits zu viel Information enthalten, insbesondere muss sich der Ersteller nicht auf den Begriff "Liste" festlegen. Auch braucht ein Kunde in der Regel nicht die Initialisierung "Empty". Wir bieten daher an:

Angebot: Datenstruktur "Stack für Zeichen"

Fähigkeiten (Leistungsumfang) dieses Angebots:

Datentyp StackZ;
Push (Zeichen) aktualisiert StackZ;
Pop aktualisiert StackZ;
Top liefert als Ergebnis ein Zeichen;
Isempty liefert als Ergebnis Boolean;

→ specification

Umgangssprachliche Erläuterungen ("Pflichtenheft"):
Push fügt ein Zeichen an den Stack an,
Pop entfernt das zuletzt eingefügte Zeichen,
Top zeigt das zuletzt eingefügte Zeichen an,
Isempty prüft, ob kein Zeichen im Stack ist.

→ im Kommentarteil von specification

Wie sieht dies im Internet-Browser aus?

Die Datenstruktur AG

- Die Firma
- Weitere Produkte
- Einsatzgebiete
- StackZ-Bereiche
- Leistungsumfang
- Impressum
- Kontakt

StackZ

Mitgeliefert werden
Push, **Pop**, **Top** und **Isempty**!
Auf Wunsch **Empty**

in den Einkaufswagen

Werden Sie bevorzugter Kunde bei uns! Betreuung, Rabatte, Sonderangebote

Was reingeht, kommt auch wieder raus!

Jetzt bestellen später zahlen

Fügen Sie mit *Push* ein Zeichen an!
Pop entfernt das letzte Zeichen!
Einmalig: *Sie können das letzte Zeichen sehen oder sogar feststellen, dass nichts mehr da ist!*

Preise, Wartung und Nachlieferungen hier

Neue Einsatzmöglichkeiten des StackZ hier

Lizenz- und Lieferbedingungen hier

Nur zufriedene Kunden! Referenzen hier.

Bestens bewährt, basiert auf dem LIFO-Prinzip.
Von Fachleuten empfohlen!

Einkauf fortsetzen

Kennen Sie schon QueueZ? hier ansehen

4.3.3.a: Formulierung als Modul (Ada ähnlich)

```
module  Stack_für_Zeichen  is     -- Der Stack soll "beschränkt" sein.
with IO; use IO;     -- IO sei ein Modul, der die Ein-/ Ausgabe von
                     -- Zeichen und Texten realisiert.
specification        -- Im Stack sind stets höchstens Max viele Elemente.
  type StZelle;
  type StackZ is access StZelle;
  type StZelle is  record
          inhalt: Character;
          next: StackZ
  end record;
  procedure Push (A: value Character);
  procedure Pop;
  function Top return Character;
  function Isempty return Boolean;
  Max: constant Natural := 2000;
```



```
implementation                                  -- Erste Version
  S: StackZ;  Unterlauf, Überlauf: exception;
  function Isempty return Boolean is
  begin return (S = nil) end;
  procedure Push (A: value Character) is
    p: StackZ;
  begin p := new StackZ; p.inhalt := A; p.next := S; S := p end;
  procedure Pop is
  begin  if S ≠ nil then S := S.next else raise Unterlauf fi  end;
  function Top return Character is
  begin if not Isempty then return S.inhalt
        else Put("Fehler! Der Stack enthält kein Element. ");
             Put("Es wird ' ' zurückgegeben."); return ' '  fi  end;
  procedure Empty is begin S := nil end;
```

Diese "implementation" kann nicht feststellen, ob der Stack mehr als "Max" Elemente enthält. Wir müssen also eine andere Implementierung wählen, die sich die Anzahl der Elemente, die aktuell im Stack sind, in einer Variablen "Anzahl" merkt. Diese Variable wird bei den Operationen Push und Pop verändert.

```
implementation                                  -- Zweite Version

   S: StackZ;  Unterlauf, Überlauf: exception;  Anzahl: Natural;

   function Isempty return Boolean is
   begin return (S = nil) end;                  -- auch (Anzahl = 0) ist korrekt

   procedure Push (A: value Character) is
     p: StackZ;
   begin if Anzahl < Max then
                p := new StackZ; p.inhalt := A;
                p.next := S; S := p; Anzahl := Anzahl + 1
           else raise Überlauf fi
    end;
```



```
   procedure Pop is
   begin if Anzahl = 0 then
                raise Unterlauf
           else S := S.next;
                Anzahl := Anzahl -1
           fi
   end;

   function Top return Character is
   begin if not Isempty then return S.inhalt
           else Put("Lesefehler! Der Stack enthält kein Element.");
                raise Unterlauf
           fi
   end;

   procedure Empty is
   begin  S := nil;  Anzahl := 0  end;
```



```
   begin  Empty;
   exception
      when Unterlauf => Put ("Der Stack ist bereits leer."); ...
      when Überlauf => Put ("Der Stack ist voll und kann keine
           weiteren Zeichen mehr aufnehmen. Ihre Operation
           kann daher nicht durchgeführt werden. Falls Sie
           weitere Zeichen speichern möchten, müssen Sie
           zunächst Elemente aus dem Stack entfernen."); ...
   end
   end module Stack_für_Zeichen;
```

Hier wurde also vorausschauend die Ausnahme "Überlauf" definiert, die eventuell mit dem Kunden noch gar nicht vereinbart wurde, die aber in der Programmierpaxis erforderlich ist, dann in das Pflichtenheft eingefügt werden muss und bei der Implementierung der Prozedur Push zu erwecken ist.

Dieser Modul lässt sich nun in einem Programm wie folgt verwenden:

```
declare
      X, Y: Character;
      module  Stack_für_Zeichen  is  ...  end module;
      ...
begin ...
      Stack_für_Zeichen.Push(X);
      ...
      if not Stack_für_Zeichen.Isempty
           then  Y := Stack_für_Zeichen.Top;
                 Stack_für_Zeichen.Pop  fi;
      ...
end;
```

4.3.3.b: Dieses Vorgehen eignet sich nicht für mehrere Stacks. In diesem Fall kann man den Datentyp StackZ vereinbaren und die Prozeduren parametrisieren.

```
module  Stacks_für_Zeichen  is      -- beachte den Plural "Stacks_..."
with IO; use IO;
specification                       -- hier lassen wir die Beschränktheit weg
   type StZelle;
   type StackZ is access StZelle;
   type StZelle is  record
          Inhalt: Character;
          Next: StackZ
   end record;
   procedure Push (A: value Character; S: access StackZ);
   procedure Pop (S: access StackZ);
   function Top (S: value StackZ) return Character;
   function Isempty (S: value StackZ) return Boolean;
```



```
implementation                  -- hier die "Methoden" programmieren
   function Isempty (S: access StackZ) return Boolean is
   begin return (S = nil) end;
   procedure Push (A: value Character; S: access StackZ) is
     p: StackZ;
   begin p := new StackZ; p.Inhalt := A; p.Next := S; S := p end;
   procedure Pop (S: access StackZ) is
   begin if S ≠ nil then S := S.Next
         else Put ("Stackunterlauf. "); raise Unterlauf fi  end;
   function Top (S: access StackZ) return Character is
   begin if S ≠ nil then return S.Inhalt
         else Put ("Lesefehler. "); raise Stack_Error fi  end;
   procedure Empty (S: access StackZ) is begin S := nil end;
begin  <Der Modul hat keine eigenen Daten, daher ist nichts zu initialisieren.
        Nur die Ausnahmebehandlungen kann man hier festzulegen ...>  end
end module Stacks_für_Zeichen;
```


Solch ein Modul lässt sich z. B. wie folgt verwenden:

```
declare
      X, Y: Character;
      module  Stacks_für_Zeichen  is  ...  end module;
      S1, S2, S3: StackZ;
      ...
begin ...
      Stacks_für_Zeichen.Push(X, S1);
      ...; S2 := S3; ...
      if not Stacks_für_Zeichen.Isempty(S2)
           then  Y := Stacks_für_Zeichen.Top(S2);
                 Stacks_für_Zeichen.Pop(S2)  fi;
      ...
end;
```

Hier ist der Datentyp StackZ bekannt, da die Modul-Deklaration abgearbeitet wurde.

Stacks_für_Zeichen.Empty(S1) ist nicht erlaubt, da die Prozedur Empty nicht in der Spezifikation aufgeführt ist.

In diesem Beispiel haben wir den Datentyp StackZ nach außen bekannt gemacht. Dies ist aber nicht nötig. Der Ersteller des Moduls könnte den Stack auch durch ein array (1..flex) of Character implementieren, wobei eine Indexvariable sich die Stelle merkt, wo das oberste (= zuletzt eingefügte) Zeichen steht.

Will man die Implementierung des Datentyps offen lassen, muss man seine Definition vor dem Benutzer verstecken. Man würde dann unter "specification" nur schreiben:

```
type StackZ
```

oder sogar diese Information weglassen und die Deklaration vollständig in den Implementierungsteil verschieben.

Datentypen, die man dem Benutzer zwar bekannt gibt, aber dessen genaue Definition man ihm nicht mitteilt, bezeichnet man als "private Typen" des Moduls, siehe 4.3.4.

with und use können in allen Deklarationsteilen stehen.

Erläuterung des Sprachelements with:
with M bedeutet, dass an dieser Stelle die Deklaration des Moduls M hineinkopiert wird. Alles, was in dessen Spezifikation steht, kann ab hier über die Punktnotation "M.xxx" benutzt werden.

Erläuterung des Sprachelements use:
use M bedeutet, dass ab dieser Stelle für einen Namen Funk, der in M vereinbart wird, die Punktnotation "M.Funk" durch Funk ersetzt werden kann, dass also die "Qualifizierung" (= der Zugriff auf die in M deklarierten Größen) ohne "M." erfolgen darf. Für Namenskonflikte ist der Programmierer selbst verantwortlich, wobei das Überladen erlaubt ist.

4.3.4 Moduln in Ada ("Pakete")

Das Schlüsselwort lautet in Ada package und man spricht von Paketen statt von Moduln.

Der Spezifikationsteil und der Implementierungsteil werden voneinander getrennt. Der Spezifikationsteil, der als Text früher im Programm als der Implementierungsteil stehen muss, hat die Form

package <Name des Pakets> is
<Folge von einfachen Deklarationen> end <Name des Pakets>;

Der Implementierungsteil heißt "body" und hat die Form

package body <Name des Pakets> is
<Folge von Deklarationen> end <Name des Pakets>;

Alle Teile der Spezifikation, die im Body programmiert werden, müssen wörtlich dort wieder vorkommen, s.u.

Besteht die Spezifikation nur aus Datentyp- und Konstanten-Deklarationen, so entfällt der Implementierungsteil.

Die Deklaration privater Typen erfolgt im Spezifikationsteil als "is private"; die Struktur wird am Ende der Spezifikation nach dem Schlüsselwort private vor dem Benutzer versteckt.

In Ada sind mit einem Datentyp (auch einem private-Datentyp) stets folgende Operationen verbunden:

- Gleichheit ("="),
- Ungleichheit ("/="),
- Zuweisung (":=").

Will man auch diese Operationen nicht für die Benutzer des Moduls zulassen, so muss man den Typ als limited private in der Spezifikation deklarieren. Auf solche Daten kann ausschließlich über die spezifizierten Operationen zugegriffen werden.

Das Standardbeispiel 4.3.3.b StackZ lässt sich leicht nach Ada übertragen (Stack_Error sei als exception hier sichtbar):

```
package  SfZ  is
with Ada.Text_IO; use Ada.Text_IO;
type StZelle;
type StackZ is access StZelle;
type StZelle is  record
                 Inhalt: Character;
                 Next: StackZ;
                 end record;
procedure Push (A: in Character; S: in out StackZ);
procedure Pop (S: in out StackZ);
function Top (S: in StackZ) return Character;
function Isempty (S: in StackZ) return Boolean;
end SfZ;
```

```
package body  SfZ  is
   procedure Push (A: in Character; S: in out StackZ) is
     p: StackZ;
   begin p := new StackZ; p.Inhalt := A; p.Next := S; S := p; end;
   procedure Pop (S: in out StackZ) is
   begin  if S /= null then S := S.Next;
      else Put ("Stackunterlauf. "); raise Stack_Error; end if;  end;
   function Top (S: in StackZ) return Character is
   begin  if S /= null then return S.Inhalt;
      else Put ("Lesefehler. "); raise Stack_Error; end if;  end;
   function Isempty (S: in StackZ) return Boolean is
   begin return (S = null); end;
end SfZ;
```


Dieser Modul lässt sich nun in einem Ada-Programm zum Beispiel wie folgt verwenden:

```
declare
        X: Character;
        package  SfZ  is  ...  end SfZ;
        S1, S2: StackZ;        -- S1 und S2 werden in Ada mit null ini-
                               -- tialisiert, da StackZ ein access-Typ ist
begin   ...
        SfZ.Push (X, S1);
        S2 := S1; ...
        if S1 = S2 then ... end if;        -- Gleichheit auf access-Typen!
        ...
        if not SfZ.Isempty (S2)
            then S1.Inhalt := SfZ.Top (S2); SfZ.Pop (S2); end if;
        ...
end;
```


Hinweis:

Die Prozedur

```
  procedure Push (A: in Character; S: in out StackZ) is
    p: StackZ;
  begin p := new StackZ; p.Inhalt := A; p.Next := S; S := p; end;
```

lässt sich in Ada kürzer mit Hilfe der Initialisierung beim Anlegen eines neuen StackZ-Elements schreiben:

```
  procedure Push (A: in Character; S: in out StackZ) is
  begin  S := new StackZ'(A,S);  end;
```

Erläuterung hierzu:
Die Initialisierung erfolgt durch '(...). Rechts vom Zuweisungszeichen wird die Next-Komponente des neuen Elements auf den alten Wert von S gesetzt, die Komponente "Inhalt" erhält den Wert von A und durch die Wertzuweisung wird dieses neue Element anschließend das oberste Stackelement.

Man kann die Definition des Datentyps StackZ verbergen:

```
package  SfZ  is
with Ada.Text_IO; use Ada.Text_IO;
type StackZ is private;
procedure Push (A: in Character; S: in out StackZ);
procedure Pop (S: in out StackZ);
function Top (S: in StackZ) return Character;
function Isempty (S: in StackZ) return Boolean;
private
    type StZelle;
    type StackZ is access StZelle;
    type StZelle is   record
                        Inhalt: Character;
                        Next: StackZ;
    end record;
end SfZ;
```

Man braucht den Implementierungsteil (package body) hierbei nicht zu ändern, auch das sonstige Programm nicht, sofern von dort nicht auf die einzelnen Komponenten von S1, S2, ... zugegriffen wird.

Nochmals der Hinweis auf "limited": Außerhalb des Implementierungsteils sind bisher Zugriffe wie S1.Inhalt oder S2.Next nicht zugelassen, allerdings sind S1 = S2 oder S1 := S2 noch erlaubt.

Schreibt man statt die Zeile
type StackZ is private;
type StackZ is limited private;
so sind außerhalb des Implementierungsteils auch keine Vergleiche auf Gleichheit und Ungleichheit sowie Zuweisungen an Variablen vom Typ StackZ zugelassen. Variablen des Datentyps können dann nur noch mit Hilfe der Operationen des Pakets manipuliert, gespeichert und ausgegeben werden.

Man kann nun leicht (und unbemerkt vom Benutzer) im package SfZ den private-Teil ersetzen durch:

```
private
   Max: constant Natural := 2000;
   type StackZ is record
      Inhalt: array (1..Max) of Character;
      Position: 0..Max;
   end record;
```

Dann muss man auch den Implementierungsteil geeignet ändern:

```
...
procedure Push (A: in Character; S: in out StackZ) is
begin if S.Position >= Max then raise Stack_Überlauf;
   else  S.Position := S.Position+1;
         S.Inhalt(S.Position) := A; end if;  end;
procedure Pop (S: in out StackZ) is ...;
...
```


4.3.5: Programmeinheiten können "Zustände" besitzen. *Bei Moduln* versteht man hierunter die aktuellen Werte aller Variablen des Moduls (also den aktuellen Speicherinhalt). *Bei Programmen* versteht man unter einem Zustand ebenfalls als den aktuellen Speicherinhalt, es kommen aber noch die Position, an der man sich im Programm befindet, und "Umgebungsinformationen" hinzu.

Vgl. 4.3.3.a: Der dortige Modul Stack_für_Zeichen besitzt Zustände (= Speicherinhalte der Variablen S).
Dagegen bietet der Modul Stacks_für_Zeichen in 4.3.3.b nur Datentypen und Methoden an; er hat also selbst keine Zustände.

Wie lange lebt ein Modul?

Unterprogramme unterliegen dem Keller-Prinzip und werden im lokalen Speicher des Programms nach dem LIFO-Prinzip verwaltet; verlässt man sie, so endet auch die Lebensdauer aller lokalen Variablen.

Dagegen bleiben die Inhalte der Variablen eines Moduls erhalten, wenn man ihn verlässt. Wenn der Modul zwischenzeitlich nicht stirbt (d. h., man bleibt im umfassenden Block) und wenn man den Modul später wieder verwendet, so besitzt er anfangs den gleichen Zustand, in dem er das letzte Mal verlassen worden ist.

Moduln geben ihren Zustand und ihre Arbeitsweise nach außen nur zum Teil bekannt (Prinzip des "information hiding"). Meist erfährt man den gesamten Zustand nicht oder kann ihn nur über die bereit gestellten Operationen feststellen.

4.4 Polymorphie

Wie entwirft man Systeme? Zunächst erstellt man gewisse Wünsche/Forderungen und schreibt auf, welche Funktionen das System erfüllen soll.

Dann beginnt man mit Plänen, Konzepten, Strukturen und Konstruktionen, wobei man schrittweise das bereits Vorhandene und das bisher Erstellte weiterentwickelt. Hierbei sollte man sich so spät wie möglich konkret festlegen. Dadurch bleibt der Entwurf flexibel und das System kann wesentlich leichter an künftige Änderungen und an die wechselnden Wünsche der Kunden angepasst werden.

Entscheidungen werden umrissen und eingeschränkt, aber Strukturen, konkrete Festlegungen und Datentypen von Variablen werden solange wie möglich offen gehalten. Diese mögliche Vielfalt bezeichnet man als Polymorphie.

Polymorphie (aus dem Griechischen: *Vielgestaltigkeit*) ist z.B. aus der Biologie, aus den Grundlagen der Mathematik und aus der Linguistik bekannt. Schauen Sie in ein Lexikon!

In der Informatik ist Polymorphie ein Grundprinzip. Mit ihr lässt sich die Wiederverwendung von Programmteilen (englisch: reuse, Aussprache ri:ju:s) erleichtern. Die Polymorphie äußert sich durch folgende Maßnahmen:

Möglichst lange den konkreten Datentyp von Variablen offen lassen. Man denke an unspezifizierte Feldgrenzen bei arrays (vgl. 1.3.5.6).

Möglichst lange die konkrete Realisierung offen lassen. Z.B. Spezifikations- und Implementierungsteil trennen.

Parametrisierung von Paketen und Unterprogrammen, um diese für möglichst viele Anwendungen einsetzen zu können (Generizität, siehe im Folgenden).

4.4.1: In der Logik bedeutet Polymorphie, dass zu einem Axiomensystem mehrere wesentlich verschiedene Modelle existieren. Siehe abstrakte Datentypen in 1.4.2.3 und 5.2.3.

Beispiel: Gesucht ist ein Datenbereich D mit Operationen "+" und "$*$" , so dass für alle Elemente $a, b, c \in D$ gilt:

(1) $a + b = b + a$, (2) $(a + b) + c = a + (b + c)$,
(3) $a * b = b * a$, (4) $(a * b) * c = a * (b * c)$,
(5) $a * (b + c) = (a * b) + (a * c)$,
(6) $(b + c) * a = (b * a) + (c * a)$,
(7) D besitzt unendlich viele Elemente.

Zu diesen Axiomen gibt es verschiedene mathematische Strukturen, die sie erfüllen (so genannte "Modelle"); zum Beispiel die natürlichen Zahlen $\mathbb{N}_0$, die ganzen Zahlen $\mathbb{Z}$, die rationalen Zahlen $\mathbb{Q}$, die reellen Zahlen $\mathbb{R}$ usw. Dieses Axiomensystem lässt also verschiedene Konkretisierungen zu, es ist daher polymorph.

4.4.2: In der Programmierung spricht man in folgenden Fällen von Polymorphie:

1. Mehrfachverwendung von ***Bezeichnern*** (hierzu zählt das Überladen, siehe 4.1.8).

2. ***Variablen*** können je nach aktueller Umgebung Elemente verschiedener Datentypen bezeichnen oder als Werte besitzen.

Beispiele hierfür haben wir schon angedeutet (1.3.5.3). Betrachte

```
subtype Natural is Integer range 0..Integer'Last;
subtype Positive is Natural range 1..Natural'Last;
X: Integer; Y: Natural; Z: Positive;
```

Y und Z werden hierbei zugleich als Variable des Typs Integer angesehen, sind also polymorph.

```
type Vektor is array (Integer range <>) of Float;

procedure Etwas (X, Y: in out Vektor) is
Summe: Vektor;
begin  ... for J in Y'Range loop Summe(J) := ... end loop;
          ...
end Etwas;
```

In diesem Beispiel sind X und Y Variablen eines Datentyps, dessen Größe (in Form der Indexgrenzen) unbekannt ist. Der Datentyp von X und Y steht also erst nach der Auswertung der zugehörigen aktuellen Parameter fest. Etwas Ähnliches gilt für die lokale Variable Summe. Die Variablen X, Y und Summe kann man in der Prozedur daher als polymorph ansehen.

3. Parametrisierung mit ***Typen***. Betrachte zum Beispiel Polynome $a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + ... + a_1 \cdot x + a_0$. Wünschenswert wäre eine Deklaration der Form

```
type polynom (n: Natural; type Ring) is
        record A: array (0..n) of Ring; end record;
```

Hinzufügen muss man die Operationen auf Polynomen, z.B. =, ≠, +, -, *, /, die auf den Operationen des Typs "Ring" beruhen. Diesen parametrisierten Typ könnte man dann im Programm benutzen, um konkrete Variablen zu deklarieren:

```
P: polynom (n => 4, Ring => Float);     oder mit Initialisierung:
Q: polynom (4, Integer) := (2, 0, 6, -5, 3);
```

Q bezeichnet also das Polynom über **Z** (A(0) ist 2, A(1) ist 0 usw.):

$q(x) = 3 \cdot x^4 - 5 \cdot x^3 + 6 \cdot x^2 + 2.$

Typen als Parameter in anderen Typen treten oft auf. Zum Beispiel hängen Listen nicht vom Typ ihrer Elemente ab, der daher möglichst spät erst festgelegt werden sollte.

4. Ein ***Unterprogramm*** oder ein ***Modul*** bzw. Paket heißt polymorph, wenn mindestens eine Spezifikation eines Datentyps oder wenn der Datentyp eines formalen Parameters oder des Ergebnisses polymorph ist. Bekannte Beispiele für solche Polymorphie sind Sortierverfahren, Integrale usw.

Diese Form von vielfacher Verwendbarkeit von Programmeinheiten (vor allem Unterprogramme und Moduln) bezeichnet man als Generizität. Es wird ein Schema angelegt, das später durch konkrete Datentypen und Konstanten ausgefüllt werden muss. Beispiel:

Parameter: Datentyp T mit Element 0 und Operationen =, ≠, +, -, *, /.

```
function Euklid (A, B: value T) return T is  R: T;
begin while B≠0 do R := A - (A / B) * B; A := B; B := R od; return A end;
```

Verwendung in der Form (polynom: siehe vorige Folie):

```
function ggT is new Euklid (T => Natural);
function ggTPolynom is new Euklid (T => polynom);
```


Beispiel: Vertausche die Werte zweier Variablen

```
procedure Tausch (type T; A, B: access T) is
  H: T;
begin  H := A;  A := B;  B := H  end;
```

Im Falle

```
        X, Y: Character;
```

könnte man nun die Inhalte von X und Y vertauschen durch den Prozeduraufruf:

```
        Tausch (Character, X, Y);
```

Effizienter wäre es, zuvor eine Prozedur für einen konkreten Datentyp aus dem "Prozedurschema Tausch" generieren, so dass man nicht erst zur Laufzeit, sondern schon beim Übersetzen den aktuellen Datentyp einsetzt.

Ziel (aus Sicht der Effizienz des Programms):
Die Polymorphie möglichst "zur Compilezeit" auflösen.
Vorgehen: Erzeuge neue Prozedur aus dem "Prozedurschema".

```
procedure Tausch (type T; A, B: access T) is
   H: T;
begin  H := A;  A := B;  B := H  end;

procedure TauschChar (A, B: access Character) is
        new Tausch (T => Character; A, B: access T  );
...
TauschChar (X, Y);
```

Dies kann auch entfallen.

Ada geht auf diese Weise vor. Schlüsselwort, um T als später auszufüllenden Parameter zu kennzeichnen: generic. Andere Sprachen erlauben es, den Typ erst zur Laufzeit mitzuteilen.

4.4.3 Beispiel für Generizität

Unter Generizität versteht man in Ada die Parametrisierung von Programmeinheiten mit Datentypen, Unterprogrammen und Moduln. Der Spezifikations- und der Implementierungsteil müssen bei generischen Einheiten getrennt angegeben werden. Der variabel gehaltene Bereich wird mit "generic" eingeleitet. Nochmals das Beispiel Vertauschen zweier Variableninhalte:

Spezifikation

```
generic
    type element is private;
procedure Tausch (A, B: in out element);
```

Implementierung

```
...
procedure Tausch (A, B: in out element) is
    H: element;
begin  H:=A; A:=B; B:=H;  end Tausch;
```



```
generic  type element is private;
procedure Tausch (A, B: in out element);
procedure Tausch (A, B: in out element) is
   H: element;
begin  H:=A; A:=B; B:=H;  end Tausch;
```

KonkreteVerwendung: Man muss aus dem Schema eine ausführbare Prozedur machen, indem man die generic-Objekte durch bereits definierte Objekte ersetzt.

```
procedure BoolTausch is new Tausch (Boolean);
procedure IntTausch is new Tausch (Integer);
X, Y: Integer;  C, D: Boolean;
...
IntTausch (X, Y); ...
BoolTausch (C, D); ...
```

Zwei Instanzen von Tausch mit konkreten Datentypen.

Beachten Sie:

Der Bezeichner "element" ist ein formaler Parameter in der Prozedur Tausch. Die Prozedur BoolTausch entsteht, indem die Prozedur Tausch mit dem aktuellen Parameter Boolean aufgerufen wird.
Dadurch wird der formale Parameter "element" textuell durch den aktuellen Parameter "Boolean" ersetzt.
Es handelt sich in diesem Fall also um eine "call-by-name" Übergabe.
Speziell muss der Ada-Compiler Namenskonflikte beseitigen (vgl. 4.2.9).
Prüfen Sie, was in folgendem Beispiel bei "BoolTausch (X,Y)" geschieht:

```
procedure Teste_generisch is
generic  type element is private;                    -- Spezifikation
procedure Tausch (A, B: in out element);
procedure Tausch (A, B: in out element) is           -- Implementierung
   type Boolean is new Integer range 0..50; H: element;
begin  H:=A; A:=B; B:=H;  end Tausch;
procedure BoolTausch is new Tausch (Boolean);        -- Konkretisierung
X: Boolean := True; Y: Boolean := False;
begin  BoolTausch (X,Y);                             -- Anwendung
   if X then Put(" 1"); else Put(" 0"); end if;      -- Überprüfung
   if Y then Put(" !"); else Put(" ?"); end if;
end;
```

4.4.4 Sortieren durch Austauschen benachbarter Elemente

Spezifikation

```
generic
   max: Positive;
   type T is private;
   type VektorT is array (1..max) of T;
procedure SORT (A: in out VektorT);
```

In der Regel ist dies nicht erlaubt.

Implementierung

```
procedure SORT (A: in out VektorT) is
procedure Austausch is new Tausch (T);
Weiter: Boolean := True;
begin
while Weiter loop  Weiter := False;
    for I in 1..max-1 loop
    if A(I) > A(I+1)
        then Weiter := True; Austausch(A(I), A(I+1)); end if;
    end loop;
end loop;
end SORT;
```

Korrekt, Austausch ist lokal definiert.

Dieser Operator ist bekannt zu machen

(Dies ist eine Variante von Bubble-Sort, siehe 7.3.3 und 10.2.1.)

Realisierung in Ada:
Ein generic-Parameter (hier: Max) darf in Ada nicht bereits im generic-Bereich verwendet werden.
Der Operator " > " muss bekannt gegeben werden ("with").
Korrekte Spezifikation mit anschließender Implementierung:

```
generic
   Max: Positive;
   type T is private;
   with function ">"(L,R:T) return Boolean;
package Sortpaket is
   type VektorT is array (1 .. Max) of T;
   procedure Sort (A: in out VektorT);
end Sortpaket;
```



```
package body Sortpaket is
  generic   type Element is private;
  procedure Tausch (A, B: in out Element);
  procedure Tausch (A, B: in out Element) is
    H: Element; begin  H:=A; A:=B; B:=H;  end Tausch;
  procedure Sort(A: in out VektorT) is
    Weiter: Boolean := True;
    procedure Austausch is new Tausch(T);
  begin
    while Weiter loop
      Weiter := False;
      for I in 1..Max-1 loop
          if A(I)>A(I+1) then  Weiter := True;
                               Austausch(A(I), A(I+1)); end if;
      end loop;
    end loop;
  end Sort;
end Sortpaket;
```


Verwendung im Programm:

```
        -- Im Deklarationsteil:
with Sortpaket; with Ada.Integer_Text_IO;
package SortpaketInt is new Sortpaket(500, Integer, ">");
R: SortpaketInt.VektorT;

        -- Im Anweisungsteil:
...                          -- Fülle das Feld R mit ganzen Zahlen.
SortpaketInt.Sort(R);   -- Das Integer-Feld R wird aufsteigend sortiert.
...
```

Hinweis: Ersetze im package SortpaketInt
">" durch "<" ⇒ Es wird absteigend sortiert.

Implementieren Sie dieses Verfahren in Ada und sortieren Sie dann auf die gleiche Art ohne großen Programmieraufwand Zeichen, Strings usw. (Wiederverwendbarkeit).

4.4.5 Hinweise zur Syntax für generische Unterprogramme und Pakete in Ada:

```
generic_declaration ::=  generic_subprogram_declaration |
                         generic_package_declaration

generic_subprogram_declaration ::=
        generic_formal_part  subprogram_specification ;

generic_package_declaration ::=
        generic_formal_part  package_specification ;

generic_formal_part ::=
    generic {generic_formal_parameter_declaration | use_clause}
```



```
generic_formal_parameter_declaration ::=
    formal_object_declaration  |
    formal_type_declaration    |
    formal_subprogram_declaration  |
    formal_package_declaration
formal_subprogram_declaration ::=
    with subprogram_specification [is subprogram_default] ;
formal_package_declaration ::=
    with package defining_identifier is new
    generic_package_name  formal_package_actual_part ;
formal_type_declaration ::=
    type defining_identifier[discriminant_part] is
                                    formal_type_definition ;
formal_package_actual_part ::= (<>) | [generic_actual_part]
```

usw.

4.4.6 *Weiteres Beispiel (Keller als generisches Paket)*

```
generic   type item is private;
package  Stack  is
   type StZelle;
   type RefStZelle is access StZelle;
   type StZelle is record Inhalt: item; Next: RefStZelle; end record;
   procedure Push (A: in item; S: in out RefStZelle);
   procedure Pop (S: in out RefStZelle);
   function Top (S: in RefStZelle) return item;
   function Isempty (S: in RefStZelle) return Boolean;
end Stack;

package body  Stack  is
   procedure Push (A: in item; S: in out RefStZelle) is
        begin  S := new RefStZelle'(A,S);  end;
   ...
end Stack;
```


Verwendung dieses Pakets:

```
package Zeichenstack is new Stack (Character);

A: Character;
X: Zeichenstack.RefStZelle;

use Zeichenstack;

begin   ...
        Push (A, X);  ...
        if Isempty (X) then Push ('C', X);
        else Pop (X); end if;
        ...
end;
```

4.5 Vererbung

Die Welt ist voller Hierarchien und Beziehungen.

1. Hat man einen Datentyp deklariert, so kann man durch Hinzufügen weiterer Komponenten hieraus weitere Datentypen ableiten ("Spezialisierung", Unterdatentypen).
2. Liegen mehrere Datentypen vor, die gewisse Komponenten gemeinsam haben, so kann man diese Gemeinsamkeiten als einen eigenen Datentyp herausziehen ("Generalisierung", Oberdatentyp).

4.5.1 Typerweiterung oder Spezialisierung: Wir formulieren dies sofort in Ada. Dort muss ein Datentyp, der später erweitert werden soll, mit dem Zusatz "tagged" deklariert werden (englisch: tag = Etikett, Zusatz).

```
type Fahrzeug is tagged record
        Hersteller: array (1..30) of Character;
        Höchstgeschwindigkeit: Positive;
        Neupreis: delta 0.01 range 0.0 .. 50_000_000.0;
end record;
```



```
type Bus is new Fahrzeug with record
        Sitzplätze, Stehplätze: Positive;
        Wendekreis: delta 0.001 range 0.0 .. 40.0;
        Achsenzahl: Positive;
end record;
```

Dies entspricht (bis auf "tagged") der folgenden Deklaration:

```
type Bus is record
        Sitzplätze, Stehplätze: Positive;
        Wendekreis: delta 0.001 range 0.0 .. 40.0;
        Achsenzahl: Positive;
        Hersteller: array (1..30) of Character;
        Höchstgeschwindigkeit: Positive;
        Neupreis: delta 0.01 range 0.0 .. 50_000_000.0;
end record;
```


Sprechweisen:

Man sagt, "Bus" ist ein aus "Fahrzeug" abgeleiteter Typ.

Man sagt, die Komponenten "Hersteller", "Neupreis" und "Höchstgeschwindigkeit" wurden vom Typ "Fahrzeug" auf oder an den Typ "Bus" vererbt.

Man nennt diesen Vorgang, Eigenschaften an andere Einheiten weiterzureichen, "Vererbung" (engl. inheritance).

Man sagt, "Bus" ist "Spezialisierung" oder "Erweiterung" oder Untertyp vom Typ oder zum Typ "Fahrzeug".

Man sagt, "Fahrzeug" ist "Generalisierung" oder Supertyp oder Obertyp vom Typ oder zum Typ "Bus".

Wie bei Bäumen nennt die Obertypen auch (direkte) "Eltern", die Untertypen (direkte) "Kinder". Setzt man dies transitiv fort, so spricht man von "Vorfahren" bzw. "Nachkommen".

Es werden also alle Deklarationen des Typs Fahrzeug auf den Typ Bus vererbt.

(Hinweis: Nicht nur Erweiterungen sind hierbei erlaubt, sondern auch Umdefinitionen bereits bestehender Typen.)

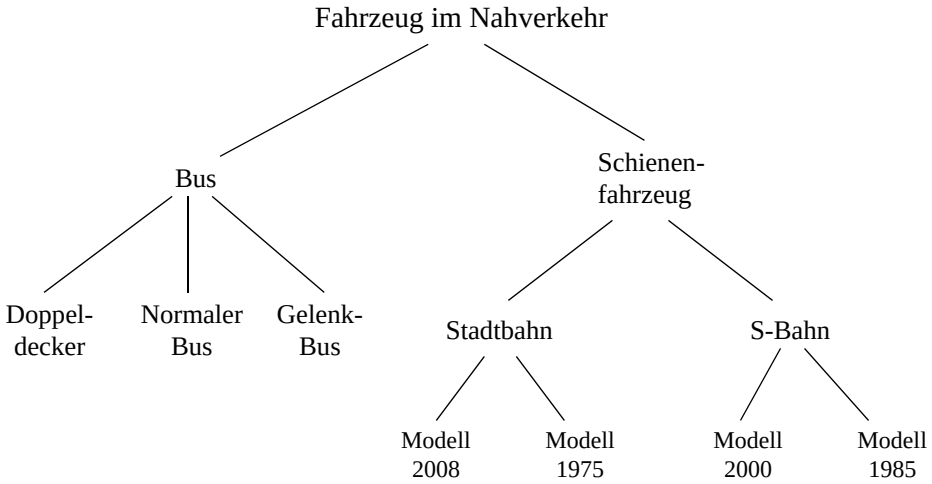
Das Gleiche für eine S-Bahn:

```
type SBahn is new Fahrzeug with record
        Sitzplätze, Stehplätze: Positive;
        Waggonlänge: delta 0.001 range 0.0 .. 400.0;
        Achsenzahl: Positive;
end record;
```


Weitere Typen (in Ada muss man explizit angeben, wenn man nichts hinzufügen will):

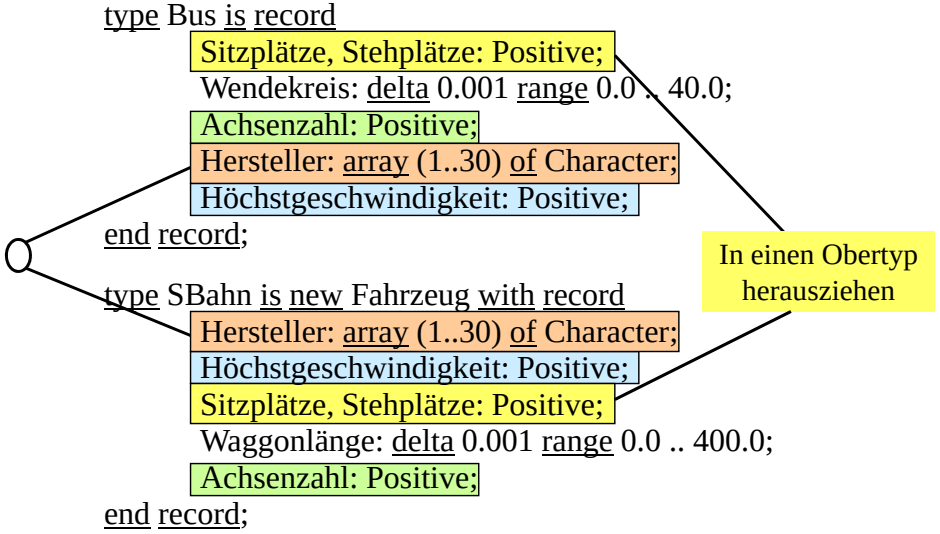
```
type Normaler_Bus is new Bus with null record;

type Doppeldecker is new Bus with record
        Höhe: Float;
end record;

type Gelenkbus is new Bus with record
        Länge: Float;
end record;

type Stadtbahn is new Fahrzeug with record
        Sitzplätze, Stehplätze: Positive;
        AnzahlKartenEntwerter: Positive;
end record;
```

Hinweise: Die Eigenschaft "tagged" vererbt sich automatisch auf alle abgeleiteten Typen. Ableitbare Typen erkennt man an "tagged" oder "with" in ihrer Definition.



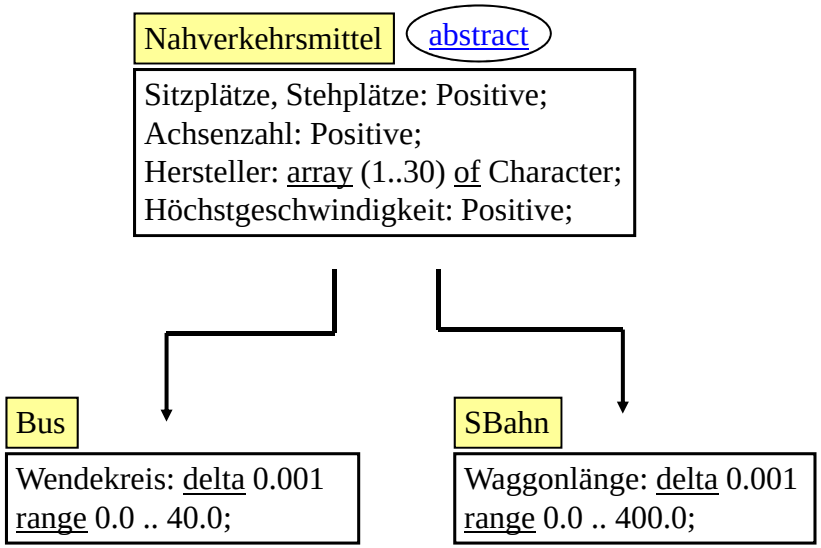
Aufbau einer Vererbungs-Hierarchie

4.5.2 Zusammenfassen oder Generalisierung: in Ada.



Neuer Typ "Nahverkehrsmittel".
Hieraus Bus und SBahn ableiten:

```
type Nahverkehrsmittel is abstract tagged record
        Sitzplätze, Stehplätze: Positive;
        Achsenzahl: Positive;
        Hersteller: array (1..30) of Character;
        Höchstgeschwindigkeit: Positive;
end record;

type Bus is new Nahverkehrsmittel with record
        Wendekreis: delta 0.001 range 0.0 .. 40.0;
end record;

type SBahn is new Nahverkehrsmittel with record
        Waggonlänge: delta 0.001 range 0.0 .. 400.0;
end record;
```

Hier tritt das Sprachelement "abstract" auf. Es bedeutet, dass man einen Datentyp deklariert hat, den man nur für die Vererbung verwendet, zu dem man aber keine Variablen oder formalen Parameter deklarieren darf. Bei dem "abstract"-Typ "Nahverkehrsmittel" sind also Deklarationen folgender Art *verboten*

```
X: Nahverkehrsmittel;
function F (A: in Nahverkehrsmittel) return Boolean; ...
```

In unserem Beispiel ist der Zusatz "abstract" nicht notwendig gewesen, doch sollte man ihn stets verwenden, wenn man den jeweiligen Typ nur in der "Vererbungs-Hierarchie" einsetzt.

(Beachte: Dieser Begriff "abstrakt" ist verschieden von dem Begriff "abstrakter Datentyp", der in Kapitel 5 erläutert wird!)

4.5.3 Umdefinitionen (redefinition)

Bei der Vererbung kann man vererbte Komponenten neu definieren. Die vererbten Komponenten sind dann wegen der Sichtbarkeitsregel automatisch ausgeblendet (können aber bei allgemeinen Strukturen wie packages über Qualifizierung mit Punkt-Notation dennoch angesprochen werden). Beispiel:

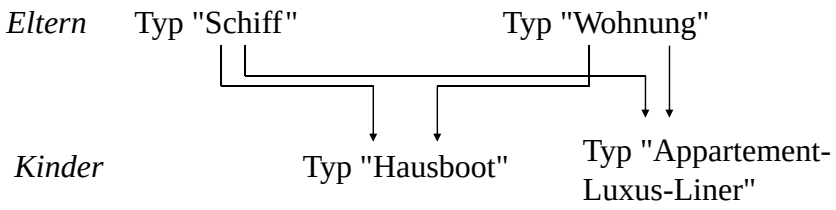
```
type wenig is 1..20;

type Kleinbus is new Bus with record
        Sitzplätze: wenig;
end record;
```

Die ererbte Komponente "Sitzplätze" wird durch diese neue Komponente "Sitzplätze" überschrieben (engl.: overridden = außer Kraft gesetzt).

4.5.4 Mehrfachvererbung

Nach dem bisherigen Konzept kann ein Datentyp höchstens einen (direkten) Obertyp besitzen:
Einfach-Vererbung (single inheritance).

Manchmal sollen aber Eigenschaften mehrerer Datentypen an einen Datentyp weitergereicht werden:
Mehrfach-Vererbung (multiple inheritance).



In Ada ist keine Mehrfachvererbung erlaubt (vor allem wegen der aufwendigeren Implementierung und der Fehleranfälligkeit bei der Verwendung mehrfacher Ableitungen, die erst zur Laufzeit nachvollzogen werden können.)

Die Programmierer sind selbst verantwortlich, wenn durch Vererbung gleicher Namen, die aber verschiedene Typen in den unterschiedlichen Eltern bezeichnen, Namenskonflikte oder andere Mehrdeutigkeiten entstehen. Generell sollte man die Mehrfachvererbung sehr sparsam und "sehr kontrolliert" verwenden.

In manchen objektorientierten Sprachen wie C++ und Eiffel ist Mehrfachvererbung möglich. In Java wurde sie nicht zugelassen. In C# lässt sie sich über sog. Interfaces nutzen.

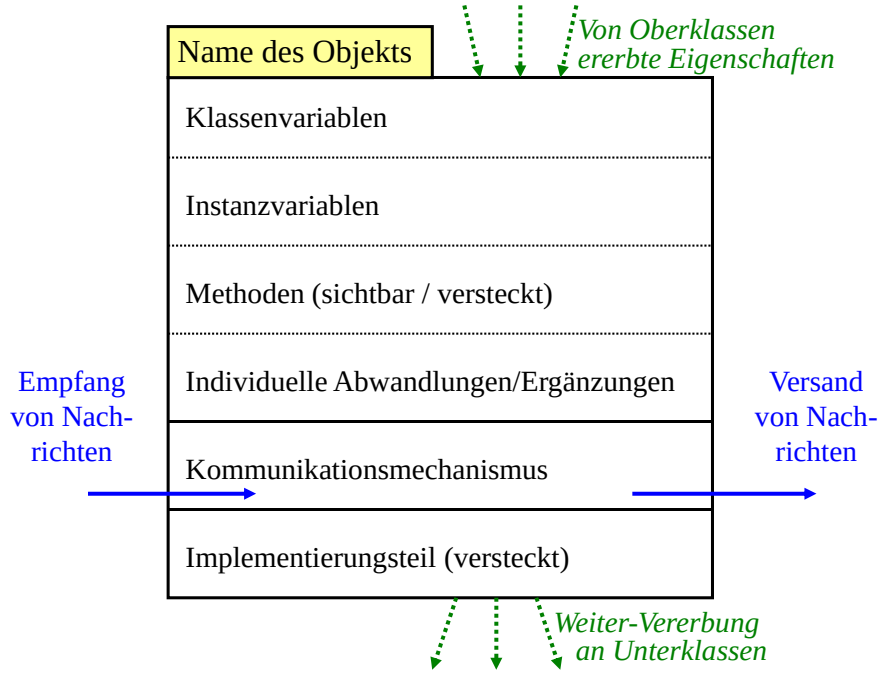
Diese Idee der Vererbung ist nicht auf Datentypen beschränkt. Wir können sie auch für Moduln einsetzen:

Wenn wir einen Modul "Listenverarbeitung" mit der Definition des Datentyps "Liste" und hierauf zugelassenen Operationen und Prozeduren eingeführt haben, so können wir hieraus einen Datentyp "Stack" durch Erweiterung gewinnen, indem wir weitere Prozeduren und Funktionen hinzunehmen (Empty und Isempty können wir übernehmen, Top, Pop und Push fügen wir unter Verwendung der bereits definierten Listen-Operationen hinzu; ggf. definieren wir auch "Einfügen" zu "Push" um).

Aber dies haben wir im Prinzip bereits getan mit dem Sprachelement with, mit dem man die sichtbaren Teile eines Pakets in eine andere Programmeinheit übernehmen kann.

4.6 Objekte (knappe Darstellung, kein Ada)

4.6.1: Objekte sind in sich geschossene Einheiten, die
- wie Moduln aufgebaut sind: Es gibt ein Schema, genannt "Klasse", das vor allem aus "Attributen" (das sind die einzelnen Datenstrukturen der Variablen) und "Methoden" (das sind die algorithmischen Teile) einschl. der Angaben zur Sichtbarkeit besteht und aus dem ein neues Objekt erzeugt werden kann; das Objekt ist eine Instanz (oder ein "Exemplar" oder eine "Ausprägung") dieser Klasse.
- einen individuellen Zustand besitzen (vor allem die Speicherbelegung der Klassen- und Instanzvariablen),
- miteinander kommunizieren können; dies geschieht durch Nachrichtenaustausch ("message passing"),
- durch Vererbung ihre Eigenschaften an neue Objekte bzw. Klassen weitergeben können.



4.6.2 Prinzipien der Objektorientierung:

1. Es gibt nur Objekte. Jedes Objekt ist eindeutig identifizierbar über seinen Namen.
2. Alles wird über Klassen, Instanzbildung, Zustände, Methoden, Nachrichten und Vererbung realisiert.
3. Objekte handeln in eigener Verantwortung (und sie geben nur bekannt, *was* sie bearbeiten, niemals, *wie* sie dies tun).
4. Klassen werden in Bibliotheken aufbewahrt und stehen allen Programmen und Klassendefinitionen zur Verfügung.
5. Programmieren bedeutet, Klassen festzulegen, hieraus Objekte zu erzeugen und diesen Aufgaben zu übertragen, indem man ihnen geeignete Nachrichten schickt. Die Auswertung der Objekte erfolgt hierbei erst zur Laufzeit (Polymorphie, dynamische Bindung der Objekte an Variable).

Wenn "alles" Objekte sind, so sind konsequenterweise auch Klassen, Nachrichten und die (formalen) Parameter Objekte.

Eine Methode der Klasse *"Klasse"* ist "new". Diese Methode erzeugt aus der Klasse eine Instanz, also ein konkretes Objekt. Jede Klasse ist eine Unterklasse der Klasse *"Klasse"* und hat somit diese Methode ererbt, kann also Instanzen von sich selbst erzeugen.

Eine konkrete Nachricht, die ein Objekt A an ein Objekt B schickt, besitzt meist aktuelle Parameter. Diese sind ebenfalls Objekte. Ebenso erwartet das Objekt A, dass das Objekt B ihm eine Nachricht mit konkreten Objekten als aktuellen Parametern zurückschickt.

4.6.3 Hinweise (1):

a. Klassen bilden *Hierarchien* oder zyklenfreie Abhängigkeitsgraphen (Ober- / Unterklassen, einfache / mehrfache Vererbung).
b. In typisierten Sprachen verweist eine Variable auf ein Objekt ihrer Klasse oder einer ihrer Oberklassen. Anderenfalls muss die Zuordnung laufend auf ihre *Zulässigkeit* überprüft werden.
c. Ist das zu aktivierende Objekt identifiziert, so muss man ermitteln, *welches die angeforderte Methode konkret ist* (eventuell muss man diverse Oberklassen durchsuchen) und ob es sie ausführen kann.
d. Zu jeder objektorientierten Sprache gibt es umfangreiche *Klassenbibliotheken*, aus denen man sich sein Programm aufbauen kann.
e. In der Praxis benötigt man eine *Entwurfsumgebung*, um Programme im Team zu entwickeln, um Erläuterungen, Entwurfsprozesse und verschiedene Dokumentationen zu erstellen und um die Klassenbibliothek zu erweitern.

Hinweise (2): Probleme in der Praxis, kleine Auswahl:

f. Die Sprache muss Erweiterungsmöglichkeiten von Moduln haben, ohne dass hierbei die privaten Informationen bekannt werden. (Das ist oft nicht realisierbar. Ada: child library units.)

g. Es müssen verschiedene Sichtweisen auf ein Objekt möglich sein. (Hierfür kann man Mehrfachvererbung nutzen.)

h. Es müssen Teile getrennt voneinander übersetzbar sein und abgelegt werden, allerdings darf hierdurch die Sichtbarkeit nicht beeinträchtigt werden. (Stichwörter in Ada: separate, stub.)

i. Die Klassenbibliotheken sollten sowohl den Quellcode als auch bereits getrennt compilierte Teile besitzen, und zwar so, dass diese leicht in ein Programm (z. B. über with und use) eingefügt werden können.

Hinweise (3): Probleme in der Praxis, kleine Auswahl:

j. Die Eindeutigkeit von Namen ist zu gewährleisten, z.B. durch Umbenennung importierter Größen (in Ada renames:

```
with Stack_für_Zeichen;
X: StackZ renames Stack_für_Zeichen.S;
function "*" (X,Y:Vektor) return Float renames Skalarprodukt;)
```

k. Die Sichtbarkeit in der Vererbungshierarchie ist genau festzulegen. (Beispiel: In der Regel sehen Unterklassen nicht, wie ihre Oberklassen die Methoden realisiert haben; daher können sie diese auch nicht verwenden, um Umdefinitionen vorzunehmen. Ändert man eine Methode ab, so muss man aber meist Zugriff auf jene soeben ausgeblendete Methode haben. In Ada: über Punkt-Notation. In Java durch "super".)

l. Die Sichtbarkeitsregeln müssen auch in der Klassenbibliothek gelten. Beispielsweise wird durch "with" die Sichtbarkeit einer anderen Klasse importiert (wann und wo endet diese?).

Hinweise (4): Probleme in der Praxis, kleine Auswahl:

m. Wie entwirft man "objektorientiert"? Man unterscheidet zwischen OOA und OOD, also "objektorientierte Analyse" und "objektorientierter Entwurf" ("D" = design = Entwurf). In der OOA wird ein Sollkonzept/Pflichtenheft erstellt und für dieses werden geeignete Klassen mit Zusatzinformationen (zeitliche Abläufe, einzuhaltende Bedingungen, Zusammenwirken der Einheiten, ...) erarbeitet. Im OOD werden hieraus die tatsächlichen Klassen, möglichst aus einer Klassenbibliothek und ergänzt um zusätzlich erforderliche Hilfs-Klassen erstellt und die Realisierung in einer Programmiersprache skizziert. (Anschließend folgt die Codierung.)

n. Die Zeit, dass man Lösungen zu Problemen von Grund auf neu schrieb, ist vorbei. Heute versucht man, eine Problemlösung so zu beschreiben, dass sie mit Hilfe der vorhandenen Klassen realisiert werden kann. Nur für wenige Probleme werden noch neue Klassen, neue Datentypen, neue Vorgehensweisen entwickelt (die anschließend in die Klassenbibliothek übernommen werden).

4.6.4 Beispiel: Addieren von Zahlen

Zur Illustration des Prinzips objektorientierter Denkweise:
Die Addition zweier Zahlen "7 + 28" läuft aus objektorientierter Sicht folgendermaßen ab:

Das Objekt 7 bekommt die Nachricht, es möge seine Methode "+" auf sich und das beigefügte Objekt (aktueller Parameter 28) anwenden und das Ergebnis zurückschicken.

Die Zahl 7 ist eine Instanz der Klasse "Integer", die aus einem Zustand (dies ist der Wert 7 seiner Instanzvariablen INH) und aus den zulässigen Methoden "+", "abs", "*", "-" usw. sowie den allgemeinen Methoden "send" (=schicke das Objekt, das auf "send" folgt, an den Sender zurück), "write" (drucke den Wert von INH aus) usw. besteht.

Wir haben also eine Klasse "Integer" (Oberklasse sei "Zahlen")

Integer	Vererbt von "Zahlen" werden: Typ "Binärfolge" und hierauf die Operationen plus, minus, mal, "<", "=", wie sie üblich sind. Der Ausbau zu "Integer" erfolgt jetzt:
Klassenvariablen	Null: constant Binärfolge := 0; Zehn: constant Binärfolge := 1010
Instanzvariablen	INH: Binärfolge
Methoden	function "+" (X: Integer) return Integer; function quad return Integer; function "abs" return Binärfolge; ...
Kommunikation	procedure return1 (A:Binärfolge) is begin X := new Integer; X.INH := A; send X end; procedure send (A:Object) is begin "schicke A an den Sender zurück" end; ...
Implementierung	procedure "+" (X: Integer) is begin return1 (plus (self.INH, X.INH)) end; procedure abs is begin if self.INH<0 then send minus(Null,self.INH) else send self.INH fi end; procedure quad (X: Integer) is begin return1 (mal (self.INH, self.INH)) end; ...

Es bleibt der jeweiligen Sprache überlassen, ob die im sichtbaren Bereich aufgelisteten Funktionen tatsächlich als Funktionen oder auf andere Weise (z.B. wie hier als Prozeduren; aber durch "return1" wird die richtige Sicht nach außen hergestellt) implementiert werden.
„self“ bezeichnet das Objekt, das diese Methode soeben verwendet.
mal (self.INH, self.INH)) multipliziert also den eigenen Inhalt mit sich selbst und es entsteht das Quadrat des eigenen Wertes.

Z := 7 + 28 wird wie folgt ausgerechnet
(die beiden Zahlen werden als Binärfolge dargestellt):

X := new Integer; X.INH := 111;
Y := new Integer; Y.INH := 11100;
Z := X."+"(Y);

X."+"(Y) bedeutet: Schicke an das Objekt, das durch X bezeichnet wird, die Nachricht, dass dessen Operation "+" ausgeführt werden soll. Diese Funktion erwartet ein Objekt der Klasse Integer als Parameter, daher wird ein solches Objekt Y als aktueller Parameter mitgegeben. Das Objekt X antwortet dann mit einem Objekt, das an den Bezeichner Z gebunden wird.

Durch Deklarationen (z.B. Z: Integer) kann man dafür sorgen, dass nicht beliebige Objekte, sondern nur Objekte bestimmten Typs an Z gebunden werden dürfen.

4.6.5 Beispiel: Geometrische Größen

Geometrische Größen entwickelt man gerne auseinander. Man beginnt mit einem Punkt, geht zur Strecke und zum Polygon über, dann betrachtet man Dreiecke, Vierecke und n-Ecke, führt Kreise (= Punkt plus Strecke) und Ellipsen ein usw.

Zugleich kann man schrittweise weitere Eigenschaften hinzufügen (insbesondere die Fläche) und Unterklassen bilden (z.B.: Viereck → Trapez → Raute → Quadrat), wobei zugleich die Flächenberechnung jeweils neu definiert werden kann.

Die Darstellung verändern wir nun etwas, indem wir die Objekte wie Moduln ausformulieren und die Oberklasse(n) explizit angeben. Es geht hier um das Prinzip und nicht um die Darstellung in einer bestimmten Programmiersprache.

Wir definieren also eine Klasse "Punkt" (Oberklasse sei Real):

Punkt	Real
X: Real; Y: Real;	
procedure Drucken	
procedure Drucken is begin < sende an das Objekt *Drucker* die Nachricht drucke_ein_Pixel_an_die_Position (self.X, self.Y) > end	

Wir definieren hiermit eine Klasse "Strecke" (beachte: mit der Klasse "Punkt" ist auch deren Oberklasse "Real" bekannt):

Strecke	Punkt
Anfang: Punkt; Ende: Punkt;	
procedure Drucken, DruckeAnfang, DruckeEnde; function Länge ( ) return Real	

```
procedure Drucken is
begin < sende an das Objekt Drucker die Nachricht
   drucke_eine_ Strecke_von_bis (self.Anfang.X,
         self.Anfang.Y, self.Ende.X, self.Ende.Y ) >  end;
procedure DruckeAnfang is begin Anfang.Drucken end;
procedure DruckeEnde is begin Ende.Drucken end;
function Länge ( ) return Real is
begin return (square_root(quad(self.Anfang.X - self.Ende.X)
         + quad(self.Anfang.Y - self.Ende.Y))) end;
```


Wir definieren nun eine Klasse "Polygon" mit dem generischen Parameter N (Oberklassen sind Strecke und Natural):

Polygon (N: Natural, N > 2)	Strecke, Natural
Eckpunkte: array [1..N] of Punkt; private Streckenzug: array [1..N] of Strecke;	
procedure Drucken; function Länge ( ) return Real	

```
procedure Drucken is  var I: Natural;
begin for I := 1 to N do Streckenzug[I].Drucken od  end;
function Länge ( ) return Real is var I: Natural; L: Real := 0.0;
begin for I := 1 to N do L := L + Streckenzug[I].Länge od; return L end;
```

Initialisierung

```
var I, K: Natural;
begin for I := 1 to N do Streckenzug[I] := new Strecke;
         Streckenzug[I].Anfang := Eckpunkte[I];
         if I = N then K:=1 else K:=I+1 fi;
         Streckenzug[I].Ende := Eckpunkte[K] od
end
```


Beachten Sie hierbei:

Mit der Klasse "Strecke" ist auch deren Oberklasse "Punkt" eine Oberklasse von Polygon. Daher können wir deren Eigenschaften hier verwenden.

Am Ende haben wir einen Initialisierungsteil angefügt, mit dem wir die Folge der Strecken, die durch die Eckpunkte definiert ist, errechnen. Diese Information geben wir nach außen nicht bekannt ("private"). Den Streckenzug verwenden wir intern zum Drucken und zur Längenberechnung.

Sie erkennen hier einen Vorteil der OOP (= objektorientierten Programmierung): Wenn wir die Klasse "Punkt" von "Real" auf "Integer" umschreiben würden, so braucht an Strecke und Polygon nichts geändert zu werden! (Wiederverwendbarer Quellcode.)

Nun müsste man eigentlich noch

- eine Boolesche Funktion "kreuzungsfrei" hinzufügen, die genau dann den Wert true ergibt, wenn sich je zwei Strecken des Polygons nicht schneiden. Hierzu würde man in der Klasse "Strecke" eine Methode einfügen:

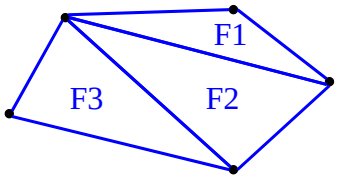
  function schnitt (S: Strecke) return Boolean is
  var ...
  begin ... end;

  (Selbst ausformulieren; Erläuterungen auf später folgenden Folien.)

- eine Boolesche Funktion "konvex" hinzufügen, die genau dann den Wert true ergibt, wenn das Polygon kreuzungsfrei ist (dann hat es ein "Inneres") und wenn alle Winkel zum Inneren des Polygons hin höchstens 180 Grad betragen.

Nun müsste man eigentlich noch

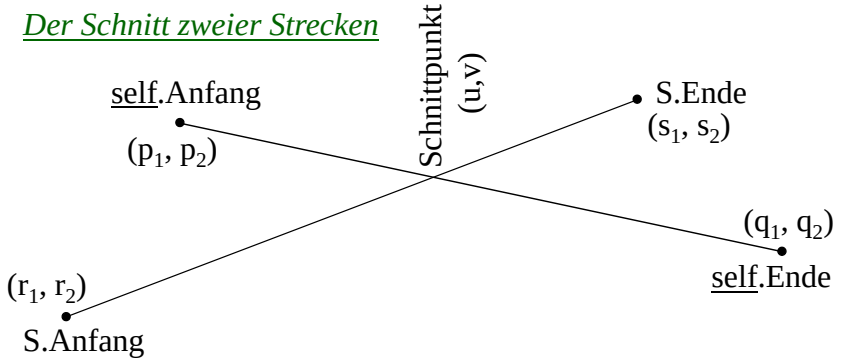
- für konvexe Polygone eine "Fläche" hinzufügen. Diese erhält man, indem man von einem Punkt aus alle aufeinander folgenden Dreiecksflächen aufsummiert (die Dreiecksfläche ist durch drei Seitenlängen eindeutig bestimmt und hieraus leicht zu berechnen); Beispiel:



Gesamtfläche =
F1 + F2 + F3.

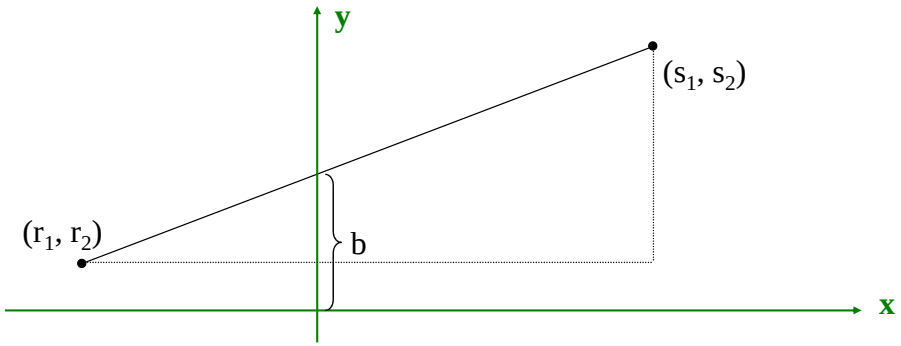
- die Flächen ausweiten auf kreuzungsfreie Polygone,
- den Schwerpunkt eines Polygons bestimmen oder andere "Mittelpunkte".

Der Schnitt zweier Strecken



Den Schnittpunkt erhält man als Schnittpunkt der beiden Geraden, die zu den Strecken gehören. Für zwei Geraden $y = a_1 \cdot x + b_1$ und $y = a_2 \cdot x + b_2$ ergibt sich der Schnittpunkt (u,v) durch $v = a_1 \cdot u + b_1$ und $v = a_2 \cdot u + b_2$, also für $a_1 \neq a_2$:

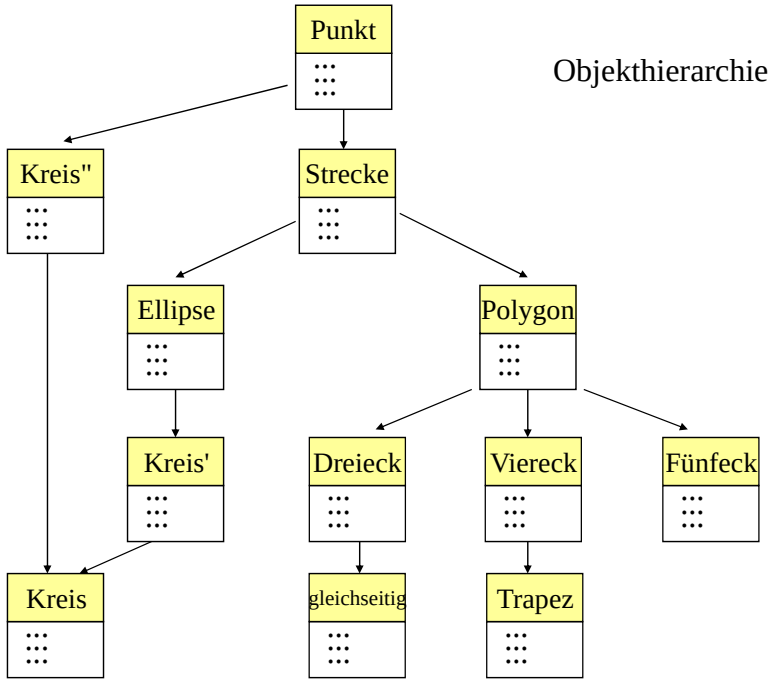
$$u = \frac{b_2 - b_1}{a_1 - a_2} \quad \text{und} \quad v = a_2 \cdot \frac{b_2 - b_1}{a_1 - a_2} + b_2$$



Ermittlung der Steigung a und des y-Abschnitts b der zu einer Strecke gehörenden Geraden $y = a \cdot x + b$:

$$a = \frac{s_2 - r_2}{s_1 - r_1} \quad \text{und} \quad b = s_2 - a \cdot s_1$$

Ein Schnitt liegt vor, wenn die x- und die y-Koordinate des Schnittpunkts echt zwischen den entsprechenden Koordinaten der beiden gegebenen Strecken liegt. (Bitte selbst zu Ende durchdenken und fertig programmieren.)



Objekthierarchie

4.6.6 Beispiel: Skat spielen

Kurzanalyse: Drei Spieler, die alle die Spielregeln kennen, die reizen, ausspielen und beilegen können, die wissen, wer den jeweiligen Stich gewonnen hat, die am Ende jedes Spiels den Sieger des Spiels einschl. der Punktzahl ermitteln und dies in eine geeignete Tabelle eintragen können. Am Ende entnimmt man der Tabelle den Gesamtsieger bzw. die individuellen Spielschulden jedes Spielers.

Es wird nur angedeutet, wie der Verlauf des Skat-Spiels modelliert werden könnte. In der Praxis würde man zunächst eine Klasse "Kartenspiel" mit diversen Methoden (Auflisten der Karten, Mischen, Verteilen nach verschiedenen Vorgaben usw.) einführen, ebenfalls eine Spiel-Verwaltung für Stand, Punkte, abgelegte Karten usw. Das Folgende soll daher nur "anregen".

Tisch

Bekannt seien aus Oberklassen: type Karte und das "array" Kartenspiel in Form von "alle_karten"

```
subtype bis_drei is Natural range 0..3;  subtype nr is bis_drei range 1..3;
Skat: array [1..2] of Karte; Blatt: array [nr, 1..10] of Karte;
Anz_gesp_Karten: bis_drei;  gespielte_Karten: array [nr] of Karte;
Anzahl_Spiele: Natural;  Punktestand: array [nr] of Integer;
Erster_Spieler: nr;  Ausspielend, Abhebend, Geber, Gewinner: nr;
Aktuelles_Spiel: Spiel_Typen; gereizt_bis: Natural; Alleinspieler: nr;
Abgelegt: array [Karte] of Boolean; constant KS := alle_karten;
```

```
procedure Karten_Verteilen; procedure reizen (S,T: Spieler);
procedure Ein_Stich; procedure Spiel_Ergebnisse;
function vor (K: nr) return nr;  function nach (K: nr) return nr;
procedure End_Abrechnung;
```

```
function vor (K: nr) return nr is
begin if K = 1 then return 3 elsif K = 2 then return 1 else return 2 fi end;
procedure Karten_Verteilen is var I: nr; M: array[1..32] of Karte;
begin M := Zufalls_Permutation (KS); Skat := "die ersten beiden Karten";
   for I in nr do Blatt [I,1..10] := "die nächsten 10 Karten" od end;
...
```


Spieler

Die Klasse Tisch sei als Oberklasse oder anderweitig "bekannt"

```
subtype bis_drei is Natural range 0..3;  subtype nr is bis_drei range 1..3;
Skat: array [1..2] of Karte; Blatt: array [1..10] of Karte;
Anz_gewonn_karten: 0..32; gewonnene_Karten: array [1..32] of Karte;
Aktuelles_Spiel: Spiel_Typen; gereizt_bis: Natural; Alleinspieler: nr;
gewonn_Punkte_bisher: Natural; Abgelegt: array [Karte] of Boolean;
Reaktionen_der_anderen: <neuen Datentyp einführen ...für Strategie>;
ich: nr; Reiz_Obergrenze: Natural; mögliches_Spiel: Spieltypen;
Spielverlauf: list of ...; Reiz_Verlauf: list of ...;
```

```
procedure Karten_Sortieren; procedure Reizen;
function Ausspielen return Karte; function Beilegen return Karte;
procedure Spielweisen_analysieren;
```

```
function Ausspielen return Karte is  var ...;  begin ... return := ... end;
procedure Blatt_bewerten is  var ...;  begin ... end;
procedure Reizen is  var ...;  begin ... end;
 ...
```


Ein Programm muss dann den Ablauf in geordneter Weise steuern, also etwa:

```
...  "Initialisiere den Tisch und die Spieler;"
while noch_nicht_Schluss do
   Tisch.Karten_verteilen; "Initialisiere die Spieler";
   Tisch.Reizen (Abhebend,Ausspielend);
   Tisch.Reizen (Geber, Gewinner);
   "Initialisieren aller Spieler und des Tisches für das Spiel";
   for I := 1 to 10 do  Tisch.Ein_Stich  od;
   Tisch.Spiel_Ergebnisse
od;
Tisch.End_Abrechnung;  ...
```

Wollen Sie dies selbst weiter durchdenken, allgemein formulieren und dann in Ada programmieren? Geschätzter Zeitaufwand bis zu einer lauffähigen ersten Version: 40 Stunden (ohne Grafik).

Gliederung des Kapitels 5

5. Abstrakte Datentypen

5.1 Datentypen

Probleme beschreiben wir mit Hilfe von Mengen und auf ihnen definierten Operationen und Relationen.

5.1.1 Mengen mit den auf ihnen definierten Operationen nennt man in der Mathematik eine Algebra.

Die Mathematik untersucht hierbei Eigenschaften, die aus Axiomen folgen, z.B. aus dem Axiom, dass die Addition kommutativ ist (d.h., für alle a und b gilt a+b=b+a) und dass die Multiplikation assoziativ [a∗(b∗c)=(a∗b)∗c] ist. Zugleich sucht sie Mengen und Strukturen, die diese Axiome erfüllen. (Beispiele: endliche Gruppen; diese sind mittlerweile kategorisiert. Oder endliche nicht-kommutative Körper; es konnte gezeigt werden, dass es solche Körper nicht gibt.)

5.1.2 Beispiele für Algebren sind:

Die natürlichen Zahlen $\mathbb{N}_0$ mit der Addition: $(\mathbb{N}_0;+)$.
Die natürlichen Zahlen $\mathbb{N}_0$ mit der Addition und dem neutralen Element "0" bzgl. der Addition: $(\mathbb{N}_0;+,0)$.
Die natürlichen Zahlen $\mathbb{N}_0$ mit der Multiplikation "$*$" und dem neutralen Element "1" bzgl. der Multiplikation: $(\mathbb{N}_0;*,1)$.
Die reellen Zahlen mit den Operationen +, -, $*$ und / und den neutralen Elementen "0.0" (bzgl. der Addition) und "1.0" (bzgl. der Multiplikation): $(\mathbb{R};+,-,0.0,*,/,1.0)$.
Die Wahrheitswerte $\mathbb{B}$ = {false, true} mit den Operationen "and", "or" und "not": $(\mathbb{B};\wedge,\vee,\neg)$.
Die Menge der reellwertigen (n×n)-Matrizen mit der Addition, der Subtraktion, der Multiplikation und den neutralen Elementen "Nullmatrix" 0 und "Einheitsmatrix" E_n: $(\mathbb{R}^{n,n};+,-,0,*,E_n)$.
Die reellen Zahlen mit den Operationen +, -, $*$ und / und den neutralen Elementen "0.0" (bzgl. der Addition) und "1.0" (bzgl. der Multiplikation) und den Vergleichsoperationen =, < und >, wobei diese von $\mathbb{R}^2$ nach $\mathbb{B}$ führen: $(\mathbb{R},\mathbb{B};+,-,0.0,*,/,1.0,=,<,>)$.
Die komplexen Zahlen $\mathbb{C}$, dargestellt als zweidimensionale Ebene $\mathbb{R}^2$ mit der Addition $(a_1,b_1)+(a_2,b_2)=(a_1+a_2,b_1+b_2)$ und der Multiplikation $(a_1,b_1)*(a_2,b_2)=(a_1*a_2-b_1*b_2,a_1*b_2+b_1*a_2)$ usw.: $(\mathbb{R}^2,\mathbb{B};+,-,(0.0,0.0),*,/,(1.0,0.0),=)$.

In der Mathematik interessieren sowohl diese "konkreten Algebren" als auch Klassen von konkreten Algebren, die gewisse Axiome erfüllen.

Beispiel:
$(H;\bullet)$ heißt Halbgruppe $\Leftrightarrow$
(1) H ist eine Menge,
(2) $\bullet: H \times H \to H$ ist eine (totale) Abbildung,
(3) $\bullet$ ist *assoziativ*, d.h., für alle $a,b,c \in H$ gilt: $a\bullet(b\bullet c)=(a\bullet b)\bullet c$.
$(M;\bullet,e)$ heißt Monoid $\Leftrightarrow$
(1) $(M,\bullet)$ ist eine Halbgruppe,
(2) $e \in M$ ist ein neutrales Element *bzgl.* $\bullet$ (auch *Einheit* genannt), d.h., für alle $a \in M$ gilt: $a\bullet e = a = e\bullet a$.

Die Mathematik untersucht Eigenschaften solcher Algebren.
Zum Beispiel: Wenn $(M;\bullet,e)$ ein Monoid ist und e' ein Element aus M ist, sodass für alle $a \in M$ gilt $a\bullet e' = a = e'\bullet a$, dann ist e = e'.

Wer mit Zahlen umgeht, bewegt sich meist in Algebren mit mindestens zwei Operationen.

Zum Beispiel ist ein Ring eine Menge mit zwei Operationen, von denen eine kommutativ, assoziativ und invertierbar ist, die andere assoziativ ist und für die die Distributivgesetze gelten. Ist die zweite Operation zusätzlich kommutativ und invertierbar, so spricht man von einem Körper.

Alle Gesetzmäßigkeiten, die man für einen Ring herleiten kann, gelten dann auch für alle "konkreten Ringe". Beispielweise gelten in Körpern Kürzungsregeln und Lösungsverfahren für Gleichungssysteme und man kann über ihnen Funktionen- und Vektorräume aufbauen (Polynome, Matrizen, d-dimensionale Räume, ...). Wir erläutern dies am Beispiel der Polynomringe.

5.1.3 Beispiel: Polynomring

Ein Ring ist eine Algebra (R;+,-,0,$\star$,1), wobei R eine Menge ist, auf der die kommutative und assoziative Operation "+", die assoziative Operation "$\star$" und die bzgl. "+" bzw. "$\star$" neutralen Elemente Null "0" und Eins "1" definiert sind. Weiterhin gibt es zu jedem Element $a \in R$ ein inverses Element "-a" (mit a+(-a)=0), es müssen die Distributivgesetze gelten ($a\star(b+c)=a\star b+a\star c$ und $(b+c)\star a=b\star a+c\star a$) und meist verlangt man noch $0 \neq 1$, da der Ring sonst nur aus einem Element bestehen könnte.

Beispiele für Ringe sind die ganzen, die rationalen, die reellen und die komplexen Zahlen, wobei in diesen Fällen die Multiplikation zusätzlich kommutativ ist. Matrizenräume sind in der Regel nicht-kommutative Ringe (genauer: die Multiplikation ist nicht kommutativ).

Es sei R ein kommutativer Ring, d.h., ein Ring, dessen Multiplikation kommutativ ist. Eine Abbildung $p: R \to R$ der Form

$$p(x) = a_n \star x^n + a_{n-1} \star x^{n-1} + \ldots + a_1 \star x + a_0 = \sum_{i=0}^{n} a_i \star x^i$$

(mit $n \in \mathbb{N}_0$, $a_i \in R$ für $i = 0, 1, \ldots, n$ und $a_n \neq 0$, falls $n > 0$ ist) heißt Polynom über R. Die natürliche Zahl n heißt der Grad von p.

Nun definiere man für zwei Polynome p und q vom Grad n bzw. m

$p(x) = \sum_{i=0}^{n} a_i \star x^i$ und $q(x) = \sum_{i=0}^{m} b_i \star x^i$ die Summe und das Produkt.

O.B.d.A. sei $n \geq m$. Fülle q mit führenden Nullen auf, d.h., setze $b_{m+1} = b_{m+2} = \ldots = b_n = 0$ und bilde dann das Polynom (p+q):

$$(p+q)(x) = \sum_{i=0}^{n} (a_i + b_i) \star x^i \qquad \text{Setze: } (-p)(x) = \sum_{i=0}^{n} (-a_i) \star x^i .$$

Das Produkt zweier Polynome ist definiert durch:

$$(p \star q)(x) = \sum_{k=0}^{n+m} \left(\sum_{i=0}^{k} a_i \star b_{k-i} \right) \star x^k \quad \text{für } q \neq 0. \text{ Im Falle } q = 0 \text{ setze } (p \star q) = 0.$$

0 sei das Nullpolynom (d.h., es besteht nur aus $a_0 = 0$) und *1* sei das Einspolynom (d.h., es besteht nur aus $a_0 = 1$). Es sei P_R die Menge aller Polynome über R, dann ist die Algebra (P_R; +, -, *0*, $\star$, *1*) ein kommutativer Ring, der sog. Polynomring über R.

■

Zwischenfrage: Kann man so etwas in Ada formulieren?

$$p(x) = \sum_{i=0}^{n} a_i \star x^i .$$

```
type Ring is ....;
type Polynom (Grad: Natural:=0) is
        record A: array (0..Grad) of Ring := (1);
        end record;
```

Dies ist in Ada realisierbar. Allerdings muss für jeden Ring alles neu geschrieben werden. Besser wäre es daher, wenn auch der Datentyp Ring als Diskriminante übergeben werden könnte:

~~type polynom (Ring: "Datentyp"; Grad: Natural) is~~
~~record A: array (0..n) of Ring; end record;~~

Dies ist jedoch in Ada nicht erlaubt. Vielmehr muss man das "generic"-Konstrukt mit einem Paket verwenden.

5.1.3.a Folgerung aus diesen Überlegungen:

Es gibt "konkrete Algebren", z.B. die ganzen Zahlen.

Diese erfüllen die Gesetzmäßigkeiten einer "abstrakten" Algebra, also einer oder mehrerer Mengensymbole mit Operationensymbolen und Axiomen. Jede konkrete Algebra, die die Gesetzmäßigkeiten einer abstrakten Algebra erfüllt, nennen wir ein Modell dieser (abstrakten) Algebra.

Zum Beispiel ist die konkrete Algebra "ganze Zahlen" ein Modell der Algebra "Ring", aber sie ist kein Modell der Algebra "Körper". Die rationalen Zahlen sind ein Modell für einen Körper.

Will man allgemeine Konstruktionsprinzipien und Lösungsverfahren für möglichst viele Strukturen verwenden, so muss man sie für eine abstrakte Algebra formulieren können. Solche Beschreibungen müssen in der Informatik möglich sein.

Die Informatik befasst sich ebenfalls mit Mengen und ihren Operationen. Statt von einer Algebra spricht man bei konkreten Mengen von einem "Datentyp". Eine (abstrakte) Algebra entspricht dem später zu definierenden "Abstrakten Datentyp". Konkrete Datentypen sind Modelle eines abstrakten Datentyps.

Es gibt konkrete Datentypen, die man nicht weiter zerlegen kann (oder will); wir nennen sie "elementare Datentypen" (in Ada "skalare Datentypen" genannt), siehe Abschnitt 2.4.

Zu den "elementaren" Datentypen zählen: *Wahrheitswerte, Zeichen, ganze Zahlen, reelle Zahlen und Zeiger auf Datentypen (Namen)* sowie selbstdefinierte endliche Mengen (Aufzählungstypen).

Es gibt sie in fast allen Programmiersprachen.

Definition 5.1.4: *Datentyp* (Wiederholung aus 1.4 und 2.4)
Eine Menge (oder mehrere Mengen) zusammen mit den hierauf definierten Operationen nennt man einen (konkreten) Datentyp.
Einen Datentyp, den man nicht auf andere Datentypen zurückführt, nennt man einen elementaren Datentyp.

"Unsere" elementaren Datentypen sind neben den Aufzählungstypen: Boolean, Character, Integer, Real und Zeiger auf Objekte gegebener Datentypen.

Um diese Datentypen exakt festzulegen, müssen wir zu jeder Menge die zulässigen Operationen hinzufügen.

Jede Programmiersprache definiert diese Operationen in der Regel etwas anders. In 1.8 haben wir für Ada 95 diese Datentypen auf 30 Folien im Detail aufgeführt.

5.1.5: Bei der genauen Beschreibung der Datentypen kann man folgendes Schema verwenden (die letzte Zeile unten lässt man in der Regel weg).

Datentyp xyz:
Zugrunde liegende Mengen: ...
Nullstellige Operationen (=besondere Konstanten): ...
Einstellige Operationen: ...
Zweistellige Operationen: ...
höherstellige Operationen (sofern vorhanden): ...
Definition gewisser Operationen: ...
Beziehungen, Gesetzmäßigkeiten: ...
Dieser Datentyp lautet in der Sprache Ada 95: ....

5.1.6: Darstellung der Operationen (vgl. auch 3.7.6):

Hat man zwei Operationen f, g: $M \times M \to M$ auf einer Menge M, so kann man zusammengesetzte Ausdrücke ("Terme") bilden:

f(g(x, y), x) oder g(g(f(x,y),z), g(z, x))

5.1.6 a: Diese Darstellung, dass der Operator *vor* seinen Operanden steht, bezeichnet man als Präfixnotation oder als preorder. Man kann dann die Klammern auch weglassen:

f g x y x oder g g f x y z g z x

und die Auswertung der Ausdrücke bleibt eindeutig.

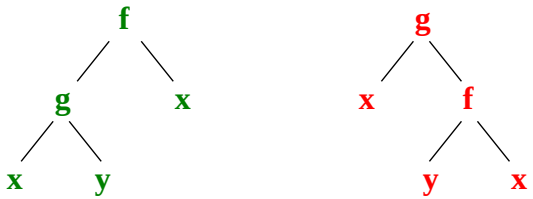
Statt dessen kann man die Operatoren auch *hinter* ihre Operanden schreiben:

x y g x f oder x y f z g z x g g

5.1.6 b: Dies nennt man Postfix-Notation (oder polnische Notation oder postorder). Auch hierbei bleibt die Auswertung der Ausdrücke eindeutig.

5.1.6 c: Im täglichen Leben verwendet man jedoch in der Regel die Infix-Notation oder als inorder, bei der man das Operationszeichen *zwischen* die Operanden stellt; obige Beispiele werden dann zu x g y f x oder x f y g z g z g x

Dies ist nicht mehr eindeutig, da nun z. B. f(g(x, y), x) und g(x, f(y, x)) die gleiche Infix-Notation x g y f x besitzen. Man erkennt dies, wenn man die Baumstruktur betrachtet:



Beide Bäume gehören zu x g y f x

Um bei der Infixnotation Eindeutigkeit zu sichern, verwendet man zum einen Klammern und zum anderen Prioritätsregeln.

Daher müssen bei den Operationen fast immer auch solche Prioritätsregeln für die Operatoren angegeben werden.

Ist nichts angegeben oder haben Operatoren die gleiche Priorität, so wird ein Ausdruck stets von links nach rechts ausgewertet.

5.1.7. Hinweis: Wenden Sie diese Überlegungen auf die Eindeutigkeit von Wörtern oder Grammatiken an (2.7.11):

Stellt man die Ableitungsbäume mit der Präfixnotation dar, so sind zwei Ableitungen genau dann gleich (im Sinne von 2.7.10), wenn sie die gleiche Präfixnotation besitzen!

Die Präfixnotation erhält man, indem man den (Ableitungs-) Baum von der Wurzel immer zunächst nach links zu den Blättern hin durchläuft und genau dann, wenn man einen Knoten erstmals besucht, dessen Inhalt herausschreibt.

Machen Sie sich dies an Beispielen klar. Dadurch kann man die Gleichheit von Ableitungen auch "algorithmisch" relativ leicht nachprüfen.

5.1.8: Hinweise zu den elementaren Datentypen

Aufzählungstypen: siehe 1.8.1.

Boolean: siehe 1.8.3. In den meisten Sprachen sind die Prioritäten anders als in Ada geregelt: and hat höhere Priorität als or und dies wieder höhere als xor. Oft gibt es auch die Implikation ("⇒") mit

impl(a,b) = not a or b

(d.h.: impl(a,b)=false ⇔ a=true und b=false), manchmal eine dreistellige Operation if-then-else-fi if x then y else z fi = (x and y) or (not x and z).

Einige Gesetzmäßigkeiten: not (not a) = a

exor (a, b) = not (equiv (a, b))

not (a and b) = (not a) or (not b)
not (a or b) = (not a) and (not b) } deMorgansche Regeln

5.1.8: (Fortsetzung)

Character: siehe 1.8.4 und 2.4.6, Wertebereich **A**.

Integer: siehe 1.8.5 und 2.4.8 bis 11, Wertebereich **Z**. Meist gibt es weitere Operationen, insbesondere:

sgn "Signum", also das Vorzeichen einer Zahl:

$$sgn(x) = \begin{cases} 1, & \text{falls } x > 0 \\ 0, & \text{falls } x = 0 \\ -1, & \text{falls } x < 0 \end{cases}$$

square Quadrat einer Zahl ($square(x) = x^2$)

odd: $\mathbf{Z} \to \mathbf{IB}$ mit: odd(x) = true ⇔ x ist ungerade ⇔ x mod 2 = 1.

even: $\mathbf{Z} \to \mathbf{IB}$ mit: even(x) = true ⇔ x ist gerade ⇔ x mod 2 = 0.

Prüfen Sie in konkreten Programmiersprachen die genaue Bedeutung von div, mod und rem. Gesetzmäßigkeiten z.B.:
odd(x) = not even(x), ((-x) div y) = -1- (x div y), x mod y = y - ((-x) mod y).

5.1.8: (Fortsetzung)

Real: siehe 2.4.12 ff. Wertebereich **IR** (die reellen Zahlen sind die Vervollständigung der rationalen Zahlen bzgl. der Null-Folgen).

Bei den Operationen sind neben sgn, square, abs auch üblich:

sqrt = Positive Wurzel (square root) einer Zahl $x \geq 0$

log, ln, ld (Logarithmen zur Basis 10, e und 2), Exponentialfunktionen, die trigonometrischen Funktionen sowie die obere und untere Gaußklammer $\lceil\ \rceil, \lfloor\ \rfloor : \mathbf{IR} \to \mathbf{Z}$, definiert durch $\lceil x \rceil = \text{Min}\{ z \in \mathbf{Z} \mid x \leq z \}$,

$\lfloor x \rfloor = \text{Max}\{ z \in \mathbf{Z} \mid z \leq x \}$.

Es gilt stets $\lfloor x \rfloor \leq \lceil x \rceil$

und nur im Falle ganzer Zahlen x ist $\lfloor x \rfloor = x = \lceil x \rceil$.

Truncate-Funktion ("truncate" = abschneiden):

trunc: $\mathbf{IR} \to \mathbf{Z}$ mit trunc(x) = if $x \geq 0$ then $\lfloor x \rfloor$ else $\lceil x \rceil$ fi.

5.1.8: (Fortsetzung)

Verschiedene Programmiersprachen gehen von unterschiedlichen elementaren Datentypen aus und erzeugen hieraus dann recht verschiedene "Datenwelten". Bei **IB** und **A** sind wenigstens die Wertebereiche meist einheitlich, bei **Z** und **IR** müssen dagegen die unendlichen Wertebereiche durch endliche Teilmengen angenähert werden, was sowohl von der Hardware als auch von den jeweiligen Compilern oft zu unterschiedlichen Implementierungen führt.

Eine mögliche Realisierung haben Sie mit Ada 95 kennen gelernt. Informieren Sie sich bei jeder Programmiersprache, die Sie benutzen, stets als erstes über die genaue Definition der jeweiligen elementaren Datentypen!

Ein Beispiel sind die Prioritäten ("precedence"):

Prioritäten der gängigen Operatoren (in Ada aber anders, siehe 1.8.5):

Im "Alltag" lauten die Prioritäten meist:

1. Priorität: Absolutbetrag abs
2. Priorität: Exponentiation exp
3. Priorität: ·, div, mod, /, rem
4. Priorität: +, - (als einstellige Operationen)
5. Priorität: +, - (als zweistellige Operationen)
6. Priorität: =, ≠, <, ≤, >, ≥
7. Priorität: not
8. Priorität: and
9. Priorität: or
10. Priorität: equiv, xor
11. Priorität: impl

5.2 Abstrakte Datentypen

Wer Probleme zu lösen hat, interessiert sich erst in zweiter Linie um die Typen und deren detaillierte Implementierung. Zunächst stellt man Forderungen an die Lösungen, meist in Form von Eigenschaften. Von einem *Keller* (Stack, 3.5.4) erwartet man, dass er die Eigenschaft
"Was als letztes hingelegt wird, kommt als erstes wieder heraus"
d.h. die Gesetzmäßigkeit Pop(Push(K, A)) = K erfüllt.
Von einer *Warteschlange* W (queue, 3.5.5) erwarten wir, dass ihr erstes Element nicht durch Hinzufügen weiterer Elemente A verändert wird, d.h., es gilt:
First (W) = First (Enter (W, A)).
Die "Abstraktion" besteht darin, dass die Implementierung oder die Darstellung in einer Sprache nicht interessiert.

5.2.1: Dies führt zum "abstrakten Datentyp":

Ein abstrakter Datentyp wird durch Bezeichnungen für Mengen, durch Bezeichnungen von Abbildungen (einschl. ihrer Stelligkeiten) und durch die zu erfüllenden Gesetze charakterisiert.

Er abstrahiert somit von den konkreten Darstellungen (Datentypen, Implementierung von Unterprogrammen und anderen Programmeinheiten) und gibt nur die Gesetzmäßigkeiten an, denen die beteiligten Mengen genügen müssen.
Man kann ihn mit dem Schema 5.1.5 beschreiben; wir wählen allerdings im Folgenden eine eigene Formulierung.

Mathematisch gesehen ist ein abstrakter Datentyp somit eine (abstrakte) Algebra.

5.2.2 Schema zur Beschreibung eines abstrakten Datentyps:

structure < Name > is
(< Liste von Parametern und benötigten Einheiten >)
modes < nicht-leere Liste von Bezeichnern >
functions < Liste von Bezeichnern mit "Stelligkeiten" >
laws < Liste von logischen Aussagen (= Gesetzmäßigkeiten) >
end structure

Man kann auch andere Schlüsselwörter verwenden, z.B.:
sorts oder sets oder types anstelle von modes,
mappings, operators oder procedures anstelle von functions,
axioms, equations oder rules anstelle von laws.
Die Auflistungen können durch Komma oder Semikolon getrennt sein und die Abbildungen post- oder preorder (siehe 5.1.6) dargestellt werden.

In diesem Schema nennt man den Teil

modes <nicht-leere Liste von Bezeichnern>;
functions < Liste von Bezeichnern mit "Stelligkeiten" >

die Signatur des Datentyps.

Manchmal bezeichnet man auch die Signatur alleine schon als abstrakten Datentyp.

Man beachte, dass die Konstanten, die verwendet werden sollen, als nullstellige Funktionen dargestellt werden.

Wir betrachten als Beispiel die Wahrheitswerte "WHW1":

5.2.3: Beispiel Wahrheitswerte 1: Wir nehmen, wir benötigen von den Wahrheitswerten nur folgende Eigenschaften:

```
structure WHW1 is
modes  B                          -- dies steht für Menge der Wahrheitswerte
functions  wahr () B;
           not (B) B;
           and, or, impl (B, B) B;
           ifthenelsefi (B, B, B) B
laws       ASS: and (and (x,y), z) = and (x, and (y,z));
           KOMM: and (x, y) = and (y, x);
           NEG: not (not (x)) = x;
           IMP: impl (x, y) = or (not (x), y);
           IF1: ifthenelsefi (wahr, x, y) = x;
           IF2: ifthenelsefi (not(wahr), x, y) = y
end structure
```


Ordnet man den Sorten Mengen und den Funktionen Abbildungen entsprechend der vorgegebenen Stelligkeiten zwischen diesen Mengen zu und sind dann alle Gesetze erfüllt, so nennt man diese Mengen mit ihren Abbildungen (also diese "konkrete Algebra") ein Modell des abstrakten Datentyps.

Man nennt diese Mengen mit ihren Abbildungen auch einen zugehörigen konkreten Datentyp oder eine Konkretisierung oder eine Ausprägung oder eine Instanz.

Besitzt ein abstrakter Datentyp (bis auf Isomorphie) nur ein Modell, so heißt er monomorph, anderenfalls polymorph.

Zu obigem abstrakten Datentyp WHW1 gibt es mehrere Modelle, also mehrere konkrete Mengen mit Abbildungen, die alle Gesetze erfüllen. Zum Beispiel:

Modell 1: Betrachte den abstrakten Datentyp WHW1.
Für die Sorte B wählen wir die Menge $\mathbb{B}$ = {false, true}, wahr werde durch true und not(wahr) durch false dargestellt und die Funktionen not, and, or, impl und ifthenelsefi entsprechen den Funktionen Negation, Konjunktion, Disjunktion, Implikation und Alternative auf $\mathbb{B}$ (vgl. 5.1.8). Dann sind alle Gesetze erfüllt, wie man leicht nachprüft.

Doch dieses Modell ist nicht das einzige. Vielmehr ist der abstrakte Datentyp WHW1 polymorph, d.h., er hat viele Modelle. Ein weiteres Modell ist:

Modell 2: Betrachte erneut den abstrakten Datentyp WHW1.
Für die Sorte B wählen wir die einelementige Menge $\mathsf{E} = \{0\}$,
wahr werde durch 0 dargestellt und die Funktionen not, and, or,
impl und ifthenelsefi liefern für alle Argumente den Wert 0.
Dann sind ebenfalls alle Gesetze erfüllt, z.B.:

Für das Gesetz

impl (x, y) = or (not (x), y);

setzen wir alle möglichen Werte ein (hier ist nur der Wert 0 möglich):

0 = impl (0,0) = or (not (x), y) = or (0,0) = 0.

Es gibt zu WHW1 sogar unendlich viele verschiedene Modelle. Ein weiteres ist:

Modell 3: Betrachte erneut den abstrakten Datentyp WHW1.
Für die Sorte B wählen wir die ganzen Zahlen $\mathbb{Z}$,
wahr werde durch die Konstante 1 dargestellt und die
Funktionen werden repräsentiert durch:

not (z) = - z (das Negative einer ganzen Zahl),
and (a,b) = a + b (Addition),
or (a,b) = a + b (Addition)
impl (a,b) = b - a (Subtraktion der ersten Zahl von der zweiten)
ifthenelsefi (1,b,c) = b und ifthenelsefi (a,b,c) = c für a≠1.
Dann sind ebenfalls alle Gesetze erfüllt, z.B.:
impl (x, y) = y - x = (-x) + y = or (not (x), y).

Kann man den abstrakten Datentyp auch monomorph machen, also so formulieren, dass es im Wesentlichen nur noch *ein* Modell zu ihm gibt? Versuchen wir es mit dem Beispiel Wahrheitswerte 2 (WHW2):


```
structure WHW2 is
modes  B
functions  wahr () B;      falsch () B;          -- 5 Funktionen
           not (B) B;      and (B, B) B;
           or (B, B) B
laws       ZWEI: wahr ≠ falsch;                  -- ≥ 2 Elemente
           A1: and (wahr,x) = x;                 -- A1 bis A3
           A2: and (x,wahr) = x;                 -- legen and fest
           A3: and (falsch,falsch) = falsch;
           OR1: or (falsch,x) = x;               -- OR1 bis OR3
           OR2: or (x,falsch) = x;               -- legen or fest
           OR3: or (wahr,wahr) = wahr;
           N1: not(falsch) = wahr;               -- N1 und N2
           N2: not(wahr) = falsch                -- legen not fest
end structure
```


Setze nun die Menge $\mathbb{B}$ = {false, true} für die Sorte B;
wahr werde durch true und falsch durch false dargestellt und
die Funktionen not, and und or entsprechen den Funktionen
Negation, Konjunktion und Disjunktion auf $\mathbb{B}$.
Dann sind alle Gesetze erfüllt.

Da die Funktionen not, and und or in den Gesetzen komplett wie in einer Funktionstabelle festgelegt sind, gibt es im Wesentlichen auch kein zweites hiervon verschiedenes Modell.

Aber es gibt natürlich Erweiterungen (oder Einbettungen in größere Bereiche).

Setze hierfür die Menge der ganzen Zahlen **Z** für die Sorte B; wahr werde durch 1 und falsch durch 0 dargestellt. Für die Funktionen wähle man:

not (x) = 1 - x

and (x, y) = x · y

or (x,y) = 1 - ((1-x) · (1-y))

Hier wurde **IB** in die Menge **Z** geeignet eingebettet: Mit den Operationen and, or und not kann man die Teilmenge {0, 1} nicht verlassen; die Operationen sind allerdings auf ganz **Z** definiert und somit haben wir ein weiteres Modell für WHW2.

Um solche erweiterten Modelle auszuschließen, kann man zwei Wege beschreiten:

a) Man fügt ein Gesetz der Form "Es gibt höchstens zwei Elemente" hinzu. In unserem Beispiel:
$x \in B \Rightarrow (x = \text{wahr} \vee x = \text{falsch})$.

b) Oder man erlaubt generell nur "erzeugbare Modelle", d.h., in den Modellen sind nur solche Mengen erlaubt, die sich aus den angegebenen Konstanten mit Hilfe der angegebenen Abbildungen erzeugen lassen.
In unserem Beispiel sind dies alle Werte, die man aus "wahr" und "falsch" mittels not, and und or erzeugen kann; man sieht, dass hierdurch keine neuen Werte hinzukommen, so dass damit nur das zweielementige Modell zugelassen ist.

5.2.4: Beispiel "längenbeschränkter Keller" (vgl. Duden Inf.)

Bei einem Keller (Stack) interessiert den Benutzer nicht die Implementierung, sondern nur "LIFO" = last-in-first-out, also die kellerartige Speicherungstechnik. Diese lässt sich durch Gleichungen beschreiben (d: Element, x: Stack):

top (push (d, x)) = d,
pop (push(d, x)) = x.

Wir modifizieren diesen Ansatz leicht, indem wir den "längenbeschränkten Keller" einführen, der nur bis zu "max" Elemente aufnehmen kann. "max" und die Sorte Δ der einzufügenden Elemente übergeben wir als Parameter; der Keller erhält die Sorte lbsΔ (= längenbeschränkter Stack mit Elementen aus Δ). Weiterhin mixen wir hier die Postorder-Notation bei den "functions" mit der Preorder-Notation in den Gesetzen.

structure LBK is (based on boolean, based on natural,
mode Δ, natural max: max > 0)
modes lbsΔ
functions () lbsΔ empty;
(lbsΔ) boolean isempty, isfull;
(lbsΔ) Δ top;
(lbsΔ) lbsΔ pop;
(Δ, lbsΔ) lbsΔ push;
(lbsΔ) natural length
laws LEER: (x = empty) ⇔ isempty(x);
VOLL: (length(x) = max) ⇔ isfull(x);
LIFO: not isfull(x) ⇒ pop(push(d,x)) = x;
KON: not isempty(x) ⇒ push(top(x), pop(x)) = x;
OBEN: not isfull(x) ⇒ top(push(d,x)) = d;
ANZ: not isfull(x) ⇒ length(push(d,x)) = length(x)+1;
ANZ0: length(empty) = 0
end structure

Diese Darstellung lässt sich relativ rasch in eine gewohnte Repräsentation z.B. von Ada übersetzen.

(In der Praxis werden Notationen für jede Anwendung vorab festgelegt, so dass Sie lernen müssen, sich zunächst mit den Darstellungen vertraut zu machen.)

Wie bei Prozeduren können wir nun hierzu einen konkreten Datentyp (also eine konkrete Implementierung) angeben, der die Programmierung der Abbildungen und ggf. Initialisierungen enthält. Dies kann man ebenfalls wie einen abstrakten Datentyp formulieren, wobei man ein Schlüsselwort (hier: implementation; in Ada benutzt man "body") verwendet. Wir wählen hier die Realisierung mit Hilfe eines Feldes.


```
structure LBK is (based on boolean, based on natural,
                  mode Δ, natural max: max > 0)
modes       lbsΔ
functions   () lbsΔ empty;
            (lbsΔ) boolean isempty, isfull;
            (lbsΔ) Δ top;
            (lbsΔ) lbsΔ pop;
            (Δ, lbsΔ) lbsΔ push;
            (lbsΔ) natural length
< wer möchte, kann hier auch alle laws nochmals angeben >
implementation    -- Implementierungsteil, hier: Ada-ähnlich
S: array (1 .. max) of Δ;
t: 0 .. max;
< Programmstücke für die Operationen siehe nächste Folie >
end structure;
```


Vorschlag für den Implementierungsteil, Ada-ähnlich formuliert:

```
implementation
S: array (1 .. max) of Δ;
t: 0 .. max;
procedure empty is begin t:= 0; end;
function isempty return boolean is  begin return (t = 0);  end;
function isfull return boolean is  begin return (t = max);  end;
function top return Δ is
        begin if isempty then „Fehlerabbruch“;
        else top:= S(t); end if; end;
procedure pop is
        begin if isempty then „Fehlerabbruch“;
        else t:= t-1; end if; end;
procedure push (d: in Δ) is
        begin if isfull then „Fehlerabbruch“;
        else t:= t+1; S(t):= d; end if; end;
function length return integer is  begin return t;  end;
begin empty; end;                          -- Initialisierung
```


Hinweise: Hier wurde die Sorte lbsΔ zwar in der Signatur aufgeführt, aber in der Implementierung stillschweigend durch

array (1 .. max) of Δ

ersetzt. Bei der Umsetzung in eine konkrete Programmiersprache kann dieser Zusammenhang deutlich gemacht oder verschwiegen ("private") werden.

Statt mit einem Feld hätte man den Datentyp LBK auch durch eine Liste realisieren können (siehe später).

In der Praxis muss auch der Zugriff auf die Sorten geklärt werden (in Ada: private, limited private). Weiterhin kann man auch abstrakte Datentypen als "abstract im Ada-Sinne" (= man darf keine konkreten Datentypen hiervon erzeugen) kennzeichnen. Weitere Beschränkungen sind in der Praxis üblich; sie hängen meist von der jeweiligen Anwendung ab.

5.2.5 Empfohlenes Vorgehen, um in der Praxis Programme zur Realisierung von Algorithmen zu schreiben.

Top-down-Phase:

(1) Zerlege den Algorithmus in kleinere Einheiten (in der Regel nicht mehr als sieben) und gib an, wie diese im Gesamtalgorithmus zusammenwirken. Gib für jede Einheit ihre Signatur an.

(2) Ermittle die Eigenschaften, die die Funktionen der Einheiten erfüllen müssen. Formuliere diese als Gesetze eines abstrakten Datentyps aus.

(3) Wiederhole (1) und (2) mit den Algorithmen, die von den einzelnen Einheiten realisiert werden, bis so kleine Einheiten entstanden sind, dass ihre Implementierung in der Programmiersprache leicht möglich ist.

Bottom-Up-Phase:

(4) Lege nun die Datenbereiche (möglichst als Moduln, in Ada als Pakete, siehe Abschnitt 4.3.4) als Typen in der gegebenen Programmiersprache fest und implementiere die zuletzt erhaltenen Algorithmen-Teile (als Prozeduren, Funktionen, Moduln oder kommunizierende Einheiten).

(5) Wiederhole (4) schrittweise für jeden Unter-Algorithmus, dessen gesamte Daten und Operationen bereits implementiert sind, bis schließlich der Gesamtalgorithmus implementiert ist.

5.3 Daten-Konstruktoren und Beispiele

5.3.1: Sind Datentypen gegeben, so kann man hieraus neue Datentypen erzeugen. Man orientiert sich hierbei an den mathematischen Operationen auf Mengen. Wichtige Operationen:

- Bildung von Unterbereichen (Intervalle, Projektion, "range"),
- kartesisches Produkt ("record"),
- kartesisches Produkt mit sich (Vektoren, Matrizen, "array"),
- disjunkte Vereinigung ("varianter record"),
- Folgen-Bildung (freies Monoid, Wörter-Bildung, "Listen"),
- Menge der Teilmengen (Potenzmenge, "set of"),
- Menge von Abbildungen (Verallgemeinerung der "function")
- Menge der Namen oder Adressen (reference, access).

5.3.2 Grundsätzliches Vorgehen

Zu jedem Konstruktor K, der auf k Datentypen T_i wirkt,

$K\,(T_1, T_2, ..., T_k)$

wird angegeben:

(i) Wie sieht die Wertemenge aus (bezogen auf die Wertemengen der Datentypen T_i)?

(ii) Wie kann man auf die einzelnen Komponenten T_i zugreifen?

(iii) Welche Operationen der Datentypen T_i werden übernommen und in welcher Form?

(iv) Werden neue Operationen hierdurch eingeführt? (Vor allem Umwandlungen in einen anderen Datentyp.)

Ein Beispiel, das Sie bereits kennen:

Der Konstruktor array wirkt auf zwei Datentypen:

array $(T_1; T_2)$

in Ada geschrieben als array (T_1) of T_2

Wenn T_1 die Wertemenge D mit n Elementen und T_2 die Wertemenge M besitzen, so ist M^D die zu array $(T_1; T_2)$ gehörende Wertemenge (bis auf Isomorphie).

Für ein beliebiges Element $a \in D$ greift man auf den zu a gehörenden Funktionswert in M zu mittels [a] oder auch (a) (in Ada: (a)).

Ein Element aus array $(T_1; T_2)$ ist somit eine als Tabelle dargestellte Abbildung von D nach M.

Großbuchstaben: G: array (1..26) of Character

1..26 → **A**

i	G(i)
1	'A'
2	'B'
3	'C'
4	'D'
5	'E'
6	'F'
7	'G'
8	'H'
9	'I'
10	'J'
11	'K'
...	...
25	'Y'
26	'Z'

Beantwortung der vier Fragen

Gegeben: zwei Datentypen T_1 und T_2 mit den Wertebereichen D und M. Der Daten-Konstruktor ist array (T_1; T_2).
V sei eine Variable von diesem neuen Datentyp.

(i) Wertemenge: M^D (= Menge der Abbildungen von D nach M).

(ii) Zugriff auf die Komponenten: V(i) für i∈ D.

(iii) Alle Operationen des Datentyps T_2 sind auch für V(i) zulässig. Statt i kann ein Ausdruck vom Typ T_1 stehen, also ein Ausdruck, der einen Wert aus D als Ergebnis hat.

(iv) Werden neue Operationen hierdurch eingeführt? Nein.

5.3.3 Unterbereiche (vgl. 1.9.1 und 2.4.20)

Unterbereich oder Intervall a..b = {x ∈ M | a ≤ x ≤ b}. In Ada

subtype <Name> is <Datentyp> [<constraint>]

z.B.: subtype Kohl_Aera is Integer 1982..1998;

Alle Operationen von <Datentyp> gelten auch für den durch <Name> bezeichneten Unterbereich (in Ada "Untertyp" genannt), sofern der Unterbereich hierbei nicht verlassen wird (man kann auch festlegen, dass bei den Zwischenrechnungen der Unterbereich nicht verlassen werden darf).

Der Unterbereich darf zunächst unbestimmt ("flex", Abk. von flexible) bleiben und die Grenzen werden dann erst bei der Konkretisierung festgelegt (in Ada gibt es hierfür *range* < >).

Beantwortung der vier Fragen für Unterbereiche

Gegeben: ein Datentyp T mit dem Wertebereich M. Der Daten-Konstruktor ist T(unten..oben). V sei eine Variable von diesem neuen Datentyp.

(i) Wertemenge: unten..oben.

(ii) Zugriff auf die Komponenten: entfällt, V enthält den Wert.

(iii) Alle Operationen des Datentyps T sind auch für V zulässig. Wird V ein Wert zugewiesen, so muss geprüft werden, ob er im Intervall unten..oben liegt.

(iv) Werden neue Operationen hierdurch eingeführt? Nein.

Anmerkung zu Unterbereichen:

Ein Unterbereich a..b ist eine "linear zusammenhängende" Teilmenge von M. Man könnte Unterbereiche auch als eine beliebige Teilmenge einführen, indem man in

subtype <Name> is <Datentyp> [<constraint>]

die Einschränkung <constraint> durch eine logische Formel (ein sog. "Prädikat") ersetzt. Zum Beispiel:

subtype Gerade is Integer: (Gerade(z) <=> z mod 2 = 0);

Ein Intervall zum Typ T wäre dann beschrieben durch:

subtype a..b is T: (a..b (z) <=> a ≤ z and z ≤ b);

Man kann sich hier viel ausdenken, allerdings werden der Compiler und seine Implementierung recht schwierig, sodass man sich bei Unterbereichen fast immer auf Intervalle beschränkt. (Vgl. die Mengen-Programmiersprache SETL.)

5.3.4 Felder (1.9 und 2.4.21)

Gegeben ein Datentyp T mit Wertemenge M und eine natürliche Zahl r. Datentyp für das r-fache kartesische Produkt M^r mit sich:

array (1..r) of T.

1..r heißt der *Indextyp*, T heißt der *Komponententyp*.

Hierbei darf T erneut eine Felddeklaration sein usw.:

array (1..r) of array (1..s) of T
array (1..r) of array (1..s) of array (1..t) of T

Dies kürzt man meist wie folgt ab: array (1..r,1..s,1..t) of T.
Statt 1..r kann auch ein anderer Indextyp verwendet werden.

Die Anzahl dieser Indexbereiche nennt man die Dimension des Feldes. In der Praxis sind n-dimensionale Felder meist für n=1, 2 und 3 üblich (z.B. Vektoren, Matrizen, Bildpunkte).

Beantwortung der vier Fragen für Felder

Datentyp T mit den Wertebereich M. Datentyp für M^r. V sei eine Variable von diesem neuen Datentyp.

(i) Wertemenge: M^r

(ii) Zugriff auf die i-te Komponente: V(i), sofern 1..r der Indexdatentyp ist (sonst das i-te Element des Indexdatentyps in die Klammer schreiben). Meist ist auch der Zugriff auf die ganze Struktur erlaubt (Aggregatbildung). Bei n Dimensionen: $V(i_1, i_2, ..., i_n)$.

(iii) Auf jeder Komponente bleiben die Operationen des Datentyps T erhalten.

(iv) Werden neue Operationen hierdurch eingeführt? Nein. (Außer der Verwendung von Aggregaten für komplette Zuweisungen und Abfragen.)

5.3.5 Kartesische Produkte (Datensätze, siehe 1.12.1)

Gegeben seien die Mengen $M_1, M_2, ..., M_n$, die zu den Datentypen $T_1, T_2, ..., T_n$ gehören. Die Zusammenfassung zum Datentyp T mit der Wertemenge $M = M_1 \times M_2 \times ... \times M_n$ ist:

type T is record S_1: T_1; S_2: T_2; ...; S_n: T_n; end record;

Hierbei sind $S_1, S_2, ..., S_n$ Namen, die so genannten "Selektoren", mit deren Hilfe man auf die einzelnen Komponenten des Datentyps T zugreifen kann. Man "selektiert" die i-te Komponente, indem man S_i durch einen Punkt getrennt hinter den Namen der Variable vom Typ T hängt (Punkt-Schreibweise, *dot-notation*).

Beantwortung der vier Fragen für kartesische Produkte

n Datentypen $T_1, T_2, ..., T_n$ mit den Wertebereichen $M_1, M_2, ..., M_n$. Datentyp T für $M_1 \times M_2 \times ... \times M_n$. V sei eine Variable von diesem neuen Datentyp T.

(i) Wertemenge: $M = M_1 \times M_2 \times ... \times M_n$

(ii) Zugriff: $V.S_i$ für die i-te Komponente. (Meist ist auch der Zugriff auf die ganze Struktur erlaubt, Aggregatbildung.)

(iii) Auf jeder Komponente bleiben die Operationen des zugehörigen Typs T_i erhalten.

(iv) Werden neue Operationen hierdurch eingeführt? Nein. (Außer der Verwendung von Aggregaten für komplette Zuweisungen und Abfragen.)

5.3.6 Disjunkte Vereinigungen (variante records, 1.12.2)

Gegeben: M_1, M_2, ..., M_n, die zu den Datentypen T_1, T_2, ..., T_n gehören. Zusammenfassung zum Datentyp T mit der disjunkten Vereinigung $M_1 \cup M_2 \cup ... \cup M_n$ als Wertemenge:

```
type T (Index: Integer range 1..n) is record
        case Index is
                when 1 => S1: T1;
                when 2 => S2: T2;   ...

                when n => Sn: Tn;
        end case;
end record;
```

Index gibt den aktuellen Wertebereich innerhalb von M an. Mit dem Selektor S_{Index} kann man auf den aktuellen Wert zugreifen.

Beantwortung der vier Fragen für disjunkte Vereinigungen

n Datentypen $T_1, T_2, ..., T_n$ mit den Wertebereichen $M_1, M_2, ..., M_n$. Datentyp T für $M_1 \cup M_2 \cup ... \cup M_n$. V sei eine Variable von diesem neuen Datentyp T.

(i) Wertemenge: $M = M_1 \cup M_2 \cup ... \cup M_n$ zuzüglich 1..n.

(ii) Zugriff: $V.S_i$, sofern der Index von V gleich i ist. Dieser Wert i ist durch V.Index zu erreichen und änderbar.

(iii) Auf jeder "Komponente" bleiben die Operationen des zugehörigen Typs T_i erhalten.

(iv) Werden neue Operationen hierdurch eingeführt? Nein.

5.3.7 Folgenbildung

Zu einer Menge M sei M* die Menge der endlichen Folgen (auch Menge der Wörter über M genannt):

$M^* = \{a_1 a_2 ... a_n \mid n \geq 0 \text{ und } a_i \in M \text{ für } i = 1, 2, ..., n\}$.

M möge zum Datentyp T gehören. Wir definieren den Datentyp seq of T durch folgende Festlegungen entsprechend 5.1.5:

Zugrunde liegende Menge: M*

Nullstellige Operationen: Alle Elemente von M*. Man schreibt diese Wörter auf, indem man sie entweder in Anführungsstriche setzt und die Elemente von M durch Zwischenräume trennt oder indem man sie als Vektoren $(a_{i_1}, a_{i_2}, ..., a_{i_n})$ notiert. Das leere Wort erhält hierbei die Darstellung ε oder ( ). Manchmal schreibt man auch null oder nil für das leere Wort. Wir werden in diesem Abschnitt die Vektorschreibweise verwenden.

Einstellige Operationen

Für empty, square, removefirst, removelast: $M^* \to M^*$ gilt:

empty(u) = ε für alle $u \in M^*$,

square$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = (a_{i_1}, a_{i_2}, ..., a_{i_n}, a_{i_1}, a_{i_2}, ..., a_{i_n})$,

removefirst$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = (a_{i_2}, ..., a_{i_n})$,

removelast$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = (a_{i_1}, a_{i_2}, ..., a_{i_{n-1}})$.

Für das leere Wort sind removefirst und removelast undefiniert.

first, last: $M^* \to M$ sind definiert durch

first$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = a_{i_1}$, sofern $n > 0$; first(( )) ist undefiniert,

last$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = a_{i_n}$, sofern $n > 0$; last(( )) ist undefiniert.

Statt "first" schreibt man meist "head", statt removefirst "tail".

Einstellige Operationen (Fortsetzung)

in: $M \to M^*$ mit in(b) = (b), für alle $b \in M$ (Einbettung).

length: $M^* \to \mathbb{N}_0$ ist definiert durch
length$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = n$; speziell gilt length$(()) = 0$.
Für jedes $a \in M$ ist $\#_a: M^* \to \mathbb{N}_0$ die Anzahlfunktion für das Element a, d.h., $\#_a(u)$ = Anzahl, wie oft a in u vorkommt.
Üblicherweise definiert man $\#_a$ rekursiv:
$\#_a(()) = 0$ und für alle $u \in M^*$ und alle $b \in M$:
$\#_a(ub) = \#_a(u) + 1$, falls a = b ist,
$\#_a(ub) = \#_a(u)$, falls $a \neq b$ ist,

isempty: $M^* \to \mathbb{B}$ ist definiert durch
isempty(u) = true genau dann, wenn u das leere Wort ist.

Zweistellige Operationen

conc: $M^* \times M^* \to M^*$ (Konkatenation) ist definiert durch
conc$((a_{i_1}, a_{i_2}, ..., a_{i_n}), (b_{j_1}, b_{j_2}, ..., b_{j_m})) = (a_{i_1}, ..., a_{i_n}, b_{j_1}, ..., b_{j_m})$
append: $M^* \times M \to M^*$ (Anhängen eines Elements) ist definiert durch append$((a_{i_1}, a_{i_2}, ..., a_{i_n}), b) = (a_{i_1}, ..., a_{i_n}, b)$.

Die Vergleichsoperationen $=, \neq: M^* \times M^* \to \mathbb{B}$ sind wie üblich definiert. Falls M eine geordnete Menge ist, so kann man auch die anderen Vergleichsoperationen $<, \leq, >$ und $\geq$ verwenden.
(Man achte aber genau darauf, wie die Ordnung von M auf M^* fortgesetzt wurde! Beispiel: lexikografisch oder längenlexikografisch.)

Beziehungen, Gesetzmäßigkeiten

Beispiele für Gesetzmäßigkeiten sind (stets für alle $u \in M^*$):
removelast(append(u, b)) = u für alle $b \in M$.
conc(u, u) = square(u).

$$\text{length}(u) = \sum_{a \in M} \#_a(u).$$

not isempty(append(u,b)) für alle $b \in M$.

length(conc(u, v)) = length(u) + length(v) für alle $v \in M^*$.

first(in(b)) = last(in(b)) = b für alle $b \in M$.

Die Datenstruktur "seq of T" in der Sprache Ada

Folgen werden in Ada durch "Listen" dargestellt, die mittels Zeigern (also mit access - Datentypen) realisiert werden.

Für die Operationen müssen geeignete Prozeduren und Funktionen geschrieben werden, sofern nicht ein vordefiniertes Paket für die "Listenverarbeitung" existiert.

Die Realisierung mit dynamischen Datenstrukturen (Objekte in der Halde) wurde bereits in 3.5 genauer beschrieben.

Ein Spezialfall sind Folgen über dem Latin-1-Alphabet. In Ada gibt es hierfür den vordefinierten Datentyp "string".

5.3.8 Potenzmengen

Zu jeder Menge M kann man die Menge 2^M ihrer Teilmengen konstruieren: $2^M = \{ M' \mid M' \subseteq M\}$. Wegen $\emptyset \in 2^M$ und $M \in 2^M$ ist die Potenzmenge einer Menge niemals leer.

M möge zum Datentyp T gehören. Wir definieren den Datentyp set of T durch folgende Festlegungen:

Zugrunde liegende Menge: 2^M

Nullstellige Operationen: Wichtige "Konstanten" sind die leere Menge $\emptyset$ und die gesamte Menge M. Grundsätzlich sollte man jede endliche Teilmenge von M notieren können.

Eine Teilmenge M' schreibt man auf, indem man ihre Elemente auflistet oder indem man einen |M|-stelligen Booleschen Vektor b': array T of Boolean benutzt mit: $b'(m) = true \Leftrightarrow m \in M'$.

(Andere Darstellungsmöglichkeiten lernen wir noch kennen.)

Einstellige Operationen

compl: $2^M \to 2^M$ (Komplement einer Teilmenge) mit $compl(M') = \{m \in M \mid m \notin M'\} \in 2^M$.

isempty: $2^M \to \mathbb{B}$ mit $isempty(M') = true \Leftrightarrow M' = \emptyset$

Zweistellige Operationen

Durchschnitt $\cap$, Vereinigung $\cup$, Differenz \: $2^M \times 2^M \to 2^M$ sind wie üblich definiert, vgl. Einschub in 2.12.

Auch die Vergleichsoperationen und die Elementbeziehungen $=, \neq, \subset, \subseteq, \supset, \supseteq, \not\subset$: $2^M \times 2^M \to \mathbb{B}$ und $\in, \notin$: $M \times 2^M \to \mathbb{B}$ sind zweistellige Operationen.

Beziehungen, Gesetzmäßigkeiten

Es gelten die Gesetze der Booleschen Algebra, z.B.:
$compl(compl(M')) = M'$,
$M' \cap M'' = M'' \cap M'$, $M' \cup M'' = M'' \cup M'$ usw.

"$\subseteq$" ist eine Ordnungsrelation (reflexiv, antisymmetrisch und transitiv).

Weitere Beziehungen sind z.B.:
$M' \cap M'' = M' \Leftrightarrow M' \subseteq M''$
Die Operation $M'\Delta M'' = (M'\backslash M'') \cup (M''\backslash M')$ ist assoziativ und kommutativ ("symmetrische Differenz").

Die Datenstruktur "set of T" ist in der Sprache Ada nicht direkt vorgesehen. Man muss sie mit anderen Strukturen simulieren.

5.3.9 Abbildungen (vgl. Einschub in 2.12)

Die Menge der Abbildungen von M nach N wird mit N^M oder mit $Abb(M,N) = \{f \mid f: M \to N\}$ bezeichnet.

Wir müssen unterscheiden zwischen der Menge der totalen N^M Abbildungen von M nach N und der Menge aller Abbildungen (also auch der partiellen Abbildungen, die nicht auf dem gesamten Vorbereich M, sondern nur auf einer Teilmenge definiert sind). In der Programmierung geht man meist von der Menge aller Abbildungen aus. Um anzugeben, dass die Funktionen auch partiell sein können, fügt man oft ein "p" (wie "partiell") an: $Abb_p(M,N) = \{f \mid f: M \to N\}$.

M möge zum Datentyp T, N zum Datentyp U gehören. Wir definieren den Datentyp mapping (T,U) entsprechend 5.1.5:

Zugrunde liegende Menge: $Abb_p(M,N)$

Nullstellige Operationen: Dies sind feste Abbildungen, z.B. der Sinus sin: $\mathbb{R} \to \mathbb{R}$ oder der größte gemeinsame Teiler ggT: $\mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ oder die überall undefinierte Funktion $f_\perp$: M → N mit: $f_\perp(m)$ ist undefiniert für alle m∈ M.
(Man realisiert $f_\perp$ leicht durch eine unendliche Schleife.)

Einstellige Operationen: Dies sind Abbildungen der Form
$F: Abb_p(M,N) \to Abb_p(M,N)$.
Im Falle M= N ist ein wichtiges Beispiel hierfür "$^{-1}$", also die Bildung der Inversen, das heißt, wenn f: M → M eine injektive Abbildung ist, so ist f^{-1}: M → M die Abbildung: $f^{-1}(b) = a$, falls f(a) = b, und undefiniert sonst. (Falls f nicht injektiv ist, kann man oft f^{-1} dennoch definieren, indem man irgendein Urbild a mit f(a) = b für $f^{-1}(b) = a$ fest auswählt.)

Einstellige Operationen (Hinweis)

Ein Beispiel für einstellige Funktionsabbildungen sind "Ableitung" und "Stammfunktion", d.h., wenn M = N = $\mathbb{R}$, dann sind das unbestimmte Integral und die abgeleitete Funktion partielle Abbildungen auf der Menge der reellwertigen Funktionen, also z.B.:
$\int: Abb_p(\mathbb{R},\mathbb{R}) \to Abb_p(\mathbb{R},\mathbb{R})$.
Partiell ist das Integral, weil nicht zu jeder reellwertigen Abbildung eine Stammfunktion existiert (in der Regel benötigt man eine gewisse Stetigkeit als Voraussetzung). Ebenso lässt sich nicht jede reellwertige Funktion differenzieren.

Zweistellige Operationen:

Hier sind vor allem zu nennen die Hintereinanderausführung von Abbildungen
$H: Abb_p(M,M) \times Abb_p(M,M) \to Abb_p(M,M)$
mit H(f,g) = f∘g und die n-fache Iteration einer Abbildung
$I: Abb_p(M,M) \times \mathbb{N}_0 \to Abb_p(M,M)$
mit $I(f,n) = f^n$ (vgl. 2.5.5).

Realisierung in Programmiersprachen: In der Sprache ALGOL 68 war dieser Datentyp vorgesehen, er lässt sich aber dort nur schwer implementieren. In funktionalen Sprachen (LISP usw.) lassen sich dagegen Abbildungen gut darstellen und manipulieren.
In Ada sind die "nullstelligen Operationen", also die Deklarationen konkreter Funktionen, vorhanden. Zusätzlich lassen sich mit dem generic-Konzept und mit "Zeigern auf Prozeduren" Funktionen über Funktionen simulieren.

Generelle Bemerkung: *(Durchdenken Sie dies selber.)*

Alle Operationen, die auf M oder auf N definiert sind, kann man auf Funktionen ausdehnen. Wenn beispielsweise auf N eine Operation $\bar{\circ}: N \times N \to N$ definiert ist, so kann man diese Operation fortsetzen zu
$\bar{\circ}: Abb_p(M,N) \times Abb_p(M,N) \to Abb_p(M,N)$
durch $f \,\bar{\circ}\, g$. Genauer: für alle a ∈ M setze
$(f \,\bar{\circ}\, g)(a) = f(a) \,\bar{\circ}\, g(a)$.
Beispiele für solche Operationen $\bar{\circ}$ sind Addition, Subtraktion, Multiplikation, Maximum und Minimum usw.
Man kann auch Vergleiche von M auf $Abb_p(M,M)$ übertragen, z.B.: $f \leq g \Leftrightarrow$ für alle a ∈ M gilt $f(a) \leq g(a)$.
Hierbei geht eine lineare Ordnung jedoch in der Regel nur in eine partielle Ordnung über.

Definition 5.3.10: Eine lineare Liste heißt Doppelschlange (engl.: deque), wenn auf ihr genau die folgenden acht Operationen zugelassen sind:

(1) Empty = Leeren der Liste.

(2) Isempty = Abfragen auf Leerheit der Liste.

(3) First = Kopieren des ersten Elements der Liste.

(4) Last = Kopieren des letzten Elements der Liste.

(5) Enter_Front = Hinzufügen eines Elements am Anfang.

(6) Remove_First = Löschen des ersten Elements der Liste.

(7) Enter_End = Hinzufügen eines Elements am Ende.

(8) Remove_Last = Löschen des letzten Elements der Liste.

Aufgabe: Formulieren Sie einen zugehörigen abstrakten Datentyp aus.

6. Komplexität von Algorithmen und Programmen

Lernziele dieses Kapitels:

Zu jedem Algorithmus und zu jedem Programm gehört der Aufwand an Zeit, Speicherplatz und sonstigen Ressourcen. Dies kann man durch Aufwandsfunktionen (in Abhängigkeit von der Länge der Eingabe oder von der Anzahl der zu bearbeitenden Elemente) beschreiben.

Den Aufwand beschreibt man über die Größenordnung. Wir stellen zunächst die entsprechenden Funktionen / Formalismen vor. Hiermit schätzen wir den Zeitaufwand von Algorithmen und Programmen (in der Regel "uniform") ab, wobei wir entweder direkt die einzelnen Operationen und Bedingungen zählen oder Gleichungen aufstellen und lösen.

Basis der Komplexität ist das Berechnungsmodell: Üblich sind die zeichenorientiert arbeitende Turingmaschine und die zahlenorientiert arbeitende Registermaschine. Sie müssen mindestens eines dieser beiden Modelle kennen und wissen, wie der Aufwand letztlich hierauf zurückgeführt und berechnet werden kann.
Die Standardbeispiele ggT, Spiegeln, Palindrom, Suchen durch Intervallschachtelung, Multiplikation in $O(n^{\log(3)})$ und TSP müssen Ihnen am Ende des Kapitels bezüglich ihrer Komplexität geläufig sein.

6.1 Aufwandsfunktionen

Wer Programme einsetzt, fragt meist nach der Zeitspanne, innerhalb derer die Ergebnisse ausgegeben werden. Man spricht von der "Zeitkomplexität" des Programms.

Diese "Rechendauer" ist abhängig von der Eingabe. In der Regel definiert man den Zeitaufwand $t_\pi: \mathbb{N}_0 \to \mathbb{N}_0$ als Funktion der Länge der Eingabe des Programms π:
$t_\pi(n)$ = maximale Zeit, die das Programm π für irgendeine Eingabe w der Länge n bis zum Anhalten benötigt.

t_π ist natürlich nur dann eine totale Funktion, wenn das Programm π stets terminiert. Meist verlangt man diese Terminierung, wenn man die Zeitkomplexität berechnet. Wir betrachten zunächst ein Beispiel.

Beispiel

```
program was is
declare  A, B: Natural;
begin Get (A);
      B := 1;
      while A>1 loop A:=A div 2; B:=B+1; end loop;
      Put (B);
end
```

Mit Ablaufprotokollen ermittelt man einige Werte der realisierten Funktion $g: \mathbb{N}_0 \to \mathbb{N}_0$:

a	g(a)	a	g(a)	a	g(a)
0	1	4	3	8	4
1	1	5	3	16	5
2	2	6	3	80	7
3	2	7	3	1024	11


```
program was is
declare  A, B: Natural;
begin Get (A);
      B := 1;
      while A>1 loop A:=A div 2; B:=B+1; end loop;
      Put (B);
end
```

Man vermutet nun, dass die realisierte Funktion $g: \mathbb{N}_0 \to \mathbb{N}_0$ lautet: g(a) = Länge der binären Darstellung der Zahl a.

Dies trifft zu, weil in jedem Schritt die Länge der Zahl A durch die Wertzuweisung A := A div 2 genau um eins verringert wird, bis eine Darstellung der Länge 1 erreicht ist. Die Schleife wird einmal weniger, als die Länge angibt, durchlaufen. Da B anfangs auf 1 gesetzt wurde, wird daher genau die Länge der Binärdarstellung der Eingabe berechnet.


```
program was is
declare  A, B: Natural;
begin Get (A);
      B := 1;
      while A>1 loop A:=A div 2; B:=B+1; end loop;
      Put (B);
end
```

Wie lange dauert nun die Berechnung? Wir nehmen an: Die Auswertung jedes Ausdrucks und die Durchführung jeder Wertzuweisung dauern gleich lange. Dann erhält man (es sei n die Länge der Binärdarstellung der Eingabezahl a):

$1 + 1 + (n-1) * (1 + 1 + 1) + 1 + 1 = 3n + 1$ Zeiteinheiten.

Die Größenordnung ist also proportional zu n *(man sagt, sie sei O(n))*, und meist interessiert nur diese Abschätzung, bei der multiplikative und additive Konstanten ignoriert werden. ■

Um die Größenordnung einer reellwertigen oder ganzzahligen Funktion zu beschreiben, verwenden wir die so genannten *Landau-Symbole* (nach dem deutschen Mathematiker Edmund Landau, 1877-1938). Hierbei werden multiplikative und additive Konstanten vernachlässigt; es wird nur *der* Term in Abhängigkeit von n, der für $n \to \infty$ alles andere überwiegt, berücksichtigt.

Formal gesehen handelt es sich bei O(f) um die Definition einer Funktionenklasse in Abhängigkeit von einer Funktion f. In O(f) sind alle Funktionen über den reellen Zahlen enthalten, die "schließlich von f dominiert" werden.

Insgesamt verwendet man folgende 5 Klassen O, o, Ω, ω und Θ, wobei von uns am häufigsten " O " eingesetzt wird.

Definition 6.1.1: "groß O", "klein O", "groß Omega", klein Omega", "Theta"

Es sei $f: \mathbb{R}^+ \to \mathbb{R}^+$ eine Funktion über den positiven reellen Zahlen $\mathbb{R}^+ \subset \mathbb{R}$ (oft wird diese Definition auf die natürlichen Zahlen eingeschränkt, also auf Funktionen $f: \mathbb{N} \to \mathbb{N}$; unten muss dann nur $g: \mathbb{R}^+ \to \mathbb{R}^+$ durch $g: \mathbb{N} \to \mathbb{N}$ ersetzt werden).

$\mathbf{O(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0: g(n) \leq c \cdot f(n)\}$,

$\mathbf{o(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0: c \cdot g(n) \leq f(n)\}$,

$\mathbf{\Omega(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0: f(n) \leq c \cdot g(n)\}$,

$\mathbf{\omega(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0: c \cdot f(n) \leq g(n)\}$,

$\mathbf{\Theta(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0: c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$.

Landau-Symbole

Erläuterungen 6.1.2: Es sei $f: \mathbb{R}^+ \to \mathbb{R}^+$

g liegt in **O(f)**, wenn g höchstens so stark wächst wie f, wobei Konstanten nicht zählen. Statt *g ist höchstens von der Größenordnung f*, sagen wir, *g ist groß-O von f,* und meinen damit, dass $g \in O(f)$ ist.

Wenn eine Funktion g zusätzlich echt schwächer als f wächst, wenn also zusätzlich gilt:

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

dann sagen wir, *g ist von echt kleinerer Größenordnung als f* oder *g ist klein-o von f,* und meinen damit, dass $g \in$ **o(f)** ist.

Erläuterungen (Fortsetzung) 6.1.2: Es sei $f: \mathbb{R}^+ \to \mathbb{R}^+$

Die Klassen Ω und ω (groß Omega und klein Omega) bilden die "Umkehrungen" der Klasse O und o. Eine Funktion g liegt genau dann in **Ω(f)** bzw. in **ω(f)**, wenn f in O(g) bzw. in o(g) liegt.

In $\Omega(f)$ liegen also die Funktionen, die mindestens so stark wachsen wie f, und in $\omega(f)$ liegen die Funktionen, die zusätzlich bzgl. n echt stärker wachsen.

In der Klasse **Θ(f)** liegen die Funktionen, die sich bis auf Konstanten im Wachstum wie f verhalten. Wenn $g \in \Theta(f)$ ist, dann sagen wir, *g ist von der gleichen Größenordnung wie f* oder *g ist Theta von f.* Da in diesem Fall g in O(f) und f in O(g) liegen müssen, folgt unmittelbar die Gleichheit $\Theta(f) = O(f) \cap \Omega(f)$.

Schreibweisen 6.1.3: Es sei $f: \mathbb{R}^+ \to \mathbb{R}^+$

Statt $g \in O(f)$ schreibt man manchmal auch $g = O(f)$, um auszudrücken, dass g höchstens von der Größenordnung f ist. Das Gleiche gilt für die anderen vier Klassen.

Anstelle der Funktionen gibt man meist nur deren formelmäßige Darstellung an. Beispiel: Statt

O(f) für die Funktion f mit $f(n) = n^2$ für alle $n \in \mathbb{N}$

schreibt man einfach $O(n^2)$.

Man schreibt auch "Ordnungs-Gleichungen", die aber nur von links nach rechts gelesen werden dürfen, z.B.:

$3 \cdot n^3 + 12 \cdot n^2 + 8 \cdot n \cdot \log(n) + 6/n = 3 \cdot n^3 + O(n^2) = O(n^3)$.

Korrekt müsste man hierfür beispielsweise schreiben:

$3 \cdot n^3 + 12 \cdot n^2 + 8 \cdot n \cdot \log(n) + 6/n \in O(3 \cdot n^3) \cup O(n^2 + n \cdot \log(n)) = O(n^3)$.

6.1.4: Einige Klassen

O(1) ist die Klasse der Funktionen, die höchstens wie ein Vielfaches der konstanten Funktion *f(n) = 1 für alle n∈IN* wachsen. Somit gehören alle konstanten Funktionen, aber auch Funktionen wie sin(n), cos(n), 1/n, $1/n^2$ oder 1/log(n) zu O(1).

Gibt es eine Klasse von Funktionen, die nicht in O(1) liegen und nur sehr schwach wachsen, also deutlich langsamer als f(n) = n ? Aus der Schule kennen Sie den Logarithmus log(n). Noch wesentlich schwächer wächst der "iterierte Logarithmus":
log*(n) = 0, für n=0 und 1,
log*(n) = Min{k | log(log(log(...log(n)...))) < 2} für n>1.

k ineinander geschachtelte Logarithmen

[Untersuchen Sie diese Funktion log*(n), vgl. danach auch 6.8.]

O(n) = Klasse der höchstens linear wachsenden Funktionen:
O(n) := $\{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c\in\mathbb{R}^+ \exists\, n_0\in\mathbb{N} \forall n\geq n_0: g(n) \leq c\cdot n\}$.
Man beachte, dass hierin auch alle Funktionen der Form $g(n) = c_1\cdot n + c_2$ (für zwei positive Konstanten c_1 und c_2) enthalten sind, weil für n≥1 gilt: $g(n) = c_1\cdot n + c_2 \leq (c_1+ c_2)\cdot n$.
Wenn g in O(n) liegt, so sagt man auch, g sei *höchstens linear*.

Zu den höchstens linear wachsenden Funktionen gehört (wegen log(x) < x für alle x>0) auch der Logarithmus. Es gilt daher: log(n) ∈ O(n). Aber auch für die Potenzen des Logarithmus gilt $\log^m(n) \in O(n)$ für alle natürlichen Zahlen m. Hierfür beachte:

Der Logarithmus wächst bekanntlich schwächer als jede noch so kleine positive Potenz, d.h.: Für jedes m≥1 gilt ab einem hinreichend großen n: $\log(n) < n^{\frac{1}{m}}$ und folglich $\log^m(n) < n$.

Ω(n) = Klasse der mindestens linear wachsenden Funktionen:
Ω(n) := $\{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c\in\mathbb{R}^+ \exists\, n_0\in\mathbb{N} \forall n\geq n_0: n \leq c\cdot g(n)\}$.
Auch hierin sind alle Funktionen der Form $g(n) = c_1\cdot n + c_2$ (für zwei positive Konstanten c_1 und c_2) enthalten, weil für n≥1 gilt: $n \leq (1/c_1)\cdot g(n) = n + (c_2/c_1)$.
Wenn g in Ω(n) liegt, so sagt man auch, g sei *mindestens linear*.

Θ(n) = Klasse der linear wachsenden Funktionen:
Θ(n) := $\{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c_1,c_2\in\mathbb{R}^+ \exists\, n_0\in\mathbb{N} \forall n\geq n_0: c_1\cdot n \leq g(n) \leq c_2\cdot n\}$.
Wegen Θ(f) = O(f) ∩ Ω(f) für alle Funktionen f gehören zu Θ(n) insbesondere alle Funktionen der Form $g(n) = c_1\cdot n + c_2$ (für zwei positive Konstanten c_1 und c_2), aber auch Funktionen wie $g(n) = n + \log^m(n) + 1/n \in \Theta(n)$ usw.

O(n^2) = Klasse der höchstens quadratisch wachsenden Funktionen.
O(n^2) := $\{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c\in\mathbb{R}^+ \exists\, n_0\in\mathbb{N} \forall n\geq n_0: g(n) \leq c\cdot n^2\}$.
Hierin sind alle Funktionen der Form $g(n) = c_1\cdot n^2 + c_2\cdot n + c_3$ (für Konstanten c_1, c_2 und c_3) enthalten, weil ab einem gewissen n≥1 dann gilt: $g(n) = c_1\cdot n^2 + c_2\cdot n + c_3 \leq (c_1+1)\cdot n^2$.
Wenn g in O(n) liegt, so sagt man, g wächst *höchstens quadratisch*.
Ω(n^2) ist die Klasse der mindestens quadratisch wachsenden Funktionen.

Für jede natürliche Zahl k ist **O(n^k)** die Klasse der höchstens wie n^k wachsenden Funktionen; O(n^k) umfasst insbesondere alle Polynome vom Grad k.

Für jede natürliche Zahl k ist **o(n^k)** die Klasse der echt schwächer als n^k wachsenden Funktionen, z.B. Funktionen wie n^{k-1} oder $n^k/\log(n)$ oder n^{k-d} für jede reelle Zahl d>0.

In der Praxis betrachtet man meist folgende Funktionsklassen:

$O(1)$: konstante Funktionen.

$O(\log n)$: höchstens logarithmisch wachsende Funktionen; wenn die Länge einer Darstellung wichtig ist, kommt oft der Logarithmus ins Spiel.

$O(n^{1/k})$: höchstens mit einer k-ten Wurzel wachsende Funktionen.

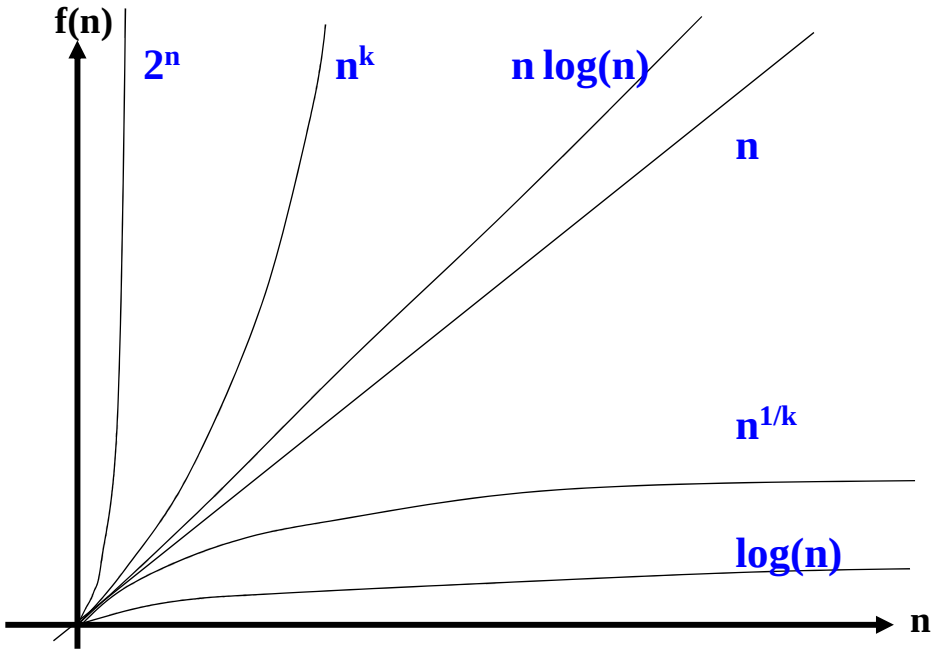
$O(n)$: lineare Funktionen.

$O(n \cdot \log(n))$: Das sind Funktionen, die "ein wenig" stärker als linear wachsen.

$O(n^2)$: höchstens quadratisch wachsende Funktionen.

$O(n^k)$: höchstens polynomiell vom Grad k wachsende Funktionen.

$O(2^n)$: höchstens exponentiell (zur Basis 2) wachsende Funktionen.



Hilfssatz 6.1.5:
Es gelten folgende Aussagen (die Beweise sind einfach):

$o(f) \subset O(f)$, $\omega(f) \subset \Omega(f)$, $\Theta(f) = O(f) \cap \Omega(f)$.

Für alle $g \in O(f)$ gelten $O(f+g) = O(f)$ und $o(f+g) = o(f)$.

Für alle $g \in \Omega(f)$ gelten $\Omega(f+g) = \Omega(g)$ und $\omega(f+g) = \omega(g)$.

Für alle $g \in \Theta(f)$ gilt $\Theta(f+g) = \Theta(f) = \Theta(g)$.

Zur Übung: Untersuchen Sie, ob folgende Formeln gelten:

$\omega(f) \cup O(f) = \Omega(f)$?

$O(f) - o(f) = \Theta(f)$?

$o(f) \cap \omega(f) = \varnothing$?

In der Regel sind Funktionen durch diese Klassen nicht vergleichbar. Zum Beispiel gilt für die beiden Funktionen

$$f(n) = \begin{cases} 1, \text{ für gerades } n \\ n, \text{ für ungerades } n \end{cases} \qquad g(n) = \begin{cases} n, \text{ für gerades } n \\ 1, \text{ für ungerades } n \end{cases}$$

weder $f \in O(g)$ noch $g \in O(f)$.

Wir werden vor allem mit den Klassen O und Θ bei der Untersuchung des Zeit- und Platzaufwands rechnen. Oft interessiert nämlich nur die Größenordnung der Komplexität und nicht der genaue Wert von Konstanten.

6.1.6 Machen Sie sich das Wachstum auch an folgender Tabelle klar. Nehmen Sie an, Ihr Rechner benötigt für eine Operation 10^{-8} Sekunden, wie viel Zeit vergeht dann für die Rechnung, wenn die Eingabelänge n = 1, 10, 100, ... beträgt? (s = Sekunde, h = Stunde, a = Jahr)

Komplex.	n = 1	n = 10	n = 100	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
n	10^{-8} s	10^{-7} s	10^{-6} s	10^{-5} s	10^{-4} s	10^{-3} s	10^{-2} s
n log(n)	-	$3 \cdot 10^{-7}$ s	$7 \cdot 10^{-6}$ s	10^{-4} s	$13 \cdot 10^{-4}$	$17 \cdot 10^{-3}$	$2 \cdot 10^{-1}$s
n^2	10^{-8} s	10^{-6} s	10^{-4} s	10^{-2} s	1 s	100 s	3 h
n^3	10^{-8} s	10^{-5} s	10^{-2} s	10 s	3 h	1/3 a	317 a
2^n	$2 \cdot 10^{-8}$ s	10^{-5} s	$3 \cdot 10^{22}$ a	--------	*viel zu*	*groß*	---------
n!	10^{-8} s	0,04 s	$3 \cdot 10^{48}$ a	--------	*viel zu*	*groß*	---------

Im Skript finden Sie eine umfassendere Darstellung. In der Vorlesung konzentrieren wir uns auf die Registermaschine (Abschnitt 6.7) und deuten die Turingmaschine nur an.
Dass es hier Unterschiede geben muss, wird z.B. an der Wertzuweisung im euklidischen Algorithmus

R := A mod B

klar. In der Registermaschine können wir annehmen, dass diese Operation zu den Basisoperationen gehört und daher unabhängig von der Größe der Werte in A und B einen Schritt dauert. Eine Turingmaschine muss jedoch eine Division zeichenweise durchführen und braucht allein hierfür mit dem aus der Schule bekannten Verfahren bereits $O(n^2)$ Schritte, wobei n die Länge der Eingabezahlen ist.
Macht man sich die Realisierung der Basisoperationen klar, so kann man von der Registermaschine rasch auf die Größenordnung der Zeitkomplexität von Turingmaschinen umrechnen.

6.2 Rechenmodell Turingmaschine

Vorbemerkungen zum Abschnitt "Turingmaschine"

Zu einem Programm gehören seine "Ressourcen", vor allem
- die Zeit, die es benötigt, und
- der Speicherplatz, den es belegt.

Um diese Ressourcen messen zu können, brauchen wir ein "Rechenmodell", also eine Maschine, die die Berechnung durchführt. Diese muss möglichst einfach sein, da man sonst fast keine Aussagen beweisen kann.

Seit längerer Zeit verwendet man zwei grundlegende Modelle:
1. die sequentielle Verarbeitung einzelner Zeichen auf einem unendlichen Band durch die Turingmaschine und
2. die sequentielle Verarbeitung von Elementen elementarer Datentypen, die in einem eindimensionalen Feld mit Direktzugriff abgelegt sind (durch die Registermaschine).

Prinzipiell ist die Turingmaschine der "realistischere" Ansatz, da beliebig große Werte nicht in einem Schritt verarbeitet werden können. Auf ihr beruhen daher die meisten theoretischen Untersuchungen.

Für die Praxis ist die Registermaschine meist das angemessenere Modell: Dort spielt sich alles in einem zwar recht großen, aber endlichen Bereich ab und die (beschränkte) Parallelität in der Hardware eines Computers erlaubt es, Zahlen und andere durch 32, 64 oder 128 Bit darstellbare Größen in einem Schritt zu verarbeiten; zugleich wird auf den Speicherort der Daten durch einen Index direkt zugegriffen.

Aktuellere Berechnungsmodelle beziehen die Parallelität ("Nebenläufigkeit", synchron, asynchron, konkurrierender Zugriff) durch gemeinsam agierende Maschinen mit ein.

Es lassen sich drei Arten der Berechnung unterscheiden, wobei das Verfahren nicht unbedingt terminieren muss:

Determinismus: Zu jedem Zeitpunkt kann es höchstens eine Fortsetzungsmöglichkeit des Programms geben. Das Verfahren liefert am Ende höchstens ein Ergebnis.

Nichtdeterminismus: Zu jedem Zeitpunkt kann es endlich viele verschiedene Fortsetzungsmöglichkeiten geben. Das Verfahren liefert kein oder irgendein Ergebnis aus einer (evtl. unendlichen) Menge von Ergebnissen.

Stochastik: Die Durchführung des Verfahrens hängt zu gewissen Zeitpunkten von zufälligen Ereignissen ab. Ein Ergebnis des Verfahrens besitzt nur mit einer gewissen Wahrscheinlichkeit die gewünschten Eigenschaften, kann aber auch ein gesichertes Ergebnis sein.

Viele Probleme der Praxis lassen sich mit nichtdeterministischen Verfahren gut beschreiben, weshalb wir mit der nichtdeterministischen Turingmaschine beginnen werden.

Programmiersprachen erlauben meist keinen Nichtdeterminismus (oder nur in beschränkter Form). Daher müssen in der Praxis deterministische Verfahren eingesetzt werden. Ihnen entsprechen die deterministischen Maschinenmodelle.

Stochastisch arbeitende Verfahren werden vor allem bei Simulationen und bei speziellen Problemstellungen (z.B. Primzahltest) verwendet. Hierauf gehen wir in der Grundvorlesung nicht näher ein.

6.2.1 Definition (nach A. M. Turing, 1912-1954, engl. Mathem.)

$M = (Q, \Sigma, \Gamma, \delta, q_0, F, \mathbf{b}, k)$ heißt (nichtdeterministische) k-Band-Turingmaschine $\Leftrightarrow$

(1) Q ist eine nicht-leere endliche Menge ("Zustandsmenge"),

(2) Σ ist eine endliche Menge ("Eingabealphabet"),

(3) Γ ist eine endliche Menge ("Bandalphabet") mit $\Sigma \subset \Gamma$,

(4) $q_0 \in Q$ ist der Anfangszustand,

(5) $F \subset Q$ ist die Menge der (akzeptierenden) Endzustände,

(6) $\mathbf{b} \in \Gamma \setminus \Sigma$ ist das "Blank"-Symbol,

(7) $k \in \mathbb{N}$ ist die Anzahl der Bänder,

(8) $\delta \subseteq Q \setminus F \times \Gamma^k \times Q \times \Gamma^k \times \{-1, 0, 1\}^k$ ist die Überführungsrelation.

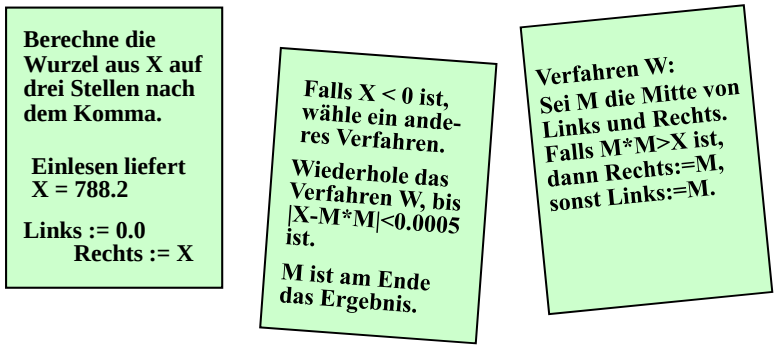
Einschub: Formalisierung (1)

Die letzten Vordiplomsprüfungen haben gezeigt, dass die Studierenden fast keinen Wert auf korrekte Formulierungen und auf den Übergang von einer meist noch vagen Idee zu einer präzisen formalen Dastellung legen. Dies ist jedoch für die Dokumentation von Software, für einen korrekten Entwurf und für die Herleitung und Messung von Eigenschaften unerlässlich.

Daher leiten wir im Folgenden die obige Definition der k-Band-Turingmaschine aus einer vagen Idee ab. Diese Idee besagt: Wir wollen ein Berechnungsmodell für die alltäglichen Situationen definieren, in denen wir Rechnungen durchführen, zum Beispiel arithmetische Operationen auf Zahlen.

Einschub: Formalisierung (2)

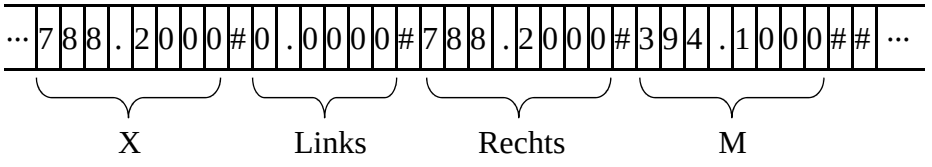
Vorstellung: Die Informationen befinden sich auf diversen Blättern und werden vom Menschen verwaltet und manipuliert.



Man muss diese Vorschriften in eine Reihenfolge bringen (=Programm) und geordnet ablegen (=Speicher).

Einschub: Formalisierung (3)

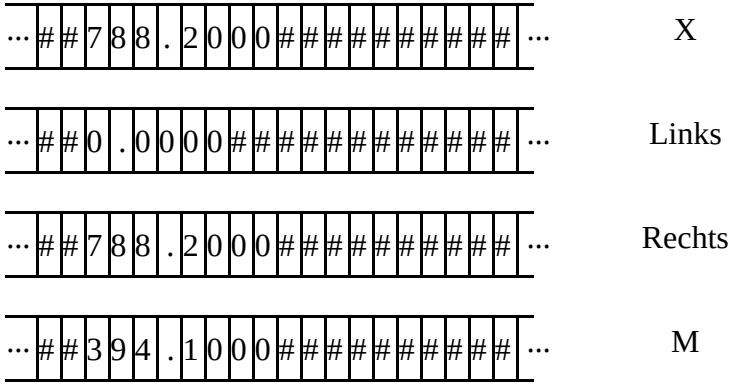
Die Variablen sind X, M, Links und Rechts. Man kann sie auf einen linearen Speicher (= ein Band) schreiben und verarbeiten:



Nun kann man die Werte ziffernweise (zum Beispiel rechts auf dem weiteren Band) bearbeiten, wie dies auch per Hand geschieht. Hierfür muss man nur wenig wissen, nämlich die Tabellen für Addition, Subtraktion und Multiplikation für die Zahlen von 0 bis 9, um M∗M und X-M∗M usw. zu berechnen.

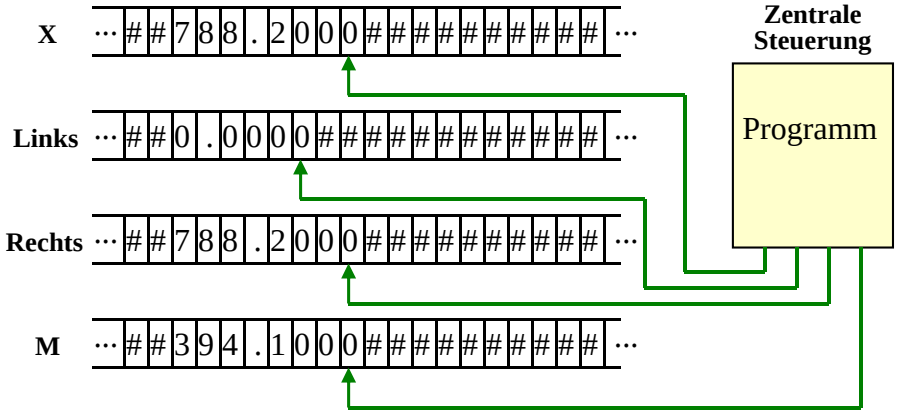
Einschub: Formalisierung (4)

Man kann die Variablen X, M, Links und Rechts aber auch auf 4 Bänder verteilen (wir verwenden das Zeichen "#" für die Felder, auf denen "nichts" steht):



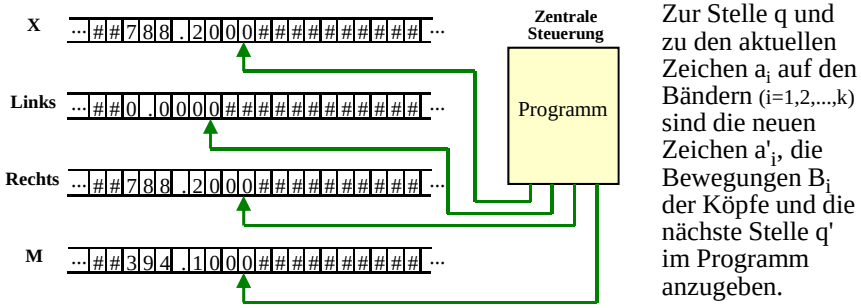
Einschub: Formalisierung (5)

Zu jedem Band muss es einen Zeiger ("Lese-Schreibkopf") auf das aktuell gelesene Zeichen geben. Wie muss dann ein Schritt einer Berechnungsvorschrift aussehen?



Einschub: Formalisierung (6)

Um M := (Links+Rechts) div 2 zu berechnen, muss man ziffernweise die Summe Links+Rechts auf das Band "M" übertragen und dort von der obersten zur untersten Ziffer hin durch 2 teilen. Hierzu muss man die aktuellen Zeichen auf den einzelnen Bändern lesen, durch andere ersetzen und die Positionen der Lese-Schreibköpfe verändern können. Zusätzlich muss man die Stelle ("Zustand") dieses Befehls im Programm kennen und den als nächstes folgenden Befehl angeben können.



Zur Stelle q und zu den aktuellen Zeichen a_i auf den Bändern (i=1,2,...,k) sind die neuen Zeichen a'_i, die Bewegungen B_i der Köpfe und die nächste Stelle q' im Programm anzugeben.

Einschub: Formalisierung (7)

Also muss der aktuellen Situation bestehend aus
Stelle q im Programm und den
Zeichen $a_1, ..., a_k$ unter den Lese-Schreibköpfen
die Reaktion der Maschine zugeordnet werden, die aus einer Folgestelle q' im Programm, den neu zu druckenden Zeichen $a'_1, ..., a'_k$ und den Bewegungen der Lese-Schreibköpfe (eine Position nach links, nach rechts oder Stehen-bleiben) besteht.

Solch eine Reaktion wird also durch ein Tupel
$(q, a_1, ..., a_k, q', a'_1, ..., a'_k, B_1, ..., B_k)$
beschrieben.

Eine Maschine ist im Wesentlichen eine Menge solcher Tupel (also eine Tabelle).

Gibt es hierbei zu jedem "Anfangs"-Tupel $(q, a_1, ..., a_k)$ höchstens ein Tupel $(q, a_1, ..., a_k, q', a'_1, ..., a'_k, B_1, ..., B_k)$, so arbeitet die Maschine *deterministisch*.

Einschub: Formalisierung (8)

Ein Tupel $(q, a_1, ..., a_k , q', a'_1, ..., a'_k, B_1, ..., B_k)$ ist - im Falle deterministischer Maschinen - programmiersprachlich eine einseitige if-Anweisung:

if Stelle=q and "auf dem ersten Band wird a_1 gelesen"
and "auf dem zweiten Band wird a_2 gelesen" ...
and "auf dem k-ten Band wird a_k gelesen"
then ersetze a_1 durch a'_1, a_2 durch a'_2, ... , a_k durch a'_k;
bewege den Lese-Schreibkopf auf Band 1 entsprechend B_1;
bewege den Lese-Schreibkopf auf Band 2 entsprechend B_2; ...
bewege den Lese-Schreibkopf auf Band k entsprechend B_k;
mache weiter im Programm an der Stelle q'
fi

Einschub: Formalisierung (9)

Nun muss man noch sagen, welches Zeichen zu Anfang auf den Bändern stehen soll (Blank **b**), an welcher Stelle die Maschine beginnt (q_0), wann die Maschine aufhören soll (Ende-Stellen F), welche Zeichen als Eingabe (Σ) benutzt werden und welche Zeichen auf dem Band stehen (Γ) dürfen. Zusammen mit der Menge der (Programm-) Stellen Q und der oben erläuterten Menge der Tupel $\delta \subseteq Q \setminus F \times \Gamma^k \times Q \times \Gamma^k \times \{-1, 0, 1\}^k$ erhalten wir als Formalisierung die Turingmaschine 6.2.1. (Hierbei steht -1 für links, 1 für rechts und 0 für Stehen-bleiben.)

Was wir als Nächstes klären müssen, sind der Begriff "deterministisch" und die genaue Arbeitsweise (siehe obige if-Anweisung) dieser Maschinen. Hieraus ergibt sich dann als "Bedeutung" die "realisierte Abbildung" (bzw. im nichtdeterministischen Fall die "realisierte Relation").

(Hinweis: Statt von der Stelle q sprechen wir vom "Zustand" q.)

6.2.2 Definition

Obiges M = (Q, Σ, Γ, δ, q_0, F, **b**, k) heißt deterministische k-Band-Turingmaschine, wenn zusätzlich gilt:

δ ist eine (partielle) Abbildung bzgl. der ersten beiden Komponenten, d.h.: Zu jedem $q \in Q$ und $a \in \Gamma^k$ existiert höchstens ein $(q, a, q', a', B) \in \delta$.

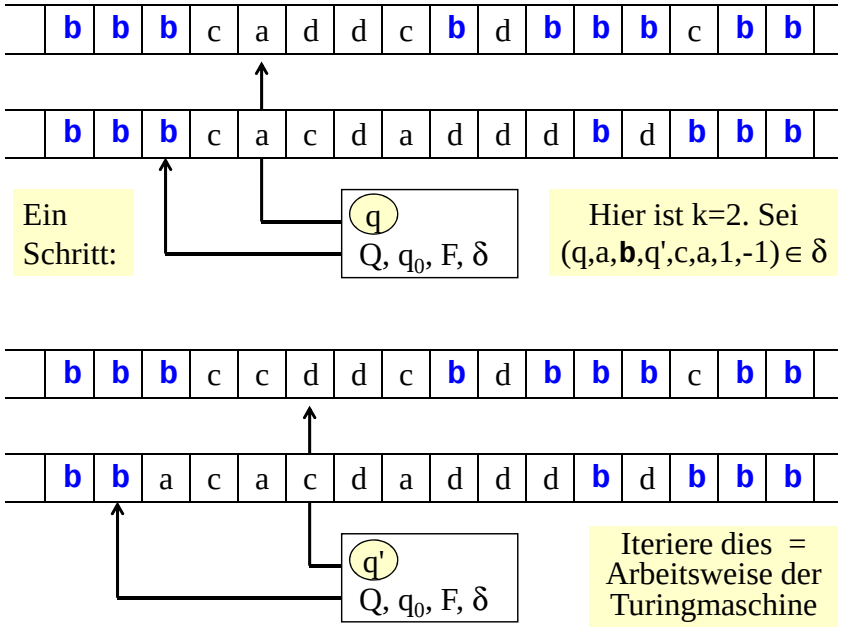
Man schreibt in diesem Fall δ als (partielle) Abbildung

$\delta: Q \setminus F \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{-1, 0, 1\}^k$

und nennt δ die Überführungs- oder Übergangsfunktion von M.

Bei einer deterministischen Maschine gibt es zu jedem Zeitpunkt höchstens eine Fortsetzungsmöglichkeit.

Eine nichtdeterministische Maschine verhält sich wie eine Grammatik: Der nächste Schritt ist nicht eindeutig bestimmt, vielmehr kann die Maschine eine von mehreren Fortsetzungsmöglichkeiten willkürlich auswählen.



1. Anfangssituation: "Ein Wort $w \in \Sigma^*$ wird eingegeben" bedeutet:

Alle Felder aller Bänder sind anfangs mit dem Blanksymbol beschriftet. Dann wird das Wort $w \in \Sigma^*$ auf das erste Band geschrieben, der Lese-Schreibkopf wird auf dem ersten Zeichen von w positioniert und die Turingmaschine befindet sich im Zustand q_0. Alle anderen Lese-Schreibköpfe befinden sich irgendwo auf den anderen Bändern.

2. Berechnung:

Es wird solange ein Schritt ausgeführt, bis man in einen Endzustand gelangt oder bis man auf eine Situation trifft, zu der es keine Folgesituation in δ gibt. Befindet sich M zu diesem Zeitpunkt in einem Endzustand, so ist die Berechnung erfolgreich abgeschlossen, anderenfalls erfolglos.

3. Ausgabe:

Am Ende wird das Wort ausgegeben, das auf dem ersten Band vom linkesten Nicht-Blanksymbol bis zum rechtesten Nicht-Blanksymbol steht. Dies ist ein Wort $v \in \Gamma^*$. Im nichtdeterministischen Fall wird einem Eingabewort also eine Resultats-Menge

$Res_M(w) = \{v \mid$ es gibt eine endlich lange Berechnung, die für die Eingabe w in einem Endzustand endet und hierbei v als Ausgabe besitzt$\}$

als Ergebnis zugeordnet (eventuell ist diese Menge leer). Im deterministischen Fall besitzt $Res_M(w)$ höchstens ein Element, d.h. Res_M ist dann eine (partielle) Abbildung $Res_M: \Sigma^* \rightarrow \Gamma^*$.

6.2.3 Definition:

Diese Resultatsrelation bzw. diese Resultatsabbildung Res_M ist die Bedeutung ("Semantik") der Turingmaschine M.

Es folgen 26 Folien mit Beispielen.

Wer dies gut nachvollziehen kann, sollte auch mit der Maschinenprogrammierung in der Praxis keine Probleme haben.

6.2.4.a: Beispiel "Addiere 1"

Aufgabe: Berechne den Nachfolger zu einer Zahl, d.h.: Zu einer binär dargestellten natürlichen Zahl soll 1 addiert werden. Konstruiere ein deterministische Turingmaschine, die dies leistet.

Aus der Aufgabenstellung folgt, dass das Eingabealphabet $\Sigma = \{0,1\}$ ist. Anfangs steht die binär dargestellte Zahl (der Länge n) auf dem Band und der Lese-Schreibkopf befindet sich auf der ersten Ziffer.

Lösungsidee: Man gehe ans Ende der Eingabe und laufe dann von hinten nach vorne, solange man auf das Zeichen "1" trifft (dieses wird jeweils in "0" umgewandelt). Man hält an, wenn das Zeichen "0" oder "**b**" erreicht wird, das dann durch "1" ersetzt wird.

$M_{Succ1} = (Q, \Sigma, \Gamma, \delta, q_0, F, \mathbf{b}, k)$ mit $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, $\Sigma = \{0,1\}$, $\Gamma = \{0,1,\mathbf{b}\}$, $k = 1$ und δ sei durch folgende Tabelle gegeben:

q	a	q'	a	B
q_0	0	q_0	0	1
q_0	1	q_0	1	1
q_0	**b**	q_1	**b**	-1
q_1	0	q_2	1	0
q_1	1	q_1	0	-1
q_1	**b**	q_2	1	0

Erläuterung: Im Zustand q_0 läuft der Lese-Schreibkopf von M_{Succ1} solange nach rechts, bis das Ende der Eingabe erreicht ist, erkennbar durch das Zeichen **b**. Dann wechselt die Maschine in den Zustand q_1. Nun wandelt sie, nach links gehend, jede "1" in eine "0" um und beendet ihre Arbeit (= geht in den Endzustand q_2), sobald eine "0" oder "**b**" gefunden wird (dieses Zeichen wird durch "1" ersetzt).

Ist diese Lösung korrekt?

Ja, wenn man führende Nullen zulässt.

Wir wollen nun verlangen, dass die Maschine nur Zahlen ohne führende Nullen (außer der Zahl 0 selbst) verarbeiten darf; andere Zahlen sollen zu einem fehlerhaften Anhalten führen.

Dies lösen wir, indem wir weitere Zustände hinzufügen:

$M_{Succ2} = (Q, \Sigma, \Gamma, \delta_2, q_0, F, \mathbf{b}, k)$ mit $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, $F = \{q_4\}$, $\Sigma = \{0,1\}$, $\Gamma = \{0,1,\mathbf{b}\}$, $k = 1$ und δ_2 sei durch folgende Tabelle gegeben (der Zustand q_5 ist ein "Fehler-Zustand"):

q	a	q'	a	B
q_0	0	q_1	**b**	1
q_0	1	q_2	1	1
q_0	**b**	q_5	**b**	0
q_1	0	q_5	0	0
q_1	1	q_5	1	0
q_1	**b**	q_4	1	0

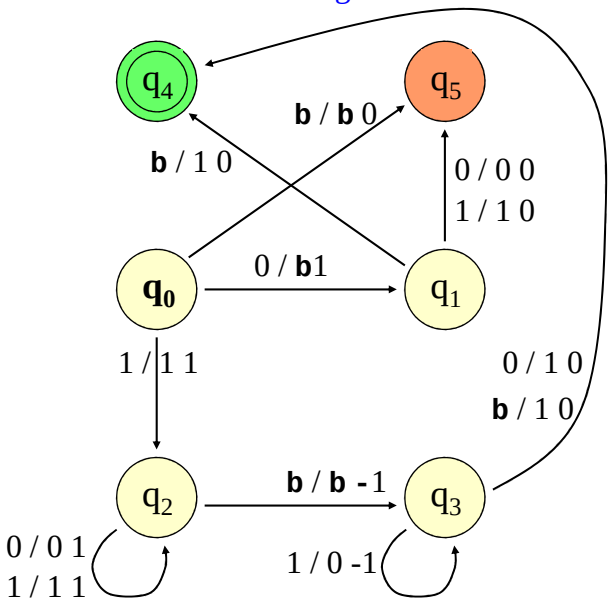
q	a	q'	a	B
q_2	0	q_2	0	1
q_2	1	q_2	1	1
q_2	**b**	q_3	**b**	-1
q_3	0	q_4	1	0
q_3	1	q_3	0	-1
q_3	**b**	q_4	1	0

Die Zustände q_0 und q_1 prüfen den Fall ab, ob eine einzelne "0" vorliegt. Falls dagegen die Eingabe mit "1" beginnt, so wird im Zustand q_2 wie bei M_{Succ1} weiter gearbeitet.

Zeichnerische Darstellung

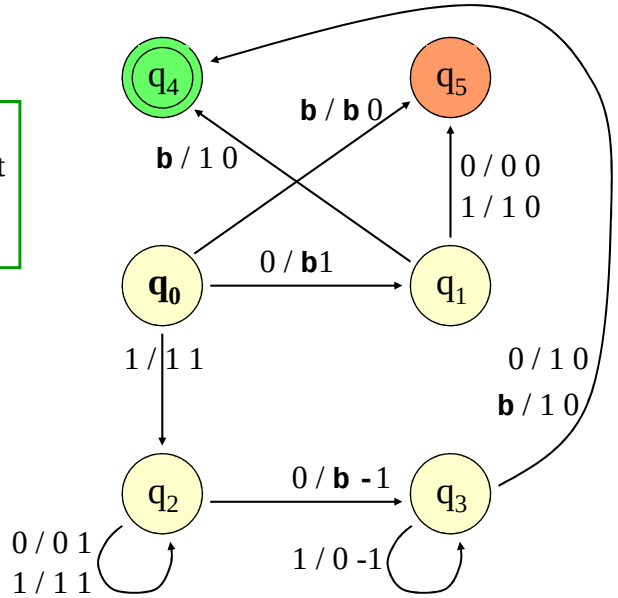
q	a	q'	a	B
q_0	0	q_1	**b**	1
q_0	1	q_2	1	1
q_0	**b**	q_5	**b**	0
q_1	0	q_5	0	0
q_1	1	q_5	1	0
q_1	**b**	q_4	1	0

q	a	q'	a	B
q_2	0	q_2	0	1
q_2	1	q_2	1	1
q_2	**b**	q_3	**b**	-1
q_3	0	q_4	1	0
q_3	1	q_3	0	-1
q_3	**b**	q_4	1	0



Ergebnis:

Dies sieht netter aus, aber es bleibt ebenso schwer durchschaubar.



Mit der Turingmaschine können wir nun genau berechnen, wie viele Schritte das Verfahren für eine Eingabe benötigt, die aus n Zeichen besteht.

Nach genau n Schritten erreicht man das **b**, das hinter der Eingabe steht. Mit einem weiteren Schritt steht der Lese-Schreibkopf auf dem letzten Zeichen der Eingabe.

Schlechtester Fall: Die Eingabe besteht nur aus Einsen, dann muss man alle n Zeichen durch Null ersetzen (n Schritte) und eine Eins links davon schreiben (1 Schritt). Im schlechtesten Fall ("worst case") benötigt man daher 2n+2 Schritte.

Im besten Fall, wenn nämlich die Eingabe mit einer "0" endet, braucht man n+2 Schritte.

Der Zeitaufwand liegt somit auf jeden Fall in O(n); er ist also linear in der Länge der Eingabe.

6.2.4.b: Beispiel "Quersumme" (n = Länge der Eingabe)

Aufgabe: Bilde die Quersumme einer binären Folge, d.h.:
$quer(z_{n-1}z_{n-2}...z_1z_0)$ = Binärdarstellung der Anzahl der Einsen unter den z_i.

Erneut ist das Eingabealphabet $\Sigma = \{0,1\}$. Anfangs steht eine Folge aus Nullen und Einsen auf dem Band und der Lese-Schreibkopf befindet sich auf dem ersten Zeichen.

Idee: Man verwende ein zweites Band für die Anzahl der Einsen. Man gehe von links nach rechts auf dem ersten Band und addiere bei jeder gefundenen Eins eine 1 auf die Zahl, die auf dem zweiten Band steht. Am Ende kopiere man den Inhalt des zweiten Bandes auf das erste Band, da ja nach Definition auf dem ersten Band das Ergebnis stehen muss.

$M_{Quer} = (Q, \Sigma, \Gamma, \delta, q_0, F, \mathbf{b}, k)$ mit $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_4\}$, $\Sigma = \{0,1\}$, $\Gamma = \{0,1,\mathbf{b}\}$, k = 2 und δ sei durch folgende Tabelle gegeben (für alle $y \in \Sigma$, für alle $x \in \Gamma$):

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_0	0	**b**	q_0	**b**	**b**	1	0
q_0	1	**b**	q_1	**b**	**b**	1	-1
q_0	**b**	**b**	q_3	**b**	**b**	0	-1
q_1	x	0	q_2	x	1	0	1
q_1	x	**b**	q_2	x	1	0	1
q_1	x	1	q_1	x	0	0	-1
q_2	x	y	q_2	x	y	0	1
q_2	x	**b**	q_0	x	**b**	0	0
q_3	**b**	y	q_3	y	**b**	-1	-1
q_3	**b**	**b**	q_4	**b**	**b**	0	0

Erläuterung: Im Zustand q_0 werden alle "0" auf dem ersten Band gelöscht. Trifft man auf eine "1", so wird diese ebenfalls gelöscht, es wird in den Zuständen q_1 und q_2 eine 1 auf den Inhalt des zweiten Bandes addiert und dann der dortige Lese-Schreibkopf wieder auf das erste Blank hinter der Zahl gesetzt. Ist das Ende der Eingabe auf Band 1 erreicht, wird im Zustand q_3 Band 2 nach Band 1 kopiert.

Auch hier kann man die Laufzeit (=Anzahl der Schritte), die die Turingmaschine benötigt, in Abhängigkeit von der Länge der Eingabe abschätzen.

Bester Fall: Sind keine Einsen in der Eingabe, so benötigt die Maschine genau n+2 Schritte (davon ein Schritt im Zustand q_3).

Schlechtester Fall: Besteht die Eingabe nur aus Einsen, so benötigt die Maschine für die Bewegung auf dem ersten Band genau n+1 Schritte. Auf dem zweiten Band werden nacheinander die binär dargestellten Zahlen 1, 2, 3, ..., n durchlaufen. Hierbei steht jedes zweite Mal eine "0" an letzter Stelle, die in eine "1" umgewandelt wird (hierfür werden 2 Schritte benötigt), jedes vierte Mal endet das Wort auf dem zweiten Band mit "01" (hierfür werden 4 Schritte benötigt), jedes achte Mal mit "011" (6 Schritte) usw. Die Länge der Zahl auf dem zweiten Band ist durch log(n)+1 beschränkt (die Basis des Logarithmus ist 2).

Die Gesamtzahl dieser Schritte lässt sich abschätzen durch

$(n/2)\cdot 2 + (n/4)\cdot 4 + (n/8)\cdot 6 + ... + (n/2^i)\cdot 2i + ...$

$$\leq \sum_{i=1}^{\log(n)+1} \frac{n}{2^i} 2i = 2n \sum_{i=1}^{\log(n)+1} \frac{i}{2^i} \leq 4n$$

Also ist im schlechtesten Fall die Anzahl der Schritte, um alle Zahlen von 1 bis maximal n zu durchlaufen, beschränkt durch 4n+1.

Beweisen Sie hierfür: $\sum_{i=1}^{k} \frac{i}{2^i} < 2$ und $\sum_{i=1}^{k} \frac{1}{2^i} < 1$

Am Ende muss Band 2 noch auf Band 1 kopiert werden. Hierfür ist mit maximal log(n)+1 Schritten zu rechnen, da die Anzahl der Einsen durch log(n) beschränkt ist.

Insgesamt erhalten wir also für die binäre Quersumme als obere Schranke:

$5n+1 + \log(n)+1 \leq 6n \in O(n)$ für $n \geq 4$.

Die Berechnung der Anzahl der Einsen in einer Folge von Nullen und Einsen mit M_{Quer} benötigt also nur linearen Zeitaufwand bzgl. der Länge der Eingabe.

Dieses Ergebnis ist wegen der bis zu n-maligen Addition einer 1, die jeweils O(log(n)) Schritte dauern kann, nicht offensichtlich (das heißt, man würde zunächst $O(n\cdot\log(n))$ erwarten).

6.2.4.c: Beispiel "Palindrome" mit einem Band

Aufgabe: Stelle fest, ob die Eingabe ein Palindrom ist, d. h., ob das Eingabewort w gleich seinem gespiegelten Wort w^R ist. Falls ja, schreibe eine "**1**", anderenfalls eine "**0**" auf das Band.

Erläuterungen: w^R ist das rückwärts gelesene Wort w.
Formal: Gegeben sei ein Alphabet Σ. Die Spiegelung ^{R} ist eine bijektive Abbildung von Σ^* nach Σ^*, die induktiv definiert ist durch:

$\varepsilon^R = \varepsilon$ und $(aw)^R = w^R a$ für alle $a \in \Sigma$ und alle $w \in \Sigma^*$.

Ein Wort, das vorwärts und rückwärts gelesen gleich ist, nennt man ein Palindrom.

$Pal = \{ w \in \Sigma^* \mid w = w^R \} \subset \Sigma^*$

ist die Menge der Palindrome über dem Alphabet Σ.

Palindrome sind folgendermaßen charakterisiert:

(1) Das leere Wort ε ist ein Palindrom.

(2) Jedes Zeichen $a \in \Sigma$ ist ein Palindrom.

(3) Wenn *a* ein Zeichen und *w* ein Palindrom sind, dann ist auch *awa* ein Palindrom.

(4) Alle Palindrome werden nach den Regeln (1) bis (3) gebildet.

Entsprechend dieser Charakterisierung kann man Palindrome wie folgt erkennen: Falls das Wort höchstens die Länge 1 hat, so ist es ein Palindrom. Anderenfalls vergleiche das erste mit dem letzten Zeichen. Falls diese gleich sind, streiche sie weg und wende das Verfahren auf das restliche Wort an, anderenfalls lag kein Palindrom vor. Dies führt zu folgender deterministischen 1-Band-Turingmaschine:

$M_1 = (Q_1, \Sigma, \Gamma, \delta_1, q_0, F_1, \mathbf{b}, 1)$ mit $\Sigma = \{a_1, ..., a_m\}$, $F_1 = \{q_3\}$, $Q_1 = \{q_0, q_1, q_2, q_3\} \cup \{q_a, q'_a, q''_a \mid a \in \Sigma\}$, $\Gamma = \Sigma \cup \{\mathbf{0}, \mathbf{1}, \mathbf{b}\}$, $k = 1$ und δ_1 sei durch folgende Tabelle gegeben (für alle $x, y \in \Sigma$, für alle $s \in \Sigma$ mit $s \neq x$; die Maschine merkt sich das gelesene Zeichen x in ihren Zuständen q_x, q'_x bzw. q''_x):

q	a	q'	a	B
q_0	x	q_x	**b**	1
q_0	**b**	q_3	**1**	0
q_x	**b**	q_3	**1**	0
q_x	y	q'_x	y	1
q'_x	y	q'_x	y	1
q'_x	**b**	q''_x	**b**	-1
q''_x	x	q_1	**b**	-1
q''_x	s	q_2	**b**	-1

beachte: $s \neq x$

q	a	q'	a	B
q_1	y	q_1	y	-1
q_1	**b**	q_0	**b**	1
q_2	y	q_2	**b**	-1
q_2	**b**	q_3	**0**	0

Erläuterung: Im Zustand q_0 wird auf ε geprüft (ja ⇒ Ende, drucke **1**) oder das aktuelle Zeichen x durch Übergang in den Zustand q_x gemerkt. In q_x wird erneut auf ε geprüft (⇒ Ende, drucke **1**) oder im Zustand q'_x ans Ende des Wortes gegangen. Das letzte Zeichen wird im Zustand q''_x mit x verglichen. Bei Gleichheit läuft man an den Anfang des Wortes und wiederholt das Verfahren, bei Ungleichheit wird das restliche Wort gelöscht und **0** gedruckt.

Laufzeitabschätzung:

Der beste Fall liegt vor, wenn die Eingabe die Form awb hat mit $w \in \Sigma^*$, $a, b \in \Sigma$ und $a \neq b$. Dann stellt man nach n+1 Schritten fest, dass kein Palindrom vorliegt, löscht in weiteren n-1 Schritten die verbliebene Eingabe und schreibt im nächsten Schritt eine **0**. Dies dauert 2n+1 Schritte.

Im schlechtesten Fall ist die Eingabe ein Palindrom. Dann benötigt die obige Turingmaschine n Schritte nach rechts, n-1 Schritte nach links, erneut n-2 Schritte nach rechts, n-3 Schritte nach links usw., insgesamt

$$n + (n-1) + (n-2) + ... + 1 = n \cdot (n+1)/2 ,$$

also $O(n^2)$ Schritte.

Hinweis: Zu den schwierigsten Problemen der Algorithmik gehört die Angabe unterer Schranken.

Im Falle der Palindrome kann man jedoch beweisen:
Satz: Jede (deterministische oder nichtdeterministische) 1-Band-Turingmaschine, die zu jeder Eingabe genau dann eine "**1**" ausgibt, wenn die Eingabe ein Palindrom ist, und sonst eine "**0**", benötigt hierfür mindestens $c \cdot n^2$ Schritte im schlechtesten Fall (für eine von der jeweiligen Turingmaschine abhängige rationale Konstante $c > 0$; n ist die Länge der Eingabe).

Unser Verfahren ist also in diesem Sinne "optimal".
M_1 besitzt die rationale Konstante 1/2.

Hat man zwei Bänder, so gibt es offensichtlich Verfahren mit linearem Zeitaufwand, wie im Folgenden gezeigt wird.

6.2.4.d: Beispiel "Palindrome" mit zwei Bändern

Aufgabe: Wir wollen erneut feststellen, ob die Eingabe ein Palindrom ist.

Mit zwei Bändern benötigt man nur O(n) Schritte.

Idee: Kopiere die Eingabe, also den Inhalt des ersten Bandes auf das zweite Band und vergleiche dann schrittweise das Wort auf dem ersten Band rückwärts mit dem Wort auf dem zweiten Band vorwärts. Dies liefert folgende deterministische 2-Band-Turingmaschine M_2.

$\Sigma = \{a_1, a_2, ..., a_m\}$ ist wiederum ein beliebiges Alphabet.

$M_2 = (Q_2, \Sigma, \Gamma, \delta_2, q_0, F_2, \mathbf{b}, k)$ mit $k = 2$, $Q_2 = \{q_0, q_1, q_2, q_3, q_4\}$, $F_2 = \{q_4\}$, $\Gamma = \Sigma \cup \{\mathbf{0}, \mathbf{1}, \mathbf{b}\}$ und δ_2 sei durch folgende Tabelle gegeben (für alle $x, y \in \Sigma$, für alle $c, d \in \Gamma$ mit $c \neq d$):

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_0	x	**b**	q_0	x	x	1	1
q_0	**b**	**b**	q_1	**b**	**b**	-1	-1
q_1	c	y	q_1	c	y	0	-1
q_1	c	**b**	q_2	c	**b**	0	1
q_2	x	x	q_2	**b**	**b**	-1	1
q_2	**b**	**b**	q_4	**1**	**b**	0	0

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_2	c	d	q_3	**b**	**b**	-1	0
q_3	x	d	q_3	**b**	d	-1	0
q_3	**b**	d	q_4	**0**	**b**	0	0

Erläuterung: Im Zustand q_0 kopiert M_2 das Eingabewort w auf das zweite Band. Im Zustand q_1 wird der Lese-Schreibkopf des zweiten Bandes an den Anfang des kopierten Wortes gesetzt. Im Zustand q_2 werden die Wörter verglichen. Lag ein Palindrom vor, so löscht man hierbei beide Bänder simultan, gelangt in den Fall "q_2 **b b**", druckt eine **1** auf das erste Band und geht in den Endzustand q_4. Waren im Zustand q_2 zwei Zeichen verschieden ($c \neq d$, also kein Palindrom), so wird im Zustand q_3 das Band 1 gelöscht und **0** gedruckt.

Laufzeitabschätzung: Diese Turingmaschine läuft in linearer Zeit. Begründung:

Die Turingmaschine M_2 verarbeitet alle Eingaben auf die gleiche Weise: Erst wird die Eingabe w auf das zweite Band kopiert (n+1 Schritte), dann wird der Lese-Schreibkopf des zweiten Bandes auf den Anfang des Wortes gesetzt (n+1 Schritte), danach werden die beiden Wörter zeichenweise verglichen und dabei gelöscht (n Schritte) und anschließend wird **0** oder **1** gedruckt (1 Schritt).

Folglich benötigt diese Turingmaschine für jede Eingabe der Länge n genau 3n+3 Schritte.

Frage: Können Sie eine deterministische Maschine konstruieren, die im schlechtesten Fall weniger Schritte benötigt? (Eine Maschine mit $2{,}5 \cdot n+3$ Schritten findet man rasch. Gibt es schnellere?)

Die Turingmaschinen M_1 und M_2 berechnen die gleiche Resultatsfunktion $Res_{M_1} = Res_{M_2}$

$Res_{M_1}: \Sigma^* \to \{0,1,\mathbf{b}\}^*$ mit

$$Res_{M_1}(w) = \begin{cases} 1, & w \text{ ist ein Palindrom} \\ 0, & w \text{ ist kein Palindrom} \end{cases}$$

Da alle Ergebniswerte aus {0,1} sind, kann man diese Abbildung auch als Abbildung von Σ^* in {0,1} auffassen.

Hinweis: Res_{M_1} ist die charakteristische Funktion χ_{Pal} der Menge Pal der Palindrome, siehe 2.3.4. und 6.2.4.c.
[Die Menge Pal ist also entscheidbar. Definition siehe 2.3.3.]

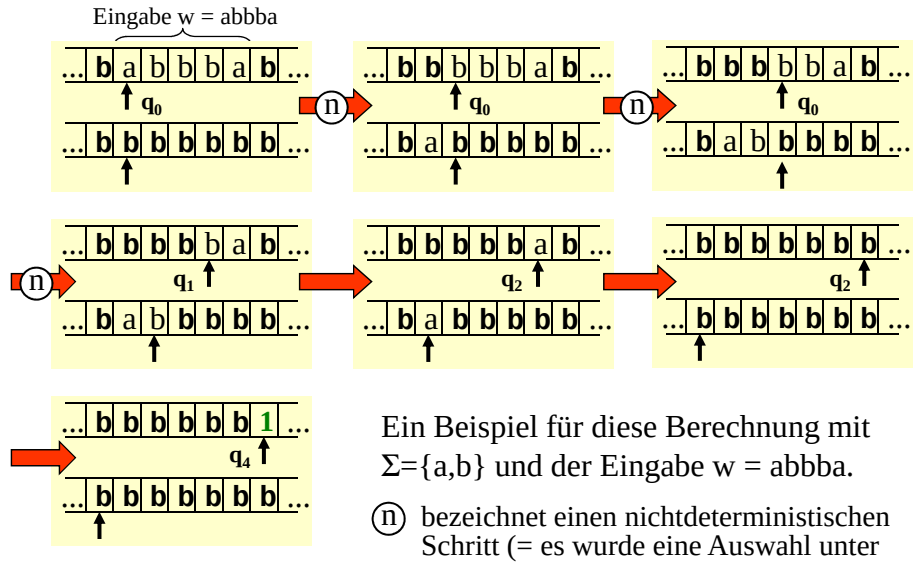
6.2.4.e: Beispiel

"Palindrome" nichtdeterministisch mit zwei Bändern

Aufgabe: Wir wollen erneut feststellen, ob die Eingabe ein Palindrom ist.

Idee: Kopiere die Eingabe, also den Inhalt des ersten Bandes auf das zweite Band, aber nur bis zur Mitte des Wortes (diese Mitte wird nichtdeterministisch geraten) und vergleiche dann schrittweise das restliche Wort auf dem ersten Band vorwärts mit dem Wort auf dem zweiten Band rückwärts. Dies liefert die nichtdeterministische 2-Band-Turingmaschine M_3.

$\Sigma = \{a_1, a_2, ..., a_m\}$ ist wiederum ein beliebiges Alphabet.



Ein Beispiel für diese Berechnung mit $\Sigma=\{a,b\}$ und der Eingabe w = abbba.

(n) bezeichnet einen nichtdeterministischen Schritt (= es wurde eine Auswahl unter mehreren Möglichkeiten getroffen).

$M_3 = (Q_3, \Sigma, \Gamma, \delta_3, q_0, F_3, \mathbf{b}, k)$ mit $Q_3 = \{q_0, q_1, q_2, q_3, q_4\}$, $F_3 = \{q_4\}$, $\Gamma = \Sigma \cup \{\mathbf{0}, \mathbf{1}, \mathbf{b}\}$, k = 2 und δ_3 sei durch folgende Tabelle gegeben (für alle $x, y \in \Sigma$, für alle $c, d \in \Gamma$ mit $c \neq d$):

Vorsicht, fehlerhaft!

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_0	x	**b**	q_0	**b**	x	1	1
q_0	x	**b**	q_1	**b**	**b**	1	-1
q_1	x	x	q_2	**b**	**b**	1	-1
q_1	x	c	q_2	**b**	c	1	0
q_2	x	x	q_2	**b**	**b**	1	-1
q_2	c	d	q_3	**b**	**b**	1	0
q_2	**b**	**b**	q_4	**1**	**b**	0	0
q_3	x	d	q_3	**b**	d	1	0
q_3	**b**	d	q_4	**0**	d	0	0

Erläuterung: Im Zustand q_0 kopiert M_3 Zeichen von Band 1 auf Band 2. Jederzeit kann M_3 dieses Kopieren beenden und in den Zustand q_1 wechseln. Dabei kann das nächste Zeichen auf Band 1 übersprungen werden. Spätestens dann werden jedoch im Zustand q_2 die beiden Zeichen unter den Lese-Schreibköpfen auf Gleichheit geprüft. Trifft man nun gleichzeitig auf das Blanksymbol, so muss ein Palindrom vorgelegen haben und man druckt **1** und geht in den Endzustand q_4. Anderenfalls wird Band 1 gelöscht, die **0** gedruckt und der Endzustand q_4 angenommen.

Arbeitet M_3 wirklich korrekt?

Wir müssen die Resultatsmenge von M_3 genau angeben.

Zunächst fällt auf, dass M_3 nichts tut, wenn das leere Wort eingegeben wird. Es liegt also sicher ein Fehler vor.

Für eine Eingabe der Länge 1 kann man zwar in den Zustand q_1 wechseln, kommt dann aber nicht mehr weiter.

Ansonsten gibt es für jede Eingabe mit mindestens zwei Zeichen eine Berechnungsfolge, die mit dem Endzustand q_4 abschließt. Dabei wird aber für alle Eingaben der Länge 2 das Zeichen **1** gedruckt, auch wenn kein Palindrom vorliegt.

Wir versuchen nun, Res_{M_3} präzise anzugeben (wobei nicht sicher ist, dass diese Formel wirklich stimmt. Prüfen Sie sie beispielhaft oder formal nach. Bei Turingmaschinen übersieht man oft nicht alle ihre Möglichkeiten.)

Res_{M_3} = {(w,**1**) | w enthält mindestens 2 Zeichen und w besitzt die Form $w=uxu^R$ oder $w=uxyu^R$ mit $u \in \Sigma^*$ und $x,y \in \Sigma$}
$\cup$ {(w,**0**) | w besitzt mindestens zwei Zeichen}

Das Ziel bei einer nichtdeterministischen Turingmaschine lautet in unserem Beispiel natürlich (der Fehlerfall (w,**0**) interessiert hierbei nicht, da er auch für Wörter aus Pal auftreten darf):

$(w,\mathbf{1}) \in Res_{M_3} \Leftrightarrow w \in Pal.$

Diese Aussage ist aber offensichtlich nicht erfüllt.
Daher werden wir die Turingmaschine M_3 zu folgender Turingmaschine M_4 korrigieren.

$\Sigma = \{a_1, a_2, ..., a_m\}$ mit $m \geq 1$.
$M_4 = (Q_4, \Sigma, \Gamma, \delta_4, q_0, F_4, \mathbf{b}, k)$
mit $Q_4 = \{q_0, q_1, q_2\}$,
$F_4 = \{q_2\}$, $\Gamma = \Sigma \cup \{\mathbf{1}, \mathbf{b}\}$,
$k = 2$ und δ_4 ist durch die nebenstehende Tabelle gegeben (für alle $x \in \Sigma$).

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_0	x	**b**	q_0	**b**	x	1	1
q_0	x	**b**	q_1	**b**	**b**	1	-1
q_0	x	**b**	q_1	**b**	x	1	0
q_0	**b**	**b**	q_1	**b**	**b**	1	-1
q_1	x	x	q_1	**b**	**b**	1	-1
q_1	**b**	**b**	q_2	**1**	**b**	1	0

Den "Fehlerfall" (w,**0**) haben wir hier vollständig weggelassen, da er im nichtdeterministischen Fall unerheblich ist. Nur bei Eingabe eines Palindroms gibt es eine Möglichkeit, in den Endzustand q_2 zu gelangen.

Beschreibt M_4 genau Pal, d.h., gilt $(w,\mathbf{1}) \in Res_{M_4} \Leftrightarrow w \in Pal$?
Prüfen Sie diese Aussage.

Laufzeit von M_4:

M_4 bewegt in jedem Schritt den Lese-Schreibkopf des ersten Bandes nach rechts, wobei maximal zwei Blanksymbole auftreten dürfen. Die Zahl der Schritte ist daher beschränkt durch n+2.

Feststellung: Mit einer nichtdeterministischen Maschine kann man in n+2 Schritten feststellen, ob einWort der Länge n ein Palindrom ist.

(Spendiert man weitere Zustände, so kann diese Prüfung auch in n+1 Schritten erfolgen. Konstruieren Sie eine geeignete Maschine.)

Ende der Beispiele 6.2.4

6.2.5 Definitionen:

(1) Für zwei Alphabete Σ und Δ heißt eine Abbildung $f: \Sigma^* \to \Delta^*$ Turing-berechenbar (oder partiell-berechenbar oder oft kurz berechenbar), wenn es eine deterministische Turingmaschine M mit $f = Res_M$ gibt.

(2) Ist f zusätzlich eine totale Funktion (also auf ganz Σ^* definiert), so heißt f total berechenbar oder (total) rekursiv.

(3) Eine Menge $L \subseteq \Sigma^*$ heißt entscheidbar, wenn ihre charakteristische Funktion χ_L total berechenbar ist (vgl. 2.3.3 u. 2.3.4).

(4) Eine Menge $L \subseteq \Sigma^*$ heißt Haltebereich einer Turingmaschine, wenn es eine (deterministische) Turingmaschine gibt, die genau für alle Wörter aus L in einem Endzustand anhält.

(5) Eine Menge $L \subseteq \Sigma^*$ heißt aufzählbar, wenn sie Haltebereich einer (deterministischen) Turingmaschine ist.

6.2.6 Beispiel "wiederholungsfreie Wörter"

Es sei Σ ein Alphabet. Ein Wort $w \in \Sigma^*$ heißt wiederholungsfrei, wenn es keine Wörter x, y und z gibt mit $w = xyyz$ (mit $y \neq \varepsilon$). Kein echtes Teilwort kommt also in w zweimal hintereinander vor. Insbesondere darf kein Buchstabe auf sich selbst folgen.

Ist Σ ein- oder zweielementig, so gibt es nur endlich viele wiederholungsfreie Wörter (bitte ausprobieren!).
Ist Σ dagegen mindestens dreielementig, so gibt es unendlich viele wiederholungsfreie Wörter. Über $\Sigma = \{a, b, c\}$ kann man beliebig lange wiederholungsfreie Wörter konstruieren:

a b a c b a b c a b a c b c a b c a c b a b c ...

Es sei WF = $\{w \in \Sigma^* \mid w \text{ ist wiederholungsfrei}\}$.
Aufgabe: Entscheide zu $w \in \Sigma^*$, ob w in WF liegt oder nicht, d.h.: Berechne die charakteristische Funktion χ_{WF} zu WF.

Lösungsansatz, deterministisch:

Sei Σ mindestens dreielementig. Sei $n = |w|$ und $w = w_1 w_2 \ldots w_n$ mit $w_i \in \Sigma$ für i =1, 2, ..n. Falls $n \leq 1$, dann ist $w \in$ WF.

Anderenfalls prüfe für jede Position i ($1 \leq i \leq n-1$) und für jede Länge k ($1 \leq k \leq (n-i+1)$ div 2 bzw. $i+2k-1 \leq n$), ob $w_i w_{i+1} .. w_{i+k-1} = w_{i+k} w_{i+k+1} \ldots w_{i+2k-1}$ gilt.
Trifft dies für kein i und k zu, so ist w wiederholungsfrei.

Wir formulieren diese Überprüfung in Ada, wobei wir hier das Sprachelement exit verwenden. Dieses hat die Syntax:
exit_statement ::= exit [*loop*-name] [when condition];

Erinnerung zur Bedeutung: Verlasse die umgebende Schleife (oder die Schleife mit dem Namen *loop*-name), sofern die Bedingung hinter when zutrifft.

Mit exit kann man Schleifen verlassen. Gibt man nichts an, so wird die innerste Schleife, die die exit-Anweisung umfasst, verlassen. Will man mehrere Schleifen gleichzeitig verlassen, wie in unserem Fall, so muss die Schleife, die zu verlassen ist, einen Namen erhalten, den man in der exit-Anweisung angibt.

```
Schleife_i:
  for i in 1..n-1 loop
    for k in 1..(n+i-1)/2 loop
      wh := true;
      for j := i to i+k-1 loop wh := wh and (W(j)=W(j+k)); end loop;
      exit Schleife_i when wh;
    end loop;
  end loop;
if wh then "das Eingabewort besitzt eine Wiederholung"
else "das Eingabewort ist wiederholungsfrei" end if;
```


Zeitaufwand dieses Algorithmus?

Der am längsten dauernde Fall liegt vor, wenn das Wort wiederholungsfrei ist. Dann werden alle Kombinationen durchprobiert.
Hierbei können i und k in der Größenordnung von n/2 liegen, d.h., die beiden äußeren Schleifen benötigen bereits $O(n^2)$ Schritte.
Die innerste Schleife erfordert nochmals bis zu n/2 Schritte. Insgesamt erhalten wir somit einen Zeitaufwand von $O(n^3)$.

Ein zusätzlicher *Speicheraufwand*, außer konstant viel Platz für die Hilfsvariablen i, j, k, wh und Zwischenergebnisse wie (n+i-1)/2, entsteht nicht, da der Algorithmus nur auf dem Wort W arbeitet.

Einspruch: Genau genommen stimmt dies aber nicht. Es fällt doch zusätzlicher Speicherplatz in Abhängigkeit von der Länge der Eingabe an.

Denn es müssen die natürlichen Zahlenwerte von i, j, k und n dargestellt werden.

Um die Zahl n darzustellen, braucht man log(n) Ziffern in der Binärdarstellung und in jeder anderen Darstellung ebenfalls O(log(n)) Ziffern. Für wh braucht man dagegen nur eine Ziffer.

Also erfordert der Algorithmus insgesamt zusätzlichen Speicherplatz in der Größenordnung $4 \cdot \log(n)$, also O(log(n)). In der Praxis ist dies eine geringe Größe.

Wie realisiert man diesen Algorithmus nun mit einer deterministischen Turingmaschine?

Das Eingabewort wird auf dem ersten Band abgelegt. Die Variablen i, j, k und n werden auf 4 Bändern gespeichert. Die Vergleiche und das Zählen ist höchstens in der Größenordnung der Länge der Darstellungen, also O(log(n)).
Da der Direktzugriff W(j) auf ein Turingband nicht möglich ist, muss man die Abfrage $W(j) = W(j+k)$ mit k Schritten durchführen, wobei man ggf. noch eine Hilfsvariable verwendet, um j oder k nicht löschen zu müssen.
Eine 6-Band-Turingmaschine müsste das Problem also in $O(n^4)$ Schritten lösen können. Man müsste mit $O(n^3)$ auskommen, wenn man das Wort W anfangs auf ein weiteres Band kopiert. Die konkrete Turingmaschine ist recht groß und wird daher nicht ausformuliert.

Lösungsansatz, nichtdeterministisch:

Sei Σ mindestens dreielementig. Sei $n = |w|$ und $w = w_1 w_2 \ldots w_n$ mit $w_i \in \Sigma$ für i =1, 2, ..n. Falls $n \leq 1$, dann ist $w \in$ WF.

Anderenfalls erzeuge man nichtdeterministisch eine Position i ($1 \leq i \leq n-1$) und eine Länge k ($1 \leq k \leq (n-i+1)$ div 2). Für diese prüft man, ob
$w_i w_{i+1} .. w_{i+k-1} = w_{i+k} w_{i+k+1} \ldots w_{i+2k-1}$ gilt.
Trifft dies zu, so ist w nicht wiederholungsfrei. Anderenfalls geht man nicht in einen Endzustand.

Diesen Nichtdeterminismus können wir nicht mehr in einer gängigen Programmiersprache ausdrücken, wohl aber mit einer nichtdeterministischen Turingmaschine mit geeignet vielen Bändern. Hier reichen ebenfalls 6 Bänder sicher aus.

Die nichtdeterministische Turingmaschine ermittelt zunächst die Länge n der Eingabe w und weist danach in O(log(n)) Schritten den Variablen i, j, k irgendwelche Werte zu.

Dann wird in $O(n^2)$ Schritten die j-Schleife simuliert. Falls anschließend wh true ist, geht die Turingmaschine in einen Endzustand, anderenfalls hält sie an, ohne in einen Endzustand zu gelangen.
Somit kann eine nichtdeterministische 6-Band-Turingmaschine das Problem, ob w nicht in WF liegt, in $O(n^2)$ Schritten lösen. Man sagt, nichtdeterministisch ist das Komplement von WF in quadratischer Zeit lösbar, während mit unserem Algorithmus oben der deterministische Fall $O(n^4)$ Schritte erfordert.

Der zusätzliche Aufwand für den Speicherplatz ist in beiden Fällen gleich.

Man kann das Problem nichtdeterministisch auch in linearer Zeit lösen.

Überlegen Sie sich ein geeignetes Vorgehen.

■

6.3 Churchsche These

Warum befassen wir uns mit Turingmaschinen?

Bisher haben wir Algorithmen in einer Programmiersprache als Programme formuliert. Diese Programme müssen einer Maschine (einem Computer, einem Rechner, einer Datenverarbeitungsanlage) übergeben werden, die sie ausführt. Mit Hilfe eines Compilers lassen sich die Programme in die Maschinensprache eines Computers übertragen und vom Computer durchrechnen. Die Behauptung lautet nun:

6.3.1. These: Jeder Algorithmus lässt sich durch eine geeignete deterministische Turingmaschine darstellen.
(oder durch eine geeignete deterministische Registermaschine, vgl. 6.6)

Dies wollen wir plausibel machen.

Beispiel 6.3.2: Betrachte folgenden Algorithmus, der hier als Block in Ada notiert ist:

```
declare z, s: Natural;
begin   Get(z); s := 3;
        while z > 1 loop
                s := s + z;
                z := z - 2;
        end loop;
        Put(s);
end;
```

Dieses Programmstück lässt sich schrittweise in eine deterministische Turingmaschine M umwandeln. Diese verwendet die Binärdarstellung für natürliche Zahlen.

Schritt 1: Lege k = 2 fest.
Verwende 2 Bänder: das erste für z und das zweite für s.

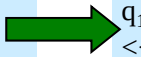
Schritt 2:
Wandle das Programm ein wenig um (auch: Binärdarstellung!).

```
declare z, s: Natural;
begin   Get(z); s := 3;
        while z > 1 loop
                s := s + z;
                z := z - 2;
        end loop;
        Put(s);
end;
```

```
2 Bänder für  z und s;
~~Get(z);~~ z steht auf Band 1;
Starte im Anfangszustand;
Schreibe 11 auf Band 2;
<< M1 >>
if z <= 1 then goto M2; end if;
s := s + z;
z := z - 2;
goto M1;
<< M2 >>: Lösche Band 1 und
      schreibe s auf Band 1;
Gehe in Endzustand
```

Schritt 3: Wandle das Programm weiter um.

2 Bänder für z und s;
z steht auf Band 1;
Starte im Anfangszustand;
Schreibe 11 auf Band 2;
<< M1 >>
if z <= 1 then goto M2; end if;
s := s + z;
z := z - 2;
goto M1;
<< M2 >>: Lösche Band 1 und
schreibe s auf Band 1;
Gehe in Endzustand



2 Bänder für z und s;
z steht auf Band 1;
Starte im Anfangszustand q_0;
q_0 c **b** q_1 c 1 0 1 für alle
q_1 c **b** q_2 c 1 0 0 c ∈ Γ
<< M1 >> (= Zustand q_2)
q_2 x y q_2 x y 1 0 ∀x,y∈Σ
q_2 **b** y q_3 **b** y -1 0
q_3 **b** y q_{14} **b** y 0 0
q_3 0 y q_4 0 y -1 0
q_3 1 y q_4 1 y -1 0
q_4 **b** y q_5 **b** y 1 0
q_4 x y q_6 x y 1 0
q_5 x y q_{14} **b** y 0 0
-- weiter mit Zustand q_6
s := s + z; z := z - 2; goto M1;
<< M2 >>: (= Zustand q_{14})
Lösche Band 1 und schreibe s auf Band 1;
Gehe in Endzustand

Schritt 4: Wandle das Programm weiter um.

2 Bänder für z und s;
z steht auf Band 1;
Starte im Anfangszustand q_0;
q_0 c **b** q_1 c 1 0 1 für alle
q_1 c **b** q_2 c 1 0 0 c ∈ Γ
<< M1 >> (= Zustand q_2)
q_2 x y q_2 x y 1 0 ∀x,y∈Σ
q_2 **b** y q_3 **b** y -1 0
q_3 **b** y q_{14} **b** y 0 0
q_3 0 y q_4 0 y -1 0
q_3 1 y q_4 1 y -1 0
q_4 **b** y q_5 **b** y 1 0
q_4 x y q_6 x y 1 0
q_5 x y q_{14} **b** y 0 0
-- weiter mit Zustand q_6
s := s + z; z := z - 2; goto M1;
<< M2 >>: (= Zustand q_{14})
Lösche Band 1 und schreibe s auf Band 1;
Gehe in Endzustand

q_0 c **b** q_1 c 1 0 1 für alle
q_1 c **b** q_2 c 1 0 0 c ∈ Γ
<< M1 >> (= Zustand q_2)
q_2 x y q_2 x y 1 0 ∀x,y∈Σ
q_2 **b** y q_3 **b** y -1 0
q_3 **b** y q_{14} **b** y 0 0
q_3 0 y q_4 0 y -1 0
q_3 1 y q_4 1 y -1 0
q_4 **b** y q_5 **b** y 1 0
q_4 x y q_6 x y 1 0
q_5 x y q_{14} **b** y 0 0
q_6 **b** **b** q_8 **b** **b** 1 1
q_6 **b** 0 q_6 **b** 0 -1 -1
q_6 **b** 1 q_6 **b** 1 -1 -1
q_6 0 **b** q_6 0 0 -1 -1
q_6 0 0 q_6 0 0 -1 -1
q_6 0 1 q_6 0 1 -1 -1
-- usw.

Schritt 5: Wandle das Programm weiter um.

Schritt 6: Wandle das Programm weiter um.

Schritt 7: Wandle das Programm weiter um.

... bis die Turingmaschine fertig ist, und zwar erhält man z.B.:

$M = (Q, \Sigma, \Gamma, \delta_M, q_0, F, \mathbf{b}, k)$ mit $Q = \{q_0, q_1, q_2, ..., q_{16}\}$, $F = \{q_{15}\}$, $\Sigma = \{0,1\}$, $\Gamma = \{0,1,\mathbf{b}\}$, $k = 2$ und δ_M sei durch folgende Tabelle gegeben (für alle $x, y \in \Sigma$, für alle $c, d \in \Gamma$):

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_0	c	**b**	q_1	c	1	0	1
q_1	c	**b**	q_2	c	1	0	0
q_2	x	y	q_2	x	y	1	0
q_2	**b**	y	q_3	**b**	y	-1	0
q_3	**b**	y	q_{14}	**b**	y	0	0
q_3	0	y	q_4	0	y	-1	0
q_3	1	y	q_4	1	y	-1	0

Erläuterung: Anfangs steht z ohnehin auf dem ersten Band (=Get(z)). Mit den Zuständen q_0 und q_1 schreibt die Turingmaschine die Zahl 3 (binär 11) auf das zweite Band. Der zweite Lese-Schreibkopf steht auf dem letzten Zeichen von s. Mit q_2 wird das Ende von z aufgesucht. Mit q_3 beginnt dann die while-Schleife. Es ist $z>1$ zu prüfen. $z \leq 1$ kann durch das leere Wort oder durch eine alleinstehende 0 oder 1 erkannt werden. Dies wird in den Zuständen q_3 und q_4 geprüft. Ist $z \leq 1$, so wird nach Zustand q_{14} verzweigt, anderenfalls nach q_6.

Additionstabelle ohne Übertrag

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_4	**b**	y	q_5	**b**	y	1	0
q_4	x	y	q_6	x	y	1	0
q_5	x	y	q_{14}	**b**	y	0	0
q_6	**b**	**b**	q_8	**b**	**b**	1	1
q_6	**b**	0	q_6	**b**	0	-1	-1
q_6	**b**	1	q_6	**b**	1	-1	-1
q_6	0	**b**	q_6	0	0	-1	-1
q_6	0	0	q_6	0	0	-1	-1
q_6	0	1	q_6	0	1	-1	-1
q_6	1	**b**	q_6	1	1	-1	-1
q_6	1	0	q_6	1	1	-1	-1
q_6	1	1	q_7	1	0	-1	-1

Erläuterung: Der Zustand q_5 dient dazu, das erste Band auf jeden Fall zu leeren, da am Ende s auf Band 1 kopiert werden muss (das Ergebnis muss am Ende auf dem ersten Band stehen, siehe 6.2.2, "3. Ausgabe").

Ist z > 1, so wird der erste Lese-Schreibkopf auf das letzte Zeichen von z gestellt. q_6 bedeutet den Eintritt in den Rumpf der Schleife. Es ist s := s+z zu berechnen. Dies geschieht wie bekannt stellenweise von rechts nach links, wobei das Blanksymbol wie die 0 behandelt wird und M sich den Übertrag im Zustand merkt (q_6 = kein Übertrag, q_7 = es liegt ein Übertrag vor). Die Addition ist beendet, wenn auf beiden Bändern das Blanksymbol gelesen wird (danach: Zustand q_8).

Additionstabelle mit Übertrag

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_7	**b**	**b**	q_8	**b**	1	1	1
q_7	**b**	0	q_6	**b**	1	-1	-1
q_7	**b**	1	q_7	**b**	0	-1	-1
q_7	0	**b**	q_6	0	1	-1	-1
q_7	0	0	q_6	0	1	-1	-1
q_7	0	1	q_7	0	0	-1	-1
q_7	1	**b**	q_7	1	0	-1	-1
q_7	1	0	q_7	1	0	-1	-1
q_7	1	1	q_7	1	1	-1	-1
q_8	**b**	**b**	q_9	**b**	**b**	-1	-1
q_8	sonst		q_8	sonst		1	1
q_9	c	d	q_{10}	c	d	-1	0

Erläuterung: Im Zustand q_8 werden beide Lese-Schreibköpfe wieder auf das letzte Zeichen der beiden Zahlen z und s gesetzt. Da beide Köpfe gleichmäßig bei der Addition nach links bewegt wurden, kann man beide Köpfe gleichzeitig nach rechts laufen lassen, bis auf beiden Bändern das Blanksymbol gelesen wird. "sonst" bedeutet in der Tabelle: alle sonstigen Kombinationen ungleich **b b**.

Ab Zustand q_9 wird nun die Wertzuweisung z := z - 2 ausgeführt. Binär bedeutet dies: Ziehe eine 1 ab der vorletzten Stelle ab. Den zweiten Lese-Schreibkopf ignoriert man daher und setzt den ersten eine Position nach links (Zustand q_{10}).

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_{10}	**b**	d	q_{16}	**b**	d	0	0
q_{10}	0	d	q_{10}	1	d	-1	0
q_{10}	1	d	q_{11}	0	d	-1	0
q_{11}	x	d	q_{11}	x	d	-1	0
q_{11}	**b**	d	q_{12}	**b**	d	1	0
q_{12}	0	d	q_{12}	**b**	d	1	0
q_{12}	**b**	d	q_{14}	**b**	d	0	0
q_{12}	1	d	q_{13}	1	d	1	0
q_{13}	x	d	q_{13}	x	d	1	0
q_{13}	**b**	d	q_3	**b**	d	-1	0
q_{14}	**b**	x	q_{14}	x	x	-1	-1
q_{14}	**b**	**b**	q_{15}	**b**	**b**	1	1

Erläuterung: Im Zustand q_{10} wird nun 1 subtrahiert, indem man nach links gehend jede 0 in eine 1 umwandelt, bis man auf ein Blanksymbol oder auf eine 1 trifft. Trifft man auf 1, so ersetze man sie durch 0 und beende die Subtraktion. Trifft man auf das Blanksymbol, dann muss ein Fehler vorgelegen haben, da z > 1 zu Beginn des Schleifenrumpfs war (wir fügen eine Halt-Zeile hierfür in δ_M ein, die unter normalen Bedingungen nicht erreicht werden kann).
Nun können aber führende Nullen entstanden sein, die die Abfrage "z>1" verfälschen würden. Also müssen erst die führenden Nullen beseitigt werden, indem man nach links zum Anfang der Zahl z geht und dann nach rechts laufend führende Nullen löscht.

Restliche Erläuterung: Damit ist das Ende des Schleifenrumpfs erreicht, und sobald man im Zustand q_{13} das rechte Ende von z gefunden hat, muss man die Schleife neu starten im Zustand q_3.
Trifft die Abfrage "z>1" nicht zu, so wird die Ausgabe vorbereitet. Hierzu muss die Zahl s auf das erste Band kopiert werden. Beachte, dass in diesem Fall das erste Band bereits (Zustände q_4 und q_5) gelöscht wurde. Man muss also im Zustand q_{14} nur von links nach rechts jedes Zeichen vom zweiten auf das erste Band schreiben, bis das Blanksymbol gelesen wird.
Damit endet der Algorithmus.

Überzeugen Sie sich durch penibles Nachvollziehen, dass diese Turingmaschine genau den vorgegebenen Algorithmus realisiert! Zugleich erkennen Sie, dass man mit Turingmaschinen while-Schleifen und beliebige Wertzuweisungen nachvollziehen kann. Auch die Alternative und andere Anweisungen lassen sich in die "Sprache der Turingmaschinen" übersetzen. ■

Die hier vorgestellte Übersetzungsmethode lässt sich automatisieren, so dass man einen Compiler angeben kann, der jedes Ada-Programm in eine gleichwertige Turingmaschine umwandelt. (Allgemeine Techniken zur Übersetzung: Veranstaltungen zum Compilerbau.)

Die Laufzeit der Turingmaschine ist im Prinzip gleich der Laufzeit des Ada-Programms, hinzu kommt aber, dass alle Operationen zeichenweise durchzuführen sind und dass kein Datenzugriff über Pointer oder Indizes erfolgt, sondern dass die Daten hintereinander auf den Bändern abgelegt sind und daher das Band sequentiell bis zu der Stelle, an der die gesuchten Daten stehen, durchlaufen werden muss.

Wir fassen diese Aussagen zusammen in drei Behauptungen:

Behauptung 6.3.3:

Jeder Algorithmus lässt sich durch eine Turingmaschine realisieren.

oder anders ausgedrückt:

Die formale Definition des intuitiven Begriffs "Algorithmus" ist die Turingmaschine.

Dies ist die Churchsche These aus dem Jahre 1936 (nach Alonzo Church, 1903-1995).

Diese Behauptung lässt sich nicht beweisen, da der Begriff "Algorithmus" nur umgangssprachlich festgelegt ist. Man hat den Begriff "Algorithmus" aber auf viele verschiedene Arten definiert und alle (z. B. die Registermaschine, 6.6.1) erwiesen sich als gleichwertig zur Turingmaschine. Daher ist man von der Gültigkeit der Churchschen These heute überzeugt.

Was wir aber beweisen können, ist die

Behauptung 6.3.4:

(1) Zu jedem Ada-Programm π gibt es eine Turingmaschine T_π, deren Resultatsfunktion gleich der von dem Ada-Programm realisierten Abbildung ist.

(2) Man kann eine berechenbare Funktion c konstruieren, die jedem Ada-Programm π eine solche Turingmaschine $c(\pi) = T_\pi$ zuordnet. (c heißt "Compiler".)

Der Beweis dieser Aussage ergibt sich daraus, dass man einen Compiler für Ada-Programme in eine Maschinensprache angibt und die Maschinensprache durch eine Turingmaschine simuliert, siehe Vorlesungen über Compilerbau.

Was wir weiterhin beweisen können, ist die

Behauptung 6.3.5:

Es gibt einen Compiler c mit folgender Eigenschaft:
Wenn das Ada-Programm π für die Eingabe w anhält und hierfür S(w) Speicherplätze belegt und T(w) Zeiteinheiten benötigt, so besitzt die Turingmaschine $c(\pi)$ einen Platzbedarf $O(S(w))$ und benötigt höchstens $O(S(w) \cdot T(w))$ Schritte.

Diese Behauptung folgt aus dem Beweis (den wir hier nicht führen können, siehe Vorlesung zur Komplexitätstheorie) zur Behauptung 6.3.4 durch genaue Analyse, wie der Speicher und wie die Anweisungen durch die Maschinensprache eines Computers und danach durch eine Turingmaschine simuliert werden. (Vgl. 6.4.10)

6.3.6: Schlussfolgerung aus diesen Überlegungen:

1. Ada-Programme und Turingmaschinen sind zwei gleichwertige Konzepte, um Algorithmen zu realisieren.

2. Jedes Verfahren, das im intuitiven Sinn ein Algorithmus ist, lässt sich durch eine Turingmaschine (oder ein Ada-Programm mit unbegrenzt großem Speicher) realisieren.

3. Wenn wir Aussagen über Algorithmen machen wollen, so genügt es, Turingmaschinen (oder Registermaschinen) zu betrachten (diese sind präzise definiert und daher formalen Untersuchungen zugänglich).

4. Wir können Komplexitätsklassen auf Programmen und auf Turingmaschinen einführen und hiermit Algorithmen genauer klassifizieren.

6.4 Komplexitätsklassen

Wir wollen nun Probleme, Algorithmen, Funktionen, Turingmaschinen oder Programme danach klassifizieren, wie viel Ressourcen sie für ihre Lösung oder Ausführung benötigen. Ressourcen sind vor allem Zeit und Platz.

Umgangssprachlich ist die Zeitkomplexität nichts anderes als die "Rechendauer".

Die "Rechendauer" ist abhängig von der Eingabe. In der Regel definiert man den Zeitaufwand $t_\pi: \mathbb{N}_0 \to \mathbb{N}_0$ als Funktion der Länge der Eingabe des Programms π:
$t_\pi(n)$ = maximale Zeit, die das Programm π für irgendeine Eingabe w der Länge n bis zum Anhalten benötigt.

Wie misst man die Zeit? Die Geschwindigkeit auf einem Computer ist ungeeignet, da sich diese mit jeder neuen Technik ändert. Wir werden die Zeit daher in "Schritten" messen. Bei Turingmaschinen ist dies "ganz einfach", da bei diesem formalen Modell genau definiert ist, was ein Schritt ist, nämlich ein einmaliges Anwenden der Übergangsrelation bzw. Übergangsfunktion δ. Bei Programmen zählen wir die elementaren Anweisungen und Ausdrücke, die man bis zum Anhalten des Programms ausführen oder auswerten muss; diese grobe Näherung wird uns zunächst reichen.

Für Komplexitätsuntersuchungen spielen nur Berechnungen, die anhalten, eine Rolle. Daher gehen wir von Algorithmen und Maschinen aus, die für alle Eingaben nach endlich vielen Schritten anhalten. Wir beginnen mit Turingmaschinen.

6.4.1: Gegeben sei eine k-Band-Turingmaschine
$M = (Q, \Sigma, \Gamma, \delta, q_0, F, \mathbf{b}, k)$, die für alle Eingaben anhalten möge, d.h., der Haltebereich von M sei Σ^*. Die Komplexität für die Eingabe w ist die minimale Anzahl der Schritte $t_M(w)$ oder die minimal erforderliche Zahl an Speicherplätzen $s_M(w)$, die M für die Eingabe von w benötigt, um seine Berechnung bis zum Anhalten durchzuführen.
(t kommt von "time complexity" und s von "space complexity".)

$t_M(w)$ = Min { i | Es gibt eine Berechnung, die bei Eingabe von w auf das erste Band von M genau i mal die Übergangsrelation δ verwendet und dann anhält}.

$s_M(w)$ = Min { i | Es gibt eine Berechnung, die bei Eingabe von w auf das erste Band von M anhält und hierbei genau j verschiedene Bandfelder besucht hat; die Felder für w werden stets mitgezählt}.

In dieser Definition wurde das Minimum gebildet. Der Grund liegt darin, dass die Turingmaschinen nichtdeterministisch sein dürfen und daher mehrere Berechnungen zu lassen.

In der Praxis werden letztlich deterministische Modelle benutzt. Wir müssen daher bei der Definition der Komplexitätsklassen zwischen deterministischen und nichtdeterministischen Turingmaschinen unterscheiden.

Man beachte auch, dass t_M und s_M nur im deterministischen Fall in der gleichen Berechnungsfolge ermittelt werden. Im nichtdeterministischen Fall kann eine Berechnungsfolge möglichst günstig für die Zeit und eine andere möglichst günstig für den Platzbedarf sein.

6.4.2: Für die Praxis ist die Definition zu speziell, da sie sich auf *jede einzelne* Eingabe bezieht. Man fasst daher alle Eingaben gleicher Länge zusammen und verwendet als Komplexität in der Regel den schlechtesten Fall ("worst case" complexity), manchmal den durchschnittlichen Fall ("average case") oder selten auch den besten Fall ("best case"). Daher definiert man die **Komplexität** wie folgt:

$t_M(n) = \text{Max}\,\{t_M(w) \mid \text{die Länge von w ist n}\}$,
$s_M(n) = \text{Max}\,\{s_M(w) \mid \text{die Länge von w ist n}\}$.

$$t_M^{av}(n) = \frac{1}{|\Sigma^n|} \sum_{w \in \Sigma^n} t_M(w), \quad s_M^{av}(n) = \frac{1}{|\Sigma^n|} \sum_{w \in \Sigma^n} s_M(w).$$

$t_M^{best}(n) = \text{Min}\,\{t_M(w) \mid \text{die Länge von w ist n}\}$,
$s_M^{best}(n) = \text{Min}\,\{s_M(w) \mid \text{die Länge von w ist n}\}$.

Im Folgenden benötigen wir noch die semicharakteristische Funktion, da bei nichtdeterministischen Maschinen bereits der Erfolgsfall ausreicht, um eine Menge zu beschreiben (während das Komplement meist nur durch Durchtesten aller Möglichkeiten erkannt werden kann).

Definition 6.4.3:

Für jedes $L \subseteq \Sigma^*$ heißt die partielle Abbildung

$\psi_L : \Sigma^* \to \{1\}$ mit

$\psi_L(w)$ = if $w \in L$ then 1 else undefiniert fi

die semi-charakteristische Funktion von L.

Die semi-charakteristische Funktion ist also für $w \in L$ gleich der charakteristischen Funktion (2.3.3) und sonst undefiniert.

6.4.4: Hieraus lassen sich nun die **Komplexitätsklassen** für das Berechnungsmodell Turingmaschinen präzise definieren. Wir verwenden hierfür den Zusatz ^{TM}.
Es seien T, S: $\mathbb{N}_0 \to \mathbb{N}_0$ zwei gegebene Funktionen, dann setze:

$\text{DTimeSpace}^{TM}(T,S) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine deterministische Turingmaschine M, die χ_L berechnet, mit $t_M(n) \in O(T(n))$ und $s_M(n) \in O(S(n)).\}$

$\text{NTimeSpace}^{TM}(T,S) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine nichtdeterministische Turingmaschine M, die ψ_L berechnet, mit $t_M(n) \in O(T(n))$ und $s_M(n) \in O(S(n)).\}$

Hinweis: χ_L ist die charakteristische Funktion von L, siehe 2.3.4.

6.4.4 (Fortsetzung)

$DTime^{TM}(T) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine deterministische k-Band-Turingmaschine M, die χ_L berechnet, mit $t_M(n) \in O(T(n)).\}$

$NTime^{TM}(T) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine nichtdeterministische k-Band-Turingmaschine M, die ψ_L berechnet, mit $t_M(n) \in O(T(n)).\}$

$DSpace^{TM}(S) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine deterministische k-Band-Turingmaschine M, die χ_L berechnet, mit $s_M(n) \in O(S(n)).\}$

$NSpace^{TM}(S) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine nichtdeterministische k-Band-Turingmaschine M, die ψ_L berechnet, mit $s_M(n) \in O(S(n)).\}$

Statt Sprachklassen kann man auf die gleiche Art auch Funktionsklassen (über die Resultatsabbildung) definieren. Diese bezeichnet man ebenfalls mit $DTime^{TM}(T)$ usw.

Bei nichtdeterministischen Berechnungen kann man sich auch andere Definitionen vorstellen. Wir verlangen hier jedoch nur, dass ψ_L berechnet wird. Im Falle, dass $w \in L$ ist, gibt es also für diese Eingabe w eine erfolgreich anhaltende Berechnung.

Weiterhin kümmern wir uns generell nicht um Konstanten, sondern nur um die Abhängigkeit von der Länge der Eingabe. Dies kann man theoretisch untermauern (siehe spätere Theorie-Vorlesungen), aber auch dadurch begründen, dass durch schnellere Computer jede einmal ermittelte Konstante hinfällig wird.

6.4.5. Einfache Folgerungen:

$DTime^{TM}(T) \subseteq NTime^{TM}(T)$ und
$DSpace^{TM}(S) \subseteq NSpace^{TM}(S)$
da deterministische Turingmaschinen ein Spezialfall der nichtdeterministischen sind.

$DTime^{TM}(T) \subseteq DSpace^{TM}(T)$ und
$NTime^{TM}(T) \subseteq NSpace^{TM}(T)$
da man, um T(n) Felder zu besuchen, mindestens T(n) Schritte gemacht haben muss.

Hinweis: Wir vertiefen nicht weiter. Für eine präzisere Bestimmung der Komplexität und daraus folgende Eigenschaften der Komplexitätsklassen siehe Veranstaltungen des Hauptstudiums. Wir wenden uns nun den Algorithmen zu.

6.4.6: Wir betrachten nur Algorithmen und Programme, die für alle Eingaben anhalten. Die Komplexität für die Eingabe w ist die Anzahl der Schritte $t_A(w)$ oder die Zahl an Speicherplätzen $s_A(w)$, die ein Programm oder ein Algorithmus A für die Eingabe von w benötigt, um seine Berechnung bis zum Anhalten durchzuführen.

Genauer: Sei A ein Algorithmus (oder Programm). Eine *Berechnung* für die Eingabe w ist eine Folge von elementaren Anweisungen und Ausdrücken. Hiermit definieren wir:

$t_A(w)$ = Anzahl der elementaren Anweisungen und Ausdrücke, die der Algorithmus A in seiner Berechnung bei Eingabe von w durchläuft, bis er anhält.

$s_A(w)$ = Anzahl der Speicherplätze, auf die A während seiner Berechnung bei Eingabe von w zugreift, bis er anhält.

6.4.7: Nun geht man wie bei Turingmaschinen weiter vor: Man fasst die Eingaben gleicher Länge zusammen und verwendet als Komplexität in der Regel den schlechtesten Fall ("worst case" complexity), manchmal den durchschnittlichen Fall ("average case") oder selten den besten Fall ("best case"). Daher definiert man die **Zeit- und Platz-Komplexität** für Programme/Algorithmen A wie folgt:

$t_A(n) = \text{Max}\ \{t_A(w) \mid \text{die Länge von w ist n}\}$,
$s_A(n) = \text{Max}\ \{s_A(w) \mid \text{die Länge von w ist n}\}$.

$$t_A^{av}(n) = \frac{1}{|\Sigma^n|} \sum_{w \in \Sigma^n} t_A(w), \qquad s_A^{av}(n) = \frac{1}{|\Sigma^n|} \sum_{w \in \Sigma^n} s_A(w).$$

$t_A^{best}(n) = \text{Min}\ \{t_A(w) \mid \text{die Länge von w ist n}\}$,
$s_A^{best}(n) = \text{Min}\ \{s_A(w) \mid \text{die Länge von w ist n}\}$.

6.4.8: Hieraus lassen sich nun die **Komplexitätsklassen** für Algorithmen definieren. Es seien T, S: $\mathbb{N}_0 \to \mathbb{N}_0$ zwei gegebene Funktionen, dann setze:

DTimeSpace (T,S) = { $L \subseteq \Sigma^*$ | Es gibt einen deterministischen Algorithmus A, der χ_L berechnet, mit $t_A(n) \in O(T(n))$ und $s_A(n) \in O(S(n))$.}

NTimeSpace (T,S) = { $L \subseteq \Sigma^*$ | Es gibt einen nichtdeterministischen Algorithmus A, der ψ_L berechnet, mit $t_A(n) \in O(T(n))$ und $s_A(n) \in O(S(n))$.}

6.4.8 (Fortsetzung)

DTime (T) = { $L \subseteq \Sigma^*$ | Es gibt einen deterministischen Algorithmus A, der χ_L berechnet, mit $t_A(n) \in O(T(n))$.}

NTime (T) = { $L \subseteq \Sigma^*$ | Es gibt einen nichtdeterministischen Algorithmus A, der ψ_L berechnet, mit $t_A(n) \in O(T(n))$.}

DSpace (S) = { $L \subseteq \Sigma^*$ | Es gibt einen deterministischen Algorithmus A, der χ_L berechnet, mit $s_A(n) \in O(S(n))$.}

NSpace (S) = { $L \subseteq \Sigma^*$ | Es gibt einen nichtdeterministischen Algorithmus A, der ψ_L berechnet, mit $s_A(n) \in O(S(n))$.}

Statt Sprachklassen kann man auf die gleiche Art auch Funktionsklassen definieren. Diese bezeichnet man ebenfalls mit DTime (T), NTime (T), DSpace (S) und NSpace (S).

Nichtdeterministische Berechnungen haben wir bei Algorithmen ansatzweise in 2.2.2 kennen gelernt. Solche Berechnungen sind denkbar, sofern man nicht-deterministische Sprachelemente in die Programmiersprache einführt. Neben der dort angegebenen Kontrollstruktur "|" sind die "guarded commands" ein Standardbeispiel, bei denen mehrere Bedingungen zutreffen können, worunter eine zutreffende Bedingung willkürlich ausgewählt und ausgeführt wird (siehe 6.6.7). Jedoch wird "echter Nichtdeterminismus" bisher von Programmiersprachen nicht unterstützt.

Auch hier kümmern wir uns generell nicht um Konstanten, sondern nur um die Abhängigkeit von der Länge der Eingabe.

6.4.9. Einfache Folgerungen (wie in 6.4.5):

DTime (T) $\subseteq$ NTime (T), DSpace (S) $\subseteq$ NSpace (S).

DTime (T) $\subseteq$ DSpace (T), NTime (T) $\subseteq$ NSpace (T).

Unterschied zu den TM-Komplexitätsklassen:

a) Die Auswertung eines Ausdrucks kostet bei Algorithmen nur einen Schritt.
b) Die Ausführung einer Elementaranweisung kostet nur einen Schritt.
c) Der Zugriff auf eine (indizierte) Variable erfolgt in einem Schritt.

Man nennt die oben definierte Komplexität bei Algorithmen und Programmen daher das "uniforme Komplexitätsmaß".

6.4.10: Aus der Behauptung 6.3.5 und anderen Überlegungen erhält man Zusammenhänge zwischen den verschiedenen Maßen (ohne Beweis):

DTimeSpace (T,S) $\subseteq$ DTimeSpace™ (T·S, S)

DTimeSpace™ (T, S) $\subseteq$ DTimeSpace (T·log(S), S)

Die Komplexitätsklassen sind daher recht ähnlich. Statt mit den üblichen Klassen der Turingmaschinen zu arbeiten, können wir daher die anschaulicheren Klassen für Algorithmen verwenden.

Polynomiell beschränkte Zeit-Komplexitätsklassen:

Für ein Polynom p(n) vom Grad k ist wegen der O-Klassen:

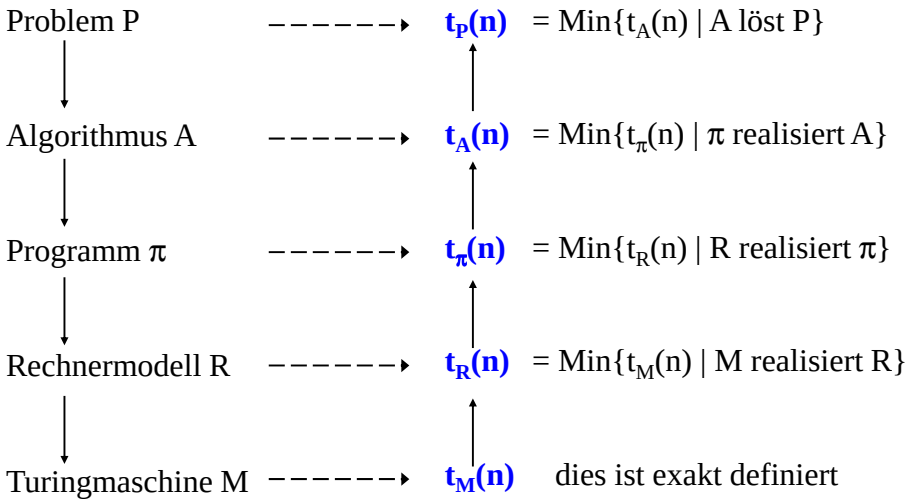
DTime™ (n^k) = DTime™ (p(n)) . Setze:

$$\mathbf{P} = \bigcup_{k\geq 0} \text{DTime}^{TM}(n^k) \qquad \mathbf{NP} = \bigcup_{k\geq 0} \text{NTime}^{TM}(n^k)$$

Berühmtestes Problem derzeit: Ist **P** = **NP** oder nicht?

Das heißt: Gibt es zu jedem **n**ichtdeterministisch **p**olynomiellen Algorithmus einen deterministischen Algorithmus, der die gleiche Aufgabe in **p**olynomieller Zeit löst? (Es gilt natürlich **P** $\subseteq$ **NP**; man vermutet **P** $\neq$ **NP**.)

6.4.11 Idee der Definition von Komplexitätsfunktionen = Rückführung auf ein universelles Rechnermodell, z.B. TM.



6.4.12 Uniforme Komplexität

Berechnet man die Komplexität $t_A(w)$ bzw. $s_A(w)$ so, dass jede elementare Anweisung und jede Abfrage genau einen Schritt dauern und dass jeder elementare Wert (z.B. jede Zahl) genau einen Speicherplatz belegt, so spricht man von der **uniformen Komplexität**. In diesem Fall braucht man in der Hierarchie der vorherigen Folie nicht auf Turingmaschinen zurückgreifen, sondern kann die Komplexität auf höherer Ebene direkt abschätzen.
Berechnet man die Komplexitäten so, dass jede Operation so viel Zeit kostet, wie die zeichenweise Ausführung erfordert, bzw. dass Werte so viel Platz belegen, wie sie Zeichen haben, so spricht man von der **logarithmischen Komplexität** (sie entspricht der Turingmaschine).

In der Praxis berechnet man zunächst die uniforme Komplexität: Sie gibt meist eine hinreichend genaue Orientierung über den zu erwartenden Aufwand. Erst für eine detaillierte Abschätzung bzw. bei der Programmierung betrachtet man die logarithmische Komplexität, die den tatsächlichen Aufwand genauer wiedergibt.

6.5 Beispiele

Die Komplexität einzelner Algorithmen wurde bereits an einigen Stellen der Vorlesung betrachtet.

Ein Beispiel ist die "binäre Suche", auch Intervallschachtelung (schnelle Suche in einem sortierten Feld) genannt.

Als Erstes präsentieren wir dieses Beispiel, wobei wir die zunächst nicht einsichtige Aussage nachweisen, dass die Abfrage auf Gleichheit sich im Mittel als schlechter herausstellt, als wenn man nur auf "<" abfragt. Danach betrachten wir den ggT, die Multiplikation von Zahlen, die transitive Hülle von Graphen und das Problem des Handlungsreisenden (traveling salesman problem, TSP):

Beispiel 6.5.1 Intervallschachtelung oder binäre Suche

Hier ist n die Anzahl der Elemente im Feld und nicht die Länge der Eingabe (diese wäre etwa um den Faktor log(n) größer).

Ein Feld A: array (1..n) of Integer sei gegeben. Das Feld sei sortiert, d.h.: $A(i) \le A(i+1)$ für $i = 1, 2, ..., n-1$.

Aufgabe: Man schreibe einen Algorithmus, der zu einer Zahl S in möglichst kurzer Zeit feststellt, ob S im Feld A liegt oder nicht. Im Falle, dass S im Feld A enthalten ist, soll ein Index m mit A(m) = S ausgegeben werden, anderenfalls sei m = 0.

Geht man das Feld von links nach rechts durch, so dauert es bis zu n Schritte, um das Ergebnis zu ermitteln.

Ein schnelleres Verfahren ist die Intervallschachtelung: Teste, ob S genau in der Mitte A(Mitte) von A liegt, falls nein und es ist A(Mitte) < S, suche rechts von der Mitte weiter, sonst links.

Programm 1 (in Ada): wir setzen hier willkürlich nmax = 100_000.

```
procedure Search1 is
Mitte, Links, Rechts, S: Integer; Gefunden: Boolean;
A: array (1..100_000) of Integer;
begin  ...;  -- "lies n, das Feld A und die zu suchende Zahl S ein"
     Links:=1; Rechts := n; Gefunden := false;
     while (Links <= Rechts) and (not Gefunden) loop
          Mitte := (Rechts+Links) / 2;
          if A(Mitte) = S then Gefunden := true;
          elsif A(Mitte) < S   then Links := Mitte+1;
                               else Rechts := Mitte-1; end if;
     end loop;
     if not Gefunden then Mitte := 0; end if;
     ...;                          -- "drucke das Ergebnis Mitte aus"
end Search1;
```

Wie lange dauert es, bis dieser Algorithmus spätestens endet?

Hierzu zählen wir die Wertzuweisungen und Bedingungen, die im ungünstigsten Fall ausgerechnet werden müssen.

In diesem Sinne dauert die Durchführung der 3 Anweisungen

Links:=1; Rechts := n; Gefunden := false;

genau 3 Schritte.

In der Schleife werden bis zu 6 Schritte benötigt, nämlich je einer für die vier Bedingungen (Links <= Rechts), (not Gefunden), A(Mitte) = S und A(Mitte) < S und je einer für zum Beispiel die Wertzuweisungen

Mitte := (Rechts+Links) / 2; und Links := Mitte+1;

[Zählt man das and noch zusätzlich, so sind es sogar bis zu 7 Schritte; das Folgende ist dann entsprechend anzupassen.]

Wie oft wird die Schleife durchlaufen? Das Intervall von Links bis Rechts halbiert sich mindestens in jedem Schritt, folglich muss nach log(n) Schleifendurchläufen Schluss sein.

3 Schritte

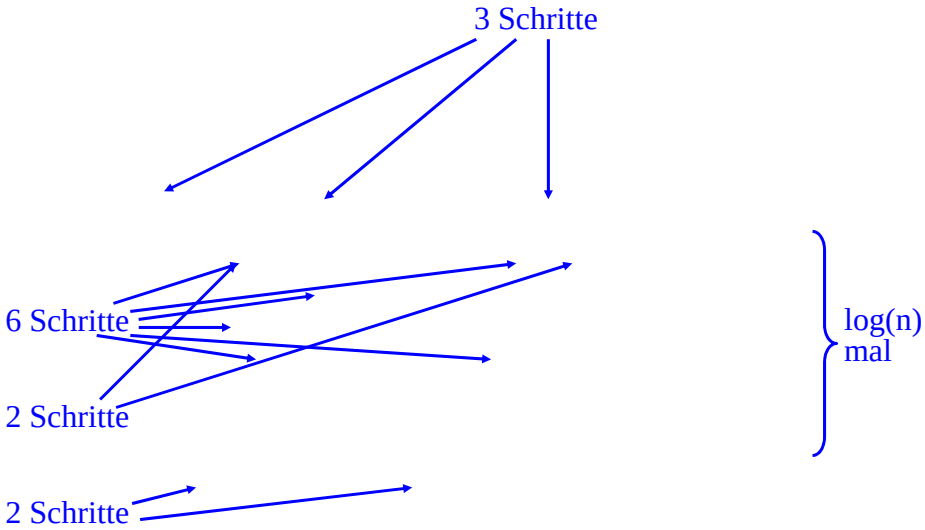
```
procedure Search1 is
Mitte, Links, Rechts, S: Integer; Gefunden: Boolean;
A: array (1..100_000) of Integer;
begin ...; -- "lies n, das Feld A und die zu suchende Zahl S ein"
   Links:=1; Rechts := n; Gefunden := false;
   while (Links <= Rechts) and (not Gefunden) loop
      Mitte := (Rechts+Links) / 2;
      if A(Mitte) = S then Gefunden := true;
      elsif A(Mitte) < S  then Links := Mitte+1;
                          else Rechts := Mitte-1; end if;
   end loop;
   if not Gefunden then Mitte := 0; end if;
   ...;                   -- "drucke das Ergebnis Mitte aus"
end Search1;
```

6 Schritte

2 Schritte

log(n) mal

2 Schritte



Gesamt: 6·log(n) + 7 Schritte ∈ O(log(n))

Schlechtester Fall (der "worst case") tritt ein, wenn das gesuchte Element S nicht im Feld A enthalten ist. Wir sagen: Die *uniforme worst case Zeitkomplexität t(n)* des Programms Search1 lautet

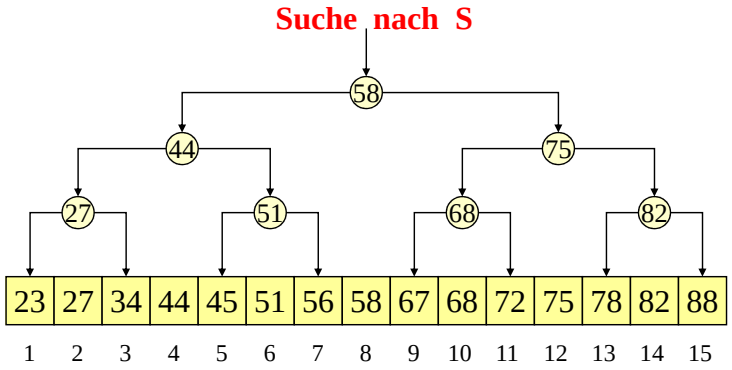
$$t(n) = 6 \cdot \log(n) + 7.$$

Beachten Sie: n ist hier die Anzahl der Elemente im Feld A. "Uniform", weil wir annehmen, alle Wertzuweisungen und Bedingungen würden die gleiche Zeit kosten.

Bester Fall (best case): In diesem Fall wird S im ersten Durchgang der while-Schleife gefunden, also nach 13 Schritten.

Durchschnittlicher Fall (average case): Hierzu nehmen wir an, dass sich das gesuchte Element S tatsächlich im Feld A befindet (sonst kann man nur die obige worst case Abschätzung verwenden).

Wir skizzieren die Verhältnisse, wobei wir hier $n=15=2^4-1$ wählen:



In $2^3 = 8$ Fällen braucht man 4 Schleifendurchläufe,
in $2^2 = 4$ Fällen braucht man 3 Schleifendurchläufe,
in $2^1 = 2$ Fällen braucht man 2 Schleifendurchläufe,
in $2^0 = 1$ Fall braucht man 1 Schleifendurchlauf.

Allgemein gilt also, wenn $n = 2^k - 1$ ist:

In 2^{k-1} Fällen braucht man k Schleifendurchläufe,
in 2^{k-2} Fällen braucht man k-1 Schleifendurchläufe,
in 2^{k-3} Fällen braucht man k-2 Schleifendurchläufe,
....
in $2^0 = 1$ Fall braucht man 1 Schleifendurchlauf.

Daher braucht man im Mittel:

$\frac{1}{n}\,(k\cdot 2^{k-1} + (k-1)\cdot 2^{k-2} + (k-2)\cdot 2^{k-3} + \ldots + 2\cdot 2^1 + 1)$ Durchläufe.

Berechne also die Summe $\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^j$

$$\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^j = \frac{1}{2}\sum_{j=1}^{k} (j-1)\cdot 2^j + \frac{1}{2}\sum_{j=1}^{k} 2^j$$

$$= \sum_{j=1}^{k} (j-1)\cdot 2^{j-1} + \frac{1}{2}(2^{k+1} - 2)$$

$$= \sum_{j=0}^{k-1} j\cdot 2^j + (2^k - 1) = \sum_{j=1}^{k} j\cdot 2^j - k\cdot 2^k + (2^k - 1), \quad \text{d.h.:}$$

$\frac{1}{2}\sum_{j=1}^{k} j\cdot 2^j = k\cdot 2^k - (2^k - 1)$. Folglich erhalten wir

$$\frac{1}{n}\,(k\cdot 2^{k-1} + (k-1)\cdot 2^{k-2} + \ldots + 2\cdot 2^1 + 1) = \frac{k\cdot 2^k - (2^k - 1)}{2^k - 1} \approx k - 1$$

Somit beträgt die *average case Zeitkomplexität* der Intervallschachtelung ziemlich genau $6\cdot\log(n) + 1$ Schritte, also nur einen Schleifendurchlauf weniger als im schlechtesten Fall.

Erkenntnis: Im Mittel spart man nur eine konstante Zahl an Operationen gegenüber dem schlechtesten Fall. Folglich lohnt sich zum Beispiel die Abfrage "A(Mitte) = S" nicht! Könnte man sie weglassen, so würde man log(n) viele Schritte sparen. Dies führt auf folgende bessere Version des Algorithmus für die Suche mittels Intervallschachtelung:

Man entscheide erst ganz am Ende, ob A(Mitte) = S ist; hierzu muss man im Falle A(Mitte) < S im rechten Teil des Feldes weitersuchen (Links:=Mitte+1), anderenfalls im linken Teil einschließlich des gerade betrachteten Feldes Mitte (Rechts:=Mitte, statt "Mitte+1", da A(Mitte) gleich S sein könnte).

Programm 2 (in Ada): im Mittel besser als Search1

```
procedure Search2 is
Mitte, Links, Rechts, S: Integer; Gefunden: Boolean;
A: array (1..100_000) of Integer;
begin  ...;  -- "lies n, das Feld A und die zu suchende Zahl S ein"
     Links:=1; Rechts := n;
     while (Links < Rechts) loop
          Mitte := (Rechts+Links) / 2;
          if A(Mitte) < S   then Links := Mitte+1;
                            else Rechts := Mitte; end if;
     end loop;
     Gefunden := A(Mitte) = S;
     if not Gefunden then Mitte := 0; end if;
     ...;                          -- "drucke das Ergebnis Mitte aus"
end Search2;
```


Weisen Sie nun nach, dass für diese Version Search2 gilt:

Die *uniforme worst case Zeitkomplexität* beträgt $5 + 4 \cdot \log(n)$; die *uniforme average case Zeitkomplexität* besitzt genau den gleichen Wert.

Hierbei ist n die Zahl der Elemente im array.

Version Search2 braucht also im Mittel und im schlechtesten Fall 2/6 ≈ 33% weniger Abfragen als Version Search1.

Weitere Suchverfahren betrachten wir im Kapitel 8 dieser Vorlesung.

Beispiel 6.5.2 Größter gemeinsamer Teiler

Der ggT (englisch: *gcd* greatest common divisor) wurde bereits in 1.6, 1.7.1, 1.7.3, 1.7.7 und 2.4.2 bis 2.4.4. im Detail behandelt.

Frage: Wie lange arbeitet der Euklidische Algorithmus, bis er anhält?

Beispielrechnung mit a = 144, b = 89:

$$ggT(144,89) = ggT(89,55) = ggT(55,34) = ggT(34,21)$$
$$= ggT(21,13) = ggT(13,8) = ggT(8,5) = ggT(5,3)$$
$$= ggT(3,2) = ggT(2,1) = ggT(1,1) = ggT(1,0) = 1.$$

Dieses Beispiel benötigt 11 Schleifendurchläufe.

6.5.2.1 Algorithmus:

```
-- es ist hier ggT(a,0) = a, auch für a=0.
declare A, B, R: Natural;
begin  Get (A); Get (B);
       while B≠0 loop  R:=A mod B; A:=B; B:=R;  end loop;
       Put (A);
end
```

Werteverlauf für die Variablen A und B bei Eingabe von a und b:

A	B
a	b
b	a mod b
a mod b	b mod (a mod b)
b mod (a mod b)	(a mod b) mod (b mod (a mod b))

Kann man etwas über die Größe dieser Werte sagen?

Hilfssatz 6.5.2.2:

b mod (a mod b) < b/2, für alle a ≥ b > 0, a mod b ≠ 0, d.h.,

nach spätestens zwei Schleifendurchläufen hat sich der Wert von B mehr als halbiert.

Beweis: Es ist immer 0 ≤ a mod b < b.

Fall 1: 0 ≤ a mod b ≤ b/2.
Dann gilt: b mod (a mod b) < a mod b ≤ b/2,
da für y > 0 stets x mod y < y ist.

Fall 2: b/2 < a mod b < b.
Dann gilt: b mod (a mod b) = b - (a mod b) < b - b/2 = b/2,
da für x/2 < y < x stets x mod y = x - y ist.

Damit ist der Hilfssatz bewiesen.

Folgerung 6.5.2.3:

Der Euklidische Algorithmus durchläuft höchstens 2·log(b) Mal seine Schleife (b = Anfangswert der Variablen B).

Die uniforme Zeitkomplexität des Euklidischen Algorithmus

```
begin Get (A); Get (B);
      while B≠0 loop R:=A mod B; A:=B; B:=R; end loop;
      Put (A);
end
```

beträgt somit höchstens 8·log(b) + 4 Zeiteinheiten.
Da log(a) + log(b) = n die Länge der Eingabe ist und log(b) somit in O(n) liegt, folgt:
Der Euklidische Algorithmus besitzt eine lineare uniforme Zeitkomplexität (d.h., er liegt in DTime (n)). Diese günstige Komplexitätsklasse liegt aber nur an der Uniformität.

6.5.2.4: Schwäche der uniformen Zeitkomplexität:

Sie misst, wie viele elementare Anweisungen ausgeführt und wie viele Ausdrücke ausgewertet werden, aber sie gibt keine Auskunft darüber, wie aufwendig die Durchführung einer elementaren Anweisung oder die Ausrechnung eines Ausdrucks ist.

Wie lange dauert denn die Wertzuweisung R := A mod B ?
Der ungünstigste Fall liegt vor, wenn A k Stellen und B ungefähr k/2 Stellen lang ist. Im Binärsystem führt man dann k/2 Subtraktionen der Länge k/2 aus, d.h., die Berechnung des Restes A mod B erfordert zeichenweise $O(k^2)$ Schritte.
Da k beim Euklidischen Algorithmus in der Größenordnung von n liegt (k = log(a)), *würde der obige Algorithmus also in Wahrheit $O(n^3)$ statt O(n) Schritte erfordern.*

Genau dieses Ergebnis würde die Komplexitätsuntersuchung mit Hilfe von Turingmaschinen ergeben, d.h., der Euklidische Algorithmus liegt in $DTime^{TM}(n^3)$.

Dennoch hat die uniforme Komplexität gewisse Vorteile. In den meisten praktischen Fällen wird die mod-Funktion durch einen Coprozessor für natürliche Zahlen, die in einem "normalen" Bereich liegen, oder mit Hilfe einer Softwaresimulation berechnet. In solchen Fällen *scheint* die Modulo-Bildung in konstanter Zeit abzulaufen, so dass sich der Euklidische Algorithmus in der Praxis meist wie ein Linearzeitalgorithmus verhält.

Aufgabe: Man kann zeigen, dass das Problem, den ggT zeichenweise zu berechnen, bereits in $DTime^{TM}(n^2)$ liegt. Versuchen Sie sich einmal an diesem Problem.

■

Beispiel 6.5.3 Multiplikation natürlicher Zahlen
(siehe Übungen!)

Wie lange dauert die ziffernweise Multiplikation, die wir in der Schule erlernen? Dort wird für die Multiplikation a·b ein Schema aufgeschrieben, das an ein Parallelogramm erinnert; dessen Fläche ist log(a)·log(b). Wenn beide Zahlen a und b ungefähr n/2 Stellen besitzen, so dauert die Multiplikation zeichenweise also $O(n^2)$ Schritte.

Als Programm nimmt man gewisse Modifikationen vor, um statt des Parallelogramms immer nur eine Zeile bzw. eine Variable mitzuführen. Eine einfache Technik beruht auf den Formeln: Sei Z (anfangs 0) das angestrebte Ergebnis, so gilt

$Z + A{\cdot}B = Z + (2A){\cdot}(B/2)$, sofern B gerade ist,
$Z + A{\cdot}B = (Z + A) + A{\cdot}(B-1)$, sofern B ungerade ist.

Dies ergibt folgendes Verfahren:

Algorithmus:

```
declare A, B, Z: Natural;
begin  Get (A); Get (B);
       Z:=0;
       while B > 0 loop
          if (B mod 2 = 0) then B := B div 2; A := A+A;
          else B := B-1; Z := Z+A;  end if;  end loop;
       Put (Z);
end
```

Wenn B > 0 ist, so beginnt die Binärdarstellung von B mit einer 1 und somit ist B im letzten Schleifendurchlauf 1; dort wird also spätestens der Wert von Z verändert.

[*Hinweis zur Semantik später:* Die Schleifeninvariante lautet {Z+A·B = a·b}, wenn a und b die eingelesenen Anfangswerte sind. D.h.: Obiger Algorithmus berechnet korrekt das Produkt.]

Wie lange dauert dieser Algorithmus für die Eingaben a, b?
Die uniforme Zeitkomplexität liefert:

```
declare A, B, Z: Natural;
begin  Get (A); Get (B);        2 Schritte
       Z:=0;                    1 Schritt
       while B > 0 loop         1 Schritt          2k+1 mal
          if (B mod 2 = 0) then 1 Schritt
             B := B div 2; A := A+A;   2 Schritte
          else                                     2k mal
             B := B-1; Z := Z+A;  end if;  end loop;  2 Schritte
       Put (Z);                 1 Schritt
end
```

Insgesamt: 8k + 5 Schritte, mit k = log(b) und wegen k ≈ n/2 liegt dies in DTime(n).

Zeichenweises Arbeiten: Betrachten wir nun die elementaren Anweisungen im Detail. Wir nehmen an, alle Zahlen seien binär dargestellt. Die Eingabelänge sei n = log(a) + log(b). Damit ist auch die Ausgabe durch n Stellen beschränkt.

Die Anweisungen Get (A), Get (B) und Put (Z) dauern zeichenweise so lange, wie die Zahlen lang sind, also O(n). Z:=0 dauert einen Schritt. Ebenso kann man "B>0" in einem Schritt entscheiden. Der Ausdruck (B mod 2 = 0) kostet einen Schritt, da man nur das letzte Zeichen von B auf Null testen muss. B := B div 2 und A := A+A kosten ebenfalls jeweils einen Schritt, da nur die letzte Null gestrichen bzw. eine Null angehängt werden muss. Auch B := B-1 kostet nur einen Schritt, da B ja ungerade ist und daher nur die letzte 1 in eine 0 umgewandelt werden muss. Z := Z+A kann dagegen bis zu n Schritte dauern. Es ist also der else-Zweig, der die Laufzeit im Wesentlichen bestimmt.

In den else-Zweig gelangt man immer, wenn B ungerade ist. Dies ist genau dann der Fall, wenn in der Binärdarstellung von B am Ende eine 1 steht. Da im then-Zweig die letzte Ziffer von B jeweils gestrichen wird, so wird der else-Zweig also genau so oft durchlaufen, wie Einsen in der Binärdarstellung von b auftreten. Der then-Zweig dagegen wird genau so oft durchlaufen, wie b lang ist, also log(b) mal.

Es sei n die Länge der Eingabe und eins(b)

$$1 \le \text{eins}(b) \le \log(b) \le n \in O(n)$$

die Anzahl der Einsen in der Binärdarstellung der Eingabezahl b, dann wird der else-Zweig insgesamt genau eins(b) Mal durchlaufen (je ein Schritt für "B:=B-1" und bis zu n Schritte für die Addition "Z:=Z+A") und der then-Zweig genau log(b) Mal mit jeweils 2 Schritten. Die Abfrage "$B \bmod 2 = 0$" erfolgt genau eins(b)+log(b) Mal und die Abfrage "$B > 0$" einmal mehr.

Somit erfordert die zeichenweise Zeitkomplexität des obigen Algorithmus zur Multiplikation höchstens

$$\text{eins}(b)\cdot(n+1) + 2\cdot\log(b) + 2\cdot(\text{eins}(b) + \log(b)) + 1 + 5$$

$$= \text{eins}(b)\cdot(n+3) + 2\cdot\log(b) + 6 \in O(n^2) \text{ Schritte.}$$

(Hierbei wurde die Ein- und Ausgabe jeweils nur mit einem Schritt berücksichtigt; realistisch sind je n Schritte, jedoch ändert dies nichts an der Größenordnung.)

Dies ist auch die Zeitkomplexität der Turingmaschine, die man erhält, wenn man obigen Algorithmus schrittweise in eine Turingmaschinendarstellung umwandelt. Das heißt: Der hier vorgestellte Multiplikationsalgorithmus liegt in $DTime^{™}(n^2)$, und weil eins(b) bis zu n/2 groß sein kann, liegt seine worst-case-Komplexität nicht in $o(n^2)$, sondern in $\Theta(n^2)$.

■

6.5.4 *Gibt es einen noch schnelleren Algorithmus für die Multiplikation?* *Oder kann man beweisen, dass es kein schnelleres Verfahren geben kann?*

Wenn man zwei natürliche Zahlen (ungleich Null), die jeweils k bzw. m Ziffern lang sind, miteinander multipliziert, so entsteht eine Zahl, die (k+m-1) oder (k+m) Ziffern besitzt.
Da das Ergebnis der Multiplikation nur so lang sein kann wie die beiden Faktoren zusammen, könnte es eventuell einen Algorithmus geben, der die Multiplikation proportional zur Eingabelänge n durchführt.
Dies würde bedeuten, dass man die Multiplikation bis auf einen konstanten Faktor genau so schnell ausführen könnte wie die Addition.
Das glaubt eigentlich niemand. Dennoch ist es bisher nicht gelungen zu beweisen, dass die Multiplikation deutlich mehr Zeit als die Addition erfordert (siehe Hinweis am Ende von 6.2.4.c).

Nun wollen wir ein einfaches Verfahren vorstellen, welches schneller als mit der Zeitkomplexität $O(n^2)$ multipliziert.

Gegeben seien zwei k-stellige natürliche Zahlen x und y (mit $k \approx n/2$); die Länge k sei eine gerade Zahl. Setze p=k/2. Dann lassen sich x und y schreiben in der Form:

$$x = A\cdot 2^p + B \quad \text{und} \quad y = C\cdot 2^p + D,$$

wobei A, B, C, D mindestens 0 und echt kleiner als 2^p sind. Dann gilt

$$x \cdot y = (A\cdot 2^p + B)\cdot(C\cdot 2^p + D) = A\cdot C\cdot 2^{2p} + (A\cdot D + B\cdot C)\cdot 2^p + B\cdot D.$$

Man kann also die Multiplikation zweier k-stelliger Zahlen durchführen, indem man sie auf 4 Multiplikationen von halber Länge k/2 und drei Additionen zurückführt. Dies ergibt erneut eine quadratische Zeitkomplexität. Kommt man vielleicht mit weniger als vier Multiplikationen halber Länge aus?

Es gilt: $(A \cdot D + B \cdot C) = (A-B) \cdot (D-C) + A \cdot C + B \cdot D$,
wie man durch Ausrechnen leicht nachprüft.

Somit erhalten wir aus obiger Gleichung:

$$x \cdot y = A \cdot C \cdot 2^{2p} + (A \cdot D + B \cdot C) \cdot 2^p + B \cdot D$$
$$= A \cdot C \cdot 2^{2p} + ((A-B) \cdot (D-C) + A \cdot C + B \cdot D) \cdot 2^p + B \cdot D$$

wobei A, B, C, D mindestens 0 und echt kleiner als 2^p sind.

Nun haben wir es schon geschafft: Auf der rechten Seite stehen nur noch drei verschiedene Multiplikationen:

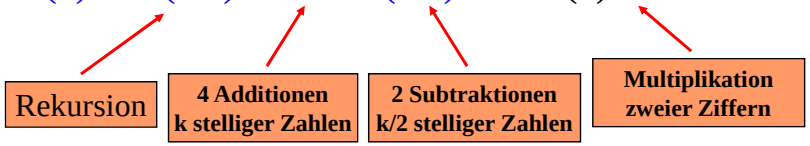
$A \cdot C$, $B \cdot D$ und $(A-B) \cdot (D-C)$

Beachte: Die Multiplikation mit 2^p bzw. 2^{2p} ist keine echte Multiplikation, sondern nur ein Anhängen von p bzw. 2p Nullen, wenn man zur Basis 2 rechnet.

Somit haben wir die Multiplikation zweier k-stelliger Zahlen auf drei Multiplikationen von k/2-stelligen Zahlen zurückgeführt. Der Preis, den wir dafür bezahlen müssen, sind zwei zusätzliche Subtraktionen zweier k/2-stelliger Zahlen und eine zusätzlich Additionen zweier k-stelliger Zahlen. Da aber die Zeit für die Addition und die Subtraktion proportional zur Länge der Zahlen ist, müssten wir dennoch schneller fertig werden. Wir prüfen dies nach.

Wenn $t(k)$ die Anzahl der durchzuführenden Operationen für die Multiplikation zweier k-stelliger Zahlen nach diesem Verfahren ist, dann erhalten wir also folgende Gleichung:

$$t(k) = 3 \cdot t(k/2) + 4 \cdot k + 2 \cdot (k/2) \quad \text{mit} \quad t(1)=1$$



Wie lautet die Lösung dieser Gleichung?

Probieren wir es aus. Ersetzen von $t(k/2)$, $t(k/4)$ usw. entsprechend der Rekursionsformel $t(k) = 3 \cdot t(k/2) + 5 \cdot k$ ergibt:

$$t(k) = 3 \cdot t(k/2) + 5 \cdot k$$
$$= 3 \cdot (3 \cdot t(k/4) + 5 \cdot k/2) + 5 \cdot k$$
$$= 3 \cdot 3 \cdot t(k/4) + 3 \cdot 5 \cdot k/2 + 5 \cdot k$$
$$= 3 \cdot 3 \cdot t(k/4) + 5 \cdot k \cdot (1 + 3/2)$$
$$= 3 \cdot 3 \cdot (3 \cdot t(k/8) + 5 \cdot k/4) + 5 \cdot k \cdot (1 + 3/2)$$
$$= 3 \cdot 3 \cdot 3 \cdot t(k/8) + 5 \cdot k \cdot (1 + 3/2 + 9/4)$$
$$= 3 \cdot 3 \cdot 3 \cdot (3 \cdot t(k/16) + 5 \cdot k/8) + 5 \cdot k \cdot (1 + 3/2 + 9/4)$$
$$= 3 \cdot 3 \cdot 3 \cdot 3 \cdot t(k/16) + 5 \cdot k \cdot (1 + 3/2 + 9/4 + 27/8)$$
$$= \ldots$$

Die allgemeine Formel nach i Schritten lautet daher:

$$t(k) = 3^i \cdot t(k/2^i) + 5 \cdot k \cdot (1 + 3/2 + 9/4 + \ldots + 3^{i-1}/2^{i-1})$$
$$= 3^i \cdot t(k/2^i) + 10 \cdot k \cdot ((3/2)^i - 1)$$

Beachte die geometrische Reihe

$$1 + a + a^2 + a^3 + \ldots + a^m = \frac{a^{m+1} - 1}{a - 1}$$

In unserem Fall ist $a = {}^3/_2$.

Diese Ersetzungen kann man vornehmen, bis $k = 2^i$ geworden ist, also bis $i = \log(k)$. Dann ist $t(k/2^{\log(k)}) = t(1) = 1$. Wir setzen dies ein und erhalten:

$$\begin{aligned} t(k) &= 3^{\log(k)} \cdot t(k/2^{\log(k)}) + 10 \cdot k \cdot ((3/2)^{\log(k)} - 1) \\ &= k^{\log(3)} + 10 \cdot k \cdot (k^{\log(1,5)} - 1) \\ &= 11 \cdot k^{\log(3)} - 10 \cdot k \approx 11 \cdot k^{1,585} - 10 \cdot k \in O(k^{1,585}) \end{aligned}$$

Beachte die Formeln:
log ist hier der Logarithmus zur Basis 2,

$a^{\log(b)} = b^{\log(a)}$ und

$k \cdot k^{\log(1,5)} = k^{1+\log(1,5)} = k^{\log(2)+\log(1,5)} = k^{\log(2 \cdot 1,5)} = k^{\log(3)}$

Die Multiplikation zweier k-stelliger Zahlen lässt sich also zeichenweise in proportional zu $k^{1,585}$ Schritten durchführen.

(Die Schulmethode benötigt k^2 Schritte, siehe Anfang von 6.5.3.)

Satz: Die Multiplikation natürlicher Zahlen liegt in $O(n^{1,585})$.

Geht es noch schneller? Ja. Der beste derzeit bekannte Algorithmus, der Algorithmus nach Schönhage und Strassen (1971), der sich allerdings nur für riesige Zahlen eignet, benötigt
$O(k \cdot \log(k) \cdot \log(\log(k)))$ Schritte.

Dies ist schon relativ dicht an der Größenordnung O(k), so dass man vielleicht eines Tages doch einen Algorithmus finden wird, der die Multiplikation in O(k) Schritten - und somit bis auf eine multiplikative Konstante genau so schnell wie eine Addition - durchführt??

Anmerkung: Ab wann lohnt sich dieses rekursive Verfahren? Hierfür müssen wir wissen, wie aufwendig eine normale Multiplikation genau ist, also einschließlich der Konstanten. Wenn $k = \log(b) = \log(a) = n/2$ ist, so benötigt der Algorithmus in 6.5.3 höchstens $k \cdot n + 3 \cdot k + 4 \cdot n + 5 = 2 \cdot k^2 + 11 \cdot k + 5$ Schritte. (Vgl. dort: Alles geht in einem Schritt, nur bei "Z:=Z+A" können bis zu n-stellige Zahlen auftreten.)

Die rekursive Methode lohnt sich daher, wenn für k gilt: $11 \cdot k^{1,585} - 10 \cdot k < 2 \cdot k^2 + 11 \cdot k + 5$. Dies trifft schon für recht kleine Werte von k zu, also etwa ab k = 18, spätestens aber für Zahlen in der Größenordnung einer Million. Die Methode kann sich also lohnen, allerdings haben wir die Verwaltung der Zwischenergebnisse bisher nicht berücksichtigt. Eine ganz exakte Analyse müsste *alle* auftretenden Operationen im übersetzten Programm (einschl. der Rekursions-Verwaltung) einbeziehen. ■

Beispiel 6.5.5 Transitive Hülle eines (gerichteten) Graphen

Die transitive Hülle eines gerichteten oder ungerichteten Graphen G = (V, E) entsteht, wenn man für alle Kanten (x, y) und (y, z) die Kante (x, z) (bzw. für alle {x, y} und {y, z} die Kante {x, z}) hinzunimmt und dieses Vorgehen iteriert, bis sich nichts mehr ändert.

Die transitive Hülle gibt für je zwei Knoten u und v direkt an, ob es einen Weg von u nach v gibt oder nicht. Gegeben sei nun ein gerichteter Graph G = (V, E) mit $V = \{x_1, ..., x_n\}$.

Üblicherweise berechnet man die transitive Hülle über die Adjazenzmatrix. Im gerichteten Fall ist die Adjazenzmatrix $A = (a_{i,j})$ definiert durch $a_{i,j} = 1$ für $(x_i, x_j) \in E$ und $a_{i,j} = 0$ sonst (für alle $i, j = 1, ..., n$), siehe 3.8.5 g.

Gesucht ist die Adjazenzmatrix D der transitiven Hülle:

$D = (d_{i,j})$ ist definiert durch (i, j = 1, 2, ..., n)

$d_{i,j} = 1 \Leftrightarrow$ es gibt einen gerichteten Weg von x_i nach x_j

$d_{i,j} = 0 \Leftrightarrow$ sonst.

Lösungsidee:

Prüfe für alle i, j = 1, ..., n, ob es ein k gibt, so dass ein Weg von Weg von x_i nach x_j über x_k existiert.

Beginne damit, dass für jeden Knoten x stets ein Weg der Länge 0 von x nach x angenommen wird und dass die Wege ohne Zwischenknoten x_k, also die Wege der Länge 1, durch die Kantenmenge E und somit durch die Adjazenzmatrix A gegeben sind.

Diese Idee liefert den Warshall-Algorithmus zur Berechnung der transitiven Hülle:

```
type adj is array (1..n, 1..n) of 0..1;
A, D: adj; ...
begin   ... ;  D := A;
        for i in 1..n loop  D(i,i) := 1;  end loop;
        for k in 1..n loop
           for i in 1..n loop
              for j in 1..n loop
                 if D(i,k)=1 and D(k,j)=1 then D(i,j) := 1; end if;
        end loop;  end loop;  end loop;
        ...
end;
```

Hinweise: k muss in der äußersten Schleife stehen!
Man kann "D(i,k)=1" eine Schleife nach außen ziehen.

Wie beweist man, dass dieses Verfahren tatsächlich die transitive Hülle berechnet?

Durch Induktion. Die Induktionsannahme lautet: Besitzt k am Ende der beiden inneren Schleifen den Wert m, so gilt
D(i,j) = 1 $\Leftrightarrow$ Es gibt $r \geq 0$ und einen Weg $(x_i, u_1, u_2, ..., u_r, x_j)$, so dass für alle $s = 1, .., r$ gilt: $u_s \in \{x_1, x_2, ..., x_m\}$.
Man sieht dann leicht ein, dass diese Aussage nach Durchlauf der beiden inneren Schleifen auch für m+1 gilt. Die transitive Hülle erfüllt genau diese Aussage für k = n, d. h., am Ende ist D die Adjazenzmatrix für die transitive Hülle.

Uniforme Zeitkomplexität: $O(n^3)$, wegen drei ineinander geschachtelter Schleifen und konstantem Aufwand in der innersten Schleife .
Zusätzlicher Platzbedarf: Für die Matrix D: $O(n^2)$ Plätze.

TM-Zeitkomplexität für diesen Algorithmus (6 Bänder): $O(n^4)$, weil man ständig für jedes D(k,j) n Felder weitergehen muss.

Beispiel 6.5.6: Das Handlungsreisendenproblem (TSP)

6.5.6.1 Definition "Permutation"
(vgl. die Aufgaben 12-14 in Abschnitt 1.15)
Es sei n eine natürliche Zahl.
Eine bijektive Abbildung $\pi: \{1, 2, ..., n\} \rightarrow \{1, 2, ..., n\}$ heißt Permutation der Ordnung n.

Erinnerung an drei mögliche Eigenschaften von Abbildungen f:
bijektiv = injektiv und surjektiv, wobei
injektiv bedeutet: für je zwei Elemente $a \neq b$ gilt $f(a) \neq f(b)$,
surjektiv bedeutet: zu jedem b existiert ein a mit $f(a) = b$.

6.5.6.2 Inverse Abbildung

Ist die Abbildung $f: A \to B$ bijektiv, dann ist $g: B \to A$ mit $g(b) = a$, sofern $f(a) = b$ ist, ebenfalls eine bijektive Abbildung, genannt die "zu f inverse Abbildung". Man schreibt f^{-1} anstelle von g.

Es gilt dann für alle $a \in A$: $f^{-1}(f(a)) = a$
und für alle $b \in B$: $f(f^{-1}(b)) = b$.
Insbesondere ist f die zu f^{-1} inverse Abbildung, d.h., es gilt $(f^{-1})^{-1} = f$.

Also besitzt jede Permutation π eine inverse Permutation

$$\pi^{-1}: \{1, 2, ..., n\} \to \{1, 2, ..., n\}$$

mit $\pi^{-1}(\pi(k)) = k$ für alle $1 \le k \le n$.

6.5.6.3 Einige Eigenschaften

(1) Es gibt genau n! verschiedene Permutationen der Ordnung n.

Beweis: *(Dies wurde bereits in 1.3 erläutert, siehe dort).* Es sei g(n) die Anzahl der verschiedenen Permutationen der Ordnung n. Für $n = 1$ ist die Aussage $g(1) = 1 = 1!$ richtig. Sei die Aussage bis zu einer natürlichen Zahl n bewiesen, dann betrachte die Ordnung (n+1). Man erhält alle Permutationen der Ordnung n+1, indem man die Zahl (n+1) an genau eine Position der Permutationen π der Ordnung n einfügt. Es gibt genau n+1 solcher Positionen in π, nämlich am Anfang oder nach der 1. oder nach 2. ... oder nach der n-ten Zahl in π, d.h., es gibt genau (n+1) Mal so viele Permutationen der Ordnung (n+1), als es Permutationen der Ordnung n gibt. Folglich gilt $g(n+1) = (n+1) \cdot g(n)$. Dies ist jedoch genau die rekursive Definition der Fakultäts-Funktion, d.h. $g(n) = n!$.

(2) Die Hintereinanderausführung $(\pi \circ \sigma)$ zweier Permutationen π und σ ist definiert durch $(\pi \circ \sigma)(k) = \pi(\sigma(k))$ für alle $1 \le k \le n$. Es gilt: Mit π und σ ist auch $(\pi \circ \sigma)$ eine Permutation der Ordnung n.

Beweis: Man muss für $(\pi \circ \sigma)$ die Eigenschaften "injektiv" und "surjektiv" beweisen.
Seien $i, j \in \{1,2,...,n\}$ mit $i \ne j$. Dann gilt $\sigma(i) \ne \sigma(j)$, weil σ eine Permutation ist, und $(\pi \circ \sigma)(i) = \pi(\sigma(i)) \ne \pi(\sigma(j)) = (\pi \circ \sigma)(j)$, weil auch π eine Permutation ist. Also ist $(\pi \circ \sigma)$ injektiv.
Sei $k \in \{1,2,...,n\}$ beliebig gewählt, dann existiert ein $j \in \{1,2,...,n\}$ mit $\pi(j) = k$, weil π eine Permutation ist, und es gibt ein $i \in \{1,2,...,n\}$ mit $\sigma(i) = j$, weil σ eine Permutation ist. Für i gilt deshalb: $(\pi \circ \sigma)(i) = \pi(\sigma(i)) = \pi(j) = k$. Zu jedem k gibt es also ein i mit $(\pi \circ \sigma)(i) = k$, d. h., $(\pi \circ \sigma)$ ist surjektiv.

Hinweis: Die zu $(\pi \circ \sigma)$ inverse Permutation $(\pi \circ \sigma)^{-1}$ ist $(\sigma^{-1} \circ \pi^{-1})$ mit $(\pi \circ \sigma)^{-1}(k) = (\sigma^{-1} \circ \pi^{-1})(k) = \sigma^{-1}(\pi^{-1}(k))$ für alle $1 \le k \le n$.

(3) Es sei $\delta_{i,j}$ die Permutation, die genau die Zahlen i und j vertauscht und alle anderen Zahlen unverändert lässt, d. h.:
$\delta_{i,j}(k)$ = if i=k then j elsif j=k then i else k fi.
Beispiel für n=5: $\delta_{2,4}(1)=1$, $\delta_{2,4}(2)=4$, $\delta_{2,4}(3)=3$, $\delta_{2,4}(4)=2$, $\delta_{2,4}(5)=5$.
Permutationen der Form $\delta_{i,j}$ bezeichnet man als Transposition oder Zweiertausch. Es gilt: Jede Permutation lässt sich als Hinterausführung von höchstens n-1 Transpositionen darstellen.

Beweis: Anschaulich ist diese Aussage klar:
Wenn in der Permutation π die Zahl k an der Position n steht, so bilde hieraus die Permutation π', die aus π durch Vertauschen der Zahlen k und n entsteht. Dann gilt: $\pi = \delta_{k,n} \circ \pi'$. Wende dann die gleiche Überlegung auf π' und n-1 an usw. Für $n = 1$ gibt es nichts zu vertauschen, so dass man nach n-1 Schritten fertig ist. Treten hierbei Transpositionen $\delta_{i,i}$ auf, so kann man diese streichen, da sie nichts verändern. Folglich reichen höchstens n-1 Transpositionen aus.

Beispiel für n = 5:

Die Permutation (3,4,5,1,2) ist gleich

(3,4,5,1,2) vertausche n=5 mit k=2.
$= \delta_{2,5} \circ (3,4,2,1,5)$ vertausche n-1=4 mit k=1.
$= \delta_{2,5} \circ \delta_{1,4} \circ (3,1,2,4,5)$ vertausche n-2=3 mit k=2.
$= \delta_{2,5} \circ \delta_{1,4} \circ \delta_{2,3} \circ (2,1,3,4,5)$ vertausche n-3=2 mit k=1.
$= \delta_{2,5} \circ \delta_{1,4} \circ \delta_{2,3} \circ \delta_{1,2} \circ (1,2,3,4,5)$.

Weil (1,2,3,4,5) die Identität ist, die nichts verändert, gilt also:
$(3,4,5,1,2) = \delta_{2,5} \circ \delta_{1,4} \circ \delta_{2,3} \circ \delta_{1,2}$

Nachprüfen (beachte die Auswertung von rechts nach links!):
$\delta_{1,2} \circ (1,2,3,4,5) = (2,1,3,4,5)$,
$\delta_{2,3} \circ (2,1,3,4,5) = (3,1,2,4,5)$,
$\delta_{1,4} \circ (3,1,2,4,5) = (3,4,2,1,5)$,
$\delta_{2,5} \circ (3,4,2,1,5) = (3,4,5,1,2)$.

(4) Für alle i und j gilt: $\delta_{i,j} = \delta_{j,i} = \delta_{i,j}^{-1} = \delta_{j,i}^{-1}$.
(Klar.)

(5) Die Permutation (1, 2, 3, ..., n) bezeichnet man als die Identität oder als die "identische Permutation", abgekürzt id.
Offensichtlich gilt für alle Permutationen π:

$(\pi \circ id) = (id \circ \pi) = \pi$ und
$(\pi \circ \pi^{-1}) = (\pi^{-1} \circ \pi) = id$.

Es sei S_n die Menge aller Permutationen der Ordnung n.

Hinweis: $(S_n, \circ, id, ^{-1})$ bildet im mathematischen Sinne eine (nicht-kommutative) Gruppe, siehe Grundvorlesungen zur Mathematik.

(6) Stirlingsche Formel (James Stirling, 1692-1770, schottischer Mathematiker), Näherungsformel für n! Für jede natürliche Zahl $n \in \mathbb{N}$ gilt: $\left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n} \le n! \le \left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n} \cdot e^{\frac{1}{12n}}$

Diese Abschätzung wird meist in folgender Form verwendet:

$$n! \approx \left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n}$$

In diesen Formeln sind $\pi = 3{,}1415926...$ die Kreiszahl und e die Eulersche Konstante $e = 2{,}71828182845945235...$

Zum Beweis siehe Lehrbücher und Vorlesungen der Mathematik.

6.5.6.4 TSP = das Problem des Handlungsreisenden

Gegeben ist ein vollständiger gerichteter und gewichteter Graph G mit n Knoten (genannt "Städte"), also: $G = (\{1,2,...,n\}, E, \gamma)$ mit $E = \{(i,j) \mid$ für alle $1 \le i \le n$ und $1 \le j \le n$ mit $i \ne j\}$ und einer Abbildung $\gamma: E \to \mathbb{R}^+$ (= nicht-negative reelle Zahlen).

γ wird auf Wege fortgesetzt, indem die γ-Werte der Kanten des Weges aufsummiert werden, d.h., ein Weg $(x_0,x_1,...,x_r)$, der aus den r Kanten $(x_0,x_1), (x_1,x_2), ..., (x_{r-1},x_r)$ besteht, erhält die Weg-länge $\gamma((x_0,x_1,...,x_r)) = \gamma((x_0,x_1)) + \gamma((x_1,x_2)) + ... + \gamma((x_{r-1},x_r))$.

Das TSP (= travelling salesman problem) lautet dann: Finde zu G eine minimale Rundreise, das heißt, einen Kreis, der jede Stadt genau einmal (und die Startstadt genau zweimal) enthält und dessen Weglänge $\gamma(...)$ minimal bzgl. aller solcher Kreise ist. Der Handlungsreisende soll also jede Stadt $1, 2, ..., n$ genau einmal besuchen und am Ende zu seiner Startstadt zurückkehren und hierbei einen kürzestmögliche Weg durchlaufen.

Dieses Problem tritt in sehr vielen Anwendungen auf. Es wurde daher recht genau bezüglich seiner Laufzeit untersucht. Jeder zu untersuchende Kreis entspricht genau einer Permutation der Städte 1, 2, ..., n, wobei man ohne Beschränkung der Allgemeinheit annehmen darf, dass "1" die Startstadt der Rundreise ist. Dann gibt es also genau (n-1)! Rundreisen, unter denen eine mit der kürzesten Weglänge gefunden werden soll.

Somit kann man das TSP-Problem lösen, indem für jede der (n-1)! Permutationen die Weglänge berechnet und die bis dahin kürzeste Weglänge mit einer zugehörigen Permutation notiert wird:

```
P := erste Permutation (mit Startstadt 1);
while noch nicht alle Permutationen geprüft wurden loop
   berechne die Weglänge W dieser Permutation P;
   if W < bisheriges Minimum then notiere W und P end if;
   P := nächste noch nicht betrachtete Permutation;
end loop;
gib notierte minimale Weglänge und zugehörige Permutation aus;
```


In "gamma" speichern wir die γ-Werte und in "P" die Permutation, die die Rundreise festlegt.

```
gamma: array (1..n,1..n) of Float;  P: array (1..n) of Natural;
```

Die Weglänge W zu einer Permutation P ist leicht zu berechnen, indem man alle gamma(P(i-1),P(i)) für i = 2, 3, ..., n aufsummiert und dann noch gamma(P(n),P(1)) hinzuzählt. Hierfür sind genau n Additionen je Permutation erforderlich.

Somit bedeutet nur die Berechnung der nächsten Permutation eine Schwierigkeit. Dieser Schritt wurde in Aufgabe 13, Abschnitt 1.15 beschrieben. Mit der kleinen Veränderung, dass wir hier i-1 (statt i) und j-1 (statt j) verwenden, ergibt sich die Prozedur, die in dem folgenden Programm enthalten ist. Die Zahl der Vergleiche, die in der Prozedur maximal durchgeführt werden, lässt sich durch 2,5·n nach oben abschätzen, so dass wir also auch hier höchstens O(n) Operationen je Permutation benötigen.

Da (n-1)! Permutationen ausprobiert werden, ergibt sich eine Gesamtlaufzeit von (n-1)!·(n) = O(n!).

Es folgt nun ein zugehöriges Programm zur Lösung des TSP.

6.5.6.5 Ein Programm zum TSP (hier in Ada formuliert)

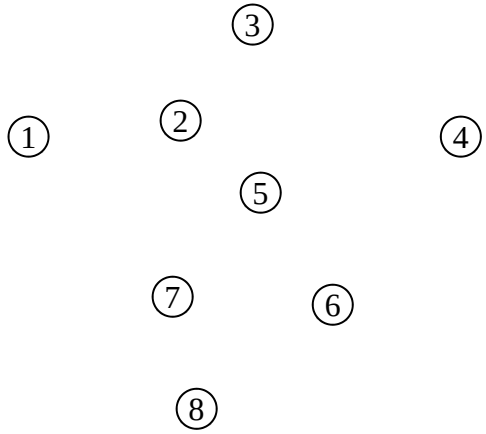
```
procedure TSP1 is                    -- es muss in diesem Programm n > 1 sein
  n: constant Positive := 8;         -- n wird hier willkürlich gesetzt
  type Permutation is array (1..n+1) of Natural range 1..n;
  type Nichtnegativreell is new Float range 0.0 .. Float'Last;

  Weiter: Boolean;                   -- wenn Weiter false wird, dann: Ende
  gamma: array (1..n,1..n) of Nichtnegativreell;   -- Eingabe: γ als Matrix
  W, Min_W: Nichtnegativreell;       -- Weglänge, bisher minimale Weglänge
  P, Min_P: Permutation;             -- aktuelle Permutation, bisher beste Permut.

  procedure Init is                  -- Initialisierung
  begin  Weiter := True; Min_W := 0.0;     -- wähle anfangs Min_W zu groß
     for i in 1..n loop
        for j in 1..n loop
            Get(gamma(i,j));  Min_W := Min_W + gamma(i,j);
        end loop;
     end loop;
     for i in 1..n loop  P(i) := i;  Min_P(i) := i;  end loop;
     P(n+1) := 1;          -- wird nur als Stopper in Naechste_Permutation benutzt
  end;
```



```
procedure Naechste_Permutation is
   i, j, H: Natural;
begin   i := n;
     -- finde zunächst das größte i mit P(i-1) < P(i)
   while P(i) < P(i-1) loop i := i-1; end loop;
   if  i > 2 then          -- sobald i = 2 ist, kann man abbrechen
      H := P(i-1);  j := i;      -- H ist eine Hilfsvariable zum Zwischenspeichern
        -- suche das kleinste j in i..n mit P(j) < P(i-1); dann ist Z = P(j-1) die kleinste
        -- Zahl in {P(n), P(n-1),...,P(i)} mit Z > P(i-1) [weil P absteigend geordnet]
      while P(j) > H loop j := j+1; end loop;
        -- hier kann j = n+1 werden, daher wurde P(n+1) = 1 als Stopper gesetzt
        -- vertausche Z=P(j-1) mit P(i-1) und sortiere danach P von i bis n
      P(i-1) := P(j-1);  P(j-1) := H;
      j := n;  -- da P von i bis n absteigend geordnet ist, muss man nur "umdrehen"
      while j > i loop
         H := P(i);  P(i) := P(j);  P(j) := H;  i:=i+1;  j:=j-1;
      end loop;
   else Weiter := False; end if;     -- sobald P(1) auf 2 gesetzt wird: Ende
end;
```

```
   function Weglaenge return Nichtnegativreell is
      Sum: Nichtnegativreell := gamma(P(n),P(1));
   begin
      for i in 2..n loop Sum := Sum + gamma(P(i-1),P(i)); end loop;
      return Sum;
   end;

begin       -- die Werte für gamma(i,i) sind egal, weil sie nie benutzt werden, sie
   Init;    -- müssen aber trotzdem eingegeben werden
   while Weiter loop
      W := Weglaenge;
      if W < Min_W then
         Min_W := W;  Min_P := P;
      end if;
      Naechste_Permutation;
   end loop;                -- Nun fehlt nur noch die Ausgabe
   Put("Länge der kürzesten Rundreise: "); Put(Min_W); New_Line;
   Put("Eine zugehörige Reihenfolge der Städte:"); New_Line;
   for i in 1..n loop Put(Min_P(i),4); end loop;  Put(Min_P(1),4);
end;
```


Beispiel: Ungefähre Lage von n = 8 Städten, wenn man sie auf die Ebene projizieren würde. Dies ergibt ungefähr die Entfernungsmatrix gamma auf der folgenden Folie.



Die folgende Entfernungsmatrix gamma ist "fast" symmetrisch, nur an den drei fett gedruckten Stellen ist sie asymmetrisch.

gamma =

0	3	5	9	5	7	4	9
3	0	3	5	2	6	5	10
4	3	0	6	6	8	9	12
9	5	6	0	4	4	9	10
5	2	6	4	0	3	3	6
7	6	8	**2**	3	0	4	4
4	5	9	9	3	4	0	3
9	10	12	10	6	**6**	3	0

Frage: Wie lautet eine kürzeste Rundreise und wie lang ist sie?

Obiges Programm muss leicht angepasst werden, weil die Ein-/Ausgabe von "Nichtnegativreell" nicht definiert ist. Wir ersetzen diesen Typ einfach durch "Natural", ersetzen "0.0" in Init durch "0" und nehmen die Zahlen der Matrix gamma von der vorigen Folie. Dann erhalten wir die Ausgabe:

```
Länge der kürzesten Rundreise:       28
Eine zugehörige Reihenfolge der Städte:
  1   7   8   6   4   5   2   3   1
```

Konstruieren Sie Ihre eigenen Entfernungsmatrizen und experimentieren Sie mit dem Programm. Bis n = 13 erhält man noch Antwort in akzeptabler Zeit. Messungen ergaben auf einem 1 GHz-Rechner Laufzeiten von knapp 7 sec für n=11, 79 sec für n=12 und 997 sec für n=13. Dies ist recht genau das vorhergesagte O(n!)-Verhalten.

6.5.6.6 Welche Laufzeit besitzt das Programm TSP1 genau?

Das obige Programm TSP1 ist für die Praxis nicht geeignet, weil es ab n = 18 auch mit sehr leistungsstarken Rechnern keine Ausgabe in überschaubarer Zeit liefert. Es bieten sich unmittelbar einige Lösungswege an:

- Parallelisierung. Man kann die Menge S_{n-1} aller Permutationen der Ordnung (n-1) irgendwie in m Teilmengen unterteilen, jede Teilmenge auf einem anderen Rechner bearbeiten lassen und am Ende aus den m Ergebnissen die kürzeste Lösung ermitteln. Mit m = 1_000_000 Rechnern kann man dann TSP1 bis etwa n = 23 verwenden.
- Suche nach schnelleren Verfahren.
- Verzicht auf die Garantie, die kürzeste Rundreise zu finden, und stattdessen mit Heuristiken/Näherungsverfahren eine möglichst kurze Rundreise berechnen.

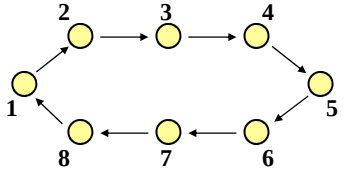
Wir diskutieren kurz die "Suche nach schnelleren Verfahren". Zunächst mache man sich klar, dass der Anteil, den die Prozedur "Naechste_Permutation" zur Laufzeit von TSP1 beiträgt, *nur proportional zu (n-1)! und nicht zu n! ist*. Grund:

Jedes zweite Mal werden nur die letzten beiden Zahlen vertauscht, weil jedes zweite Mal $P(n-1) < P(n)$ gilt;
fünf von sechs Malen betrachtet die Prozedur nur die letzten 3 Zahlen, weil nur jedes sechste Mal $P(n-2) > P(n-1) > P(n)$ ist;
23 von 24 Malen werden nur die letzten 4 Zahlen behandelt, weil nur jedes 24. Mal $P(n-3) > P(n-2) > P(n-1) > P(n)$ gilt;
usw. Im Durchschnitt erfolgen also pro Permutation höchstens

$$2{,}5 \cdot (2 \cdot 1/2 + 3 \cdot 1/6 + 4 \cdot 1/24 + \ldots + k \cdot 1/k! + \ldots + (n-1) \cdot 1/(n-1)!)$$
$$= 2{,}5 \cdot (1 + 1/2! + 1/3! + \ldots + 1/(n-2)!) < 2{,}5 \cdot (e-1) \approx 4{,}3$$

Vergleiche und entsprechend konstant viele Umspeicherungen. Das Erzeugen aller (n-1)! Permutationen geschieht also in O((n-1)!). Folglich wird der Zeitaufwand von TSP1 von der Funktion "Weglänge" bestimmt, die in der Tat jeweils n Addition, insgesamt also O(n!) Schritte zur Laufzeit beiträgt.

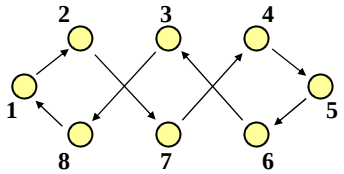
Um die Laufzeit zu verringern, muss man die jeweils nächste Weglänge nicht in n, sondern in konstant vielen Schritten berechnen. Dies ist möglich, falls die nächste Permutation stets mit einer Transposition aus der vorigen Permutation berechnen wird. Dann reichen 8 Additionen/Subtraktionen, um die neue Länge W_{neu} der Rundreise aus der vorherigen Länge W_{alt} zu berechnen.



Permutation (1,2,3,4,5,6,7,8)

vertausche 3 und 7

Permutation (1,2,7,4,5,6,3,8)



$$W_{neu} = W_{alt} + \gamma((2,7)) + \gamma((7,4)) + \gamma((6,3)) + \gamma((3,8)) - \gamma((2,3)) - \gamma((3,4)) - \gamma((6,7)) - \gamma((7,8)).$$

Tatsächlich kann man alle Permutationen so nacheinander durchlaufen, dass die nächste Permutation stets durch einen "Zweiertausch" (also durch eine Transposition) entsteht; man kann dies sogar so organisieren, dass die Transposition immer nur benachbarte Positionen vertauscht, was weitere 2 Additionen/ Subtraktionen einspart (im symmetrischen Fall sogar 4).

Dies ist gar nicht schwer. Probieren Sie es einmal aus und geben eine Vorschrift an, wie man auf diese Weise alle Permutationen "doppelpunktfrei" durchläuft.

Hierdurch kann man die TSP1-Laufzeit also auf O((n-1)!) senken.

Es gibt andere Verfahren, die das TSP in $O(n^2 \cdot 2^n)$ lösen mit dem sog. "dynamischen Programmieren", wobei sehr viel Speicherplatz benötigt wird. Man kennt heute jedoch kein Lösungsverfahren für das TSP, dessen Laufzeit nicht exponentiell mit n wächst (Stichwort: P=NP-Problem).

6.6 Registermaschinen und andere Rechenmodelle

Bei den Rechenmodellen hängt es davon ab, welche "Mächtigkeit" das Modell besitzen soll. Gängige Modelle, die beliebige Algorithmen ausführen können (die so mächtig wie Turingmaschinen sind; man sagt, "die *turingmächtig* sind"), sind die Registermaschine, der Lambda-Kalkül, prädikatenlogische Kalküle, Markov-Algorithmen usw.

Deutlich eingeschränkt in ihren Fähigkeiten sind die "linear beschränkten Automaten" (= Turingmaschinen mit O(n) Speicherzellen auf dem Band), die Keller- oder Pushdownautomaten (neben Ein- und Ausgabeband besitzen sie einen Keller als Speicherband) und endliche Automaten oder endliche Akzeptoren ("EA" oder engl. "FA") ohne zusätzlichen Speicher, jeweils deterministisch und nichtdeterministisch.

Wir stellen alle Modelle kurz vor, hierbei etwas genauer die EAs.

6.6.1 Veranschaulichung von Registermaschinen

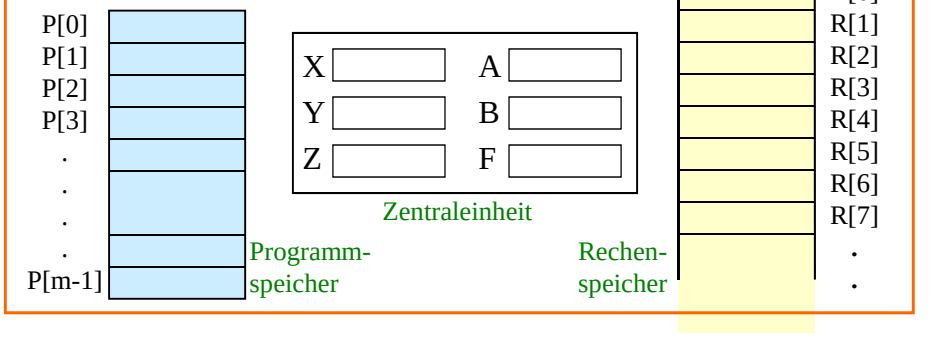
Eine Registermaschine ist wie ein kleiner Mikroprozessor aufgebaut. Sie besteht aus

1. einer *Zentraleinheit* mit 6 Registern (Speicherzellen): 3 Register X, Y und Z für arithmetische und logische Operationen, ein Adressregister A für den Zugriff auf Rechenspeicherzellen, ein Befehlsregister B, in dem die Adresse des auszuführenden Befehls steht, und ein "Flag"-Register F, in dem das Ergebnis von Operationen abgelegt wird;
2. einem *Programmspeicher* P, in dem nacheinander die Befehle des abzuarbeitenden endlichen Programms stehen;
3. einem (unendlich langen) *Rechenspeicher* R mit durchnummerierten Speicherzellen, die jeweils eine (beliebig große) ganze Zahl aufnehmen können.

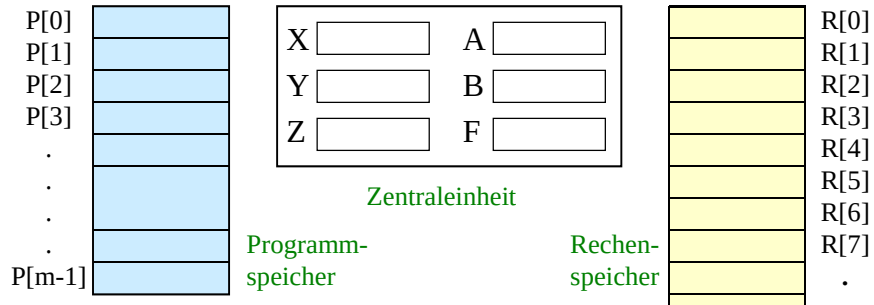
Der übliche Befehlssatz einer Registermaschine (V und V' seien Register, c eine ganze Zahl, $b \in \mathbb{N}_0$, R_k die k-te Speicherzelle, $\sigma \in \{>, \geq, <, \leq, =, \neq\}$ eine Vergleichsoperation; außer bei jump wird nach jedem Befehl B um 1 erhöht):

Befehl	Bedeutung	Befehl	Bedeutung
load V,c	V := c	copy V,V'	V := V'
read V	V := $R_{<A>}$	write V	$R_{<A>}$:= V
succ	X := X+1	add	X := Y+Z
sub	X := Y-Z	shift	X := X div 2
comp (σ)	if XσY then F:=1 else F:=0 fi		
jump b	if F=1 then B:=b else B:=B+1 fi		
stop	Anhalten, Ende der Berechnung		

Überblick: Einfachste Registermaschine



Erläuterungen der Überblickszeichnung



Typische Befehle sind:

add	Addiere die Inhalte von X und Y und lege das Ergebnis in das Register Z
jump 37	Falls F=1 ist, mache weiter mit dem Befehl mit der Nummer 37, sonst mit dem nächsten
LeftShift X	Schiebe den Inhalt von X eine Stelle nach links und füge eine 0 an
comp (Y > Z)	if der Inhalt von Y ist größer als der von Z then F:=1 else F:=0 fi
read X	X := $R_{<A>}$, d.h., sei a der Inhalt von A, so weise zu X:=R_a
load A,4	A := 4 (Wertzuweisung einer Konstanten an ein Register)

Der übliche Befehlssatz einer Registermaschine lautet (hierbei bezeichnen V und V' beliebige Register, c ist eine ganze Zahl, $b \in \mathbb{N}_0$, R_k ist die k-te Speicherzelle, $\sigma \in \{>, \geq, <, \leq, =, \neq\}$ ist eine Vergleichsoperation; außer beim jump-Befehl wird nach jedem Befehl B um 1 erhöht):

Befehl	Bedeutung	Befehl	Bedeutung
load V,c	V := c	copy V,V'	V := V'
read V	V := $R_{<A>}$	write V	$R_{<A>}$:= V
succ	X := X+1	add	X := Y+Z
sub	X := Y-Z	shift	X := X div 2
comp (σ)	if XσY then F:=1 else F:=0 fi		
jump b	if F=1 then B:=b else B:=B+1 fi		
stop	Anhalten, Ende der Berechnung		

Diese Befehle finden sich bei allen Mikroprozessoren; diese haben in der Regel aber noch viel mehr Befehle, insbesondere bzgl. der Adressierung, der Verschiebungen von Daten, der eingebauten Kellermechanismen und der Unterbrechungsbefehle. Die Ausformulierung von Algorithmen als Registermaschine ähnelt daher der Maschinenprogrammierung.

Beispiel: Test auf Teilbarkeit durch 2 (die Zahl $n \geq 0$ stehe anfangs in R_0).
Naheliegende Lösung: Subtrahiere von n ständig die Zahl 2. Die Zahl n sei hier der Variablen I zugewiesen. Dann lautet das Programmstück: **while I>1 do I:=I-2 od;**
Anschließend steht in I der Rest der Division von n durch 2.
Übertrage dieses Programmstück in den Befehlssatz der Registermaschine ($I \Leftrightarrow$ Register Y, Zahl $2 \Leftrightarrow$ Z; anfangs steht n in der Rechenspeicherzelle R_0, am Ende stehe das Ergebnis in R_0; sei a der Inhalt von A, so bezeichnet $R_{<A>}$ den Inhalt der Rechenspeicherzelle R_a). Man erhält folgendes Registermaschinenprogramm:

Nummer	Befehl	Erläuterung
0:	load A,0	A := 0
1:	read Y	Y := n (= $R_0 = R_{<A>}$)
2:	load Z,2	Z := 2
3:	load X,1	X := 1
4:	comp (≥)	teste, ob X ≥ Y ist (beachte: in X steht eine 1)
5:	jump 10	if 1 ≥ Y then weiter bei Befehl mit Nummer 10 fi
6:	sub	X := Y-Z (also X:=Y-2)
7:	copy Y,X	Y := X
8:	load F,1	F:=1 (um einen Sprung nach 3 zu erzwingen)
9:	jump 3	weitermachen beim Befehl mit der Nummer 3
10:	write Y	in A steht noch 0, also: R_0 := Y
11:	stop	

Betrachtet man Registermaschinen, die nur auf ganzen Zahlen arbeiten, so muss man zuvor prüfen, ob n < 0 ist oder nicht. Wir fügen daher vor das oben angegebene Programmstück noch die Anweisung if I < 0 then I := 0-I fi und übertragen dieses Programm wiederum in ein Registermaschinenprogramm. So erhalten wir:

Man kann sich überzeugen, dass jedes Ada-Programm in ein solches Registermaschinenprogramm übersetzt werden kann und dass sich dieser Prozess automatisieren lässt (Compiler). Solche einfachen Sprachen heißen in der Praxis "Maschinensprachen" oder "Maschinencode"; sie werden meist in Form von Assemblern ein wenig lesbarer und komfortabler gemacht.

```
 0: load A,0
 1: read Y
 2: load X,0
 3: comp (≤)
 4: jump 9
 5: copy Z,Y
 6: load Y,0
 7: sub
 8: copy Y,X
 9: load Z,2
10: load X,1
11: comp (≥)
12: jump 17
13: sub
14: copy Y,X
15: load F,1
16: jump 10
17: write Y
18: stop
```


Die verschiedenen Typen von Registermaschinen unterscheiden sich in ihren (elementaren) Datentypen und in ihren Befehlssätzen. Meist fügt man noch ein Eingabeband, das nur gelesen werden darf, und ein Ausgabeband, das nur beschrieben werden darf, hinzu (vgl. Kellerautomat und endlicher Automat unten).

Wie bei Turingmaschinen kann man für dieses Rechenmodell, das über das Register A einen wahlfreien Zugriff auf die Rechenspeicherzellen erlaubt, Komplexitätsmaße und -klassen definieren. Hiermit lassen sich Aussagen beweisen, wie Sie sie in den übersprungenen Abschnitten 6.4.9 und 6.4.10 finden.

Die Komplexitätsmaße von Registermaschinen geben in der Regel die Verhältnisse von Programmen recht gut wieder. Sie werden daher vielen theoretischen Untersuchungen zugrunde gelegt.

Hinweis zur Übersetzung von Ada-Programmen

In den Programmen von Registermaschinen gibt es nur zwei Kontrollstrukturen:

- Übergang zum nächsten Befehl (dies entspricht dem ";" in Ada). Dies wird dadurch realisiert, dass nach jeder Ausführung eines Befehls B um 1 erhöht wird.
- Bedingter Sprung (falls F=1) zum Befehl mit der Nummer b (jump b), sofern die Nummer b im Programm vorkommt (anderenfalls Fehlerabbruch).

Man kann alle strukturierten Anweisungen in höheren Programmiersprachen durch diese beiden Kontrollstrukturen simulieren. In Ada sind Sprünge mit dem Schlüsselwort goto zugelassen. Anstelle der Nummern verwendet man *Marken*, dies sind Bezeichner, die in <<...>> eingeschlossen und vor eine Anweisung gesetzt werden.

if X ≥ Y then Z := 1; else X := Y; end if; X := X+1; ...

wird im Registerprogramm zu (willkürlich wurde 20 als Nummer des ersten Befehls gewählt)

20: comp (<)	vergleiche X mit Y bzgl. "kleiner"
21: jump 25	springe zum else-Zweig
22: load Z,1	Z := 1 (then-Zweig ausführen)
23: load F,1	bereite einen "unbedingten" Sprung vor
24: jump 26	überspringe den else-Zweig
25: copy X,Y	X := Y (else-Zweig ausführen)
26: succ	X := X+1

In Ada kann man dieses Programmstück direkt nachbilden:

```
F := X < Y; if F then goto ELSE_ZWEIG; end if;
Z := 1; goto DANACH;
<<ELSE_ZWEIG>>  X := Y;
<<DANACH>>  X := X+1; ...
```


while X < Y loop X := X+1; end loop; Z:=X;...

wird im Registerprogramm zu

30: comp (≥)	vergleiche X mit Y bzgl. nicht-"kleiner"
31: jump 25	überspringe die Schleife
32: succ	X := X+1 (Schleifenrumpf)
33: load F,1	bereite einen "unbedingten" Sprung vor
34: jump 30	zurück zur Schleifen-Bedingung
35: copy Z, X	Z := X ...

In Ada lautet dieses Programmstück:

```
<<Schleife>> F := X ≥ Y; if F then goto danach; end if;
X := X+1; goto Schleife;
<<danach>>  Z := X; ...
```


Ähnliche Übersetzungen von Ada-Konstrukten in einfache Programme, die nur aus Wertzuweisungen, Sprüngen und Hintereinanderausführungen (einschließlich Marken) bestehen, lassen sich leicht angeben.

Zumindest für die Kontrollstrukturen (=Anweisungsteil) sollte daher klar sein, dass man sie in eine Registermaschine übertragen kann. Bei der Übertragung der Datenstrukturen muss man sich den erforderlichen Speicherbedarf aller Teilstrukturen merken und beim Zugriff auf Variablen oder deren Komponenten entsprechende Werte in das Adressregister laden.

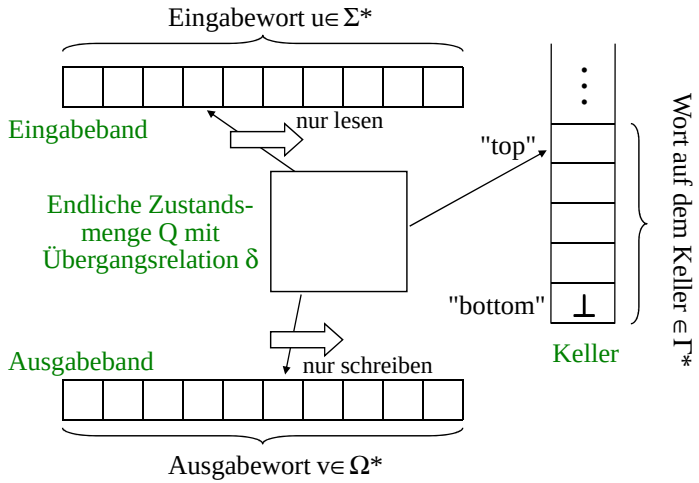
Somit ist im Prinzip klar, dass man jedes Ada-Programm in eine Registermaschine umwandeln kann, die die gleiche Resultatsfunktion berechnet. In der Praxis ist dieser Prozess aber sehr kompliziert (siehe Compilerbau-Vorlesungen).

6.6.2: Linear beschränkte Automaten sind Turingmaschinen, deren k Bänder nur so lang sind wie die Länge des Eingabewortes. Sie beschreiben also genau die Komplexitätsklasse

DSpace^{TM}(n), sofern die Maschine deterministisch ist, bzw.
NSpace^{TM}(n) im nichtdeterministischen Fall.

Fast alle Probleme, die in der Praxis auftreten, gehören zur Klasse NSpace^{TM}(n). Diese Klasse kann genau die Probleme lösen, die sich mit kontextsensitiven Grammatiken erzeugen lassen (vgl. 2.7.17). Die Laufzeit dieser Maschinen ist in der Regel exponentiell mit der Länge der Eingabe, so dass man in der Praxis nur linear beschränkte Automaten verwendet, deren Zeitkomplexität durch ein Polynom (geringen Grades) beschränkt ist.

6.6.3: Die Keller- oder Pushdownautomaten besitzen ein Eingabe- und ein Ausgabeband sowie als Arbeitsspeicher ein Keller-Band.



6.6.3.a Definition (nichtdeterministischer Kellerautomat)

$A=(Q,\Sigma,\Omega,\Gamma,\delta,Q_0,F,\perp)$ heißt Keller- oder Pushdownautomat $\Leftrightarrow$

1. Q ist eine endliche nicht-leere Menge (Zustandsmenge),
2. Σ ist eine endliche nicht-leere Menge (Eingabealphabet),
3. Ω ist eine endliche nicht-leere Menge (Ausgabealphabet),
4. Γ ist eine endliche nicht-leere Menge (Kelleralphabet),
5. $Q_0 \in Q$ ist die Menge der Anfangszustände (meist einelementig),
6. $F \subseteq Q$ ist die Menge der Endzustände,
7. $\perp \in \Gamma$ ist das Bottomsymbol (zeigt an, ob der Keller leer ist)
8. $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^* \times \Omega^*$, endliche Menge, ist die Überführungsrelation.

Hinweis zur formalen Definition: A heißt deterministisch $\Leftrightarrow$
(1) Wenn es ein $(q,\varepsilon,\alpha,q',z,w) \in \delta$ gibt, so gibt es kein anderes $(q,a,\alpha,q'',z'',w'') \in \delta$ (also mit gleichem q und α) für alle $a \in \Sigma \cup \{\varepsilon\}$.
(2) Liegt Fall (1) nicht vor, so gibt es zu jedem Tripel $(q,a,\alpha) \in Q \times \Sigma \times \Gamma$ höchstens ein Tripel $(q',z,w) \in Q \times \Gamma^* \times \Omega^*$ mit $(q,a,\alpha,q',z,w) \in \delta$.

6.6.3.b: Hinweis:
Fortgeschrittenere Beschreibungen erfordern kompliziertere Formalisierungen! Aber ohne die Fähigkeit zum Formalisieren bleibt das Programmieren vordergründig und "unsicher".

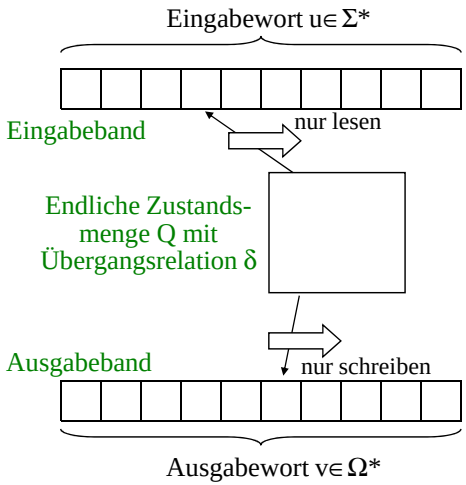
Die Bedeutung für die Praxis wird hierbei aber größer, d.h., Informatiker(innen) müssen in kompliziertere Sachverhalte vordringen, wenn sie Programmiersprachen und deren Übersetzer verstehen wollen.

Zwischen den kontextfreien Grammatiken (BNF), die Programmiersprachen erzeugen, und den Automaten, mit denen die korrekten Programme erkannt (und anschließend übersetzt) werden sollen, kann man nun einen engen Zusammenhang herstellen:

6.6.3.c: Folgende Aussagen lassen sich beweisen (siehe Vorlesung Theoretische Informatik I oder Übersetzerbau):

1. Alle Probleme, die Kellerautomaten bearbeiten können, liegen in $DTime^{TM}(n^3)$ und in $NTime^{TM}(n)$.
2. Kellerautomaten können genau die Probleme bearbeiten, die man mit kontextfreien Grammatiken (also mit der BNF, siehe 1.10, 2.7.1 und 2.7.19) beschreiben kann.
3. Nur die Programmiersprachen, deren Syntaxanalyse sich im Wesentlichen mit deterministischen Kellerautomaten durchführen lässt, sind für die Praxis interessant.
4. Kellerautomaten können genau für hierarchisch aufgebaute Probleme mit Klammerstrukturen eingesetzt werden. (Zum Beispiel für arithmetische Ausdrücke oder für binäre Bäume.)

Lässt man nun noch das Kellerband weg, so erhält man den "endlichen Automaten", also eine Maschine, die die Eingabe von links nach rechts liest, die synchron hierzu ein Ausgabeband beschreibt und die nur *endlich viel Information* in ihrer Zustandsmenge speichern kann.



6.6.4 Definition (nichtdeterministischer endlicher Automat)

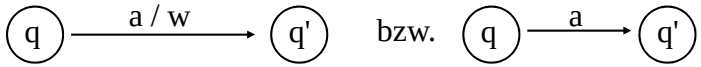
$A = (Q, \Sigma, \Omega, \delta, Q_0, F)$ heißt endlicher Automat
(engl.: finite state machine, finite automaton) $\Leftrightarrow$

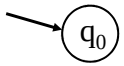
1. Q ist eine endliche nicht-leere Menge (Zustandsmenge),
2. Σ ist eine endliche nicht-leere Menge (Eingabealphabet),
3. Ω ist eine endliche nicht-leere Menge (Ausgabealphabet),
4. $Q_0 \in Q$ ist die Menge der Anfangszustände,
5. $F \subseteq Q$ ist die Menge der Endzustände,
6. $\delta \subseteq Q \times \Sigma \times Q \times \Omega^*$, endliche Menge, ist die Überführungsrelation.

A heißt deterministisch, wenn es zu jedem Paar $(q,a) \in Q \times \Sigma$ höchstens ein Paar $(q',w) \in Q \times \Omega^*$ mit $(q,a,q',w) \in \delta$ gibt.
A heißt endlicher Akzeptor, wenn Ω entfällt, $\delta \subseteq Q \times \Sigma \times Q$ gilt und es genau einen Anfangszustand gibt.

Die Arbeitsweise eines endlichen Automaten A ist sehr einfach: Anfangs befindet man sich in einem Anfangszustand und es steht ein Wort $u \in \Sigma^*$ auf dem Eingabeband. Zu dem jeweiligen Zustand q und dem nächsten Eingabezeichen a suche ein Tupel $(q,a,q',w) \in \delta$; gehe dann in den Zustand q' und drucke w ans Ende des Ausgabebandes; w darf auch das leere Wort sein, dann wird nichts gedruckt. Falls es kein solches Tupel gibt, brich ab. Wenn man auf diese Weise die gesamte Eingabe gelesen hat und sich am Ende in einem Endzustand (aus F) befindet, dann steht auf dem Ausgabeband das Resultat der Eingabe $Res_A(u) = v$. Im Falle des Akzeptors spielt die Ausgabe keine Rolle, sondern nur die Frage, ob man am Ende in einem Endzustand ist oder nicht.
Der endliche Automat funktioniert also wie eine 2-Band-Turingmaschine, die das Eingabeband nur von links nach rechts lesen und synchron hierzu auf das zweite Band die Ausgabe von links nach rechts drucken darf.

Grafische Darstellung:
Die Zustände werden durch Kreise dargestellt, die Übergänge durch Kanten, an die die Eingabe und die Ausgabe, getrennt durch "/" geschrieben werden. Wenn $(q,a,q',w) \in \delta$ bzw. beim Akzeptor $(q,a,q') \in \delta$ ist, so zeichnet man dies in der Form



Anfangszustände erhalten einen Pfeil 

und Endzustände werden doppelt umkringelt. 

Dadurch werden Automaten sehr anschaulich. Da sie zugleich formal definiert sind, lassen sie sich auch maschinell sehr gut verarbeiten.

Was ist die Bedeutung eines endlichen Automaten?

Interpretation als Automat:
Zu jeder Eingabe gibt es eine Menge von Ausgaben, die bei dieser Eingabe möglich wären:
$L(A) = \{(u,v) \mid$ Es gibt eine Folge von Übergängen, die einen Anfangszustand aus Q_0 bei der Eingabe $u \in \Sigma^*$ in einen Endzustand aus F überführen und hierbei die Ausgabe $v \in \Omega^*$ erzeugen $\} \subseteq \Sigma^* \times \Omega^*$.

Falls A deterministisch ist, so gibt es zu jedem $u \in \Sigma^*$ höchstens ein $v \in \Omega^*$ mit $(u,v) \in L(A)$. In diesem Fall wird L(A) zu einer (partiellen) Abbildung $Res_A: \Sigma^* \to \Omega^*$.

L(A) heißt die von A realisierte Relation, Res_A heißt die von A realisierte Abbildung oder die Resultatsfunktion von A.

Was ist die Bedeutung eines endlichen Akzeptors?

Interpretation als Akzeptor:

Jede Eingabe u überführt den Akzeptor in einen Endzustand oder nicht. Demnach definiert man die von A erkannte Sprache:
$L(A) = \{u \in \Sigma^* \mid$ Es gibt eine Folge von Übergängen, die einen Anfangszustand aus Q_0 bei der Eingabe u in einen Endzustand aus F überführen$\} \subseteq \Sigma^*$.

Wir werden im Folgenden vor allem Beispiele zu Akzeptoren angeben und nur ein Beispiel zu einem deterministischen Automaten.

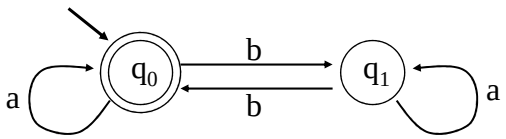
Beispiel a: Gegeben sei der Akzeptor $A = (Q, \Sigma, \delta, q_0, F)$ mit

δ	a	b
q_0	q_0	q_1
q_1	q_2	q_0

$Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$ und $F = \{q_0\}$.

Gesucht wird die Menge
$L(A) = \{u \in \Sigma^* \mid$ u wird vollständig eingelesen und dann befindet sich A im Endzustand $q_0\}$.
In diesem Beispiel ist
$L(A) = \{u \in \Sigma^* \mid$ die Anzahl der b in u ist gerade$\}$.

Skizze:



■

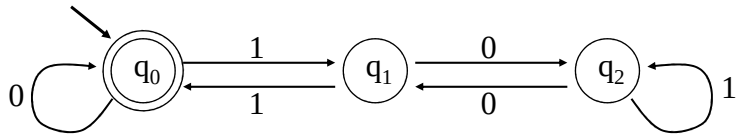
Beispiel b: Gegeben sei der Akzeptor $A^{(3)} = (Q, \Sigma, \delta^{(3)}, q_0, F)$

$\delta^{(3)}$	0	1
q_0	q_0	q_1
q_1	q_2	q_0
q_2	q_1	q_2

mit $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$
und $F = \{q_0\}$.

Gesucht wird die Menge
$L(A^{(3)}) = \{u \in \Sigma^* \mid$ u wird vollständig eingelesen und dann befindet sich $A^{(3)}$ im Endzustand $q_0\}$.

Skizze:



Wie lautet die Menge $L(A^{(3)})$?

Hierzu betrachten wir die Beziehungen der Eingaben. Es gilt:

Wenn der Akzeptor nach Eingabe von u im Zustand q_0 ist, dann führt auch u0 in den Endzustand q_0.
Wenn der Akzeptor nach Eingabe von u im Zustand q_1 ist, dann führt u1 in den Endzustand q_0.
Wenn der Akzeptor nach Eingabe von u im Zustand q_2 ist, dann führt u01 in den Endzustand q_0.

Dies erinnert an die Teilbarkeit durch die Zahl 3:
Wenn u (binär dargestellt) durch 3 teilbar ist, dann ist auch u0 durch 3 teilbar.
Wenn u (binär dargestellt) durch 3 den Rest 1 lässt (d.h. u = 3k+1), dann ist $u1 = (3k+1)\cdot 2 + 1 = 3\cdot(2\cdot k+1)$ durch 3 teilbar.
Wenn u (binär dargestellt) durch 3 den Rest 2 lässt (d.h. u = 3k+2), dann ist $u01 = (3k+2)\cdot 4 + 1 = 3\cdot(4\cdot k+3)$ durch 3 teilbar.

Tatsächlich ist die vom Akzeptor erkannte Sprache
$L(A^{(3)}) = \{u \in \{0,1\}^* \mid$ u ist die Binärdarstellung einer Zahl, die durch 3 teilbar ist$\}$.

Führende Nullen sind hierbei zugelassen.

Der Beweis ergibt sich aus der Tatsache, dass sich der Akzeptor $A^{(3)}$ genau dann im Zustand q_i befindet, wenn die bis dahin gelesene Eingabe, aufgefasst als binär dargestellte Zahl (mit führenden Nullen), bei der Division durch 3 den Rest i besitzt.

Dies lässt sich unmittelbar durch Auflisten aller Fälle nachweisen. Damit ist $L(A^{(3)})$ charakterisiert.

■

Beispiel c: Gegeben sei der Akzeptor $A = (Q, \Sigma, \delta, q_0, F)$ mit

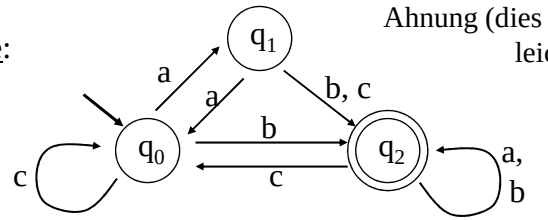
δ	a	b	c
q_0	q_1	q_2	q_0
q_1	q_0	q_2	q_2
q_2	q_2	q_2	q_0

$Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b, c\}$
und $F = \{q_2\}$.

Gesucht wird die Menge
$L(A) = \{u \in \Sigma^* \mid$ u wird vollständig eingelesen und dann befindet sich A im Endzustand $q_2\}$.

Was mag L(A) sein? Keine Ahnung (dies ist auch nicht leicht zu sehen).

Skizze:



■

Wenn der endliche Automat $A=(Q,\Sigma,\Omega,\delta,Q_0,F)$ deterministisch ist, dann kann man (wie bei Turingmaschinen) δ als (partielle) Funktion $\delta: Q\times\Sigma \to Q\times\Omega^*$ auffassen. Man füge nun einen "Fangzustand" q_f zu Q hinzu, zu dem alle nicht erfassten Übergangsmöglichkeiten führen, wodurch man den Automaten A_v erhält: $A_v=(Q\cup\{q_f\},\Sigma,\Omega,\delta_v,Q_0,F)$, wobei δ_v eine Erweiterung von δ ist mit $\delta_v(q,a) = (q_f,\varepsilon)$ für alle $(q,a)\in(Q\cup\{q_f\})\times\Sigma$, für die δ nicht definiert ist. Solch einen Automaten mit totaler Übergangsfunktion δ nennt man vollständig definiert. A_v verhält sich genauso wie A, liest aber jede Eingabe stets vollständig ein.

Analog kann man jeden nichtdeterministischen Automaten vervollständigen, so dass wir also stets annehmen können, dass es zu jedem Paar $(q,a)\in Q\times\Sigma$ mindestens ein Paar $(q',w)\in Q\times\Omega^*$ mit $(q,a,q',w)\in\delta$ gibt.

Meist wird der endliche Automat nur für $\delta\subseteq Q\times\Sigma\times Q\times\Omega$ definiert, d.h., zu jedem gelesenen Eingabezeichen wird genau ein Ausgabezeichen erzeugt. Im deterministischen Fall gilt dann $\delta: Q\times\Sigma\to Q\times\Omega$, was man mit zwei Abbildungen $\delta: Q\times\Sigma\to Q$ und $\gamma: Q\times\Sigma\to\Omega$ darstellt. Weiterhin verlangt man meist, dass der Automat genau einen Anfangszustand q_0 besitzt. Das zugehörige deterministische Modell $A=(Q,\Sigma,\Omega,\delta,\gamma,q_0,F)$ bezeichnet man als Mealy-Automaten.

Hängt die Funktion γ nicht von der Eingabe, sondern nur vom Zustand ab, d.h. $\gamma: Q\to\Omega$, so spricht man von einem Moore-Automaten. Mealy- und Moore-Automaten werden meist dem Entwurf von Schaltwerken zugrunde gelegt

Hinweis: In der Literatur bezeichnet man unsere Automaten aus der Definition 6.6.4 oft als *"verallgemeinerte"* Automaten ("gsm" = generalized sequential machine).

Endliche Automaten gehören zu den einfachsten Algorithmen. Sie arbeiten nur mit einem Eingabeband, das sie zeichenweise von links nach rechts lesen. Dabei drucken sie in jedem Schritt ein Wort (dies kann auch das leere Wort ε sein) aus Ω^* auf das Ausgabeband. Nach genau so vielen Schritten, wie die Eingabe lang ist, ist die Arbeit des endlichen Automatens beendet, d.h. die Laufzeit liegt stets in O(n). Die exakte Arbeitsweise lässt sich leicht algorithmisch formulieren:

```
q := irgendein Zustand aus Q0;
while not end_of_file do read (a);        -- genau ein Zeichen lesen!
    wähle ein Paar (q',w) mit (q,a,q',w) ∈ Q×Σ×Q×Ω*;
    (falls dies nicht existiert => Abbruch "Erfolglose Rechnung");
    q := q';  write (w)                   -- w ausgeben
od;
if q∈F then write (" Erfolgreiche Rechnung ")
else write (" Erfolglose Rechnung ") fi
```

In der Praxis verwendet man nur deterministische Automaten, also solche, bei denen es immer genau ein Paar (q'a) gibt.

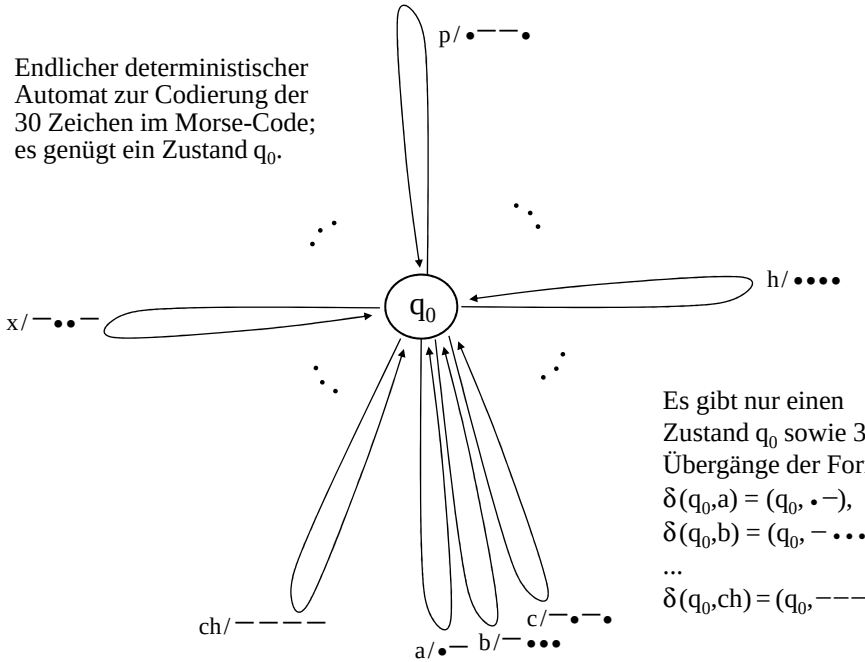
6.6.5: Beispiel Morse-Code

a •−	**b** −•••	**c** −•−•	**d** −••	**e** •
f ••−•	**g** −−•	**h** ••••	**i** ••	**j** •−−−
k −•−	**l** •−••	**m** −−	**n** −•	**o** −−−
p •−−•	**q** −−•−	**r** •−•	**s** •••	**t** −
u ••−	**v** •••−	**w** •−−	**x** −••−	**y** −•−−
z −−••	**ä** •−•−	**ö** −−−•	**ü** ••−−	**ch** −−−−

Dieser Code besteht aus zwei Zeichen • "kurz" und − "lang". Faktisch gibt es aber noch ein drittes Zeichen ◊ "Pause", mit dem man feststellt, wann eine kurz-lang-Folge zu Ende ist.

Die Codierung $\{a, b, ..., z, ä, ö, ü, ch\}^* \to \{•, −, ◊\}^*$ lässt sich mit einem deterministischen endlichen Automaten mit nur einem Zustand realisieren; für die Umkehrung ("Decodierung") braucht man 31 Zustände.

Endlicher deterministischer Automat zur Codierung der 30 Zeichen im Morse-Code; es genügt ein Zustand q_0.

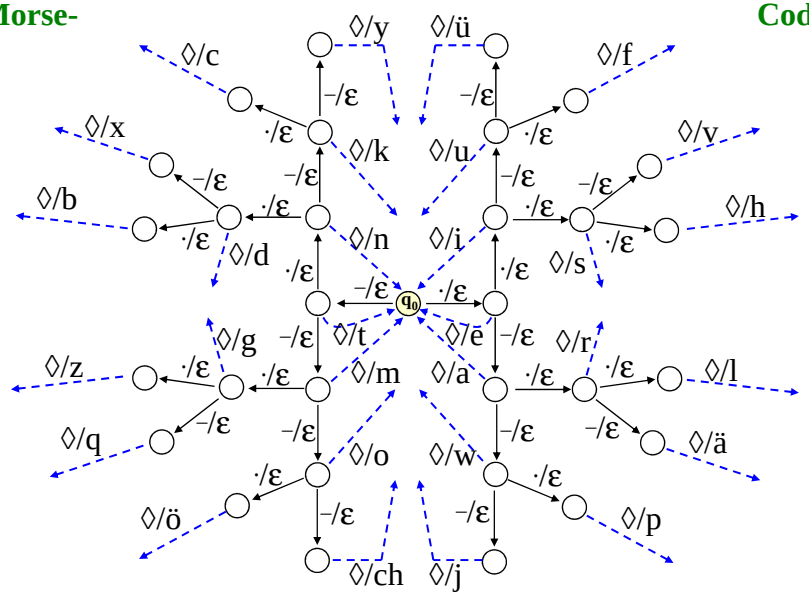


Es gibt nur einen Zustand q_0 sowie 32 Übergänge der Form
$\delta(q_0,a) = (q_0, \bullet-)$,
$\delta(q_0,b) = (q_0, -\bullet\bullet\bullet)$,
...
$\delta(q_0,ch) = (q_0, ----)$.

Um zu entscheiden, zu welchem Zeichen eine Folge aus "•" und "—" gehört, muss man die Folge einlesen, bis das Pausenzeichen "◊" auftritt. Dann kann man eindeutig das gesuchte Zeichen ermitteln und ausgeben. Zu diesem Zweck muss man sich jedoch alle Zeichenfolgen bis zur Länge 4 merken; dies sind $2^1 + 2^2 + 2^3 + 2^4 = 30$ Möglichkeiten. Zusammen mit dem Anfangszustand benötigt man daher 31 Zustände für die Decodierung.

Der zugehörige endliche deterministische Automat lautet $A_{Morse} = (Q, \Sigma, \Omega, \delta, \{q_0\}, \{q_0\})$ mit $Q = \{q_0, q_1, q_2, ..., q_{30}\}$, $\Sigma = \{\bullet, -, \lozenge\}$, $\Omega = \{a, b, ..., z, ä, ö, ü, ch\}$, und die Übergangsfunktion δ folgt aus der Zeichnung auf der nächsten Folie (die Nummerierung der Zustände ist belanglos).

Morse- **Code**



■

Alle gestrichelten Linien führen zum Anfangszustand in der Mitte.

6.6.6: "Reguläre Sprachen": Eine Menge $L \subseteq \Sigma^*$ heißt "endlich akzeptierbar" oder regulär, wenn es einen endlichen Akzeptor gibt, der diese Sprache akzeptiert.

Hierbei ist es egal, ob dieser Akzeptor deterministisch oder nichtdeterministisch ist, da man beweisen kann, dass man zu jedem nichtdeterministischen Akzeptor A einen deterministischen Akzeptor A' mit L(A) = L(A') konstruieren kann.

Weiterhin kann man beweisen, dass jede reguläre Sprache mit einer rechtslinearen Grammatik erzeugt werden kann und umgekehrt, vgl. 2.7.17.

Der Begriff "reguläre Sprache" besitzt eine ebenso zentrale Bedeutung, wie dies die Begriffe "aufzählbare Sprache" und "entscheidbare Sprache" haben. Siehe viele weitere Veranstaltungen im Laufe Ihres Studiums.

6.6.7 Nichtdeterminismus

Bereits in den 70er Jahren wurden die "bewachten Anweisungen" (guarded commands) vorgeschlagen:

Schema:

```
if Bedingung1 => Anweisung1;
[] Bedingung2 => Anweisung2;
. . .
[] Bedingungk => Anweisungk
fi
```

Bedeutung: Alle Bedingungen werden ausgewertet. Falls keine Bedingung true ist, so wird die Anweisung übersprungen, anderenfalls wird genau eine "Anweisung i", deren "Bedingung i" zu true ausgewertet wurde, nichtdeterministisch ausgewählt und ausgeführt.

Wir betrachten das Standardbeispiel ggT. Es gelten folgende Eigenschaften:

(1) $ggT(a,b) = ggT(b,a)$ für alle $a,b \in \mathbb{N}_0$,

(2) $ggT(a,0) = ggT(a,a) = a$ für alle $a \in \mathbb{N}_0$,

(3) $ggT(a,b) = ggT(a-b,b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$,
$ggT(a,b) = ggT(a+b,b)$ für alle $a,b \in \mathbb{N}_0$,

(4) $ggT(a,b) = ggT(b, a \bmod b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$.

Hieraus kann man sofort ein nichtdeterministisches Programm zur Berechnung des ggT angeben: Wähle ständig irgendeine der obigen Gleichungen und führe die zugehörige Wertzuweisung aus, bis irgendwann einmal Fall (2) eintritt.

Das Programm mit bewachten Anweisungen lautet:


```
while B ≠ 0 loop
    if true => H := A; A := B; B := H;
    [] A ≥ B => A := A - B;
    [] true => A := A + B;
    [] A ≥ B => H := A mod B; A := B; B := H;
    fi ;
end loop;
Put (A);
```

Hier ist keine "Strategie" vorgegeben, wie man zum Ende der Schleife kommen kann. Wie beim Nichtdeterminismus üblich lautet die Aussage nur: Wenn es eine Möglichkeit gibt, das Programmstück zu beenden, so wähle eine solche und fahre anschließend mit der nachfolgenden Anweisung fort.

Mit dem Nichtdeterminismus kann man "schwierige" Probleme leicht beschreiben, z.B. das *Binpacking-Problem:*
Es sollen n Gegenstände, die die Gewichte $g_1, g_2, ..., g_n$ besitzen, so in m Behälter gepackt werden, dass in jedem Behälter das Maximalgewicht Max nicht überschritten wird.

Anwendungen:

1. Es sollen n Kisten, die jeweils $g_1, g_2, ..., g_n$ Tonnen wiegen, mit m LKWs transportiert werden, wobei jeder LKW die maximale Zuladung Max Tonnen besitzt. Geht dies?
2. Kann man n Produkte mit m Maschinen in insgesamt Max Zeiteinheiten herstellen, wobei das i-te Produkt g_i Zeiteinheiten zu seiner Herstellung benötigt?
3. Optimierungsproblem:Verteile möglichst viele von n Goldstücken der Gewichte $g_1, g_2, ..., g_n$ so auf m Personen, die jede höchstens Max Gewichtseinheiten tragen können, dass das Gewicht der zurückbleibenden Goldstücke minimal ist.

Formalisierung des BPP als Entscheidungsproblem:

Definition 6.6.8: Binpacking Problem (BPP)

Gegeben: natürliche Zahlen n, m, Max, g_1, g_2, ..., g_n.
Gesucht: eine Abbildung f: {1, 2, ..., n} → {1, 2, ..., m},
so dass für jedes i ($1 \le i \le m$) gilt:

$$\sum_{f(j)=i} g_j \le \text{Max}.$$

f(j) = i bedeutet also: Der j-te Gegenstand wird in den i-ten Behälter gelegt.

Um das Problem, ob es zu n, m, Max, g_1, g_2, ..., g_n ein solches f gibt, zu lösen, kann man deterministisch alle m^n Möglichkeiten durchprobieren. Nichtdeterministisch lässt sich eine Lösung dagegen leicht beschreiben.

Nichtdeterministisches Programmstück nach der Methode "guess and check" (raten und dann nachprüfen):

```
declare n, m, Max: Natural;
begin  read(n, m, Max); ...
declare  g: array (1..n) of Natural; f: array (1..n) of Natural;
     B: array (1..m) of Natural; Fehler: Boolean; i, j: Natural;
begin   < Lies die Gewichte g ein >;
     Fehler := False;  for i:=1 to n do B(i) := 0 od;
raten (guess)
     for j := 1 to n do
          if True => f(j) := 1; [] True => f(j) := 2; ...
          [] True => f(j) := m-1; [] True => f(j) := m  fi  od;
prüfen (check)
     for j := 1 to n do B(f(j)) := B(f(j)) + g(j);
          if B(f(j)) > Max then Fehler := true fi od;
     if not Fehler then < gib die Lösung f aus > fi
end  end;
```

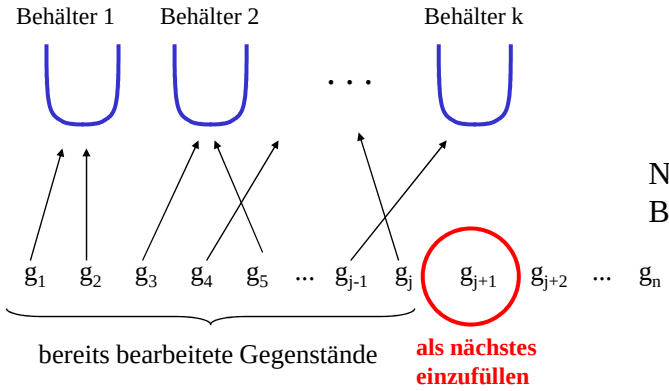

6.6.9 Backtracking

Beachten Sie, dass bei einem nichtdeterministischen Programm der Misserfolg nicht beachtet wird. Nur im Erfolgsfall (also der Fall "Es existiert eine Lösung") wird eine Lösung ausgegeben.

Ein nichtdeterministisches Programm dient der Klärung des Problems; für die Ermittlung einer Lösung ist es in der Praxis kaum hilfreich. Um nun eine Lösung zu finden, geht man alle Möglichkeiten durch: Die verschiedenen Möglichkeiten werden baumartig aufgeschrieben und systematisch durchlaufen und abgeprüft.

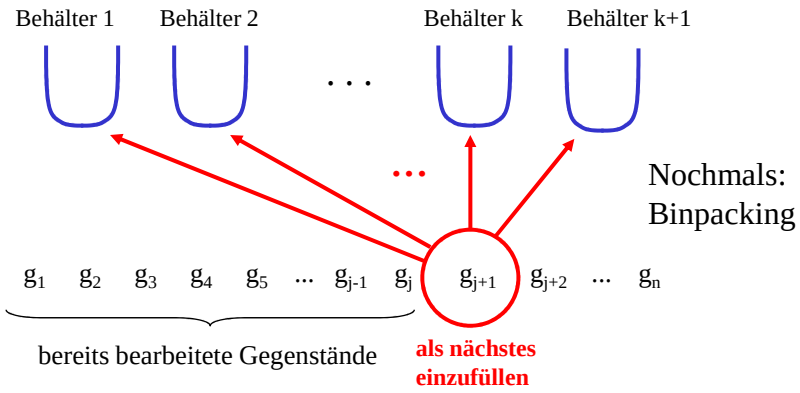
Ein solches systematisches Durchprobieren aller Möglichkeiten bezeichnet man als Backtracking = "Rückverfolgen durch den Lösungsbaum".

Backtracking geht stets von einer vorhandenen Situation aus und reduziert das Problem auf einfachere Probleme. Eine typische Situation beim Füllen der m Behälter ist: Man hat die ersten j Gegenstände bereits konfliktfrei in die ersten k Behälter gefüllt und muss nun den (j+1)-ten Gegenstand einfüllen.



Nochmals: Binpacking

Wir können den (j+1)-ten Gegenstand nun einem der bereits vorhandenen Behälter 1, 2, ..., k zuordnen oder einen neuen Behälter (k+1) verwenden, sofern k+1 ≤ m, d.h., k < m ist.



Nochmals: Binpacking

Alle diese k+1 Fälle probieren wir im Backtracking durch. Die folgende Prozedur vollzieht genau die obige Rekursion nach.

Entwurf der rekursiven Prozedur (es fehlen noch Details):

```
procedure BTBPP (j, k: Natural) is
declare i: Natural;
begin
      if  j = n  then  ..... < Lösung gefunden > .....
      else  for i:=1 to k do
                if  Gegenstand (j+1) passt noch in Behälter i
                      then  BTBPP (j+1, k)  fi   od;
            if k < m then BTBPP (j+1, k+1) fi
      fi
end;
```

Um zu überprüfen, ob der Gegenstand (j+1) noch in den Behälter i passt, verwenden wir wieder das globale Feld B: array(1..m) of Natural, in welchem wir die Summe der Gewichte aller Gegenstände, die aktuell im jeweiligen Behälter liegen, notieren. Der Gegenstand (j+1) darf in den Behälter i nur gelegt werden, wenn dadurch das Maximalgewicht nicht überschritten wird, also nur, wenn B(i)+g(j+1) ≤ Max ist.

So erhalten wir die Verfeinerung des Prozedurschemas (man beachte, dass vor und nach jedem rekursiven Aufruf das Gewicht des jeweiligen Behälters B(i) angepasst werden muss):

```
procedure BTBPP (j, k: Natural) is
declare i: Natural;
begin
      if  j = n  then  < Lösung gefunden >
      else  for i:=1 to k do
                if  B(i) + g(j+1) ≤ Max  then
                      B(i) := B(i) + g(j+1);  BTBPP(j+1, k);
                      B(i) := B(i) - g(j+1)  fi
            od;
            if k < m then B(k+1) := g(j+1);  BTBPP(j+1, k+1);
                      B(k+1) := 0  fi
      fi
end;
```


Diese Prozedur stellt bisher nur fest, ob eine Lösung existiert, aber sie gibt keine mögliche Zuordnung zu den Behältern aus. Hierfür müssen wir uns noch zu jedem Gegenstand merken, in welchen Behälter er gelegt wurde. Dies geschieht in dem globalen Feld f: array (1..n) of Natural. Dieses Feld beschreibt genau die Abbildung f aus Definition 6.6.8.

Wir müssen nicht alle m^n möglichen Abbildungen f durchprobieren. Wir können annehmen, dass der Gegenstand mit der Nummer 1 stets im Behälter 1 liegt, der Gegenstand mit der Nummer 2 stets in einem der Behälter 1 oder 2 usw. Durch Umnummerieren der Behälter lässt sich also stets erreichen, dass f(j) ≤ j für alle j=1,2,...,n gilt. Speziell ist dann f(1)=1. In der Prozedur stellen wir automatisch sicher, dass f(j) ≤ j für alle weiteren j gilt, indem für den Gegenstand j+1 neben den bereits betrachteten Behältern nur der Behälter k+1 (und nicht k+2, k+3 usw.) ausprobiert wird. Weil m die höchste Nummer eines Behälters ist, brauchen wir also nur Abbildungen f zu betrachten mit:

f(j) ≤ Min(j,m), wobei Min(j,m) das Minimum der beiden Zahlen j und m ist.

So erhalten wir BTBPP (Backtrackingprozedur für das Binpackingproblem):

```
procedure BTBPP (j, k: Natural) is
< Beachte: Die Gegenstände 1 bis j sind bereits eingefügt, wobei k Behälter verwendet wurden >
declare i: Natural;
begin
    if  j = n  then  < Lösung gefunden, drucke das Feld f aus >
    else  for i:=1 to k do
              if  B(i) + g(j+1) ≤ Max  then
                  f(j+1):=i;  B(i):=B(i) + g(j+1);  BTBPP (j+1, k);
                  B(i):=B(i) - g(j+1);  f(j+1):=0  fi
          od;
          if k < m then  f(j+1):=k+1;  B(k+1):=g(j+1);
                  BTBPP (j+1, k+1);  B(k+1):=0; f(j+1):=0  fi
    fi
end;
```


Globale Variablen sind wiederum:

```
Max, m, n: Natural;
g, f: array(1..n) of Natural;
B: array(1..m) of Natural;
```

Der Aufruf der Prozedur BTBPP lautet, sofern bereits alle Daten in die globalen Variablen Max, m, n und g eingelesen wurden:

```
if (n<1) or (m<1) then
        < Abbruch, da keine Lösung zu suchen ist > fi; ...
for j:=1 to n do if g(i) > Max then
        < Abbruch, da keine Lösung möglich > fi od; ...
for i:=1 to m do f(i):=0 od;
for j:=1 to n do B(j):=0 od;
B(1):=g(1); f(1):=1; BTBPP(1,1);
```


Manche der obigen Anweisungen erscheinen überflüssig (z.B. f(j+1):=0 oder for i:=1 to n do f(i):=0 od;). Wir haben sie dennoch aufgeführt, da sie eventuell sehr hilfreich werden können, wenn Fehler auftreten oder wenn die Prozedur noch von anderen Programmen aufgerufen wird.

Die Übertragung nach Ada ist nun einfach.

Aufgabe: Schreiben Sie das Programm für die Lösung des Binpacking-Problems fertig und testen Sie es an einigen Daten. Messen Sie Zeit und Speicherplatz. Machen Sie sich das systematische Durchprobieren mittels Backtracking klar und üben Sie diese Technik auch an dem in 1.15, Aufgabe 15 genannten Erbschaftsproblem; zur Illustration siehe das Zahlenbeispiel in 6.8.

Hinweise: Nichtdeterministische Sprachelemente sind in imperativen Programmiersprachen unüblich. In Ada gibt es eine nichtdeterministische Auswahl ("select-Anweisung" mit den Schlüsselwörtern select, or und else, zusammen mit task, accept, entry, delay und terminate), allerdings ist sie auf die Kommunikation zwischen Prozessen beschränkt, siehe Programmierübungen bzw. Literatur über Ada.

In prädikativen Programmiersprachen (wie PROLOG) gehören der Nichtdeterminismus als Beschreibung und das deterministische Backtracking als Lösungsmethode zu den Grundprinzipien. Dies wird in der Vorlesung über Logik angesprochen und in anderen Vorlesungen näher behandelt.

6.7 Entwurfsmethoden für Algorithmen

Für Algorithmen gibt es einige Entwurfsprinzipien. Die vier bekanntesten haben wir teilweise bereits behandelt:

Greedy-Technik (= gieriges Vorgehen): Man arbeitet sich schrittweise zur Lösung vor, indem man in jedem Schritt die Maßnahme, die den größten lokalen Nutzen verspricht, durchführt.
(Stichwörter: Dijkstra-Verfahren für kürzeste Wege, Prim-Algorithmus für spannende Bäume, Hillclimbing Techniken)

Divide and Conquer (= teile-und-herrsche): Zerlege das Problem in Teilprobleme, löse diese nach der gleichen Technik und setze dann aus den Teilen eine Lösung zusammen.
(Stichwörter: Quicksort, Sortieren durch Mischen, schnelle Multiplikation in $O(n^{1,585})$ nach 6.5.4.)

Dynamisches Programmieren: Setze die Lösung aus allen kleineren Lösungen zusammen.
(Stichwörter: Warshall-Algorithmus 6.5.5, optimale Suchbäume, Syntaxanalyse kontextfreier Sprachen "CYK".)

Backtracking: Systematisches baumartiges Durchmustern sämtlicher Lösungsmöglichkeiten.
(Beispiele: Binpacking, Erbschaftsproblem, Syntaxanalyse kontextsensitiver Sprachen,)

Diese und andere Entwurfsmethoden werden Sie in weiteren Veranstaltungen, insbesondere in "Entwurf und Analyse von Algorithmen", vertiefen.

6.8 Historische Anmerkungen

Vor 100 Jahren hoffte man, dass sich alle Probleme, die durch Gleichungen über den natürlichen Zahlen formal darstellbar sind, mit Algorithmen lösen lassen. 1931 bewies der österreichische Logiker K. Gödel (1906-1978), dass dies unmöglich ist.

Aus dem Unentscheidbarkeitssatz 2.3.2 wurde eine Fülle von algorithmisch unlösbaren Problemen abgeleitet, z.B.
- entscheide, ob Programme für *alle* Eingaben anhalten,
- entscheide, ob Programme für *alle* Eingaben nicht anhalten,
- entscheide, ob zwei beliebige Programme dasselbe berechnen, also ob sie die gleiche realisierte Abbildung besitzen,
- entscheide, ob eine kontextfreie Grammatik alle Wörter erzeugt.

Zur Lösung solcher Probleme ist in jedem Einzelfall eine gesonderte Analyse und Beweisführung erforderlich, d.h., hier ist stets ein gut ausgebildeter Experte erforderlich. Dies bezeichnen wir als das Arbeitsplatzerhaltungstheorem der Informatik.

Dass Algorithmen lange Laufzeiten haben können, ist seit altersher bekannt. Der Euklidische Algorithmus oder das Newtonverfahren zur Berechnung von Nullstellen sind relativ schnell arbeitende Verfahren. Die Gaußsche Elimination zur Lösung linearer Gleichungssysteme benötigt $O(n^3)$ Schritte, wobei n die Zahl der Variablen ist. Dies ist für die Berechnung per Hand meist schon zu aufwendig.

Die Entwicklung von Rechenmaschinen wurde durch die zeitraubenden Rechnungen, vor allem im technischen Bereich, beflügelt. Systematische Untersuchungen zur Komplexität begannen in den 1950er Jahren. Das exponentielle Wachstum mancher Algorithmen führte zu der Frage, ob man die Komplexitätsklasse **P** charakterisieren und ihr Verhältnis zur Klasse **NP**, in der viele praktische Probleme liegen, klären könne.

1971 zeigte S. Cook, dass es in **NP** Probleme mit folgender Eigenschaft gibt: Liegt nur eines dieser Probleme in **P**, so fallen die Klassen **P** und **NP** zusammen. Diese Probleme sind unter dem Begriff "NP-vollständige Probleme" bekannt geworden. Man kennt mittlerweile rund 10.000 solcher schwerer Probleme.

Hierzu zählen Zuordnungs- und Optimierungsprobleme wie die Erstellung von Stundenplänen oder optimale Standorte oder Rundreisen mit vorgegebenen Eigenschaften. Ein besonders einfaches Problem ist das Erbschaftsproblem (engl.: "partition problem"): Gegeben seien n natürliche Zahlen $g_1, g_2, ..., g_n$; gibt es hierzu eine Menge von Indizes $I = \{i_1, i_2, ..., i_r\} \subseteq \{1, 2, ..., n\}$ mit

$$\sum_{j=1}^{r} g_{i_j} = G, \quad \text{wobei } G = \left(\sum_{i=1}^{n} g_i\right) / 2 \quad ?$$

Kann man also die n Zahlen in zwei Teilmengen zerlegen, deren Summen gleich sind? (vgl. 1.15)

Falls Sie einmal an diesem einfach aussehenden Problem üben wollen, so stellen Sie für folgende 20 Zahlen, deren Summe 33.480.070 ist (d.h. G = 16.740.035), fest, ob es eine Teilmenge gibt, deren Summe gleich G ist:

1.976.834, 1.864.558, 1.755.621, 1.575.931, 2.169.504,
1.567.429, 2.001.571, 1.682.544, 1.289.337, 1.223.752,
1.884.283, 1.671.449, 1.400.530, 1.547.733, 1.338.626,
1.438.792, 2.010.563, 1.422.589, 1.863.866, 1.794.558

Mittlerweile ist die "Komplexitätstheorie", die eng mit der "Algorithmik" zusammenhängt, ein etablierter Zweig der Informatik. Auch wenn hier viele abstrakte Erkenntnisse gewonnen wurden, so hat man bisher noch keine Techniken entwickeln können, um zu beweisen, dass es keine deterministischen Turingmaschinen (oder Programme) geben kann, die gewisse Probleme in polynomieller Zeit lösen, obwohl das Problem in exponentieller Zeit gelöst werden kann.

Schon um 1955 gelang es dagegen, eine Klasse von Problemen zu entdecken, die sich in Realzeit durch Schaltwerke mit endlichem Gedächtnis lösen lassen. Ausgangspunkt waren Arbeiten von Kleene über Nervennetze.

E. F. Moore veröffentlichte 1956 "Gedanken Experiments on Sequential Machines" (in den Automata Studies, Princeton Univ., Princeton, N.J.) und G. H. Mealy untersuchte "A Method for Synthesizing Sequential Circuits" (Bell System Technical Journal 34, 1955). Beide Arbeiten waren grundlegend für die Beschreibung von Schaltwerken auf höherer Ebene (= endliche Automaten) und für die Synthese von Hardwarebausteinen. Ab 1959 wurden diese regulären Probleme exakt klassifiziert.

Man erkannte rasch, dass man komplexere Rechenmodelle benötigte: Kellerautomaten, Stackautomaten, linear beschränkte und weitere Automaten wurden ab 1961 definiert und analysiert.

Immer wieder stieß man auf das Problem, nichtdeterministische Verfahren deterministisch simulieren zu müssen. Es entstanden Hunderte von Klassen und Hierarchien, ohne dass man essentielle Hinweise zur Lösung des **P**=**NP**-Problems erhielt.

Heute weiß man von sehr vielen Problemen ziemlich genau, in welcher Komplexitätsklasse sie liegen; z.B. lässt sich die Frage, ob eine natürliche Zahl eine Primzahl ist, in polynomieller Zeit (bzgl. der Länge der eingegebenen Zahl) lösen. Doch wir wissen noch viel zu wenig über die (Nicht-) Existenz mathematischer Räume, in denen man sehr effizient rechnen und zwischen denen man relativ schnell hin- und herschalten kann.

Alle diese Fragen werden durch parallele und verteilte Algorithmen noch verschärft. Hier kommen neue Komplexitätsklassen ins Spiel, auf die Sie erst im Hauptstudium stoßen werden.

Zur Aufgabe aus 6.1.4: Untersuchen Sie die Funktion log*.

Hinweise: Man kann die Funktion log* auch induktiv definieren:

$$\log^*(x) = \begin{cases} 0, & \text{für } x < 2, \\ \log^*(\log(x)) + 1, & \text{für } x \geq 2.0 \end{cases}$$

So lassen sich einige Werte leicht bestimmen:

$\log^*(2) = \log^*(\log(2)) + 1 = \log^*(1) + 1 = 1$

$\log^*(3) = \log^*(\log(3)) + 1 = \log^*(1.58496...) + 1 = 1$

$\log^*(4) = \log^*(\log(4)) + 1 = \log^*(2) + 1 = 2$

$\log^*(40) = \log^*(5.32193...) + 1 = \log^*(2.40...) + 1 + 1$
$= \log^*(1.5...) + 1 + 1 + 1 = 3$

$\log^*(65536) = \log^*(16) + 1 = \log^*(4) + 2 = 4$

$\log^*(2^{65536}-1) = \log^*(65535.99...) + 1 = \log^*(15.99...) + 2$
$= \log^*(3.99...) + 3 = \log^*(1.99...) + 4 = 4$

$\log^*(2^{65536}) = \log^*(65536) + 1 = \log^*(16) + 2 = \log^*(4) + 3 = 5$

Beachte: $2^{65536} \approx 10^{19726}$ ist bereits eine sehr große Zahl mit fast 20000 Stellen. log* wächst extrem langsam, da erst $\log(2^{10^{19726}}) = 6$ ist.

Aufgabe aus 6.1.4 (Fortsetzung): Die Funktion log*.

Die Funktion log* besitzt eine "Umkehrfunktion" (eingeschränkt auf die natürlichen Zahlen **IN**), nennen wir sie h: $\mathbf{IN} \to \mathbf{IN}$, mit den Eigenschaften $\log^*(h(n)) = n$ und $h(\log^*(n)) = n$ für alle $n \in \mathbf{IN}$.

Offenbar gilt

$h(1) = 2$, $h(2) = 4$, $h(3) = 16$, $h(4) = 65536$, $h(5) = 2^{65536}$.

Die Funktion h erfüllt für alle $n \in \mathbf{IN}$ die Eigenschaft

$$h(n+1) = 2^{h(n)}.$$

Diese Funktion h wächst also viel viel schneller als beispielsweise die Exponentialfunktion 2^n.

h lässt sich folgendermaßen beschreiben:

$$h(n) = 2^{2^{2^{\cdot^{\cdot^{2}}}}}$$ n Zweien stehen hier insgesamt.

7. Axiomatische Semantik von Programmen

Lernziele dieses Kapitels:

Ein Algorithmus beschreibt eine Abbildung der Eingabewerte auf die Ausgabewerte. Ein Algorithmus kann in einer Programmiersprache durch (unendlich) viele Programme dargestellt werden. Woher weiß man aber, dass ein Programm tatsächlich eine gewünschte Abbildung realisiert oder einen abstrakt dargestellten Algorithmus implementiert?

In der Praxis wird meist "getestet": Einige Werte werden in das Programm eingegeben, und wenn die gewünschten Ausgabewerte herauskommen, so *glaubt* man, das Programm werde schon richtig sein. Doch dies ist völlig unzureichend; denn auf diese Weise kann man mit etwas Glück Fehler entdecken, aber niemals *die Abwesenheit von Fehlern* beweisen!

In diesem Kapitel erfahren Sie, wie die Arbeitsweise eines Programms mithilfe von logischen Formeln nachvollzogen und das Programm als korrekt nachgewiesen werden kann. Die von Hoare erarbeitete Methode wird an mehreren Beispielen eingeübt. Mit Hilfe der weakest precondition (wp) lässt sich diese Methode sogar teilweise automatisieren. Schließlich wird die Wichtigkeit des Terminierungs-Beweises hervorgehoben.

7.1 Verifikation und Korrektheit

7.1.1 Begriffe

Semantik = Bedeutung (in einem objektiven Sinn, also ohne persönliche Interpretationen und Aufforderungen zu Handlungen)

Beispiel: Wenn jemand sagt, "hier zieht es", so ist die Semantik dieses Satzes die Aussage, dass ein Luftzug vorhanden ist. Solche Aussagen kann man mit "ja" oder "nein" beantworten. In der Umgangssprache ist aber mehr damit verbunden. Wer "hier zieht es" sagt, will meist nicht diese Tatsache bestätigt hören, sondern er möchte, dass der Urheber ermittelt wird oder jemand umgehend ein offenes Fenster schließen möge. Solche zusätzlichen Handlungen interessieren jedoch bei der Semantik nicht; sie zählen eher zur Pragmatik von Sätzen.

In der Informatik werden Programmiersprachen untersucht. Deren Syntax haben wir bereits ab 1.10 kennen gelernt. Die Semantik eines Programms π ist die zugehörige realisierte Abbildung f_π, wie sie in 2.5 definiert wurde. Die Aufgabe lautet nun, entweder die realisierte Abbildung f_π aus dem Programm π zu ermitteln oder auf andere Weise zu zeigen, dass π die gewünschte Abbildung tatsächlich realisiert.

Verifikation eines Programms (oder eines Algorithmus)
= Nachweis, dass das Programm gewisse (zu prüfende) Eigenschaften besitzt.

Verifikation (im engeren Sinne)
= Beweis, dass ein Programm π genau das berechnet, was es berechnen soll, dass also $f_\pi = f$ gilt für eine vorgegebene Abbildung f.

Dieser Beweis besteht meist aus zwei Teilen:
- Terminierung: Das Programm π hält für alle Eingaben an.
- Korrektheit: Das Programm π arbeitet für jede Eingabe richtig.

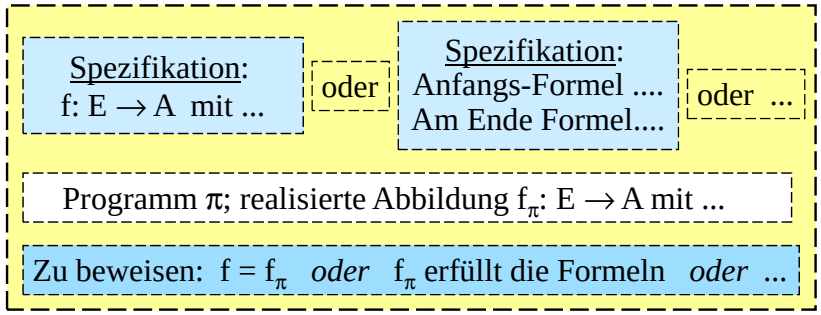
Diese beiden Teile sind unabhängig von einander. Sie lassen sich auf das Halteproblem (siehe 2.3.2) zurückführen und sind daher algorithmisch nicht lösbar, siehe künftige Theorie-Vorlesungen.

In diesem Kapitel behandeln wir vor allem die Korrektheit.

Präzisierung: Für den Beweis der Korrektheit muss man zweierlei angeben bzw. ermitteln und auf Gleichheit testen:
- die Spezifikation, die angibt, *was* zu realisieren ist,
- die vom Programm π realisierte Abbildung $f_\pi: E \to A$

(E = Menge der zulässigen Eingaben, A = Ausgabemenge).

Spezifikation = präzise, vom Programm unabhängige Beschreibung der angestrebten Funktion oder des Ein-Ausgabe-Verhaltens. Diese erfolgt z.B. durch Gleichungen, Formeln, Eigenschaften usw.



7.1.2 Hier verwendete Vorgehensweise

Der konkrete Ablauf eines Programms besteht aus einer Folge von "elementaren Aktionen", nämlich elementaren Anweisungen und Bedingungen, wie wir dies mit einem Ablaufprotokoll nachvollziehen. Zwischen je zwei Aktionen schreiben wir logische Formeln (bzw. "Aussagen") und beweisen, dass die jeweils nächste Formel aus der vorhergehenden Formel gefolgert werden kann, wenn man dazwischen die "elementare Aktion" ausführt.

Vorgehensweise: Im Programmablauf ist für jede Situation
{A} <elementare Aktion> {B}
zu beweisen, dass durch die <elementare Aktion> aus der Formel "A" die Formel "B" gefolgert werden kann.

Beispiel: Die logische Formel $x > 3$ wird durch die Anweisung "x := x+7" in die Formel $x > 10$ überführt. Wir schreiben dies in der Form:

{x > 3} x := x+7 {x > 10}

Beispiel 7.1.3:

Spezifikation: Realisiert werden soll die Funktion
f: $\mathbb{Z} \to \mathbb{Z}$ mit f(n) = 2n für alle n $\in \mathbb{Z}$.

Behauptung: Folgendes Programm realisiert diese Funktion.

```
x, y, i: Integer;
begin
    Get(x);
    y := x;
    i := 0;
    while i < x loop
        i := i+1;
        y := y+1;
    end loop;
    Put(y);
end;
```

Dies sieht zunächst richtig aus, wie die Ablaufprotokolle für die Eingaben 3, 0 oder 6 nahe legen. Aber ist es wirklich korrekt?

Unsere Vorgehensweise: Schreibe zwischen Aktionen Formeln:

```
x, y, i: Integer;
{logische Formel einfügen}
begin  Get(x);
    {logische Formel einfügen}
    y := x;
    {logische Formel einfügen}
    i := 0;
    {logische Formel einfügen}
    while i < x loop
        {logische Formel einfügen}
        i := i+1;
        {logische Formel einfügen}
        y := y+1;
        {logische Formel einfügen}
    end loop;
    {logische Formel einfügen}
    Put(y);
    {logische Formel einfügen}
end;
```


Offensichtlich einzusetzende Formeln bzw. Aussagen:

```
x, y, i: Integer;
{x, y, i enthalten eine ganze Zahl, ihr Inhalt ist hier aber undefiniert}
begin  Get(x);
    {eine ganze Zahl a wird eingegeben und nun ist x = a}
    y := x;
    {logische Formel einfügen}
    i := 0;
    {logische Formel einfügen}
    while i < x loop
        {logische Formel einfügen}
        i := i+1;
        {logische Formel einfügen}
        y := y+1;
        {logische Formel einfügen}
    end loop;
    {logische Formel einfügen}
    Put(y);
    {2a wird ausgegeben}
end;
```


Hieraus folgende einzufügende Formeln bzw. Aussagen:

```
x, y, i: Integer;
{x, y, i enthalten eine ganze Zahl, ihr Inhalt ist hier aber undefiniert}
begin  Get(x);
    {eine ganze Zahl a wird eingegeben und nun ist x = a}
    y := x;
    {hier gilt x = a und y = a}
    i := 0;
    {hier gilt x = a und y = a und i = 0. Also gilt auch y = a + i}
    while i < x loop
        {hier gilt x = a und y = a + i und i < x}
        i := i+1;
        {logische Formel einfügen}
        y := y+1;
        {logische Formel einfügen}
    end loop;
    {i ≥ x und weitere logische Formel einfügen}
    Put(y);
    {2a wird ausgegeben}
end;
```

sonst würde der Schleifenrumpf nicht betreten!

sonst würde die Schleife nicht verlassen!

Hieraus folgende einzufügende Formeln bzw. Aussagen:

```
x, y, i: Integer;
{x, y, i enthalten eine ganze Zahl, ihr Inhalt ist hier aber undefiniert}
begin  Get(x);
    {eine ganze Zahl a wird eingegeben und nun ist x = a}
    y := x;
    {hier gilt x = a und y = a}
    i := 0;
    {hier gilt x = a und y = a und i = 0. Also gilt auch y = a + i}
    while i < x loop
        {hier gilt x = a und y = a + i und i < x}
        i := i+1;
        {hier gilt x = a und y = a + i-1 und i ≤ x}
        y := y+1;
        {hier gilt x = a und y = a + i und i ≤ x}
    end loop;
    {i ≥ x  und weitere logische Formel einfügen}
    Put(y);
    {2a wird ausgegeben}
end;
```

Diese Formel folgt aus dieser Formel, wenn man im Rumpf der Schleife bleibt, wenn also auch nächstes Mal "i<x" gilt!

Offensichtlich gilt bisher: Wenn im Programm die Folge

{A} c {B} (für eine Aktion c und zwei Formeln A und B)

auftritt, dann ist folgende Aussage stets richtig:

Wenn A zutrifft und c ausgeführt wird, dann trifft anschließend B zu.

Wie gehen wir nun aber mit der noch fehlenden Formel

{i ≥ x und weitere logische Formel einfügen}

um? Sie muss zutreffen, wenn die while-Schleife abbricht. Folglich kann man die Formeln, die zu Beginn der while-Schleife gültig sind, hier einsetzt. Dann erhält man hier:

{i ≥ x und x = a und y = a+ i}

Durch Einsetzen von i ≥ x und x = a in y = a+ i folgt hieraus:

{i ≥ x und x = a und y = a+ i ≥ a + x = a + a = 2a}

{i ≥ x und x = a und y = a+ i ≥ a + x = a + a = 2a}

Aus dieser Formel erkennt man: Nur wenn i = x ist, ist y = 2a. Der Wert von i wird aber nur verändert, wenn anfangs x = a > 0 war. Anderenfalls ist a = x ≤ 0 = i und der Wert von i bleibt gleich 0, da der Schleifenrumpf nicht ausgeführt wird. In diesem Fall ist der Wert von y am Ende y = a + i = a.

Die vom Programm berechnete Funktion f' lautet daher:

$$f'(a) = \begin{cases} 2a, & \text{falls } a > 0 \\ a, & \text{falls } a \le 0 \end{cases}$$

Beachten Sie: Wir haben hiermit bewiesen, dass diese Funktion f' die vom Programm berechnete Funktion ist!

Wir geben noch das Gesamtprogramm an:

Gesamtprogramm mit Formeln bzw. Aussagen:

```
x, y, i: Integer;
{x, y, i enthalten eine ganze Zahl, ihr Inhalt ist hier aber undefiniert}
begin  Get(x);
    {eine ganze Zahl a wird eingegeben und nun ist x = a}
    y := x;
    {hier gilt x = a und y = a}
    i := 0;
    {hier gilt x = a und y = a und i = 0. Also gilt auch y = a + i}
    while i < x loop
        {hier gilt x = a und y = a + i und i < x}
        i := i+1;
        {hier gilt x = a und y = a + i-1 und i ≤ x}
        y := y+1;
        {hier gilt x = a und y = a + i und i ≤ x}
    end loop;
    {hier gilt i ≥ x und x = a und y = a+ i ≥ a + x = 2a}
    Put(y);
    {ausgegeben wird 2a für a > 0, sonst wird a ausgegeben}
end;
```

Vorgehen 7.1.4: Dieses Vorgehen mit logischen Formeln zwischen den Aktionen liegt der *"axiomatischen Semantik"* zugrunde:

Schreibe zwischen je zwei elementare Aktionen eine Formel, die hier erfüllt sein muss. Gemeint sind "prädikatenlogische Formeln". Solche Formeln sind Ihnen aus der Mathematik geläufig. Sie werden aufgebaut
- aus Booleschen Ausdrücken (true, false, logischen Variablen, Vergleichsausdrücken und den Operatoren $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (impl)) sowie
- aus den Quantoren $\forall$ und $\exists$ ("für alle" und "es existiert").

Typische prädikatenlogische Formeln sind:

$(x = y) \wedge (x+1= y)$ d.h., x ist gleich y und x+1 ist gleich y.

$\forall\, x : (x = x)$ d.h., für alle Werte x gilt, dass x gleich x ist.

$\forall\, x : (\exists\, y : (x+y = 0))$

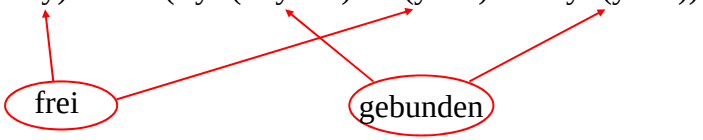
d.h., zu jedem x gibt es ein y, so dass x+y Null ist.

Solche Formeln können wahr oder falsch sein (man sagt auch, sie sind *erfüllt* oder nicht), wobei stets der Wertebereich, aus dem die Werte der Variablen sein müssen, zu beachten ist. Die erste der obigen Formeln ist beispielsweise falsch, wenn man die ganzen Zahlen zugrunde legt; sie ist aber richtig, wenn man die einelementige Menge {1} mit 1+1=1 wählt. Die zweite Formel ist stets wahr. Die dritte Formel ist für ganze Zahlen wahr, für natürliche Zahlen falsch.

Definition 7.1.5: Ein Vorkommen einer Variablen x in einer Formel heißt *gebunden*, wenn dieses Vorkommen im Bereich eines Quantors liegt, wenn sich also dieses Vorkommen von x irgendwo in einer Teilformel $\exists\, x : (...)$ oder $\forall\, x : (...)$ befindet. Ein Vorkommen von x, das außerhalb jeder solchen Teilformel liegt, heißt *frei*.

Eine Variable kann in einer Formel sowohl frei als auch gebunden vorkommen. Zum Beispiel kommt die Variable y in folgender Formel zweimal frei und zweimal gebunden vor:

$(a = y) \vee \forall\, x : (\exists\, y : (x+y = 0) \wedge (y > a) \wedge \forall\, y : (y < x))$



Die Variable a kommt in dieser Formel nur frei vor, jedes Vorkommen der Variablen x ist dagegen gebunden.

7.1.6: Solche Bedingungen / prädikatenlogische Formeln nennt man im Zusammenhang mit der Korrektheit von Programmen ***Zusicherungen*** oder ***assertions*** und schreibt sie in geschweifte Klammern zwischen die Anweisungen.

Jede Anweisung verändert diese Zusicherungen. Wenn vor der Durchführung einer Anweisung c die Zusicherung A erfüllt war und nach der Ausführung von c die Zusicherung B erfüllt ist, so schreibt man hierfür

$\{A\}\ c\ \{B\}$.

Als Wertebereich wählen wir hier stets die ganzen Zahlen **Z**.
Der Fall "undefiniert" wird dargestellt durch das Zeichen ⊥.

Beispiel 7.1.7: Wir betrachten ein sehr einfaches Beispiel, wobei wir die prädikatenlogischen Formeln links umgangssprachlich und rechts formal hinschreiben:

```
x: Integer;                          x: Integer;
{x ist undefiniert}                  {x = ⊥}
Get(x);                              Get(x);
{x ist eine ganze Zahl a}     ⇒      {a ∈ Z ∧ x = a}
x := x+2;                            x := x+2;
{x ist a+2}                          {x = a+2}
Put(x);                              Put(x);
{a+2 wird ausgegeben}                {Ausgabe = a+2}
```


Beispiel 7.1.8: Wir betrachten folgendes Programmstück:

```
x: Natural := 0; q: Natural := 1;

Get(x);

while x > 0 loop

    q := q + q;

    x := x - 1;

end loop;

Put (q);
```

Schwierigkeiten macht die while-Schleife. Da sie immer wieder durchlaufen wird, muss die zugehörige Bedingung am Ende und erneut am Anfang gelten, d.h., sie darf durch den Rumpf der while-Schleife nicht verändert werden. Solche Bedingungen nennt man "Schleifen-Invarianten". In diesem Beispiel lautet sie

$$q = 2^{a-x}$$

wobei a die eingelesene Zahl ist.

Was berechnet dieses Programm? Selbst lösen, dann erst weiterlesen.

Wir fügen die Zusicherungen ein:

```
x: Natural := 0; q: Natural := 1;
{ x, q ∈ IN0 ∧ x = 0 ∧ q = 1 }
Get(x);
{ a ∈ IN0 ∧ x = a ∧ q = 2^(a-x) }
while x > 0 loop
{ a ∈ IN ∧ x > 0 ∧ q = 2^(a-x) }
    q := q + q;
{ a ∈ IN ∧ x > 0 ∧ q = 2^(a-x+1) = 2^(a-(x-1)) }
    x := x - 1;
{ a ∈ IN ∧ x ≥ 0 ∧ q = 2^(a-x) }
end loop;
{(a∈IN ∧ x=0 ∧ q=2^(a-x)) ∨ (a=0 ∧ q=1)}  ⇒ {a∈IN0 ∧ q=2^a}
Put (q);
{ Ausgabe = 2^a }
```


7.2 Die Hoareschen Regeln

Diese Idee der schrittweisen Beweise scheint stets einen Menschen als Beweiser zu erfordern. Große Teile solcher Beweise lassen sich aber automatisieren.

Der englische Informatiker C.A.R.Hoare hat vor über 40 Jahren Regeln entwickelt, wie man aus den Einzelteilen einer Anweisung die Zusicherungen für die gesamte Anweisung erhalten kann. Die Ein- und Ausgabe berücksichtigen wir hierbei nicht; sie müssen zusätzlich hinzugefügt werden. Zugrunde gelegt werden folgende 5 Kontrollstrukturen (skip = nichts tun):

skip, Zuweisungen x := ß, „**;**“, „**if**“ und „**while**“.

Vorbemerkung: Jede Regel $\frac{X}{Y}$ bedeutet:

Wenn X zutrifft, dann trifft auch Y zu.

7.2.1: Die sechs Regeln von Hoare:

Für alle Zusicherungen A, B, C, für alle Wertzuweisungen **x:=ß** und für alle Anweisungen **c**, **d**:

1 $\dfrac{\text{true}}{\{A\}\ \text{skip}\ \{A\}}$, wobei skip die leere Anweisung (null) ist.

2 $\dfrac{\text{true}}{\{A'\}\ \mathbf{x:=ß}\ \{A\}}$, wobei A' aus A entsteht, indem man in A alle freien Vorkommen von x durch ß ersetzt.

(außer: x war zuvor undefiniert; wir können aber stets verlangen, dass jede Variable bei ihrer Deklaration initialisiert wird)

3 $\dfrac{\{A\}\ \mathbf{c}\ \{B\}\ \wedge\ \{B\}\ \mathbf{d}\ \{C\}}{\{A\}\ \mathbf{c;d}\ \{C\}}$

Für alle Zusicherungen A, A", B, B", für alle Booleschen Ausdrücke b und für alle Anweisungen **c**, **d**:

4 $\dfrac{\{A \wedge b\}\ \mathbf{c}\ \{B\}\ \wedge\ \{A \wedge \underline{\text{not}}(b)\}\ \mathbf{d}\ \{B\}}{\{A\}\ \mathbf{if}\ b\ \mathbf{then\ c\ else\ d\ end\ if}\ \{B\}}$

5 $\dfrac{\{A \wedge b\}\ \mathbf{c}\ \{A\}}{\{A\}\ \mathbf{while}\ b\ \mathbf{loop\ c\ end\ loop}\ \{A \wedge \underline{\text{not}}(b)\}}$

Ein solches A heißt Schleifen-invariante.

6 $\dfrac{A \Rightarrow A''\ \wedge\ \{A''\}\ \mathbf{c}\ \{B''\}\ \wedge\ B'' \Rightarrow B}{\{A\}\ \mathbf{c}\ \{B\}}$

Konsequenzregel ("⇒" ist hier die logische Implikation)

Hinweis: Regel 2 klingt auf den ersten Blick merkwürdig:

2 $\dfrac{\text{true}}{\{A'\}\ \mathbf{x:=ß}\ \{A\}}$ wobei A' aus A entsteht, indem man in A alle freien Vorkommen von x durch ß ersetzt.

Man hätte erwartet, dass man A aus A' ableitet und nicht umgekehrt. Betrachte jedoch ein Beispiel:

$\{X \geq -3\}\ \ X := 2*X + 1;\ \ \{\ ?\ \}$

Offensichtlich kann man nicht X in $\{X \geq -3\}$ durch 2*X+1 ersetzen, weil sich dann $\{2*X+1 \geq -3\}$, also $\{X \geq -2\}$ als Nachbedingung ergibt, während im Falle, dass anfangs X = -3 ist, anschließend X = -5 ist, also die Nachbedingung $\{X \geq -2\}$ falsch wäre. Offenbar muss aber $X := 2*X + 1 \geq 2*(-3) + 1 = -5$ sein

Die Einsetzung von X in die Nachbedingung $\{\ X \geq -5\ \}$ ist daher korrekt: $\{\ ?\ \}\ \ X := 2*X + 1;\ \ \{\ X \geq -5\ \}$

liefert die Vorbedingung $\{\ 2*X + 1 \geq -5\ \}$, also $\{\ X \geq -3\ \}$.

7.2.2: Man verifiziere für einige Beispiele, dass diese Regeln korrekt sind:

dort muss man X durch X+1 ersetzen!

$\{X > -1\}\ \ X := X + 1;\ \ \{X > 0\}$ nach Regel 2

$\{X \geq 0\}\ \ X := X+1;\ X:=X+1;\ \ \{X \geq 2\}$ nach Regel 3

$\{X > 0\}$ if X mod 2 = 0
then X := X+1;
else X := X+2; end if; $\{X > 1\}$ nach Regel 4

$\{X \geq 0\}$ while X > 1 loop
X := X-2; end loop; $\{X \geq 0 \wedge X \leq 1\}$ nach Regel 5 mit A = (X ≥ 0)

$\{X+Y=a\}$ if X > Y then X := X-2;
else Y := Y-2; end if; $\{X+Y=a-2\}$ nach Regel 4

7.3 Beispiele

Beispiel 7.3.1: Was realisiert folgendes Programmstück ("Block")?

```
declare x, y, z: Natural;
begin
    Get(x); Get(y); z:=0;
    while y > 0 loop
        if (y mod 2 = 0) then
            y:=y div 2;
            x:=x+x;
        else
            y:=y-1;
            z:=z+x;
        end if;
    end loop;
    Put(z);
end;
```

Füge geeignete gültige Zusicherungen ein! Dann erst weiterlesen.


```
x, y, z: Natural;
{x=⊥ ∧ y=⊥ ∧ z=⊥}
begin Get (x); Get (y);
{a∈ℕ0∧a≥0∧x=a ∧ b∈ℕ0∧b≥0∧y=b ∧ z=⊥}
z:=0;
{a∈ℕ0∧a≥0∧x=a ∧ b∈ℕ0∧b≥0∧y=b ∧ z=0}
while y > 0 loop
    if (y mod 2 = 0) then
        y:=y div 2;
        x:=x+x;
    else
        y:=y-1;
        z:=z+x;
    end if;
end loop;
Put(z); end;
```

Wir betrachten nun nur die while-Schleife.


```
{a∈ℕ0∧a≥0∧x=a ∧ b∈ℕ0∧b≥0∧y=b ∧ z=0} ⇒
    {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
while y > 0 loop
    {x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
    if (y mod 2 = 0) then
    {x≥0 ∧ y>0 ∧ ∃k: y=2k ∧ z≥0 ∧ z+x·y=a·b}
        y := y div 2;
    {x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·2y=a·b}
        x := x + x;
    {x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
    else
        y:=y-1;
        z:=z+x;
    end if;
end loop;
```

then-Zweig betrachtet


```
{a∈ℕ0∧a≥0∧x=a ∧ b∈ℕ0∧b≥0∧y=b ∧ z=0} ⇒
    {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
while y > 0 loop
    {x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
    if (y mod 2 = 0) then
        y := y div 2;
        x := x + x;
    else
    {x≥0 ∧ y>0 ∧ ∃k: y=2k+1 ∧ z≥0 ∧ z+x·y=a·b}
        y:=y-1;
    {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y+x=a·b}
        z:=z+x;
    {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
    end if;
end loop;
```

else-Zweig betrachtet

Es gilt also

$\{x \geq 0 \wedge y > 0 \wedge z \geq 0 \wedge z+x \cdot y = a \cdot b\}$

```
if (y mod 2 = 0) then
        y := y div 2;
        x := x + x;
```
$\{x \geq 0 \wedge y > 0 \wedge z \geq 0 \wedge z+x \cdot y = a \cdot b\}$
```
else
        y:=y-1;
        z:=z+x;
```
$\{x \geq 0 \wedge y \geq 0 \wedge z \geq 0 \wedge z+x \cdot y = a \cdot b\}$
```
end if;
```


Mit der Konsequenzregel folgt hieraus wegen $y > 0 \Rightarrow y \geq 0$:

$\{x \geq 0 \wedge y > 0 \wedge z \geq 0 \wedge z+x \cdot y = a \cdot b\}$

```
if (y mod 2 = 0) then
        y := y div 2;
        x := x + x;
```
$\{x \geq 0 \wedge y \geq 0 \wedge z \geq 0 \wedge z+x \cdot y = a \cdot b\}$
```
else
        y:=y-1;
        z:=z+x;
```
$\{x \geq 0 \wedge y \geq 0 \wedge z \geq 0 \wedge z+x \cdot y = a \cdot b\}$
```
end if;
```


Also gilt nach der vierten Regel:

$\{x \geq 0 \wedge y > 0 \wedge z \geq 0 \wedge z+x \cdot y = a \cdot b\}$

```
if (y mod 2 = 0) then
        y := y div 2;
        x := x + x;
else
        y:=y-1;
        z:=z+x;
end if;
```
$\{x \geq 0 \wedge y \geq 0 \wedge z \geq 0 \wedge z+x \cdot y = a \cdot b\}$

Wir setzen dies ein:

$\{a \in \mathbb{N}_0 \wedge a \geq 0 \wedge x = a \wedge b \in \mathbb{N}_0 \wedge b \geq 0 \wedge y = b \wedge z = 0\} \Rightarrow$
$\{x \geq 0 \wedge y \geq 0 \wedge z \geq 0 \wedge z+x \cdot y = a \cdot b\}$

```
while y > 0 loop
        {x ≥ 0 ∧ y > 0 ∧ z ≥ 0 ∧ z+x·y = a·b}
        if (y mod 2 = 0) then
                y := y div 2;
                x := x + x;
        else
                y:=y-1;
                z:=z+x;
        end if;
        {x ≥ 0 ∧ y ≥ 0 ∧ z ≥ 0 ∧ z+x·y = a·b}
end loop;
```
$\{x \geq 0 \wedge y \geq 0 \wedge z \geq 0 \wedge z+x \cdot y = a \cdot b \wedge y \leq 0\}$
$\Rightarrow \{y = 0 \wedge z = a \cdot b\}$
```
Put(z);
```

```
declare x, y, z: Natural;
begin
    Get(x); Get(y); z:=0;
    {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
    while y > 0 loop
        if (y mod 2 = 0) then
            y:=y div 2;
            x:=x+x;
        else
            y:=y-1;
            z:=z+x;
        end if;
    end loop;
    {y=0 ∧ z=a·b}
    Put(z);
end;
```

Also wird das Produkt zweier natürlicher Zahlen berechnet.

Beispiel 7.3.2: ***Plateau-Problem***

Gegeben: n ≥ 1 und ein sortiertes Feld ganzer Zahlen

a: array (1..n) of Integer;

Gesucht ist die maximale Anzahl gleicher Zahlen, also

Max {d≥1 | ∃i mit a(i) = a(i+1) = ... = a(i+d-1)}.

Dieser Wert heißt auch "maximale Plateaulänge" von a.

Lösungsvorschlag:

" Lies n und das array a ein. Sortiere a. Danach: "

```
    j := 1; p := 1;
    while j /= n loop
        j := j+1;
        if a(j) = a(j-p) then p := p+1; end if;
    end loop;
```

abgekürzt durch **μ**

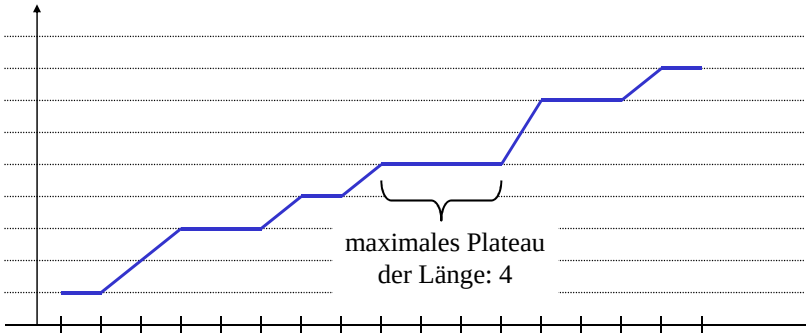
" Ergebnis ist p "

Erläuterung des Namens "Plateau-Problem" am Beispiel:

Stelle das sortierte Feld mit der maximalen Plateaulänge 4

1, 1, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5, 7, 7, 7, 8

grafisch dar:



Behauptung: Dieses Programmstück **μ** berechnet die maximale Plateaulänge des sortierten Feldes a.

Beweis: Zur Abkürzung definieren wir:

MP(j,p) ⇔ p ist die maximale Plateaulänge des Teilfelds a(1..j).

sort(a) ⇔ a ist sortiert

Wenn für alle j gilt: MP(j,p), wobei p der aktuelle Wert der Variablen p nach Durchlauf der Schleife für den Wert j ist, dann gilt MP(n,p) für den letzten Wert n von p, womit die Behauptung bewiesen wäre. Zu Beginn von **μ** gilt:

n≥1 ∧ a ist sortiert.

Hieraus folgt, dass anfangs gilt: MP(1,1) ∧ sort(a).

Nun gehen wir die einzelnen Anweisungen durch:

{MP(1,1) ∧ sort(a)}
j := 1;
{MP(j,1) ∧ sort(a)}
p := 1;
{MP(j,p) ∧ sort(a)}
while j /= n loop
    {MP(j,p) ∧ sort(a) ∧ j≠n}
    j := j+1;
    {MP(j-1,p) ∧ sort(a) (∧ j-1≠n) }
    if a(j) = a(j-p) then p := p+1; end if;
    Behauptung: {MP(j,p) ∧ sort(a)}
end loop;

Regel 2:

$$\frac{\text{true}}{\{A'\}\ \mathbf{x:=ß}\ \{A\}}$$

, wobei A' aus A entsteht, indem man in A alle freien Vorkommen von x durch ß ersetzt.

{MP(j-1,p) ∧ sort(a)}
if a(j) = a(j-p) then
  {MP(j-1,p) ∧ sort(a) ∧ a(j) = a(j-p)}
  ⇒ {MP(j,p+1) ∧ sort(a)} -- es gibt eine um 1 längere Folge
  p := p+1;
  {MP(j,p) ∧ sort(a)} -- Regel 2
else
  {MP(j-1,p) ∧ sort(a) ∧ a(j)≠a(j-p)}
  ⇒ {MP(j,p) ∧ sort(a)} -- es ist keine längere Folge entstanden
  null; -- null in Ada entspricht skip
  {MP(j,p) ∧ sort(a)} -- Regel 1
end if;
{MP(j,p) ∧ sort(a)} -- Regel 4

{MP(1,1) ∧ sort(a)}
j := 1;
{MP(j,1) ∧ sort(a)}
p := 1;
{MP(j,p) ∧ sort(a)}
while j /= n loop
    {MP(j,p) ∧ sort(a) ∧ j≠n}
    j := j+1;
    {MP(j-1,p) ∧ sort(a)}
    if a(j) = a(j-p) then p := p+1; end if;
    {MP(j,p) ∧ sort(a)}
end loop;
{MP(j,p) ∧ sort(a) ∧ j=n} ⇒ **{MP(n,p)}**

Erinnerung:
MP(j,p) ⇔
p ist die maximale Plateaulänge des Teilfelds a(1..j).

was zu beweisen war

μ berechnet also tatsächlich die maximale Plateaulänge, sofern **μ** terminiert.
Offenbar endet das Programm aber stets nach genau n Schleifendurchläufen.

Bei solchen Beweisen muss man stets darauf achten, dass man alle Voraussetzungen auch verwendet.

Frage an Sie: Wo genau wurden im Beweis die beiden Voraussetzungen

" n >= 1 " sowie
" a ist sortiert "

benutzt? Oder hätte man sie weglassen können?

Gehen Sie den Beweis nochmals im Detail durch. Machen Sie sich auch die Terminierung klar.

Beispiel 7.3.3: ***BubbleSort und "falsches Quicksort"***

Gegeben: n ≥ 1 und ein Feld ganzer Zahlen
A: array (1..n) of Integer;
Dieses Feld A soll sortiert werden.

Das klassische Verfahren "Bubble-Sort" lautet (in 4.4.4 haben wir eine Variante hiervon kennen gelernt):

```
for i in reverse 1..n-1 loop
    for j in 1..n-1 loop
        if A(j) > A(j+1) then "Vertausche A(j) und A(j+1)" end if;
    end loop;
end loop;
```

Hinweis:
Statt "for j in 1..n-1 loop" kann man auch "for j in 1..i loop" schreiben, da die letzten n-i Elemente nach jedem Durchlauf genau die n-i größten Elemente in sortierter Reihenfolge sind und daher nicht mehr durchsucht werden müssen. So spart man den Faktor 2.

Wir ersetzen die for- durch while-Schleifen und erhalten:

```
i := n;
while i >= 1 loop
    i := i - 1;
    j := 1;
    while j <= i loop
        if A(j) > A(j+1) then "Vertausche A(j) und A(j+1)" end if;
        j := j + 1;
    end loop;
end loop;
```

Die Verifikation dieses Bubble-Sort-Programm ist einfach. Wir schreiben zur Abkürzung:
sg(k) steht für ["sortiert sein und größer ab Position k"]
$A(k) \le A(k+1) \le ... \le A(n-1) \le A(n) \ \wedge \ \forall\, i < k: A(i) \le A(k)$
wobei stets *sg(n+1)* = true ist.
max(j) steht für $A(j) = \mathrm{Max}\{A(i) \mid 1 \le i \le j\}$,
wobei stets *max(1)* = true ist.
Beachte im Beweis: Aus $sg(i+2) \wedge max(i+1)$ folgt $sg(i+1)$.


```
i := n;                                  -- i, n: Natural, d.h., n ≥ 0 genügt für den Beweis
{ i ≥ 0 ∧ sg(i+1) ∧ max(1) } ⇒ { sg(i+1) ∧ max(1) }
while i >= 1 loop
    { sg(i+1) ∧ i ≥ 1 ∧ max(1) }
    i := i - 1;
    { sg(i+2) ∧ i ≥ 0 ∧ max(1) }
    j := 1;
    { sg(i+2) ∧ j ≤ i+1 ∧ max(j) }
    while j <= i loop
        { sg(i+2) ∧ max(j) ∧ j ≤ i }
        if A(j) > A(j+1) then "Vertausche A(j) und A(j+1)" end if;
        { sg(i+2) ∧ max(j+1) ∧ j ≤ i }
        j := j + 1;
        { sg(i+2) ∧ max(j) ∧ j ≤ i+1 }
    end loop;
    { sg(i+2) ∧ max(j) ∧ j ≤ i+1 ∧ j > i } ⇒      -- also ist hier j = i+1
    { sg(i+1) ∧ max(1)}                  -- s.o. und weil max(1) stets true ist
end loop;                                -- hier muss i = 0 sein, daher n ≥ 0 anfangs
{ sg(i+1) ∧ max(1) ∧ i < 1 } ⇒ { sg(1) }                    -- weil i = 0
```

sg(1) besagt, dass $A(1) \le A(2) \le ... \le A(n-1) \le A(n)$ gilt, d.h., Bubble-Sort erzeugt ein sortiertes Feld A.

Bubblesort benötigt stets $\Theta(n^2)$ Vergleiche; es ist deshalb in der Praxis nur für kleinere Felder geeignet (bis etwa n=50; für größere n gibt es deutlich schnellere Verfahren).

Wir wenden uns nun Quicksort zu, siehe nächste Folie. Um einen Fehler aufspüren zu können, betrachten wir zunächst eine (nicht richtig arbeitende) Version "Aufteilen" von Quicksort.

"Quicksort" (1962 von C.A.R.Hoare publiziert) wählt ein "Pivot-Element" p aus und ordnet das Feld so in zwei Teilfelder A(1..m) und A(m+1..n) um, dass im ersten Teil alle Elemente kleiner oder gleich p und im zweiten Teil alle Elemente größer oder gleich p sind. Das Verfahren ist rekursiv. Es arbeitet im Mittel (average case) besonders schnell, siehe später 10.2.4.

```
...; type Index is 1..n; ...; A: array (Index) of Integer; ...

procedure Aufteilen (L, R: Index) is        -- A ist global, A(L..R) wird sortiert
i, j, m: Index;  p, h: Integer;             -- p wird das Pivot-Element
begin
if L < R then
    i := L;  j := R;  m := (L+R)/2;  p := A(m);
    while i <= j loop                       -- die Indizes i und j laufen aufeinander zu
        while  A(i) <= p  loop  i := i+1;  end loop;
        while  A(j) >= p  loop  j := j-1;  end loop;
        if i <= j then  h := A(i);  A(i) := A(j);  A(j) := h;
                        i := i+1;  j := j-1;
        end if;
    end loop;
    Aufteilen(L, j); Aufteilen(i, R);
end if;
end Aufteilen;

... Aufteilen(1,n); ...                     -- Aufruf des Sortierverfahrens
```

Vorsicht: Diese Prozedur enthält einen Fehler!


```
procedure Aufteilen (L, R: Index) is
i, j, m: Index;  p, h: Integer;
{A(L..R) ist ein Feld ganzer Zahlen}
if L < R then
{A(L..R) ist ein Feld mit mindestens 2 ganzen Zahlen (R-L ≥ 1)}
    i := L; j := R; m := (L+R)/2; p := A(m);
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p}
    while i <= j loop
        ...
    end loop;
    Aufteilen(L, j); Aufteilen(i, R);
end if;
end Aufteilen;
```



```
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p}
    while i <= j loop
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p ∧ i ≤ j}
        while A(i) <= p loop i := i+1; end loop;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ A(i) > p ∧ alle Elemente in A(j+1..R) sind ≥ p}
        while A(j) >= p loop j := j-1; end loop;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ A(i) > p ∧ alle Elemente in A(j+1..R) sind ≥ p ∧ A(j) < p}
        if i <= j then  h:=A(i); A(i):=A(j); A(j):=h;
                        i := i+1; j := j-1;
    end loop;
```



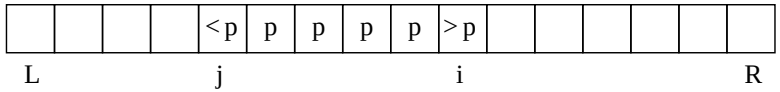
```
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ A(i) > p ∧ alle Elemente in A(j+1..R) sind ≥ p ∧ A(j) < p}
        if i <= j then  h:=A(i); A(i):=A(j); A(j):=h;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i) sind ≤ p ∧ alle Elemente in A(j..R) sind ≥ p ∧ i ≤ j}
                        i := i+1; j := j-1;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p}
        end if;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p}
    end loop;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p ∧ i > j}
```

⇒ (Wunsch!) {R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..j) sind ≤ p ∧ alle Elemente in A(i..R) sind ≥ p ∧ i > j}

Diese letzte Folgerung ist gesondert zu beweisen. Hierfür berechnen wir, welche Werte die Variablen i und j am Ende der äußeren while-Schleife besitzen können.

Fall 1: Unmittelbar vor dem Ende der while-Schleife traf die if-Bedingung zu. Falls hierbei i = j war, so muss A(i) > p und A(j) < p gegolten haben. Dies ist aber unmöglich.
Folglich müssen i < j und A(i) > p und A(j) < p gewesen sein. Da anschließend die while-Schleife abbricht, muss "neues i" = i+1 > j-1 = "neues j" sein; zusammen mit i < j folgt hieraus j - i = 1, d.h., *nach* Abarbeiten des then-Zweigs muss i - j = 1 gelten, wobei A(i) > p und A(j) < p ist. Folglich sind alle Element in A(L..j) kleiner oder gleich p und alle Elemente in A(i..R) größer oder gleich p.

Fall 2: Unmittelbar vor dem Ende der while-Schleife traf die if-Bedingung nicht zu. Dann muss die Abbruchbedingung i > j der while-Schleife durch die inneren while-Schleifen erreicht worden sein. Es muss A(i) > p und A(j) < p gelten, weil dies die Abbruchbedingungen der inneren while-Schleifen sind. Dann kann wiederum nicht i = j gelten. Da aber A(i-1) <= p gewesen sein muss (sonst wäre die innere while-Schleife bereits mit i-1 terminiert), müssen alle Elemente in A(j+1..i-1) gleich p sein:



Es folgt: Alle Elemente in A(L..j) sind ≤ p ∧ alle Elemente in A(i..R) sind ≥ p ∧ alle Elemente in A(j+1..i-1) sind = p ∧ i > j .

Fazit: Wenn wir nun die Rekursion aufrufen, so genügt es, dies für den Bereich von L bis j und von i bis R zu tun. Unter der Annahme, dass durch

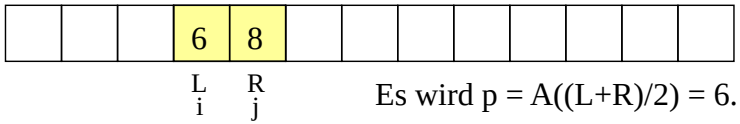
Aufteilen(L, j); Aufteilen(i, R);

die Bereiche A(L..j) und A(i..R) sortiert werden, und zusammen mit der Tatsache, dass A(j+1..i-1) nur aus Werten gleich p bestehen kann (sofern dieser Bereich nicht leer ist), folgt, dass die Prozedur "Aufteilen" den Bereich von L bis R korrekt sortiert.

Sie terminiert aber nur, sofern i und j den Bereich L..R nicht verlassen und der rekursive Aufruf stets mit kleineren Bereichen erfolgt!

Und genau dieser Fehlerfall tritt bei obigem Programm auf !

Irgendwann stößt die Rekursion auf einen kleinen, z.B. einen zweielementigen Teilbereich (L+1 = R):



Es wird p = A((L+R)/2) = 6.

In der ersten inneren while-Schleife läuft i nach rechts, bis i = R ist, wegen A(R) = 8 > p. Danach läuft der Index j absteigend von R über die linke Grenze L hinaus, weil jedes Mal A(j) ≥ p ist. *Fehler!*
Diesen Fehler kann man beseitigen, indem man entweder zusätzliche Abfragen bzgl. der Bereichsgrenzen einfügt oder indem man in den inneren while-Schleifen auch bei Gleichheit mit p die Vertauschung durchführt; das heißt, die Bedingungen müssen dann lauten

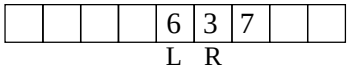
```
while A(i) < p loop i := i+1; end loop;      -- statt A(i) <= p
while A(j) > p loop j := j-1; end loop;      -- statt A(i) >= p
```

Wegen der jetzt geltenden Zusicherung
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R)
∧ alle Elemente in A(L..i-1) sind ≤ p ∧ **A(i) ≥ p**
∧ alle Elemente in A(j+1..R) sind ≥ p ∧ **A(j) ≤ p**}
können die Indizes i und j nun nicht mehr übereinander hinweglaufen. Vielmehr kann der zweite Index j höchstens um eine Stelle nach links über den Index i gelangen, d.h., beide innere while-Schleifen enden stets mit Werten i und j, für die gilt

$$i - j \leq 1.$$

Da p ein Element von A(L..R) ist, müssen beide Indizes spätestens stehen bleiben, wenn sie auf p treffen, d.h., die if-Bedingung ist mindestens einmal erfüllt und damit gilt anschließend i > L und j < R. Die Rekursion erfolgt also stets mit kleineren Grenzen. Dies sichert die Terminierung.

Hinweis: Es kann bei der Verwendung von A(i) <= p (statt A(i) < p) und von A(i) >= p statt (A(i) > p) auch der Fall einer unendlichen Rekursion auftreten. Zum Beispiel kann im Laufe der Rekursion die folgende Situation entstehen:



Dann werden p = 6, i = L und j = R gesetzt. i wird erhöht, bis i = R+1 ist, j bleibt auf R stehen, weil nicht 3 >= 6 gilt. Nun ist aber i > j und die Werte von i und j werden nicht mehr verändert. Anschließend wird "Quicksort(L, j);", also "Quicksort(L, R);" *mit den gleichen Grenzen* erneut aufgerufen, d.h., es liegt ein unendlicher rekursiver Aufruf vor.

Wir formulieren auf der nächsten Folie das korrekte Verfahren Quicksort und führen auf den anschließenden Folien nochmals den Nachweis der Korrektheit. Vergleichen Sie diese Version mit der früher bereits vorgestellten Fassung.

Hinweis: Die Laufzeit des Verfahrens beträgt im Mittel $O(n \cdot \log(n))$ und im worst case $O(n^2)$. Das Pivotelement kann man beliebig wählen. Dieser Nichtdeterminismus führt aber stets zum gleichen Ergebnis, nämlich zu der sortierten Folge.

7.3.4: Das richtige Quicksort

...; type Index is 1..n; ...; A: array (Index) of Integer; ...

```
procedure Quicksort(L, R: Index) is    -- A ist global, A(L..R) wird sortiert
i, j: Index; p, h: Integer;            -- p wird das Pivot-Element
begin
    if L < R then
        i := L; j := R; "wähle p beliebig aus A(L..R)";
        while i <= j loop              -- die Indizes i und j laufen aufeinander zu
            while A(i) < p loop i := i+1; end loop;
            while A(j) > p loop j := j-1; end loop;
            if i <= j then  h:=A(i); A(i):=A(j); A(j):=h;
                            i := i+1; j := j-1;  end if;
        end loop;      -- auch bei Gleichheit A(i) = p oder A(j) = p vertauschen!
    Quicksort(L, j); Quicksort(i, R);
    end if;
end Quicksort;
```

... Quicksort(1,n); ... *-- Aufruf des Sortierverfahrens*

Korrektheit dieses Verfahrens:

procedure Quicksort (L, R: Index) is
i, j, m: Index; p, h: Integer;
{A(L..R) ist ein Feld ganzer Zahlen}
if L < R then
{A(L..R) ist ein Feld mit mindestens 2 ganzen Zahlen (R-L ≥ 1)}
    i := L; j := R; m := (L+R)/2; p := A(m);
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p}
    while i <= j loop

    end loop;
    Quicksort(L, j); Quicksort(i, R);
end if;
end Quicksort;

{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p}
    while i <= j loop
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p ∧ i < j}
        while A(i) < p loop i := i+1; end loop;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ **A(i) ≥ p** ∧ alle Elemente in A(j+1..R) sind ≥ p}
        while A(j) > p loop j := j-1; end loop;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ **A(i) ≥ p** ∧ alle Elemente in A(j+1..R) sind ≥ p ∧ **A(j) ≤ p**}
        if i <= j then h:=A(i); A(i):=A(j); A(j):=h;
                i := i+1; j := j-1; end if;
    end loop;

{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ A(i) ≥ p ∧ alle Elemente in A(j+1..R) sind ≥ p ∧ A(j) ≤ p}
        if i <= j then h:=A(i); A(i):=A(j); A(j):=h;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i) sind ≤ p ∧ alle Elemente in A(j..R) sind ≥ p}
            i := i+1; j := j-1;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p}
        end if;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p}
    end loop;
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind ≤ p ∧ alle Elemente in A(j+1..R) sind ≥ p ∧ i > j} ⇒
{R-L ≥ 1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..j) sind ≤ p ∧ alle Elemente in A(i..R) sind ≥ p ∧ i > j}

Die weiteren Folgerungen können von früher übernommen werden.

Man beachte, dass Quicksort terminiert, da die if-Bedingung mindestens einmal erfüllt sein muss, die Prozedur also den then-Zweig ausführt und daher i um mindestens 1 erhöht und j um mindestens 1 erniedrigt wird.

Hinweis zum Speicherplatzbedarf: Quicksort benötigt Platz für die Rekursion. Wenn man die Rekursion jeweils mit dem größeren Bereich zuerst durchführt, so kann der zusätzliche Platz bis zu 2n Speicherplätze betragen, um jedes Mal das Paar (i,R) abzulegen. Dies ließe sich auf 2·log(n) verringern, falls man immer zuerst den kürzeren Bereich nimmt, d.h., falls man die beiden rekursiven Aufrufe ersetzt durch:

if (j-L) < (R-i) then Quicksort(L, j); Quicksort(i, R);
else Quicksort(i, R); Quicksort(L, j); end if;

Dummerweise nützt uns diese Formulierung jedoch nichts, weil innerhalb der Prozedur beim erneuten Aufruf von

Quicksort(L, j); Quicksort(i, R);

die Rekursion ja nicht beendet wird und daher immer noch bei

Quicksort(L, j); bzw. bei Quicksort(i, R);

die vollständige Aufruf-Kette erhalten bleibt!

Man muss also Quicksort so abwandeln, dass der Aufruf

Quicksort(L, R);

beendet wird, **bevor** der Aufruf

Quicksort(L, j); (im then-Zweig) bzw.
Quicksort(i, R); (im else-Zweig)

erfolgt; zugleich müssen aber die Indexgrenzen i und R für den später folgenden Aufruf "Quicksort(i, R);" (bzw. L und j für "Quicksort(L, j);") aufbewahrt werden! Es ist klar, dass sich dies realisieren lässt, aber wie? Selbst durchdenken!

7.4 Schwächste Vorbedingung

7.4.1: Vorgehensweise zum Nachweis der Korrektheit:
Schreibe zwischen je zwei Anweisungen eine prädikatenlogische Formel und versuche mit Hilfe der Hoareschen Regeln zu beweisen, dass jeweils {A} c {B} gilt.
Da die Hoareschen Regeln sich nur auf die Anweisungen "leere Anweisung", "Wertzuweisung", "Hintereinanderausführung von Anweisungen", "Alternative" und "while-Schleife" beziehen, können wir dieses Schema bisher auch nur auf Programme anwenden, die höchstens aus diesen Bestandteilen bestehen.
Will man beliebige Ada-Programme untersuchen, so muss man weitere Regeln einführen.
(Solche weiteren Regeln kann man aufstellen. Wir tun dies nicht, weil es hier um das Prinzip geht und weil weitere Regeln rasch recht kompliziert werden können.)

Kann man dieses Vorgehen automatisieren, d.h., kann man ein Programm schreiben, das
- ein Programm einliest,
- die Spezifikation einliest,
- schrittweise die erforderlichen Zusicherungen ermittelt und
- den Beweis der Korrektheit führt?

Nein, denn das Problem, die Korrektheit eines Programms zu beweisen, ist algorithmisch nicht lösbar. Es lässt sich auf das Halteproblem zurückführen, vgl. Abschnitt 2.3. Formale Beweise hierzu werden in der Theoretischen Informatik geliefert (z.B. der Satz von Rice).

Aber: Man kann natürlich ein interaktives Programm, also ein "Unterstützungssystem" bauen, welches
- ein Programm einliest,
- die Spezifikation einliest,
- diese Spezifikation als letzte Zusicherung verwendet und versucht, die vor der letzten Anweisung einzufügende Zusicherung zu konstruieren oder den Benutzer aufzufordern, einen Vorschlag einzugeben,
- einen Beweis für die Korrektheit dieses einen Schrittes zu führen oder den Benutzer aufzufordern, einen solchen Beweis einzugeben und diesen nachzuvollziehen,
- das Gleiche für die Anweisung davor zu wiederholen usw.

Auf diese Weise kann für viele Programme ein Beweis der Korrektheit ermöglicht und teilweise sogar automatisiert werden. Entsprechende "Programmbeweiser" sind verfügbar.

Wie muss ein Programmbeweiser vorgehen?

Ausgehend von einer Zusicherung und der davor stehenden Anweisung muss man versuchen die Zusicherung zu konstruieren, die vor der letzten Anweisung einzufügen ist.

Idee: Finde zu der Situation

c {B}

eine (möglichst schwache) Zusicherung A, so dass

{A} **c** {B}

gilt.

Beispiel: Zu { ? } x := x+1 {x > 8}
ist offensichtlich { x > 7 } x := x+1 {x > 8}
solch eine schwächste Zusicherung. Korrekt wäre auch

{ x > 12 } x := x+1 {x > 8}

aber x > 12 ist eine "stärkere Aussage" als x > 7.
("Stärker" bedeutet, dass sie auf weniger ganze Zahlen zutrifft.
Formal ist A eine stärkere Aussage als B, wenn $A \Rightarrow B$ gilt.)

7.4.2 Bezeichnungen: Wir müssen nun in der Formel

{A} **c** {B}

zwischen den Zusicherungen A und B unterscheiden. Statt "Zusicherung" sagt man oft auch "Bedingung", und daher
nennt man A die Vorbedingung zur Anweisung **c** (englisch: *precondition*)
und B die Nachbedingung zur Anweisung **c** (englisch: *postcondition*).

Unsere Aufgabe lautet also:
Suche zu der Situation **c** {B}
eine Vorbedingung A so, dass {A} **c** {B}
gilt. Gibt es mehrere solche Vorbedingungen, dann suche die "schwächste" Vorbedingung, d.h., suche ein A mit:
1. {A} **c** {B}
2. wenn für irgendeine Vorbedingung A' gilt: {A'} **c** {B}, dann ist A eine Folgerung aus A', d.h., dann gilt $A' \Rightarrow A$.

Definition 7.4.3:
Eine solche Zusicherung A heißt "schwächste Vorbedingung" zu **c** und B (englisch: *weakest precondition*, abgekürzt wp).

Man schreibt: **A = wp(c,B)**.

Als erstes müssen wir fragen, ob diese Definition in sich stimmig ist (d.h., ob sie "wohldefiniert" ist). Eventuell gibt es zu **c** und B mehrere oder gar keine schwächste Vorbedingung!

Ohne Beweis notieren wir hier:
Es gibt stets eine schwächste Vorbedingung zu **c** und B.

Die schwächste Vorbedingung ist aus Sicht der Logik stets eindeutig. Gäbe es nämlich zwei schwächste Vorbedingungen A und A', dann gelten $A' \Rightarrow A$ und $A \Rightarrow A'$ gleichzeitig. Zwei solche Zusicherungen sind aber logisch gleichwertig, d.h., es gilt dann: $A' \Leftrightarrow A$.

Beispiel 7.4.4:

c ≡ if (X mod 2 = 1) then X:=X+3; end if;
B ≡ X mod 6 = 0
Gesucht ist wp(c,B).

Betrachte hierzu alle Belegungen a der Variablen X, für die nach Durchführung der Anweisung **c** die Zusicherung B gilt:
{a ∈ Z | falls a ungerade ist, dann muss a+3 durch 6 teilbar sein;
falls a gerade ist, dann muss a durch 6 teilbar sein}
= {a ∈ Z | a ist durch 3 teilbar}.

Eine Zusicherung, die genau für die Werte dieser Menge erfüllt ist, muss zur schwächsten Vorbedingung gehören. Also gilt:

wp(c,B) ≡ X mod 3 = 0

Beispiel 7.4.5: Wenn die Aktion **c** eine Wertzuweisung, eine Hintereinanderausführung oder eine Alternative ist, dann lässt sich wp(**c**,B) oft leicht aus den Bestandteilen berechnen und dies kann auch automatisiert werden. Zum Beispiel (selbst nachvollziehen!):

(1) Es sei **c** = if X<0 then X:=-X+1; else X:=X-1; end if
Dann gilt: wp(**c**, X > 5) = (X < -4) ∨ (X > 6).

(2) Es sei **c'** = X := (X*X) mod 8;
Dann gilt: wp(**c'**, {X = 1}) = {∃Y: X = 2*Y+1}, d.h., die Variable X muss eine ungerade Zahl als Inhalt haben. Um dies zu zeigen, braucht man allerdings Zusatzwissen, und zwar die Tatsache, dass das Quadrat jeder ungeraden Zahl gleich 1 modulo 8 ist.

7.4.6: Wesentlich schwieriger ist die Behandlung der while-Schleife. Als Beispiel betrachte man:

c ≡ while (X /= 0) loop X := X-2; end loop;
B ≡ X mod 3 = 0
Wie lautet nun wp(**c**, B)?

Man benötigt eine Schleifeninvariante. "X mod 3 = 0" ist aber keine Schleifeninvariante, da die Formel
{(X mod 3 = 0) ∧ (X /= 0)} X := X-2; {X mod 3 = 0}
nicht erfüllt ist (vgl. Hoaresche Regel 5).

Dagegen sind sowohl "X mod 2 = 0" als auch "X mod 2 = 1" Schleifeninvarianten.

Diese Information nützt uns aber nichts. Betrachten wir stattdessen noch einmal die 5. Hoaresche Regel:

$$\frac{\{A \wedge b\}\ \mathbf{c}\ \{A\}}{\{A\}\ \text{while } b \text{ loop } \mathbf{c} \text{ end loop } \{A \wedge \text{not}(b)\}}$$

In unserem Fall ist not(b) ≡ (X = 0), so dass nach Verlassen der while-Schleife B ≡ X mod 3 = 0 stets erfüllt ist, egal mit welchem Wert von X die Schleife begonnen wurde. Das sieht zunächst unsinnig aus; man beachte jedoch, dass die Frage, ob die Schleife terminiert, nach der Definition von {A} **c** {B} keine Rolle spielt: Denn es soll B nach der Ausführung von **c** erfüllt sein; wenn jedoch die Schleife nicht endet, "dann ist danach alles erfüllt".

Somit erhalten wir als die schwächste Vorbedingung obiger Schleife die Zusicherung: X ∈ **Z** bzw. wp(**c**,B) = true, d.h., die Nachbedingung ist für jede ganze einzulesende Zahl erfüllt.

Im Allgemeinen lässt sich die schwächste Vorbedingung für eine Schleife nicht algorithmisch berechnen. Wer jedoch ein Programm entwickelt, kennt mindestens eine Schleifeninvariante, nämlich die, die er bei der Formulierung der Schleife im Sinn hatte. Mit einem interaktiven System kann diese Zusicherung eingegeben werden und das System kann versuchen zu beweisen, dass sie tatsächlich eine Schleifeninvariante ist.

Hinweis: Statt der schwächsten Vorbedingung kann man auch die "stärkste Nachbedingung" einführen und ermitteln.

Wie soll man nun vorgehen, wenn man beweisen will, dass ein Programm genau das berechnet, was es berechnen soll?

7.4.7: Vorgehen beim Beweisen der Korrektheit

Gegeben sei eine Spezifikation, in der Regel ist dies eine Formel, aus der hervorgeht, welche Ausgaben das Programm für welche Eingaben zu liefern hat.

Diese Formel F schreiben wir als Zusicherung {F} unmittelbar vor die zugehörige Ausgabeanweisung Put(...) im Programm. Vor der Ausgabeanweisung mögen Anweisungen c_k, c_{k-1}, ..., c_2, c_1 stehen. Dann liegt also folgende Situation vor:

... c_k; c_{k-1}; ...; c_2; c_1; {F} Put(...); ...

Berechne nun schrittweise $wp(c_1,\{F\}) = F_1$, $wp(c_2,F_1) = F_2$, $wp(c_3,F_2) = F_3$, ..., so erhält man:

... $\{F_k\}$ c_k; $\{F_{k-1}\}$ c_{k-1}; $\{F_{k-2}\}$...; $\{F_2\}$ c_2; $\{F_1\}$ c_1; {F} Put(...); ...

Das Programm wird hierdurch mit Zusicherungen gefüllt.

Auf diese Weise entsteht ein kompletter Beweis, dass aus den ersten Zusicherungen im Programm die korrekte Ausgabe gemäß der Formel F folgt.

Man prüfe nun nach, ob die am Anfang des Programms stehenden Zusicherungen genau der Eingabe entsprechen. Ist dies der Fall, so hat man bewiesen, dass das Programm korrekt arbeitet.

Die Terminierung muss man noch gesondert nachweisen.

Hier brechen wir die Erläuterung der Korrektheit mit Hilfe der axiomatischen Semantik ab. Es wurden die Idee vorgestellt und ein mögliches Unterstützungssystem angedeutet. Überprüfen Sie beim Schreiben eines Programms stets, welche Zusicherungen zwischen zwei Aktionen erfüllt sein müssen und verbessern Sie hierdurch Ihre Fähigkeit, Programme zu erstellen, die "auf Anhieb korrekt" sind. Schreiben Sie diese Zusicherungen in Ihr Ada-Programm mittels "pragma Assert (<Formel>,<Fehlertext>)" hinein!
Vertiefungen siehe Vorlesungen über Formale Semantik, Sichere Systeme, Interaktive Systeme, Wissensverarbeitung, Programmiersprachen/Übersetzer.

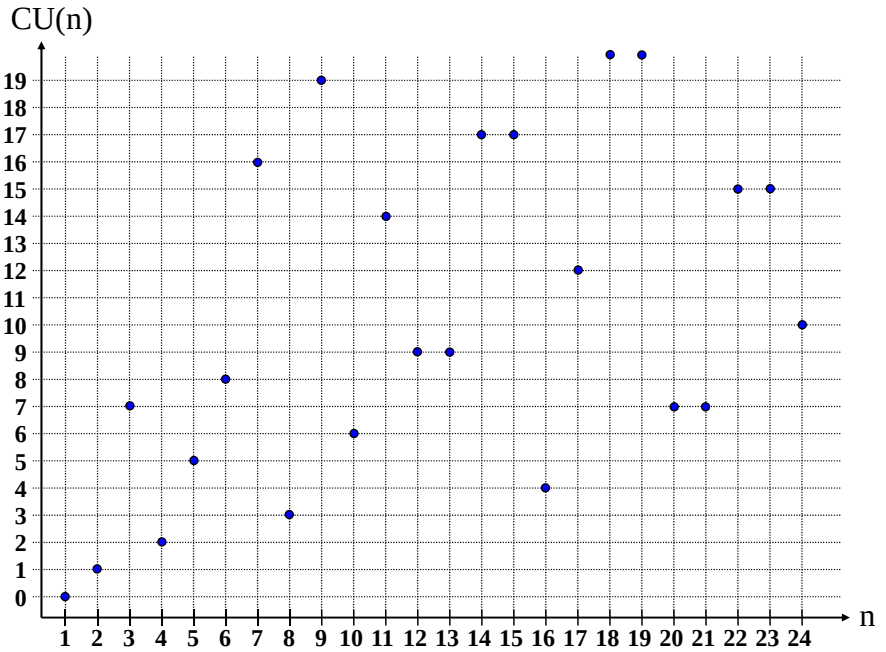
7.5 Terminierung

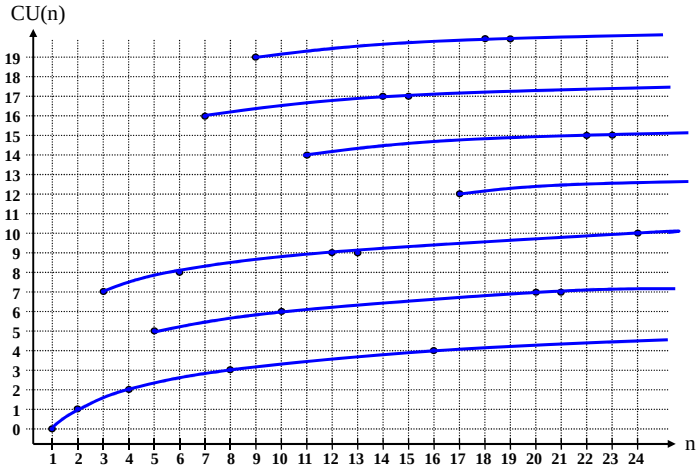
Die Terminierung und die Korrektheit eines Programms müssen stets getrennt nachgewiesen werden.

7.5.1 Standardbeispiel: Collatz- oder Ulam-Funktion:

```
function CU(x: Positive) return Natural is
begin
    if x=1 then return 0;
    elsif x mod 2 = 0 then return CU(x/2) + 1;
    else return CU(3*x+1) + 1;
    end if;
end CU;
```

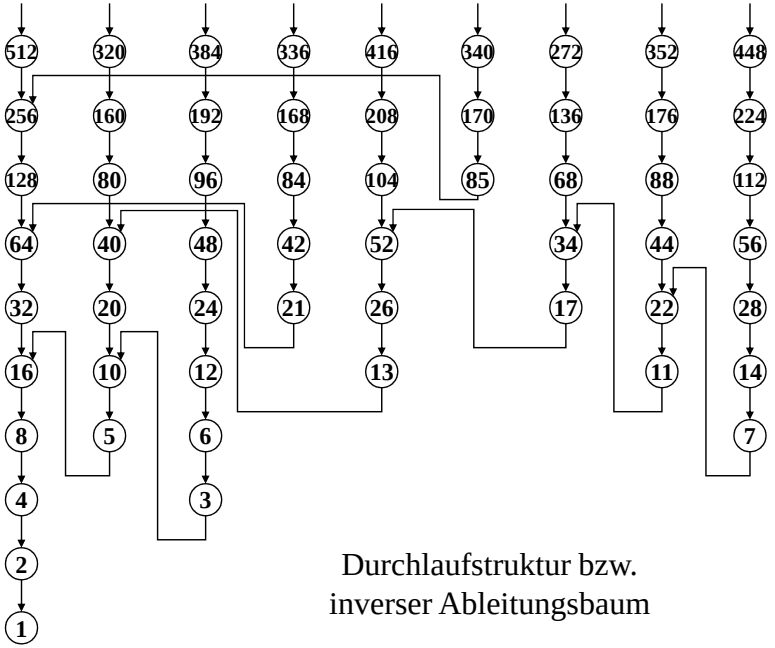
Von dieser Funktion CU ist bis heute nicht bekannt, ob sie total ist, d.h., ob der obige rekursive Algorithmus für jede natürliche Zahl terminiert.
Veranschaulichung auf der nächsten Folie.





Die Werte liegen auf logarithmischen Kurven, die jeweils mit einer ungeraden Zahl beginnen.

Die Funktion CU wird meist nach Stanislaw Ulam, 1909-1984, Prof. an der Univ. Florida, benannt, der sich in den 1950er Jahren mit ihr beschäftigte. Aber bereits in den 1930er Jahren stellte der Hamburger Mathematikprofessor Lothar Collatz diese Funktion vor, weshalb sie zugleich Collatz-Funktion heißt, manchmal auch bezeichnet als Hasse's Problem oder Thwaite's Problem oder als "the syracuse algorithm" (nach der Syracuse University, wo sie genauer untersucht wurde).



Durchlaufstruktur bzw.
inverser Ableitungsbaum

7.5.2: Die Terminierung ist bei rekursiven Definitionen nicht leicht nachzuweisen. Relativ einfach geht dies allerdings bei folgender doppelt-rekursiv dargestellten Funktion, der sog. McCarthy 91-Function:

```
function MC91 (x: Natural) return Natural is
begin if x > 100 then return x-10;
      else return MC91(MC91(x+11)); end if;
end;
```

Übung: Beweisen Sie, dass die hiervon realisierte Funktion f lautet:
f(x) = x-10, für x >100, und f(x) = 91 sonst.

Erst dann zur nächsten Folie gehen!

Beweis (direkt):

Für x > 100 ist MC91(x) = x-10 nach Definition der Funktion.

Für 90 ≤ x ≤ 100 gilt (mit y = x+11 > 100):
MC91(x) = MC91(MC91(x+11)) = MC91(MC91(y))
= MC91(y-10) = MC91(x+1)
= ... = MC91(101) = 91.

Für x < 90 folgt nun unter der Annahme, dass MC91(z) = 91 für alle x < z ≤ 100 bereits bewiesen wurde:
MC91(x) = MC91(MC91(x+11)) = MC91(91) = 91.

7.5.3. Einige Übungsaufgaben zur Rekursion

Übertragen Sie die Funktionsprozeduren in Ada-Programme und stellen Sie eine Tabelle der ersten Werte auf. Versuchen Sie danach, eine Formel für die Funktionen zu finden und beweisen Sie Ihre Vermutung. (Lösungen am Ende von 7.6.)

Überlegen Sie sich für Ihre Eingaben vorher die Laufzeit und die Größenordnung der Ergebnisse!

```
function FA(x: Natural) return Natural is
s: Natural := 0;
begin
  if x <= 1 then return 1;
  else  for k in 1..(x-1) loop s := s + FA(k); end loop;
        return s; end if;
end FA;
```



```
function FB(x: Natural) return Natural is
s: Natural := 0;
begin
  if x <= 1 then return 1;
  else  for k in 1..(x-1) loop s := s + k*FB(k); end loop;
        return s; end if;
end FB;
```

```
function FC(x: Natural) return Natural is
begin
  if x <= 1 then return x;
  else  return FC(x-1) + FC(x-2); end if;
end FC;
```



```
function FD(x: Natural) return Natural is
s: Natural := 0;
begin
  if x <= 1 then return 1;
  else  for k in 1..(x-1) loop s := s + (x-k)*FD(k); end loop;
        return s; end if;
end FD;
```

```
function FE(x: Natural) return Integer is
d, s: Integer := 0;
begin
   for k in 1..x loop s := s + k*k*k; d := d + k; end loop;
   return (s - d*d);
end FE;
```



```
function FF(x: Natural) return Natural is
begin
  if x = 0 then return 0;
  else  return (FF(x-1) + x-1) mod x; end if;
end FF;
```

```
function FG(x: Natural) return Natural is
s: Natural := 0;
begin
  if x = 0 then return 1;
  else  for k in 1..x loop s := s + FG(k-1)*FG(x-k); end loop;
        return s; end if;
end FG;
```

7.6 Historische Anmerkung

Das Problem, die Richtigkeit von Programmen nachzuweisen, beginnt mit der praktisch verwendbaren Programmierung, also seit den 1950er Jahren. Dies führte rasch auf die Frage nach der Bedeutung ("Semantik") eines Programms und dessen präziser Formulierung in Kalkülen.
Die ersten Arbeiten hierzu stammen von R. Floyd (Einführung von festen Stellen im Programm, an denen die Bedeutung ermittelt wird), C.A.R. Hoare (Aufstellung eines Regelsystems) und E. W. Dijkstra (Einführung der weakest precondition) in den 1960er Jahren. Ab 1970 entwickelt sich eine Fülle von Arbeiten zu diesem Gebiet (denotationale Semantik, Semantik nebenläufiger Systeme, Entwicklung von Kalkülen und Beweissystemen, Semantik nebenläufiger Prozesse, konkrete Methoden wie model checking usw.).

Die Terminierung ist ein eher mathematisches Problem. Seine Unentscheidbarkeit lässt sich zum einen aus dem Gödelschen Unvollständigkeitssatz (1931) ableiten, zum anderen war sie Gegenstand der ersten Arbeiten über berechenbare Funktionen (Church, Kleene, Turing, 1936; später: Post, Markov).

Die terminierenden Eingaben eines Programms sind aufzählbar, im Allgemeinen aber nicht die Menge der Eingaben, für die ein Programm divergiert. Zur Terminierung gibt es viele äquivalente Probleme: Busy Beaver, Postsches Korrespondenzproblem, Maximalität kontextfreier Grammatiken, Wortproblem für Halbgruppen, Game of Life (Conway) usw. Man spricht auch vom "Arbeitsplatzerhaltungstheorem" für Informatiker(innen): Im Prinzip muss man für jedes Programm und jede Eingabe einen eigenen Beweis führen, ob es mit dieser Eingabe anhält oder nicht - und hierfür braucht man eine Informatikausbildung.

Lösungen zu den Funktionen in Abschnitt 7.5.3:

FA(n) = 2^{n-2} für n > 1

FB(n) = n!/2 für n > 1

FC(n) = n-te Fibonaccizahl Fib(n) (n≥0, also 0, 1, 1, 2, 3, 5, 8, 13, 21, ...)

FD(n) = Fib(2n-3) für n > 1

FE(n) = 0, weil $(1^3 + ... + k^3 + ... + n^3) = (1 + ... + k + ... + n)^2$.

FF(n) = $2^{k+1} - n - 1$ für das k mit $2^k \le n < 2^{k+1}$ (für n > 0)

FG(n) = n-te Catalansche Zahl, für n ≥ 0

(FG(n) wächst nicht ganz so stark wie 4^n. Zu FG und FC siehe später unter "binäre Bäume" und "AVL-Bäume" in den Abschnitten 8.2.7 und 8.4.7).

8. Suchen

Ziel dieses 8. Kapitels:

Die meisten Probleme werden mit Mengen beschrieben und die Lösungsalgorithmen arbeiten auf Mengen. Dieses Kapitel stellt Ihnen einige Datenstrukturen (Felder, Bitvektoren, Suchbäume, optimale Suchbäume, AVL-, B- und digitale Bäume, Tries) vor, um Mengen darzustellen.

Am Ende sollen Sie gelernt haben, welche dieser Strukturen welche Eigenschaften besitzen. So sind (binäre) Suchbäume im Mittel gut zum Speichern und Wiederfinden von Elementen geeignet, will man jedoch eine logarithmische Such-, Einfüge- und Löschzeiten garantieren, so muss man spezielle Bäume verwenden (z.B. die höhenbalancierten Bäume). Deren Vor- und Nachteile lernen Sie soweit kennen, dass Sie in konkreten Anwendungen entscheiden können, welche Datenstruktur die günstigste ist.

Vorbemerkungen: Grundaufgabe des Suchens

Gegeben: Menge $A = \{a_1, ..., a_n\} \subseteq B$ sowie ein Element $b \in B$.

Realisiere A in einer geeigneten Datenstruktur, so dass die folgenden drei Operationen "effizient" durchgeführt werden:

- Entscheide, ob b in A liegt (und gib ggf. an, wo). **FIND**
- Füge b in A ein. **INSERT**
- Entferne b aus A. **DELETE**

Statt des meist sehr umfangreichen Elements b betrachten wir nur eine Komponente s von b, durch die b eindeutig bestimmt ist. Dieses s nennen wir "Suchelement" oder meistens "**Schlüssel**" (englisch: "**key**").

Hinweis: Weitere Operationen sind denkbar, zum Beispiel:

Weitere Aufgaben: Wähle eine Datenstruktur so, dass alle oder einige der folgenden Tätigkeiten für zwei Mengen $A_1, A_2 \subseteq B$ effizient durchführbar sind.

- Vereinige A_1 und A_2. UNION
- Schneide A_1 und A_2. INTERSECTION
- Bilde das Komplement $B \setminus A_1$. Complement
- Entscheide, ob A_1 leer ist. EMPTINESS
- Entscheide, ob $A_1 = A_2$ ist. EQUALITY
- Entscheide, ob $A_1 \subseteq A_2$ ist. SUBSET
- Entscheide, ob $A_1 \cap A_2$ leer ist. Empty Intersection

Wir beschränken uns in diesem Kapitel aber nur auf die drei Operationen FIND, INSERT und DELETE.

Prinzipielle Überlegung:

Die Mengen können eine Struktur haben oder nicht (meist sind sie dann nicht geordnet).

In geordneten Mengen kann man viel schneller suchen als in ungeordneten.

Wir könnten aber im Folgenden stets annehmen, dass die zugrunde liegenden Mengen angeordnet sind.

Grund: Alle Elemente werden im Rechner durch eine Folge von Nullen und Einsen dargestellt. Dies impliziert eine Anordnung wie auf binär dargestellten Zahlen, die wir stets ausnutzen könnten.

8.1 Suchen in "flachen" Strukturen

Flache oder sequentielle Strukturen sind üblicherweise
- (eindimensionale) Felder,
- Listen,
- Bitvektoren,

wobei egal ist, ob die Felder linear oder zyklisch sind und ob die Listen zusätzlich einfach oder doppelt verkettet werden.

Eindimensionale **Felder** und **Listen** werden von vorne nach hinten oder von hinten nach vorne durchsucht. Im Falle zyklischer Strukturen muss man sich mit einem Index oder einem Zeiger merken, ab wo die Suche begonnen wurde.

Struktur 1: Das array. *Durchsuchen eines Feldes:*

```
s: element; i: Integer; A: array (1..n) of element;
....; i:=1;
while i <= n and then A(i) /= s loop i := i+1; end loop;
if i <= n then < das gesuchte Element s steht an der Position i >
else  < s ist nicht im Feld A vorhanden > end if; ...
```

Mit einem Stopp-Element kann man eine Abfrage sparen:

```
s: element; i, n: Positive; A: array (1..n+1) of element;
....; i:=1; A(n+1) := s;        -- Stopp-Element setzen!
while A(i) /= s loop i := i+1; end loop;
if i <= n then < das gesuchte Element s steht an der Position i >
else  < s ist nicht im Feld A vorhanden > end if; ...
```


Struktur 2: Die Liste. *Durchsuchen einer linearen Liste:*

Suche s in einer Liste, auf die die Variable Anker zeigt:

```
p := Anker;
while p /= null and then p.inhalt /= s loop
                        p := p.next;  end loop;
if p = null then < s ist nicht in der Liste enthalten >
else < p verweist auf das erste Element mit Inhalt s > end if;
```

Aufwand:

Zeit: Die lineare Suche erfordert O(n) Vergleichsschritte, dauert also relativ lange. Ist das Feld geordnet, dann benötigt die binäre Suche maximal O(log(n)) Vergleiche.

Platz: Konstant viele Speicherplätze sind zusätzlich erforderlich.

Wie sieht es mit den Operationen der Grundaufgabe aus?

Wir wählen ein array als Datenstruktur, in das die Menge der Größe nach geordnet eingetragen wird. Dann

- dauert FIND nur O(log(n)) Schritte, sofern man ein geordnetes array hat und binäre Suche verwendet,
- dauert INSERT aber O(n) Schritte, da beim Einfügen alle größeren Elemente um eine Komponente nach hinten verschoben werden müssen,
- dauert DELETE ebenfalls O(n) Schritte, da beim Löschen alle größeren Elemente um eine Komponente nach vorne verschoben werden müssen.

Darstellung als Liste: Alles dauert hier O(n) Schritte, sofern man beim Einfügen keine doppelten Elemente zulässt (sonst kann man INSERT in O(1) ausführen).

FIND: Kann man die binäre Suche 6.5.1 für geordnete Felder noch beschleunigen? Oft ja.

Beachte: Bei der binären Suche halbieren wir jeweils das gesamte noch verbleibende Feld A(links..rechts). Als Teilungsindex "Mitte" wählen wir:
Mitte = (rechts + links) / 2 = links + (rechts - links) / 2.

Verbesserung: Wenn man mit den Schlüsseln "rechnen" darf (wenn sich die Schlüssel also als Zahlen darstellen lassen) und wenn die Schlüssel recht gleichmäßig über den Indexbereich verteilt sind, so kann man den ungefähren Bereich, wo ein Schlüssel s im Feld A(links..rechts) liegen muss, genauer angeben durch den folgenden Teilungsindex

$$p = links + \frac{s - A(links)}{A(rechts) - A(links)} (rechts - links).$$

So geht man beispielsweise beim *Suchen in einem Lexikon* vor.

Man kann zeigen, dass mit dieser "Interpolationssuche" die Operation FIND bei einem geordneten Feld nur noch den uniformen Zeitaufwand O(log(log(n))) benötigt. Falls der Datenbestand groß und die Schlüssel gleichverteilt sind, so braucht man für den Suchprozess nur mit $1 \cdot \log(\log(n)) + 1$ Schritten zu rechnen (ohne Beweis; man muss darauf achten, dass A(p) zwischen A(links) und A(rechts) bleibt; überlegen Sie sich Details der Implementierung).

Da log(log(n)) eine sehr schwach wachsende Funktion ist, sollte man die Interpolationssuche einsetzen, wo immer es möglich ist. (In der Praxis liegt log(log(n)) fast immer unterhalb von 10.)

Struktur 3: Darstellung der Menge durch Bitvektoren

Da $A = \{a_1, ..., a_n\} \subseteq B = \{b_1, ..., b_r\}$ eine Teilmenge ist (mit $a_i \le a_j$ und $b_i \le b_j$ für $i < j$), kann man A auch durch einen Bitvektor $x = (x_1, ..., x_r)$, mit $x_i \in \{0, 1\}$, der Länge r darstellen, wobei für alle i gilt: $x_i = 1 \Leftrightarrow b_i \in A$.

Wird nach dem Element $s \in B$ gesucht und kennt man die Nummer, die s in der Anordnung von B hat (also dasjenige i mit $s = b_i$), dann gilt für eine Teilmenge A mit Bitvektor x:
FIND: $s = b_i \in A \Leftrightarrow x_i = 1$.
INSERT: Setze $x_i := 1$.
DELETE: Setze $x_i := 0$.
Im uniformen Komplexitätsmaß laufen alle drei Operationen in O(1), also in konstanter Zeit ab!

Ist |B| = r nicht allzu groß, so ist dies eine gute Darstellung. Die Methode versagt, wenn r sehr groß ist, z.B. r > 100_000, speziell bei kontinuierlichen Mengen, da dann jede Menge A r Binärstellen benötigt, auch wenn |A| = n klein ist.

Weiterhin: alle eventuellen weiteren Operationen mit Mengen erfordern den Aufwand O(r), während man in der Praxis höchstens auf O(n) oder $O(n^2)$ kommen darf. Oft lässt sich r gar nicht bestimmen, z.B. wenn man die Menge aller Namen zugrunde legt.

8.2 Bäume und (binäre) Suchbäume

Sie können sofort ab 8.2.3 weiterlesen, wenn Ihnen die Abschnitte 3.7 und 3.8 geläufig sind. (Dort wurden die Begriffe Baum, Graph, Wald, Nachbar, Nachfolger, Wurzel, Zyklus, Weg, Vorgänger, direkter Vorg., Blatt, Höhe usw. erläutert.)

Wir stellen diese Begriffe nochmals auf den folgenden Folien zusammen. Wir setzen voraus, dass die Begriffe ungerichteter und gerichteter Graph, Nachbar (im ungerichteten Fall) bzw. Vorgänger und Nachfolger (im gerichteten Fall), Weg in einem Graphen, Kreis (oder Zyklus), azyklischer Graph, Zusammenhang und Zusammenhangskomponente bekannt sind. Die auf Graphen bezogenen Definitionen finden Sie in Abschnitt 3.8. Weiterführende Definitionen sind in Abschnitt 8.8 zusammengefasst.

Definition 8.2.1: Es sei $G=(V,E)$ ein Graph, $|V|=n\geq 0$.

(1) Ein Knoten $w\in V$ heißt Wurzel von G, wenn es von w zu *jedem* Knoten des Graphen einen Weg gibt (im gerichteten Fall muss der Weg natürlich auch gerichtet sein).

(2) Der ungerichtete Graph G = (V, E) heißt Baum, wenn er azyklisch und zusammenhängend ist (insbesondere ist dann jeder Knoten des Baums auch Wurzel).

(3) Der gerichtete Graph G = (V, E) heißt Baum, wenn er eine Wurzel w besitzt, die keinen Vorgänger hat, und jeder Knoten ungleich der Wurzel genau einen Vorgänger besitzt, d.h., zu jedem $x\in V$, $x\neq w$ gibt es genau einen Knoten y mit $(y,x)\in E$, und es gibt kein $u\in V$ mit $(u,w)\in E$.

(4) Ein Graph heißt Wald, wenn alle seine (schwachen) Zusammenhangskomponenten Bäume sind.

Folgerung: Überzeugen Sie sich von folgenden Aussagen:

(a) Es sei $G=(V,E)$ ein ungerichteter Baum. Wähle irgendeinen Knoten w als Wurzel aus und ersetze jede ungerichtete Kante {x,y} durch die gerichtete Kante (x,y), wobei x näher an der Wurzel liegt als y, d.h., die Richtung zeigt stets von der Wurzel weg. So erhält man aus G in eindeutiger Weise einen gerichteten Baum $G'=(V,E')$ mit Wurzel w. Anwendung für die Programmierung: Man kann einen Baum stets als gerichteten Baum auffassen/implementieren.

(b) Es sei $G'=(V,E')$ ein gerichteter Baum mit Wurzel w. Ersetze jede gerichtete Kante (x,y) durch die ungerichtete Kante {x,y}, so erhält man aus G' in eindeutiger Weise den ungerichteten Baum $G=(V,E)$.

Folgerung (a) begründet, warum wir Bäume mit Hilfe von Zeigern darstellen. Fügt man für jeden Knoten noch einen Inhalt hinzu, so erhält man folgende Ada-Darstellung; hierbei ist MaxG der maximale Ausgangsgrad des Baums:

```
MaxG: constant Positive := ...;
type Grad is 1..MaxG;
type Element is ...;
type Baum;  type Ref_Baum is access Baum;
type Baum (ausgangsgrad: Grad := MaxG) is record
        Inhalt: Element;
        Nachf: array (1..ausgangsgrad) of Ref_Baum;
end record;
```

Hier hat jeder Knoten mindestens einen Nachfolger. Knoten ohne Nachfolger müssen daher den ausgangsgrad 1 und dann einen null-Zeiger erhalten.

Folgerung (Fortsetzung):

(c) In einem ungerichteten Baum gibt es von jedem Knoten zu jedem Knoten *genau* einen doppelpunktfreien Weg.

(d) In jedem gerichteten Baum gibt es zu jedem Knoten $x \in V$ *genau* einen Weg von der Wurzel w nach x.

(e) Wenn es in einem gerichteten Baum einen Weg vom Knoten x zum Knoten y gibt, dann führt der Weg von der Wurzel w nach y über den Knoten x.

Definition 8.2.2: Es sei $G = (V, E)$ ein Graph.

(1) Zu jedem Knoten x eines ungerichteten Graphen $G = (V, E)$ heißt $N(x) = \{y \mid \{x,y\} \in E\}$ die Menge der *Nachbarn* von x. Ihre Anzahl $|N(x)|$ heißt *Grad des Knotens* x. Der maximale Knotengrad heißt *Grad des Graphen* G.

(2) Zu jedem Knoten x eines gerichteten Graphen $G = (V, E)$ heißt $Vor(x) = \{y \mid (y,x) \in E\}$ die Menge der *Vorgänger* von x und $Nach(x) = \{y \mid (x,y) \in E\}$ die Menge der *Nachfolger* von x. Ihre Anzahlen $|Vor(x)|$ bzw. $|Nach(x)|$ heißen *Eingangsgrad* bzw. *Ausgangsgrad des Knotens* x. Der jeweils maximale Grad heißt *Eingangsgrad* bzw. *Ausgangsgrad des Graphen* G.

Definition 8.2.2 (Fortsetzung): Es sei $G = (V, E)$ ein Baum.

(3) Ist G ein gerichteter Baum mit Wurzel w, so ist $|Vor(x)| = 1$ für alle Knoten $x \neq w$ und $|Vor(w)| = 0$. Der eindeutig bestimmte Knoten $vorg(x) = y \in Vor(x)$ heißt direkter Vorgänger oder Elternknoten oder Vaterknoten von x. Jeder Knoten, der auf dem Weg von der Wurzel w nach x liegt, heißt Vorfahr von x (aber nicht x selbst). Knoten, die den gleichen direkten Vorgänger besitzen, heißen Geschwister oder Brüder. Jeder Knoten $x \in Nach(y)$ heißt direkter Nachfolger oder Kind oder Sohn von x.

(4) Gerichteter Baum: Ein Knoten x mit $x \neq w$ und mit $|Nach(x)| = 0$ heißt Blatt des Baums.
Ungerichteter Baum: Ein Knoten x mit $x \neq w$ und mit $|N(x)| = 1$ heißt Blatt des Baums.

Definition 8.2.2 (Fortsetzung):

(5) Es sei $G = (V, E)$ ein Baum mit Wurzel w. Der von einem Knoten x in G erzeugte Unterbaum (oder Teilbaum) ist $G_x = (V_x, E_x)$ mit
$V_x = \{y \mid$ jeder doppelpunktfreie Weg von w nach y führt über $x\}$,
$E_x = E|_{V_X}$ (= alle Kanten zwischen Knoten aus V_x).
Offenbar ist G_x ein Baum mit Wurzel x. Beachte, dass G_x nicht leer ist, da stets $x \in G_x$ gilt. Speziell ist $G_w = G$.

Folgerungen:

(g) Wenn es in einem gerichteten Baum einen Weg vom Knoten x zum Knoten y gibt, so liegt y in dem von x erzeugten Unterbaum.

(h) Wenn G ein Baum mit n Knoten ist, so besitzt G genau n-1 Kanten (für n > 0).

(i) Fügt man in einem ungerichteten Baum eine Kante hinzu, so entsteht ein Zyklus. Streicht man aus diesem Zyklus danach wieder irgendeine Kante, so entsteht erneut ein Baum.

Folgerung: Überzeugen Sie sich von folgender Aussage:

(j) Jeder Baum lässt sich mit zwei Farben färben, d.h., es gibt eine Abbildung $f: V \to \{1,2\}$ mit $f(x) \neq f(y)$ für alle Kanten $\{x,y\}$ bzw. (x,y).

Bäume sind 2-färbbar. Definition hierzu: Sei $k \in \mathbb{N}$.
Ein beliebiger Graph $G=(V,E)$ lässt sich mit k Farben färben (man sagt auch, G ist k-färbbar), wenn eine Abbildung

$$f: V \to \{1,2,...,k\}$$

existiert mit $f(x) \neq f(y)$ für alle Kanten $\{x,y\}$ bzw. (x,y). Die minimale Zahl k, so dass sich G mit k Farben färben lässt, heißt Färbungszahl von G; sie zu bestimmen, heißt "Färbungsproblem". Diese Zahl lässt sich nach heutiger Kenntnis für beliebige Graphen nur mit großem Zeitaufwand berechnen. (Das Problem liegt in **NP**, aber man weiß nicht, ob es auch in **P** liegt.)

Definition 8.2.3:

(1) Es sei $G=(V,E)$ ein Baum. Ist für jeden Knoten x die Menge N(x) bzw. im gerichteten Fall die Menge Nach(x) geordnet (d.h., für die Knoten y_i der Menge N(x) bzw. Nach(x) gilt: $y_1 < y_2 < ... < y_k$), dann heißt G ein geordneter Baum.

(2) Sei $k \in \mathbb{N}$. Sei $G=(V,E)$ ein gerichteter Baum mit $0 \notin V$. Dieser Baum zusammen mit einer Abbildung
$\nu: V \times \{1,...,k\} \to V \cup \{0\}$, so dass für alle $x \in V$ gilt:
$\text{Nach}(x) = \{\nu(x,i) \mid i = 1,...,k\} \setminus \{0\}$ und aus
$\nu(x,i) \neq 0$, $\nu(x,j) \neq 0$ und $i \neq j$ folgt $\nu(x,i) \neq \nu(x,j)$,
heißt k-närer Baum. ($\nu(x,i)$ ist der i-te Nachfolger von x; ist dieser Wert 0, dann ist der i-te Nachfolger der leere Baum.)
(Die direkten Nachfolger von x stehen also in einem k-stelligen Vektor, wobei Komponenten mit leerem Unterbaum - durch 0 bezeichnet - auftreten dürfen/müssen, sofern der Ausgangsgrad von x kleiner als k ist.)

Speziell: Im Fall $k=2$ heißt der Baum binär oder Binärbaum.

Zugehörige Datenstruktur in Ada:

```
K: constant Positive := ...;
type Element is ...;
type K_Baum;  type Ref_K_Baum is access K_Baum;
type K_Baum is record
        Inhalt: Element;
        Nachf: array (1..K) of Ref_K_Baum;
end record;
```

Definition 8.2.4: Es sei $G=(V,E)$ ein Baum mit Wurzel w.

(1) Die Anzahl der Knoten in einem längsten Weg von der Wurzel zu einem Blatt heißt die Tiefe des Baumes G. (Dies ist also die Länge des längsten Weges im Baum, der von w ausgeht, plus 1.) Der leere Baum hat die Tiefe 0.

(2) Zu jedem Baum mit Wurzel w gehört die Levelfunktion level: $V \to \mathbb{N}_0$, rekursiv definiert durch
level(w) = 1 und
level(x) = level(vorg(x)) + 1 für $x \neq w$.

Hinweise: Die Tiefe des Baumes ist das maximale Level eines Knotens x im Baum:

Tiefe von G = Max{level(x) | $x \in V$}.

Statt "Tiefe" verwendet man auch das Wort Höhe. Achten Sie in der Literatur genau auf die Definition; oft werden auch die um 1 verringerten Werte als Höhe oder Tiefe bezeichnet.

8.2.5: Rekursive Definition für k-näre Bäume (für $k \in \mathbb{N}$)

1) Die leere Menge ist ein k-närer Baum.

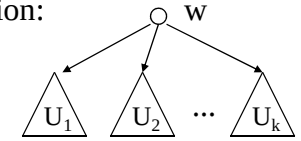
2) Wenn w ein Knoten und $U=(U_1, U_2, ..., U_k)$ ein k-Tupel von k-nären Bäumen ist, so ist auch $w\,U$ ein k-närer Baum.

w bildet die Wurzel des Baums $w\,U$. Die Komponenten von U sind Unterbäume oder Teilbäume im Baum $w\,U$.
U_i ist der i-te Unterbaum von w.

Skizze: Leerer Baum: $\varnothing$ Tiefe (leerer Baum) = 0

gerichtet oder ungerichtet

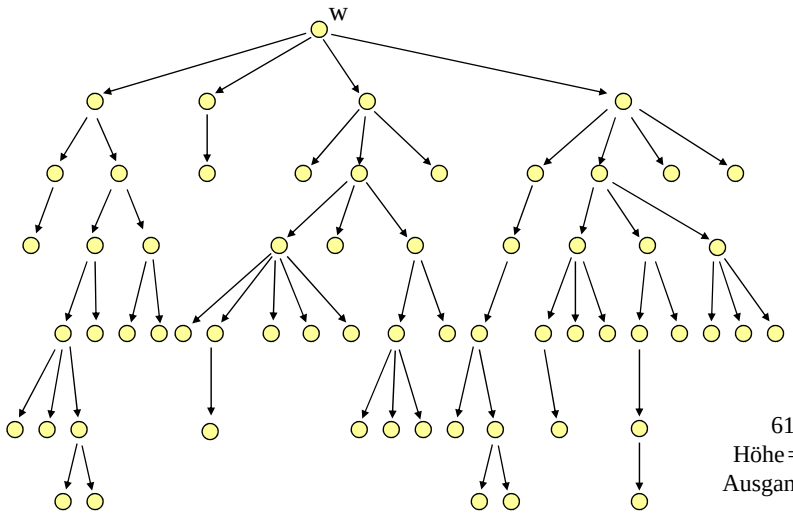
Rekursion:



Tiefe $(w\,U)$ = Max{Tiefe(U_i) | i=1,...,k} + 1

Die k Nachfolger von w sind hier geordnet.

Beispiel für einen "beliebigen geordneten Baum":



61 Knoten,
Höhe = Tiefe 7,
Ausgangsgrad 5

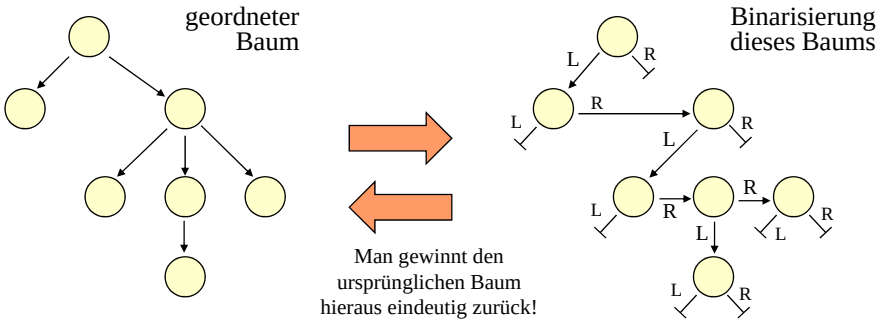
Spezialfall: Binäre Bäume

Binäre Bäume sind k-näre Bäume für k = 2. Jeder Knoten besitzt genau einen linken und einen rechten Nachfolger (diese Nachfolger können leer sein; *wichtig ist, dass der linke und der rechte Nachfolger stets unterschieden werden*).

8.2.6 Binarisierung von beliebigen geordneten Bäumen

Jeder geordnete Baum lässt sich eindeutig in einen binären Baum umwandeln, indem

- der linke Zeiger L stets auf das erste Kind und
- der rechte Zeiger R stets auf den nächsten Geschwisterknoten zeigt.



Folgerung aus dieser eindeutigen Umwandlung:

Es sei $\mathbf{C_n}$ die Anzahl aller binärer Bäume mit n Knoten.

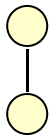
Es sei $\mathbf{B_n}$ die Anzahl aller geordneter Bäume mit n Knoten.

Dann gilt: $\mathbf{B_n = C_{n-1}}$ für alle $n > 0$.

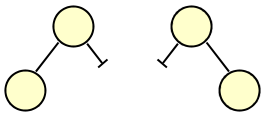
Kann dies sein (?), weil die binären Bäume doch eigentlich eine Teilmenge der geordneten Bäume sein müssten?!

Dies stimmt aber nicht, weil geordnete Bäume keine zusätzlichen leeren Unterbäume haben. Z. B.:

∃ nur ein geordneter Baum mit nur einem Nachfolger:



aber zwei binäre Bäume mit nur einem Nachfolger:



Hinweis: Eine zu unserer Binarisierung verwandte Darstellung ist die Ordnerhierachie in einem Dateisystem.

Zentrale Frage:

Wie viele binäre Bäume mit n Knoten gibt es?

Das heißt: Berechne C_n.

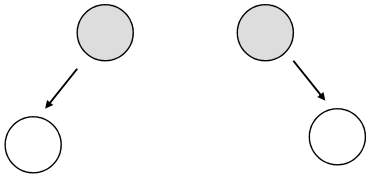
Zunächst berechnen wir C_0, C_1, ..., C_4 durch Aufzählen aller zugehöriger Binärbäume.

Wir listen alle binären Bäume mit höchstens 4 Knoten auf. Die Wurzel des binären Baums ist hier grau dargestellt. Die leeren Zeiger wurden weggelassen.

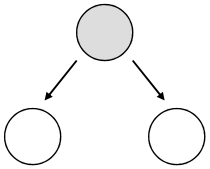
n = 0 <leerer Baum> $C_0 = 1$

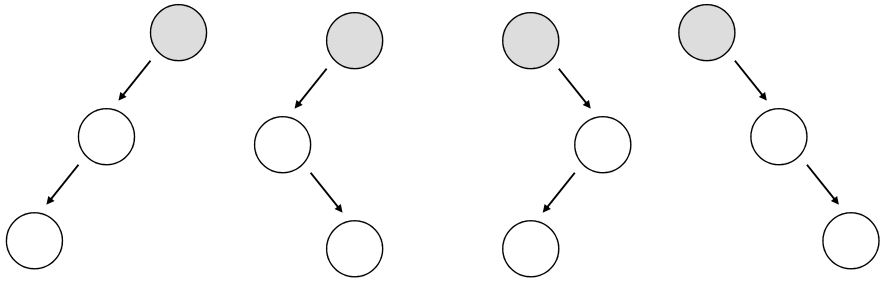
n = 1 $C_1 = 1$

n = 2 $C_2 = 2$

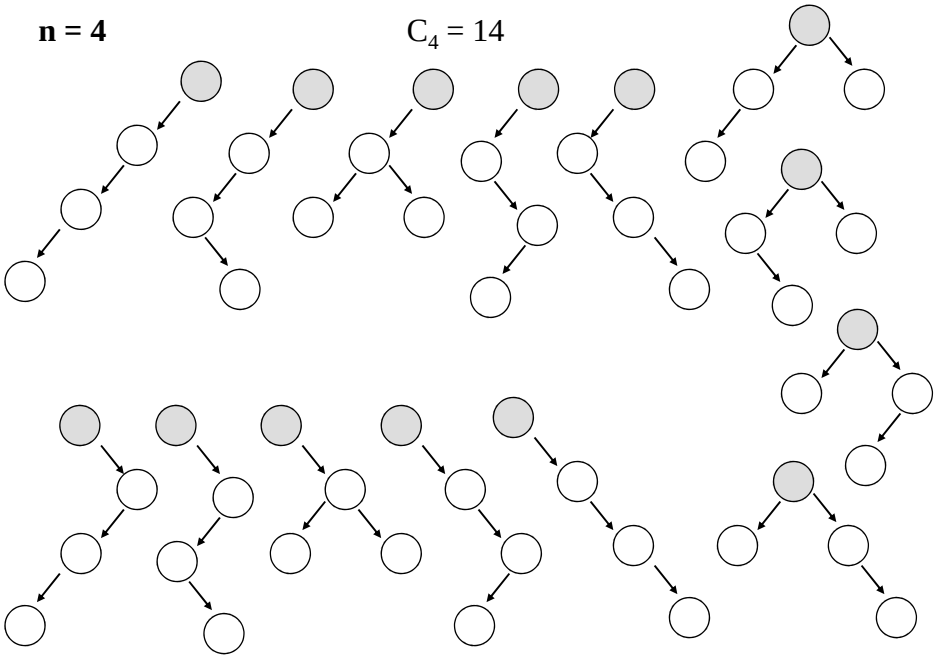


n = 3 $C_3 = 5$





n = 4 $C_4 = 14$



Durch Ausprobieren erhält man die Werte:

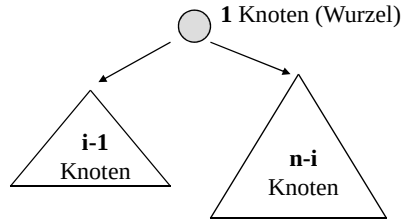
n	Anzahl
0	1
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	1430

Für C_n gilt die Rekursionsformel:

$C_0 = 1$, und für alle $n \geq 1$:

$$C_n = \sum_{i=1}^{n} C_{i-1} \cdot C_{n-i}$$

wegen (i = 1, 2, ..., n):



Erinnerung aus der Mathematik: Binomialkoeffizient

$$\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!} = \frac{n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot(n-k+1)}{1\cdot 2\cdot 3\cdot\ldots\cdot k}$$

Bedeutung für die Anwendung:
Es gibt genau " n über k " Möglichkeiten, um k verschiedene Dinge aus n verschiedenen Dingen (ohne Zurücklegen und Wiederholungen) auszuwählen.

Beispiel "Lotto 6 aus 49": Es gibt genau " 49 über 6 " = (49·48·47·46·45·44)/(1·2·3·4·5·6) = 13 983 816 Möglichkeiten, aus 49 Zahlen 6 Zahlen auszuwählen, sofern die Reihenfolge des Auswählens keine Rolle spielt.

8.2.7 Satz "Catalansche Zahlen"

$$\mathbf{C_n = \frac{1}{n+1}\binom{2n}{n}}$$

Näherung: $C_n \approx \frac{4^n}{(n+1)\sqrt{\pi\cdot n}}$ für $n > 0$

Wir beweisen nur den Satz. Die Näherung ergibt sich aus der Stirlingschen Formel (6.5.6.3).

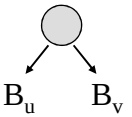
Wir zeigen zunächst: Die Anzahl C_n der Binärbäume ist gleich der Anzahl der Möglichkeiten, um n „Klammer auf“ und n „Klammer zu“ wie in korrekt geklammerten Ausdrücken aneinander zu reihen.
Z.B. gibt es genau 5 korrekte Klammerungmöglichkeiten für n = 3:
((())), **(()())**, **(())()**, **()(())**, **()()()**.

Behauptung: Die Anzahl C_n der Binärbäume ist gleich der Anzahl der Möglichkeiten, um n „Klammer auf“ und n „Klammer zu“ wie in korrekt geklammerten Ausdrücken aneinander zu reihen.

Wir geben diesen Zusammenhang präzise an:

1. Jeder korrekt geklammerte Ausdruck fängt mit "(" an.
2. Es gibt zu dieser "(" genau eine zugehörige ")", nämlich die erste ")", bei der die Anzahl der "(" und ")" gleich sind, von links nach rechts gezählt.
3. Also hat jeder nicht-leere Klammerausdruck die Form (u)v, wobei sowohl u als auch v korrekt geklammerte Ausdrücke sind und kein echter Anfang von u korrekt geklammert ist. u und v sind eindeutig bestimmt; u und v können leer sein.

4. Ordne dem leeren Wort den leeren Binärbaum zu.
5. Ordne dann rekursiv dem Ausdruck **(**u**)**v folgenden Baum zu:



wobei B_u der zu u und B_v der zu v gehörige Baum ist.

Umgekehrt gewinnt man aus diesem Baum den Ausdruck **(**u**)**v.

Auf diese Weise lässt sich jedem korrekt geklammerten Ausdruck umkehrbar eindeutig ein binärer Baum zuordnen.

Folglich ist auch die Anzahl der korrekten Klammerungen aus n Klammerpaaren gleich C_n.

Damit ist die Behauptung bewiesen.

[Für u=v=ε wird **( )** also der einknotige Baum zugeordnet: ○]

Wie viele korrekt geklammerte Ausdrücke gibt es?

Man muss n "Klammer auf" auf 2n Positionen verteilen.

Hiervon gibt es genau $\binom{2n}{n}$ Möglichkeiten.

Von dieser Anzahl muss man die abziehen, die keine korrekte Klammerungen bilden. Sei w eine nicht korrekte Klammerung aus n "(" und n ")". Diese besitzt eine erste Position, bis zu der mehr ")" als "(" stehen. w hat also die Form:

x) y

wobei x korrekt geklammert ist und y genau eine "(" mehr besitzt als ")".

Ersetze nun in y jede "(" durch ")" und umgekehrt. So möge die Klammerfolge y' entstehen. Für x) y' gilt dann:

x möge k Klammerpaare besitzen ($0 \le k \le n$).
Dann besitzt y n-k "Klammer auf" und n-k-1 "Klammer zu".
Dann besitzt y' n-k-1 "Klammer auf" und n-k "Klammer zu".
Also besitzt x)y' n-1 "Klammer auf" und n+1 "Klammer zu".

Weiterhin gilt: Geht man von zwei verschiedenen unkorrekten Klammerungen aus, so erhält man auch zwei verschiedene Ausdrücke der Form x)y'. Beweis: Wäre nämlich $x)y' = x_1)y_1'$, dann muss $x=x_1$ sein, da x und x_1 beide korrekt geklammert sind und ")" an der ersten Position steht, an der die Anzahl der "Klammer zu" die Anzahl der "Klammer auf" übersteigt. Ebenso muss dann $y' = y_1'$ sein, da die Längen der beiden Ausdrücke gleich lang, nämlich 2n, sind. Daher gilt auch $y = y_1$. Die Zuordnung $w \rightarrow x)y'$ ist daher injektiv.

Wir haben also gezeigt: Jeder unkorrekten Klammerung w von n Klammerpaaren lässt sich eindeutig eine Folge von n-1 "Klammer auf" und n+1 "Klammer zu" zuordnen.

Die Umkehrung gilt aber auch:

Ist eine Folge z aus n-1 "Klammer auf" und n+1 "Klammer zu" gegeben, so muss es genau eine erste Stelle geben, an der die Zahl der "Klammer zu" die Zahl der "Klammer auf" übersteigt. Die Folge hat also die Form x)y', wobei x ein korrekt geklammerter Ausdruck ist und in x überall die Anzahl der "Klammer auf" größer oder gleich der Anzahl der "Klammer zu" ist. In y' gibt es dann eine "Klammer auf" weniger, als es "Klammer zu" gibt. Wandle nun y' in ein y um, indem jede "(" durch ")" ersetzt wird und umgekehrt. So erhält man eine Folge x)y, die gleich viele "Klammer auf" und "Klammer zu" besitzt und die nicht korrekt geklammert ist, wegen der "(" zwischen x und y.

Diese Zuordnung $z \rightarrow x)y$ ist offenbar ebenfalls injektiv.

Daher gilt:

Jeder unkorrekten Klammerung von n Klammerpaaren lässt sich umkehrbar eindeutig eine Folge von n-1 "Klammer auf" und n+1 "Klammer zu" zuordnen. Hieraus folgt:

Die Anzahl der unkorrekten Klammerungen mit n Klammerpaaren ist gleich der Anzahl der Folgen von n-1 "Klammer auf" und n+1 "Klammer zu".

Deren Anzahl ist aber $\binom{2n}{n-1}$.

Wir haben also gezeigt:

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n}.$$

Damit ist Satz 8.2.7 bewiesen. ■

Folgerung 1:

$$C_{n+1} = C_n \cdot \frac{4n+2}{n+2} = 4 \cdot C_n \cdot (1 - \frac{3}{2n+4})$$

Dieser Formel kann man zum einen das exponentielle Wachstum entnehmen, das in der Näherungsformel ausgedrückt wird. Zum anderen lassen sich hiermit die Catalanschen Zahlen, ausgehend von $C_0 = 1$, leicht iterativ berechnen.

Hinweis: Aus dem Satz lässt sich sofort schließen:

Der Binomialkoeffizient $\binom{2n}{n}$ ist stets durch n+1 teilbar.

(Unter welchen Bedingungen auch durch n+2?

Man erhält manchmal solche "nebensächlichen" Resultate.)

Folgerung 2 aus diesem Satz:

Wir werden oft Mengen in geordneten oder in binären Bäumen speichern. Will man n Elemente auf diese Weise ablegen, so gibt es C_n verschiedene Bäume, die hierfür in Frage kommen. Deren Anzahl wächst größenordnungsmäßig wie 4^n, so dass man "geeignete" Bäume in der Praxis *nicht durch Ausprobieren* aller Möglichkeiten finden kann!

Vielmehr folgern wir aus Satz 8.2.7:
Wir müssen "intelligente" Verfahren entwickeln, um spezielle Bäume, die für gewisse Anwendungsprobleme nützlich sind, aufzuspüren. Wie solche speziellen Bäume aussehen können und welche "intelligenten" Verfahren es für ihre Bearbeitung gibt, werden wir im weiteren Verlauf dieses Kapitels vorstellen.

8.2.8 "Suchbaum"

Erinnerung an Abschnitt 3.7: Ein binärer Baum kann preorder, inorder oder postorder in linearer Zeit durchlaufen werden, siehe nächste Folie (L ist der Verweis zum linken Unterbaum, R der zum rechten).

Man kann eine Folge sortieren, indem man ihre Elemente nacheinander in einen Binärbaum nach ihrer Größe einfügt und diesen am Ende inorder ausgibt (siehe 3.7.7).

Hierzu muss jeder Knoten einen "Inhalt" erhalten und die Elemente müssen in die Knoten "richtig" eingeordnet werden. Solch einen Baum nannten wir einen "Suchbaum".

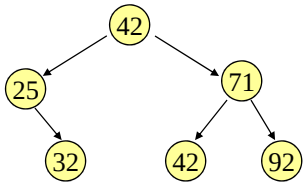
Hinweis: Der Inorder-Vorgänger eines Knotens x in einem binären Baum ist der Knoten, der in der inorder-Reihenfolge unmittelbar vor dem Knoten x ausgewertet wird. Analog: Inorder-Nachfolger, Postorder-Vorgänger usw.


```
procedure Preorder (K: Ref_BinBaum) is
begin  if K /= null then  < bearbeite den Knoten K >;
                          Preorder (K.L);
                          Preorder (K.R);
       end if;
end Preorder;
```

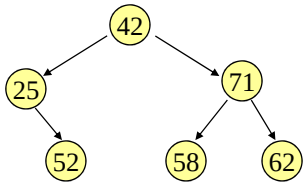
```
procedure Inorder (K: Ref_BinBaum) is
begin  if K /= null then  Inorder (K.L);
                          < bearbeite den Knoten K >;
                          Inorder (K.R);
       end if;
end Inorder;
```

```
procedure Postorder (K: Ref_BinBaum) is
begin  if K /= null then  Postorder (K.L);
                          Postorder (K.R);
                          < bearbeite den Knoten K >;
       end if;
end Postorder;
```

Definition: Ein binärer Baum, dessen Inhalts-Datentyp eine geordnete Menge ist (z.B. ganze Zahlen), heißt (binärer) Suchbaum, wenn für jeden Knoten u gilt: Alle Inhalte von Knoten im linken Unterbaum von u sind echt kleiner als der Inhalt von u und alle Inhalte von Knoten im rechten Unterbaum von u sind größer oder gleich dem Inhalt von u.



Dies ist ein Suchbaum



Dies ist kein Suchbaum

8.2.9 Suchbäume zu einer festen Folge

Es sei $a_1, a_2, ..., a_n$ eine sortierte Folge von n ganzen Zahlen und es sei B ein Suchbaum mit dem Inhalts-Datentyp Integer. Dann kann man die Zahlen a_i auf genau eine Art so in die Knoten von B legen, dass der Inorder-Durchlauf von B die sortierte Folge $a_1, a_2, ..., a_n$ ergibt.

Dies ist klar: Man durchlaufe B inorder und ordne dem i-ten Knoten bei dieser Besuchsreihenfolge die Zahl a_i zu.

Da es exponentiell viele (genauer: C_n) binäre Bäume mit n Knoten gibt, kann man den für eine gegebene Fragestellung "besten Baum" nicht durch Ausprobieren aller Suchbäume finden, sondern muss ihn mit 'guten Algorithmen' ermitteln. Wir untersuchen nun Binärbäume für die 3 Grundoperationen und gehen ab Kapitel 8.3 zu speziellen Bäumen über.

8.2.10 Binäre Bäume und die 3 Grundoperationen

Aufgabe: Gegeben sei eine geordnete Menge. Hierfür werden wir im Folgenden die ganzen Zahlen verwenden. Eine Folge solcher Elemente (dabei können Elemente mehrfach vorkommen) soll in einem Binärbaum gespeichert werden. Man gebe konkrete Verfahren für das Suchen (FIND), das Einfügen (INSERT) und das Löschen (DELETE) an und berechne deren Aufwand.
Die Datenstruktur für Binärbäume ist hier stets:

```
type BinBaum;
type Ref_BinBaum is access BinBaum;
type BinBaum is record
        Inhalt: Integer;
        L, R: Ref_BinBaum;
end record;
```


Verfahren für alle Suchbäume (auch für die später vorzustellenden AVL- und B-Bäume) und alle Schlüssel s:

Suchen (FIND): Durchlaufe einen Weg von der Wurzel zu einem Blatt, wobei man entweder s findet oder an einem Ende feststellt, dass s im Baum nicht vorkommt.

Einfügen (INSERT): Gehe so vor, als ob s gesucht werden soll; hierbei gelangt man schließlich an einen leeren Verweis; hänge genau hier einen neuen Knoten mit dem Schlüssel s an.

Löschen (DELETE): Suche den Knoten u mit Inhalt s. Falls dieser Knoten keinen oder nur einen Nachfolger besitzt, kann man ihn leicht löschen. Falls er zwei Nachfolger hat, so suche den Inorder-Vorgänger v (oder den Inorder-Nachfolger v) von u; dessen Inhalt sei s'; ersetze s in u durch s' und entferne v (zu Inorder-Vorgänger siehe 8.2.8).

8.2.11 Suchen in einem Binärbaum

Der Binärbaum ist durch den Zeiger "Anker" auf seine Wurzel gegeben. Wir formulieren die Funktion *Suche*, die zu dem Zeiger Anker und dem zu suchenden Schlüssel s einen Verweis q auf den Knoten zurückgibt, dessen Inhalt s ist. Ist s nicht im Binärbaum enthalten, wird der Verweis null zurückgegeben.

```
function Suche (Anker: in Ref_BinBaum; s: in Integer)
    return Ref_BinBaum is
q: Ref_BinBaum := Anker;
begin   while q /= null and then (q.Inhalt /= s) loop
            if s < q.Inhalt then q := q.L; else q := q.R; end if;
        end loop;
        return q;
end Suche;
```


Rekursive Darstellung:

```
function SucheRek (p: Ref_BinBaum; s: Integer)
    return Ref_BinBaum) is
begin
    if (p = null) or else (s = p.Inhalt) then  return p;
    elsif s < p.Inhalt then return SucheRek(p.L, s);
    else return SucheRek(p.R, s);
    end if;
end SucheRek;
```


8.2.12 Einfügen in einen Binärbaum

Prinzipiell werden bei binären Suchbäumen die neuen Elemente als neues Blatt in den Baum eingetragen.

```
procedure Einfügen(Anker: in out Ref_BinBaum; s: Integer) is
    p,q: Ref_BinBaum := Anker;
begin
    if  p = null  then  Anker := new BinBaum'(s, null, null);
    else
        while p /= null loop  q := p;
            if p.Inhalt > s then p := p.L; else p := p.R; end if;
        end loop;
        if s < q.Inhalt then q.L := new BinBaum'(s, null, null);
        else q.R := new BinBaum'(s, null, null); end if;
    end if;
end Einfügen;
```

Aufgabe: Schreiben Sie für diese Operation eine rekursive Prozedur.

8.2.13 Löschen in einem Binärbaum

Der Schlüssel s soll gelöscht werden. Die Suche ergibt, dass s im Knoten u steht. Hat u keinen Nachfolger, so wird u einfach gelöscht und der Verweis vom Vorgängerknoten von u wird auf null gesetzt.

Hat u genau einen Nachfolger, so wird u gelöscht und der Verweis des Vorgängerknotens auf u wird auf den einzigen Nachfolger von u gesetzt.

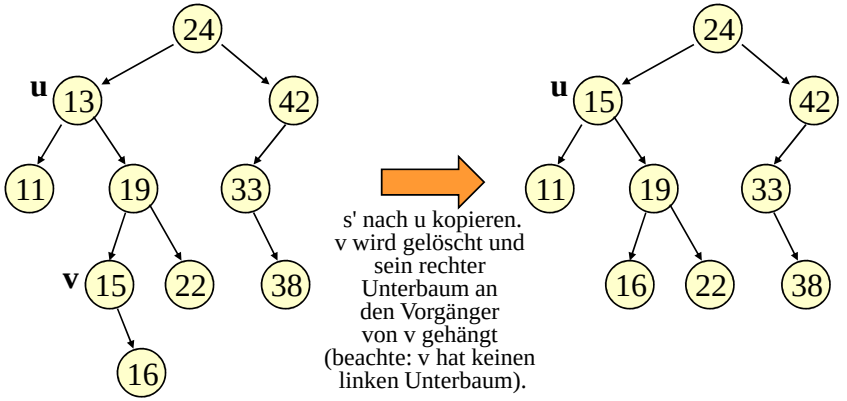
Hat u zwei Nachfolger, dann kann u nicht einfach gelöscht werden. Vielmehr ersetzt man den Inhalt von u durch den Schlüssel s', der im Inorder-Vorgänger v von u steht; dieser hat höchstens einen Nachfolger und man kann nun v wie oben angegeben löschen. Auf diese Weise entsteht wieder ein Suchbaum. (Statt des Inorder-Vorgängers kann man auch den Inorder-Nachfolger nehmen.)

Wie findet man den Inorder-Vorgänger v von u?
Sehr einfach: v ist der rechteste Knoten im linken Unterbaum von u. Man geht also zum linken Nachfolger von u und folgt dann immer dem rechten Verweis, bis dieser null ist.

Analog ist der Inorder-Nachfolger der linkeste Knoten im rechten Unterbaum von u.

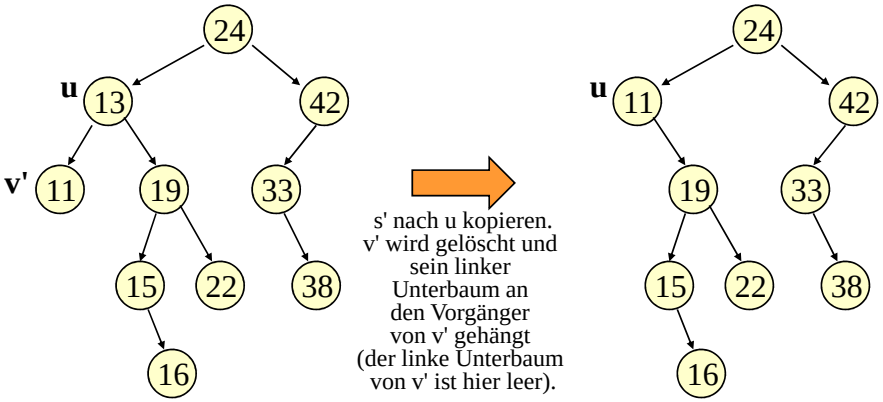
In der Praxis entscheidet man sich jedes Mal zufällig, ob man den Inorder-Vorgänger oder den Inorder-Nachfolger für das Löschen heranzieht. Hierdurch vermeidet man, dass der Suchbaum nach rechts (bzw. nach links) hin eine größere Tiefe erhält. Dadurch würde sich bei vielen Löschungen die mittlere Suchdauer allmählich erhöhen, siehe 8.2.22.

Skizze: Lösche s = 13. s steht im Knoten **u**. **u** hat zwei Kinder.



Der Inorder-Nachfolger-Knoten **v** von **u** hat hier den Inhalt s' = 15.

Man hätte auch den Inorder-Vorgänger-Knoten v' nehmen können:



Der Inorder-Vorgänger-Knoten **v'** von **u** hat hier den Inhalt s' = 11.

Löschen in einem Binärbaum (über den Inorder-Nachfolger)

```
procedure Löschen (Anker: in Ref_BinBaum; s: in Integer) is
u, v: Ref_BinBaum := null;
begin  u := Suche (Anker, s);   -- 8.2.11; falls u = null wird, nichts tun
   if  u /= null  then
      if (u.L = null) or (u.R = null) then
         < lösche u, hänge Unterbaum um, sofern einer existiert > ;
      else                              -- Suche den Inorder-Nachfolgerknoten v
         v := u.R;
         while  v.L /= null  loop  v := v.L; end loop;
         u.Inhalt := v.Inhalt;          -- s' wird nach u kopiert
         < lösche v, hänge Unterbaum um, sofern einer existiert > ;
      end if;
   end if;
end Löschen;
```

Hier ist noch ein Problem: Für < lösche u > und < lösche v > muss man den jeweiligen Vorgänger von u bzw. v kennen. Also muss die Prozedur Löschen modifiziert werden.

Hinweis zur Realisierung: Man lässt einen Zeiger "vorg" mitlaufen, der auf den zuvor betrachteten (Vorgänger-) Knoten zeigt:

```
procedure Löschen (Anker: in Ref_BinBaum; s: in Integer) is
u, v, vorg: Ref_BinBaum := null; links: Boolean;
begin  "Suche(Anker, s, u, vorg, links);"
    -- dies ist neu zu programmieren: vorg zeigt auf den Elternknoten von u
    -- (sofern vorhanden) und links ist true, falls u linkes Kind von vorg ist
    if  u /= null  then
        if (u.L = null) or (u.R = null) then  < lösche u, ... > ;
        else                          -- Suche den Inorder-Nachfolgerknoten v
            v := u.R; vorg := u;
            while  v.L /= null  loop  vorg := v; v := v.L; end loop;
            u.Inhalt := v.Inhalt;          -- s' wird nach u kopiert
            < lösche v > ;
        end if;
    end if;
end Löschen;
```

Vorsicht, unvollständig!

Dieses Programm ist noch unvollständig.
Bedeutet < lösche v > nun einfach:

```
vorg.L := v.R;   ? (Vorsicht: Fehler!)
```

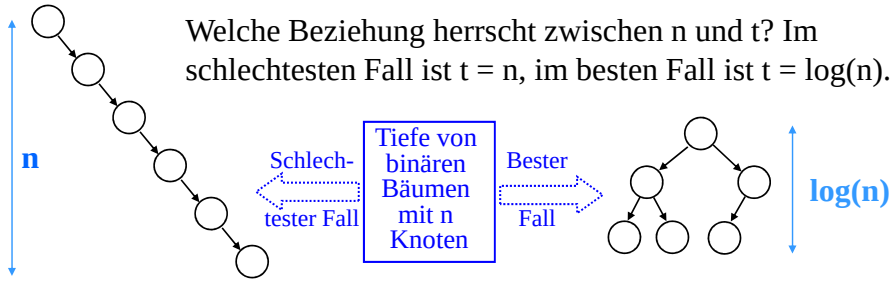
Wie müssen < lösche u > und Suche neu programmiert werden?
Was ist in dem Fall, dass der Baum nur aus einem Knoten besteht?
Wie muss man die Prozedur Löschen modifizieren, um alle Vorkommen von s im Baum zu löschen?

Selbst lösen! (Übungen)

8.2.14: Welchen Aufwand erfordern die Operationen Suchen, Einfügen und Löschen, wenn der Suchbaum n Knoten besitzt?

Man misst diesen Aufwand meist in der Anzahl der Vergleiche, die erforderlich sind, um die Operation durchzuführen.

Suchen, Einfügen und Löschen: Im schlechtesten Fall benötigt man jeweils t Vergleiche, wobei t die Tiefe des Baumes mit n Knoten ist; zur Definition "Tiefe" siehe 8.2.4 (1).



Welche Beziehung herrscht zwischen n und t? Im schlechtesten Fall ist t = n, im besten Fall ist t = log(n).

Ein Baum kann also zu einer Liste "entarten" und man braucht dann entsprechend viele Vergleiche.

Besonders günstig ist ein Baum, in dem jeder Knoten höchstens das Level log(n+1) besitzt; zu "Level" siehe 8.2.4 (2).

Was genau ist hier "log" (der Logarithmus zur Basis 2)? Wir benötigen ihn als eine Funktion zwischen natürlichen Zahlen und modifizieren daher die gewohnte reellwertige Funktion wie folgt (nach oben aufgerundeter Logarithmus):

Definition 8.2.15: Diskreter Zweierlogarithmus. Wenn nicht anders vermerkt sei in Zukunft log: $\mathbb{N} \to \mathbb{N}_0$ definiert durch
$\log(1) = 0$, und für $n \geq 2$:
$\log(n) = k$ für das eindeutig bestimmte k mit $2^{k-1} < n \leq 2^k$.

Es gilt dann also log(2)=1, log(3)=log(4)=2, log(5)=3 usw. Vom üblichen Logarithmus weicht dieses log höchstens um 1.0 ab.

Folgerung 8.2.16:
log sei wie in 8.2.15 definiert, dann gilt für die Tiefe t jedes binären Baums mit n Knoten $\log(n+1) \le t \le n$ für alle $n \ge 0$.

Beweis: $t \le n$ ist klar, da jeder doppelpunktfreie Weg in einem Baum mit n Knoten höchstens n Knoten besitzen kann.

Ein binärer Baum der Tiefe k kann höchstens 2^k-1 Knoten haben, wie man durch Induktion leicht sieht:
Für k=1 ist dies richtig, und wenn Bäume der Tiefe k höchstens 2^k-1 Knoten enthalten, dann kann ein Baum der Tiefe k+1 höchstens "Wurzel plus zwei Unterbäume der Tiefe k", also $1+2^k-1+2^k-1=2^{k+1}-1$ Knoten besitzen.
Folglich ist $n \le 2^t-1$, also $\log(n+1) \le \log(2^t) = t$.
(Für den Fall n = 0 trifft die Aussage ebenfalls zu.) ■

Hinweis: In der Ungleichung $\log(n+1) \le t \le n$ für alle $n \ge 0$ werden beide Grenzen angenommen, d.h., es gibt Bäume mit n Knoten, deren Tiefe n ist, und es gibt Bäume mit n Knoten, deren Tiefe log(n+1) ist. Wie groß ist aber die Tiefe im Mittel?

Übungsaufgaben:

a) Berechnen Sie, wie viele verschiedene binäre Bäume mit n Knoten es gibt, die genau die Tiefe n besitzen.

b) Versuchen Sie zu berechnen, wie viele verschiedene binäre Bäume mit n Knoten es gibt, die genau die Tiefe log(n+1) besitzen.

c) Berechnen Sie die "mittlere Suchzeit im besten Fall" für den Fall $n = 2^k-1$, d.h., summieren Sie für einen Baum mit diesen n Knoten und minimaler Tiefe alle Level seiner Knoten auf und dividieren diesen Wert durch n. Führen Sie die gleiche Berechnung für den schlechtesten Fall durch.

Für die Praxis ist die "mittlere Suchzeit" eines binären Baums wichtig. Hier gibt es zwei verschiedene Ansätze. Ansatz 1:

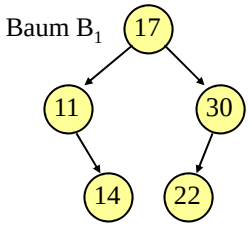
Definition 8.2.17: Man betrachte alle C_n binären Bäume mit n Knoten ($n > 0$). Für jeden Baum B mit n Knoten $v_1, ..., v_n$ berechne man das mittlere Level (oder die mittlere Suchzeit) *ml*:

$$ml(B) = \frac{1}{n} \sum_{i=1}^{n} \text{level}(v_i)$$

und ermittle hiermit das mittlere Level ML_n aller binären Bäume mit n Knoten:

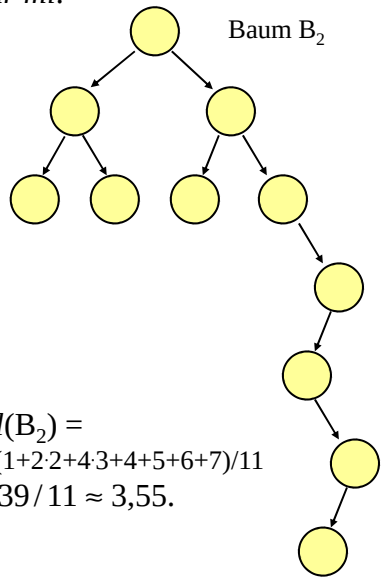
$$ML_n = \frac{1}{C_n} \sum_{\substack{B \text{ ist binärer} \\ \text{Baum mit n} \\ \text{Knoten}}} ml(B)$$

Zwei Binärbäume als Beispiele für ml:



$ml(B_1)$
$= (1+2+2+3+3)/5$
$= 11/5 = 2{,}2$

ml ist unabhängig von den Inhalten in den Knoten.



$ml(B_2) =$
$= (1+2\cdot2+4\cdot3+4+5+6+7)/11$
$= 39/11 \approx 3{,}55.$

Machen Sie sich die Definition von ML_n genau klar: Unter der Annahme, dass alle C_n binären Bäume mit n Knoten gleichwahrscheinlich sind, ist ML_n deren mittlere Suchzeit.

Beispiel (selbst nachrechnen: Listen Sie alle binären Bäume für n = 1, ..., 5 auf, vgl. Auflistung bis n = 4 in 8.2.6):

$ML_1 = 1/1 = 1$

$ML_2 = (1/2)\cdot((1+2)/2 + (1+2)/2) = 1{,}5$

$ML_3 = (1/5)\cdot((1+2+2)/3 + 4\cdot(1+2+3)/3) = 29/15 = 1{,}9333...$

$ML_4 = (1/14)\cdot(80+32+18)/4 = 130/56 = 2{,}3214...$

$ML_5 = (1/42)\cdot(562/5) = 2{,}67619...$

Man kann beweisen, dass ML_n mit $O(\sqrt{n})$ wächst.
(Der Beweis ist relativ aufwendig, siehe Literatur.)

Wir kommen zum Ansatz 2:

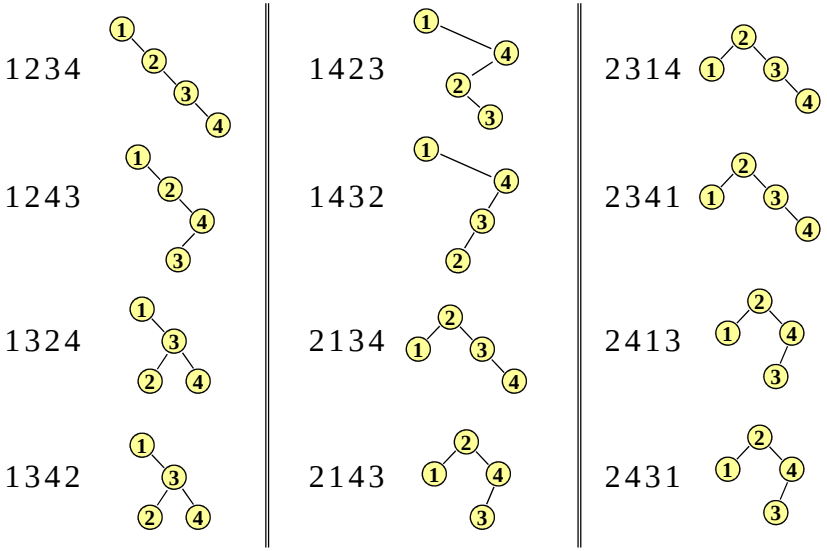
Definition 8.2.18: Gesucht wird die mittlere Suchzeit aller Bäume, die zu allen Folgen (= zu allen Permutationen, siehe 6.1.7) von n Elementen gehören. Das heißt: Man gehe von den n! Folgen mit n Elementen aus, ordne jeder Folge a entsprechend ihrer Reihenfolge einen binären Suchbaum B_a zu und frage nach der mittleren Suchdauer dieser n! Bäume.

Es sei also $a = a_1 a_2 ... a_n$ eine Folge aus n Elementen. Man konstruiere hierzu den binären Suchbaum B_a, indem man den INSERT-Algorithmus 8.2.12 auf $a_1, a_2, ..., a_n$ anwendet. Dann ist die mittlere Suchzeit *MS*:

$$MS_n = \frac{1}{n!} \sum_{\pi(a) \text{ ist eine Permutation der Folge a}} ml(B_{\pi(a)})$$

Die Folge a ist irrelevant. Wichtig ist nur, dass sie aus n Elementen besteht.

Beispiel für n = 4: Es sei a = 1234. Neben jede Permutation $\pi(a)$ wird der zugehörige Baum $B_{\pi(a)}$ gezeichnet:



Die Bäume zu den restlichen 12 Permutationen 3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312 und 4321 mögen Sie selbst zusammenstellen.

Hieraus lässt sich nun MS_4 leicht berechnen:

$MS_4 = (1/24)\ (10/4 + 10/4 + 9/4 + 9/4 + 10/4 + 10/4$
$+ 8/4 + 8/4 + 8/4 + 8/4 + 8/4 + 8/4 + ...\ )$
$= 212/96 = 2{,}20833...$

MS_4 ist kleiner als ML_4. Dies ist kein Zufall, sondern es gilt stets $MS_n \le ML_n$.

Übungsaufgabe: Diese Aussage kann man anschaulich leicht einsehen; überlegen Sie sich, wie. (Notfalls finden Sie später Hinweise in 8.7.5 und in 8.7.7.)

8.2.19 Problem: Mit welcher mittleren Suchzeit MS_n muss man rechnen, wenn man einen binären Suchbaum aus einer zufälligen Folge mit n Elementen erzeugt?

Bevor Sie weiterlesen: Versuchen Sie eine Rekursionsformel aufzustellen, um diese Suchzeit zu beschreiben. Im Folgenden wird eine Formel und deren Herleitung angegeben.

Lösungsansatz: Wir betrachten nur Binärbäume. Es sei F(n) die mittlere Suchzeit, um nach allen n Knoten eines Binärbaums zu suchen. Die mittlere Suchdauer nach einem einzigen Knoten ist dann $MS_n = F(n)/n$. Es gilt F(0)=0 und F(1)=1.

Betrachte eine beliebige Folge mit n > 1 Elementen und den zugehörigen Binärbaum. Dieser besitzt dann eine Wurzel, die einen linken Unterbaum mit i-1 Knoten und einen rechten Unterbaum mit n-i Knoten besitzt für ein i mit $1 \le i \le n$. Dieses i besagt, dass am Anfang der beliebigen Folge das Element steht, welches in der sortierten Reihenfolge an der Position i stehen würde. Betrachtet man alle möglichen Folgen und kommt jede Folge mit der gleichen Wahrscheinlichkeit vor, dann kommt auch jedes i mit der gleichen Wahrscheinlichkeit vor (nämlich 1/n). Wir können daher annehmen, dass jedes i gleichwahrscheinlich ist. Dann erhält man:

$$F(n) = (1/n)\cdot\big((F(0) + F(n-1) + \text{Erhöhung der Pfadlängen}) + (F(1) + F(n-2) + \text{Erhöhung der Pfadlängen}) + \ldots + (F(n-1) + F(0) + \text{Erhöhung der Pfadlängen})\big)$$

Die "Erhöhung der Pfadlängen" berücksichtigt die Verlängerung aller Pfade durch die Wurzel. Diese Erhöhung ist aber für *jeden* Knoten 1, also für n Knoten ist sie n, d.h.:
"Erhöhung der Pfadlängen" = n.

Somit erhalten wir die Formeln: F(0) = 0 und F(1) = 1 und

$$F(n) = (1/n)\cdot\big((F(0) + F(n-1) + n) + (F(1) + F(n-2) + n) + \ldots + (F(n-1) + F(0) + n)\big)$$
$$= (1/n)\cdot 2\cdot(F(0) + F(1) + F(2) + \ldots + F(n-1)) + n$$

Folglich gilt auch

$$F(n-1) = (1/(n-1))\cdot 2\cdot(F(0) + F(1) + F(2) + \ldots + F(n-2)) + n-1$$
$$= (n/(n-1))\cdot(1/n)\cdot 2\cdot\big(F(0) + F(1) + F(2) + \ldots + F(n-2)\big) + n-1$$

In die Formel für F(n) setzen wir den Wert
$(1/n)\cdot 2\cdot\big(F(0) + F(1) + F(2) + \ldots + F(n-2)\big) = (n-1)/n\cdot(F(n-1) - n+1)$
ein und erhalten die Rekursionsformel:

$$F(n) = (1/n)\cdot 2\cdot F(n-1) + (n-1)/n\cdot(F(n-1) - n+1) + n$$
$$= (n+1)/n\cdot(F(n-1) - (n-1)^2/n + n$$
$$= (n+1)/n\cdot F(n-1) + (2n-1)/n$$

Für Interessierte: Wie findet man Lösungen für solche Gleichungen? Siehe hierzu 8.9.1.

Hiermit kann man die mittlere Suchzeit F(n)/n, die man für zufällig aufgebaute Binärbäume erwarten muss, bereits berechnen:

F(2)/2 = (3/2·F(1) + 3/2)/2 = 3/2 = 1,5
F(3)/3 = (4/3·F(2) + 5/3)/3 = 17/9 = 1,8888...
F(4)/4 = (5/4·F(3) + 7/4)/4 = 106/48 = 2,20833...

Um eine geschlossene Formel für die exakte Lösung zu erhalten, probiert man einige Umformungen aus, zum Beispiel vereinfacht man die Formel durch Division durch (n+1); anschließend setzt man F(n-1) und danach F(n-2), F(n-3) usw. ein, bis man eine geschlossene Darstellung erhält.
F(n) = (n+1)/n·F(n-1) + (2n-1)/n wird also umgewandelt in

$$\frac{F(n)}{n+1} = \frac{2n-1}{n\cdot(n+1)} + \frac{F(n-1)}{n}$$

Wir ersetzen F(n-1)/n nun mit dieser Formel:

$$\begin{aligned}\frac{F(n)}{n+1} &= \frac{2n-1}{n\cdot(n+1)} + \frac{F(n-1)}{n}\\ &= \frac{2n-1}{n\cdot(n+1)} + \frac{2n-3}{(n-1)\cdot n} + \frac{F(n-2)}{n-1}\\ &= \frac{2n-1}{n\cdot(n+1)} + \frac{2n-3}{(n-1)\cdot n} + \frac{2n-5}{(n-2)\cdot(n-1)} + \frac{F(n-3)}{n-2}\end{aligned}$$

usw. So erhält man folgende Summenformel (F(0)=0):

$$\begin{aligned}\frac{F(n)}{n+1} &= \sum_{i=0}^{n-1}\frac{2(n-i)-1}{(n-i)\cdot(n-i+1)} + \frac{F(0)}{1}\\ &= \sum_{i=0}^{n-1}\frac{2}{(n-i+1)} - \sum_{i=0}^{n-1}\frac{1}{(n-i)\cdot(n-i+1)}\end{aligned}$$

$$\begin{aligned}\frac{F(n)}{n+1} &= \sum_{i=0}^{n-1}\frac{2}{(n-i+1)} - \sum_{i=0}^{n-1}\frac{1}{(n-i)\cdot(n-i+1)}\\ &= 2\cdot\sum_{i=2}^{n+1}\frac{1}{i} - \sum_{i=1}^{n}\frac{1}{i\cdot(i+1)}\\ &= 2\cdot\big(H(n+1)-1\big) - \sum_{i=1}^{n}\Big(\frac{1}{i}-\frac{1}{i+1}\Big)\\ &= 2\cdot H(n+1) - 2 - 1 + \frac{1}{n+1}\end{aligned}$$

mit $H(n) = \sum_{i=1}^{n}\frac{1}{i}$

$$F(n) = 2\cdot(n+1)\cdot H(n+1) - 3\cdot(n+1) + 1 = 2\cdot(n+1)\cdot H(n) - 3\cdot n$$

Definition 8.2.20: H(n) heißt "harmonische Funktion".
Hierbei wird H(0) = 0 gesetzt.
(Eigenschaften der Funktion H siehe 1.5.2, Illustration in 8.9.2.)

Lösung: $F(n) = 2\cdot(n+1)\cdot H(n) - 3\cdot n$

Aus der Analysis ist bekannt: $H(n) - \ln(n) \longrightarrow C$ für $n \to \infty$.
ln ist der natürliche Logarithmus (zur Basis e = 2,7182818284...),
C = 0,5772156649... ist die Eulersche Konstante. Einsetzen ergibt:

$$F(n) \approx 2\cdot(n+1)\cdot(\ln(n) + C) - 3\cdot n$$

(Hier ist $\log_2$ der reellwertige Zweierlogarithmus.)

$$\approx 2\cdot(n+1)\cdot\log_2(n)/\log_2(e) + 2\cdot(n+1)\cdot C - 3\cdot n$$

$$\approx 1{,}3863\cdot n\cdot\log_2(n) - 1{,}8456\cdot n + O(\log_2(n)).$$

wegen $2 / \log_2(e) \approx 1{,}3863$ und $2\cdot C - 3 \approx 1{,}8456$.

$MS_n = F(n)/n$ ist die gesuchte mittlere Suchzeit. Wir haben also bewiesen: $MS_n = 2\cdot(n+1)/n\cdot H(n) - 3$, näherungsweise ist dies:

Satz 8.2.21: Die mittlere Suchzeit MS_n

Die mittlere Suchzeit in einem Binärbaum mit n Knoten, der zufällig aus einer Folge von n Elementen aufgebaut wurde, beträgt $\approx 1{,}3863 \cdot \log_2(n) - 1{,}8456$. Sie ist um rund 39% schlechter als die mittlere Suchzeit im besten Fall.

Mittlere Suchzeit im besten Fall $\approx \log(n+1) - 1$, siehe 8.2.16.

Wegen $0 \leq \log(x) - \log_2(x) \leq 1.0$ gilt die Näherung auch, wenn man den diskreten Zweierlogarithmus log verwendet, wobei das konstante Glied 1,8456 im konkreten Fall um bis zu 1,3863 kleiner sein kann.

8.2.22: Zeitkomplexität des Sortieralgorithmus Baumsortieren (⇒ 3.7.7): Sortiere n Elemente, indem sie nacheinander in einen Suchbaum eingefügt und anschließend mit einem Inorder-Durchlauf ausgelesen werden.

Nach Satz 8.2.21 benötigt dieses Verfahren im Mittel (average case) $1{,}3863 \cdot n \cdot \log(n) + O(n)$ Schritte. Im schlechtesten Fall kann allerdings ein zu einer Liste entarteter Suchbaum entstehen, so dass das Verfahren im worst case $O(n^2)$ Schritte braucht.

Hinweis: Da Quicksort (siehe 7.3.3) ein Verfahren ist, welches zufällig einen binären Suchbaum erzeugt (das erste Pivot-Element wird die Wurzel dieses Baumes, danach rekursiv links und rechts weitermachen), ist $1{,}3863 \cdot n \cdot \log(n) - 1{,}8456 \cdot n + O(\log(n))$ zugleich die mittlere Zeitkomplexität für Quicksort (wir kommen hierauf in Kapitel 10 zurück).

Haben wir nun wirklich bewiesen, dass beim Einfügen und Löschen stets Binärbäume entstehen, deren mittlere Suchdauer O(log(n)) ist? Nein, *wenn* wir im Verfahren beim Löschen stets den "Inorder-Nachfolger" auswählen würden, also einen Knoten, der immer im rechten Unterbaum liegt!

Dann ist nach einer längeren Folge von Einfüge- und Löschoperationen damit zu rechnen, dass immer mehr Knoten mit großen Inhalten nach oben wandern und somit die entstehenden Binärbäume ständig "links-lastiger" werden! (Umgekehrt würde bei ständiger Wahl des Inorder-Vorgängers ein immer rechtslastigerer Baum entstehen.)

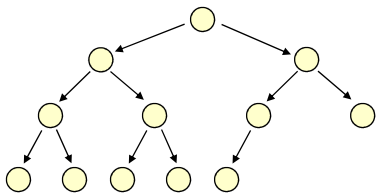
Dies tritt in der Praxis auch tatsächlich ein. Es konnte sogar theoretisch gezeigt werden, dass nach etwa n^2 Einfüge- und Löschoperationen die mittlere Suchzeit bereits in der Größenordnung von "Wurzel(n)", also weit über 1,3863 log(n) liegt, wenn man stets nur den Inorder-Nachfolger beim Löschen verwendet. Man bleibt jedoch in der Größenordnung 1,3863 log(n), wenn man beim Löschen zufällig den Inorder-Vorgänger oder den Inorder-Nachfolger auswählt und so die "Links-Rechts-Lastigkeit" vermeidet. (⇒ Buch von Ottmann/Widmayer, dort Abschnitt 5.1.3.)

Kann man den Suchbaum so einschränken, dass Entartungen möglichst nicht eintreten können und auch im schlechtesten Fall die mittlere Suchdauer in O(log(n)) liegt? Ja, siehe später AVL-Bäume oder B-Bäume.

Doch zunächst wollen wir uns den "besten" Suchbäumen zuwenden.

8.3 Optimale Suchbäume ("optimal" für die Operation FIND)

Eine geordnete Folge von Elementen $a_1, a_2, a_3, ..., a_n$ (mit $a_i \leq a_j$ für $i < j$) wird oft als Binärbaum gespeichert. Unter den C_n möglichen Binärbäumen wird man denjenigen wählen, in dem man jedes Element möglichst schnell finden kann. Wird nach allen Elementen mit gleicher Wahrscheinlichkeit gesucht, so wird man einen möglichst gleichförmigen Baum nehmen, z.B.:



Hier haben alle Knoten zwei Nachfolger, außer den Knoten mit Level log(n) oder log(n)-1.

Welchen Suchbaum würde man wählen, wenn nach verschiedenen Elementen verschieden häufig gesucht wird?

Wir nehmen also an, dass nach jedem Element a_i im Baum mit der Wahrscheinlichkeit (oder der Häufigkeit) p_i gesucht wird. Dann werden selbstverständlich *den* Binärbaum auswählen, für den die mittlere Suchdauer, d.h., die Summe der Suchzeiten gewichtet mit den Häufigkeiten, minimal ist.

Definition 8.3.1: Gegeben sind eine geordnete Folge von n Elementen $(a_1, a_2, a_3, ..., a_n)$, mit $a_i \le a_j$ für $i < j$, mit zugehörigen Häufigkeiten oder Wahrscheinlichkeiten $p_1, p_2, p_3, ..., p_n$ (insbesondere sind alle $p_i \ge 0$) sowie ein binärer Suchbaum B mit n Knoten $v_1, ..., v_n$, wobei a_i der Inhalt des Knotens v_i ist.

Die gewichtete mittlere Suchdauer S(B) von B ist definiert als

$$S(B) = \sum_{i=1}^{n} p_i \cdot \text{level}(v_i) .$$

Die Werte p_i können Häufigkeiten sein. Für ein recht großes p führt man p Anfragen durch und zählt hierbei, wie oft nach jedem Element a_i gefragt wurde; diese Anzahl sei jeweils p_i. Es gilt $p_1 + p_2 + p_3 + ... + p_n = p$.

Man kann in der Definition zum einen diese Werte p_i verwenden, man kann aber auch die relativen Häufigkeiten p_i/p benutzen; diese sind eine Näherung für die Wahrscheinlichkeit, dass nach dem i-ten Element a_i gesucht wird. Wir brauchen später von den Werten p_i nur zu wissen, dass $p_i \ge 0$ ist.

Das folgende Lösungsverfahren arbeitet sowohl mit Häufigkeiten als auch mit Wahrscheinlichkeiten korrekt.

Hinweis: Die in 8.2.17 bereits definierte mittlere Suchzeit *ml*(B) eines Baumes B ist die gewichtete mittlere Suchdauer S(B) für den Fall, dass alle Elemente die gleiche Wahrscheinlichkeit $p_i = 1/n$ besitzen.

Wenn die p_i Wahrscheinlichkeiten sind, so gibt S(B) die zu erwartende Anzahl der Vergleiche an, um irgendein beliebiges Element im Suchbaum B zu finden.

Hierbei suchen wir zunächst nur nach Elementen a_j, die in der ursprünglichen Folge $(a_1, a_2, a_3, ..., a_n)$ vorkommen. Wird auch nach Elementen gesucht, die nicht zu den a_i gehören, so muss man die Bäume leicht abändern; dies wird am Ende dieses Abschnitts 8.3 im Hinweis 4 erläutert.

Erinnerung an 8.2.9:

Gegeben sei eine sortierte Folge $a_1, a_2, a_3, ..., a_n$. Dann gibt es zu jedem binären Baum B mit n Knoten genau eine Bijektion $f: \{a_1, a_2, a_3, ..., a_n\} \to \{v_1, v_2, v_3, ..., v_n\}$ der Folgenelemente a_i zu den Knoten v_j, so dass B hierdurch zu einem Suchbaum wird (und folglich der Inorder-Durchlauf von B die sortierte Folge $a_1, a_2, a_3, ..., a_n$ liefert). f ergibt sich, indem man B inorder durchläuft und dem i-ten Knoten v_i in dieser Reihenfolge das Element $a_i = f^{-1}(v_i)$ als Inhalt zuordnet.

Jeder solche binäre Baum B mit der Zuordnung f heißt ein zur Folge $a_1, a_2, a_3, ..., a_n$ gehörender Suchbaum. Zu jeder Folge gehören genau C_n solche Suchbäume.

Ein Suchbaum, dessen mittlere Suchdauer unter allen zur Folge gehörenden Suchbäumen minimal ist, heißt optimal. Formal:

Definition 8.3.2: Gegeben sei eine geordnete Folge von n Elementen $a_1, a_2, a_3, ..., a_n$ (mit $a_i \le a_j$ für $i < j$) mit ihren Wahrscheinlichkeiten $p_1, p_2, p_3, ..., p_n$ ($p_i \ge 0$ und $p_1+p_2+...+p_n = 1$) oder Häufigkeiten $p_1, p_2, p_3, ..., p_n$ ($p_i \ge 0$).

Ein Suchbaum B mit n Knoten $v_1, ..., v_n$, wobei a_i der Inhalt des Knotens v_i ist, heißt optimaler Suchbaum (zur Folge $p_1, p_2, p_3, ..., p_n$), wenn für alle zur Folge $a_1, a_2, a_3, ..., a_n$ gehörenden Suchbäume B' gilt: $S(B) \le S(B')$.

Auch in dieser Definition sind nicht die konkreten Elemente a_i, sondern nur deren Wahrscheinlichkeiten bzw. Häufigkeiten p_i von Bedeutung.

Die Aufgabe lautet nun, zu n und $p_1, p_2, p_3, ..., p_n$ einen optimalen Suchbaum zu konstruieren. Da es C_n mögliche Suchbäume gibt (Satz 8.2.7), dauert das systematische Durchprobieren viel zu lange.

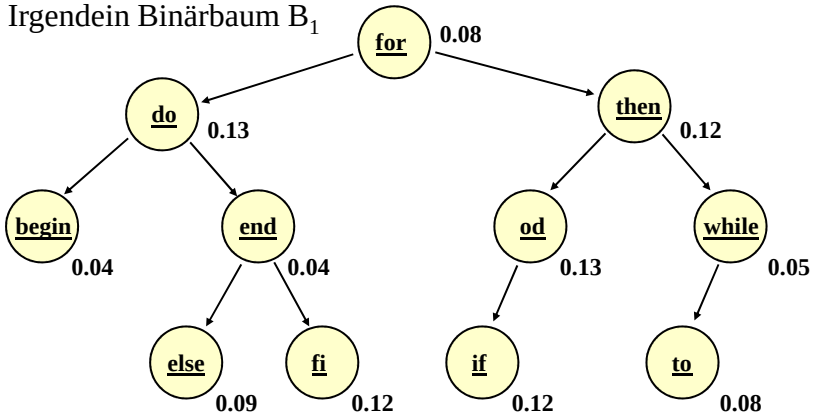
8.3.3 Beispiel

Als Beispiel betrachten wir einen Compiler, der ein Programm übersetzen soll. Hierzu muss er zunächst die Schlüsselwörter der Sprache erkennen. Diese Wörter treten mit gewissen Wahrscheinlichkeiten in einem Programm auf und wir nehmen an, wir hätten diese Wahrscheinlichkeiten gemessen. Wir verwenden hier die Folge aus n=11 Wörtern begin, do, else, end, fi, for, if, od, then, to, while. Ihre Wahrscheinlichkeiten seien

begin: 0.04, do: 0.13, else: 0.09, end: 0.04, fi: 0.12, for: 0.08, if: 0.12, od: 0.13, then: 0.12, to: 0.08, while: 0.08.

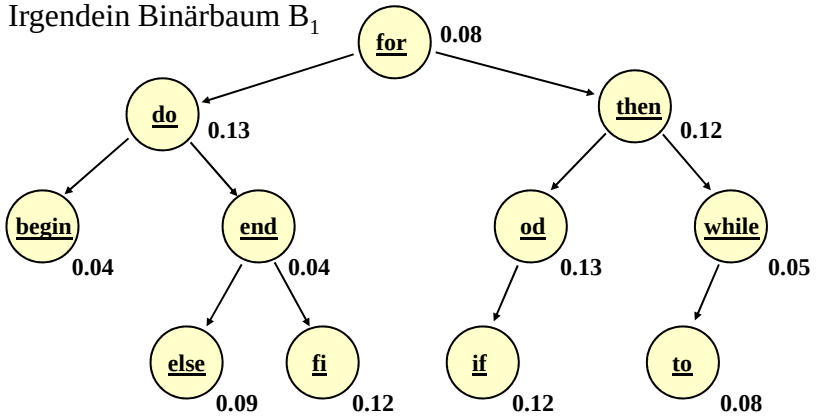
Wir fügen die 11 Folgenelemente zunächst in einen beliebigen Binärbaum B_1 ein und berechnen dessen gewichtete mittlere Suchdauer $S(B_1)$.

Irgendein Binärbaum B_1



Schlüsselwörter eintragen (inorder-Durchlauf)
Wahrscheinlichkeiten hinzufügen
Gewichtete mittlere Suchdauer nun ausrechnen (bitte selbst durchführen, danach zur nächsten Folie wechseln).

Irgendein Binärbaum B_1

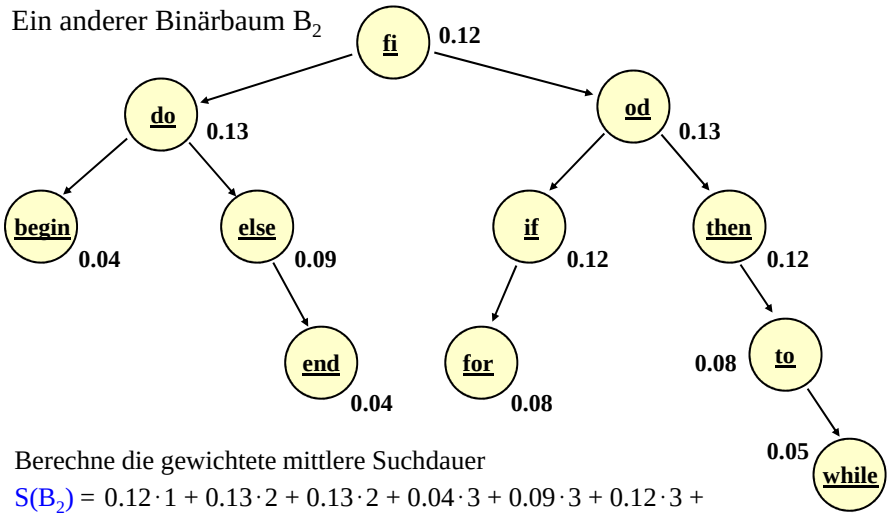


Berechne die gewichtete mittlere Suchdauer

$S(B_1) = 0.08 \cdot 1 + 0.13 \cdot 2 + 0.12 \cdot 2 + 0.04 \cdot 3 + 0.04 \cdot 3 + 0.13 \cdot 3 + 0.05 \cdot 3 + 0.09 \cdot 4 + 0.12 \cdot 4 + 0.12 \cdot 4 + 0.08 \cdot 4 = 3.00$

Konstruieren Sie nun einen Suchbaum mit kleinerem S(B). [Man sieht sofort, dass man to ein Level hinauf und while eines hinab schieben sollte, usw.]

Ein anderer Binärbaum B_2

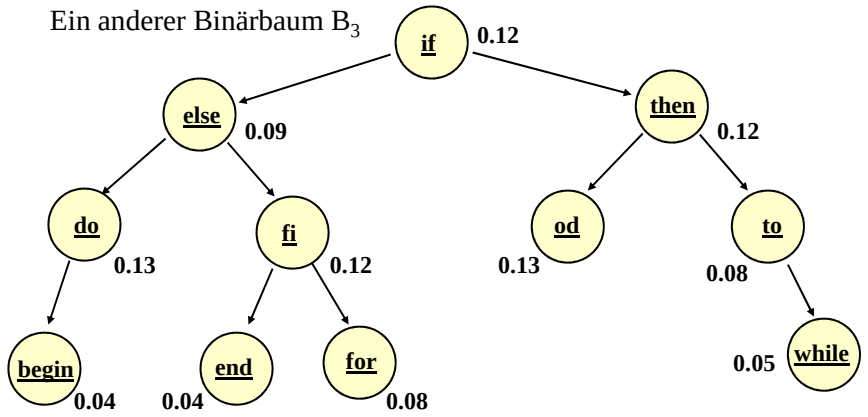


Berechne die gewichtete mittlere Suchdauer

$S(B_2) = 0.12 \cdot 1 + 0.13 \cdot 2 + 0.13 \cdot 2 + 0.04 \cdot 3 + 0.09 \cdot 3 + 0.12 \cdot 3 + 0.12 \cdot 3 + 0.04 \cdot 4 + 0.08 \cdot 4 + 0.08 \cdot 4 + 0.05 \cdot 5 = 2.80$

Konstruieren Sie nun weitere Suchbäume mit kleinerem S(B). Versuchen Sie, den optimalen Suchbaum zu finden. Erkennen Sie hierbei ein Verfahren?

Ein anderer Binärbaum B_3



Berechnen Sie selbst die gewichtete mittlere Suchdauer dieses Suchbaums. Ist dieser Baum B_3 besser als B_2? Gibt es bessere? Welche? Sollte man das mittlere Element der Folge (hier: for) möglichst in die Wurzel setzen? Oder "od" mit der hohen Wahrscheinlichkeit 0.13? (Lösung siehe 8.9.3.) ■

Hilfssatz 8.3.4:

Jeder Unterbaum eines optimalen Suchbaums ist ebenfalls ein optimaler Suchbaum.

Anschaulich ist dies "klar". Formaler *Beweis:* Wäre der Unterbaum U eines optimalen Suchbaums B nicht optimal, so gäbe es einen anderen Suchbaum U', der bzgl. der in U enthaltenen Elemente optimal ist, für den insbesondere S(U') < S(U) gilt. Tausche dann U gegen U' im Baum B aus, wodurch der Baum B' entsteht. Es gilt dann ($a_i \in$ U bezeichnen die Elemente, die im Unterbaum U liegen; x+1 sei das Level der Wurzel von U in B; in U und U' liegen natürlich die gleichen Elemente):

$$S(B) = \sum_{i=1}^{n} p_i \cdot level(v_i) = \sum_{a_i \in B-U} p_i \cdot level(v_i) + \sum_{a_i \in U} p_i \cdot level(v_i)$$

$$S(B) = \sum_{a_i \in B-U} p_i \cdot level(v_i) + \overbrace{\sum_{a_i \in U} p_i \cdot (level(v_i) - x)}^{S(U)} + \sum_{a_i \in U} p_i \cdot x$$

Hierbei ist x+1 das Level der Wurzel des Unterbaums U in B; dies ist zugleich das Level der Wurzel des Unterbaums U' in B'.

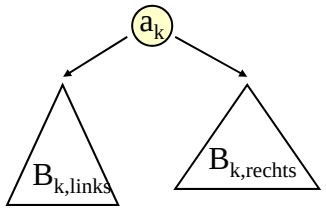
$$S(B) = \sum_{a_i \in B-U} p_i \cdot level(v_i) + S(U) + \sum_{a_i \in U} p_i \cdot x$$

$$> \sum_{a_i \in B'-U'} p_i \cdot level(v'_i) + S(U') + \sum_{a_i \in U'} p_i \cdot x$$

$$= \sum_{a_i \in B'-U'} p_i \cdot level(v'_i) + \sum_{a_i \in U'} p_i \cdot (level(v'_i) - x) + \sum_{a_i \in U'} p_i \cdot x = S(B')$$

Also war B kein optimaler Suchbaum, im Widerspruch zur Voraussetzung. Folglich muss U optimal gewesen sein. ■

Hieraus folgt, dass man einen optimalen Suchbaum schrittweise aus seinen (optimalen) Unterbäumen aufbauen kann.
Um den optimalen Suchbaum für $a_i, a_{i+1}, ..., a_{j-1}, a_j$ zu finden, berechnet man die mittlere Suchdauer für alle Paare von Unterbäumen zu $a_i, a_{i+1}, ..., a_{k-1}$ und $a_{k+1}, a_{i+2}, ..., a_j$ und wählt die Kombination aus, deren Summe den kleinsten Wert ergibt.
Skizze:



Der beste Fall liegt vor, wenn $S(B_{k,links}) + S(B_{k,rechts})$ minimal ist für ein $i \le k \le j$.

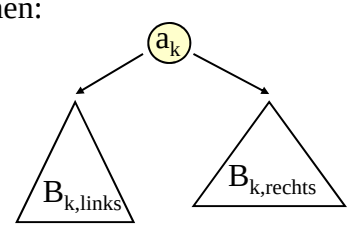
Dieser Baum enthält genau die Elemente $a_i, a_{i+1}, ..., a_{j-1}, a_j$.

8.3.5 Bezeichnungen und Formeln: Gegeben seien n und die Häufigkeiten P[1], P[2], ..., P[n]. Sei $1 \le i \le j \le n$.

Es sei G[i,j] = P[i] + P[i+1] + ... + P[j] (man bezeichnet diesen Wert auch als das "Gewicht" der Teilfolge a_i bis a_j).
Mit S[i,j] bezeichnen wir die gewichtete mittlere Suchdauer für einen optimalen Suchbaum für die Elemente $a_i, a_{i+1}, ..., a_{j-1}, a_j$.
S[i,j] lässt sich leicht rekursiv berechnen:

S[i,i] = P[i] für i = 1, 2, ..., n.

Sei i < j. Dann ist S[i,j] gleich S[i,k-1] + S[k+1,j] plus der Erhöhung aller Level um 1 (dieser Anteil ist genau G[i,j]):



$S[i,j] = Min\{S[i,k-1] + S[k+1,j] \mid i \le k \le j\} + G[i,j]$ für i < j.

S[i,j] lässt sich berechnen, wenn alle S[r,s] mit s-r < j-i bekannt sind. Setze also diff := j - i und berechne für diff = 0, 1, 2, ..., n-1 und i=1,..., n-diff alle Werte S[i, i + diff].
Für diff = 0 setze S[i,i] = P[i]. Für diff > 0 verwende die Rekursionsformel für S[i,j] mit j = i + diff.
Zeitaufwand: Drei ineinander geschachtelte Schleifen:

```
for diff in 1..n-1 loop
    for i in 1..n-diff loop
        j := i + diff;
        for k in i..j loop berechne das Minimum aller Werte
            S[i,k-1] + S[k+1,j]; end loop;
        setze S[i,j] entsprechend;
    end loop;
end loop;
```

Das gesuchte Ergebnis S(B) steht am Ende in S[1,n]. Da alle Schleifen linear von n abhängen, beträgt der Aufwand $\Theta(n^3)$.

Erläuterungen zum folgenden Programm im Pseudocode:

Wir verwenden G[i,j] und S[i,j] wie angegeben, wobei wir rund 50% des Speicherplatzes verschwenden, da wir diese Werte nur für $i \le j$ brauchen.
Weiterhin berechnen wir in diesem Programm auch die Wurzeln der Unterbäume zu a_i bis a_j. Diese legen wir in einem Feld R ab, wobei gilt
R[i,j] = k $\Leftrightarrow$ a_k steht in der Wurzel des optimalen Suchbaums von $a_i, a_{i+1}, ..., a_j$. (Dieses k wird im Programm in der innersten Schleife als k_min berechnet.)
Mit den R[i,j] kann man am Ende den optimalen Suchbaum rekonstruieren (dies programmieren wir aber nicht aus).
Die Minimumbildung erfolgt wie üblich, indem man alle Werte durchprobiert und das aktuelle Minimum speichert.
Die Werte G[i,j] kann man zu Beginn des Programms berechnen; wir führen dies jedoch in der mittleren Schleife durch.

8.3.6: Programm für einen optimalen Suchbaum in Zeit $\Theta(n^3)$

Global: Die Zahl n und n Häufigkeiten P[1], ..., P[n].

```
var   i, j, k, k_min, diff: natural;  min: real;
   S, G: array [1..n+1, 1..n] of real; R: array [1..n, 1..n] of natural;
begin
   for i:=1 to n do S[i+1,i] := 0.0; S[i,i] := P[i]; G[i,i] := P[i]; R[i,i]:=i od;
   for diff:=1 to n-1 do
      for i:=1 to n-diff do
         j := i + diff; G[i,j] := G[i,j-1] + P[j]; min := S[i+1,j]; k_min := i;
         for k:=i+1 to j do
            if  S[i,k-1] + S[k+1,j] < min  then
                min := S[i,k-1] + S[k+1,j]; k_min := k  fi
         od;
         S[i,j] := min + G[i,j]; R[i,j] := k_min
      od
   od          -- Der Suchbaum kann aus den R[i,j] konstruiert werden. (Wie?)
end            -- Ada-Programm: selbst programmieren, siehe auch 8.9.3.
```

Ergebnis: S[1,n]

8.3.7: Hinweise

Hinweis 1: Dieses ist ein "garantiertes" n^3-Verfahren. In der Praxis ist es deshalb nur für kleinere Werte von n einsetzbar.

Hinweis 2: Das Suchen (FIND) lässt sich optimal schnell durchführen. Dagegen muss man beim Einfügen (INSERT) und beim Löschen (DELETE) den Baum neu aufbauen. Daher setzt man optimale Suchbäume nur dann ein, wenn der Datenbestand sich über längere Zeiträume nicht ändert. Beispiele hierfür sind Lexika oder die Erkennung von Schlüsselwörtern und anderen Textteilen durch einen Compiler.

In zeitkritischen Anwendungen kann man eine Doppel-Strategie verfolgen: Der "große Datenbestand" ist in einem optimalen Suchbaum gespeichert, Löschungen werden nur als gelöscht im Baum markiert, die (seltenen) Neueintragungen speichert man in einem gesonderten Binärbaum, bis dieser eine gewisse Größe erreicht hat; dann baut man aus den beiden Bäumen einen neuen optimalen Suchbaum auf.

Hinweis 3: Man kann nachweisen, dass die Wurzel des jeweils zu konstruierenden Unterbaums innerhalb bestimmter Grenzen liegen muss, die von den im letzten Durchlauf (für diff-1) konstruierten Wurzeln bestimmt werden, genauer:
Es gilt stets $R[i,j-1] \le R[i,j] \le R[i+1,j]$. Dies nennt man die "Monotonie der Wurzeln".
Beweis: siehe Lehrbücher oder weiterführende Vorlesung "Entwurf und Analyse von Algorithmen".
Mit dieser Eigenschaft lässt sich obiges Verfahren zu einem $\Theta(n^2)$-Verfahren beschleunigen. (Selbst durchdenken. Obiges Programm muss nur leicht modifiziert werden.)
Aber auch diese Beschleunigung reicht für die Praxis nicht aus, wenn sich der Datenbestand und/oder die Häufigkeiten oft ändern. In solchen Fällen verwendet man in der Praxis meist AVL-Bäume oder B-Bäume (siehe 8.4 und 8.5).

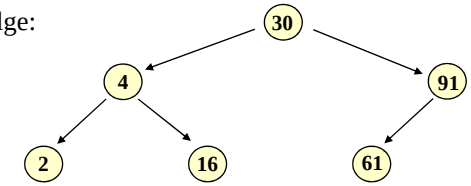
Hinweis 4: Zum Abschluss erläutern wir kurz, wie man dieses Verfahren auf den Fall der erfolglosen Suche erweitert.

Sucht man nach einem Element, das nicht in der Folge $a_1, a_2, a_3, ..., a_n$ vorkommt, so endet die Suche bei einem der n+1 null-Zeiger des Suchbaums.

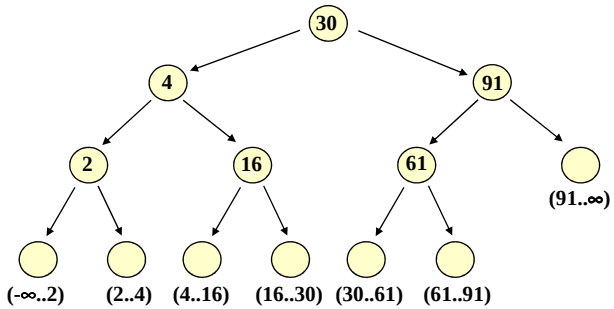
Man ersetzt nun diese null-Zeiger durch Knoten, deren Inhalt alle die Elemente bilden, mit denen man im Suchbaum hierhin gelangt. Ihre Inhalte sind daher offene Intervalle der Form $(i..j) = \{r \mid i < r < j\}$.

Jetzt muss man den Intervallen ebenfalls Häufigkeiten zuordnen, mit denen nach einem Element in ihnen gesucht wird. Auf diese Weise kann man einen optimalen Suchbaum auch für den Fall der "erfolglosen Suche" mit dem gleichen Verfahren konstruieren.

Beispiel: Gegebene Folge: 2, 4, 16, 30, 61, 91 und ein zugehöriger Suchbaum.



Füge an die sieben null-Zeiger neue Blätter an, die die nicht enthaltenen Elemente repräsentieren; diese neuen Blätter werden mit den offenen Intervallen (i..j) beschriftet.



8.4 Balancierte Bäume, AVL-Bäume

Unter der Balance eines Knotens in einem Binärbaum versteht man ein *Verhältnis*, in dem seine beiden Unterbäume zueinander stehen.

Dieses „Verhältnis“ kann sehr verschieden festgelegt werden. Üblich sind zwei Festlegungen:

- Die *Gewichts-Balance* von u gibt die Knotenanzahl eines Unterbaums relativ zum gesamten Unterbaum an.
- Die *Höhen-Balance* von u gibt die Differenz der Höhen (bzw. Tiefen) der beiden Unterbäume von u an.

Entscheidend ist: Liegt die Balance in gewissen Bereichen, dann ist die Tiefe des Baums in O(log(n)). Dadurch wird die Suchzeit stets logarithmisch beschränkt.

Definition 8.4.1:

Es sei u ein Knoten und es seien UB_{links} und UB_{rechts} seine beiden Unterbäume. Es bezeichnen

$|V_{UB_{links}}|$ die *Anzahl der Knoten* im linken Unterbaum und

$|V_{UB_{rechts}}|$ die *Anzahl der Knoten* im rechten Unterbaum von u.

$|V_{UB_{links}}| + |V_{UB_{rechts}}|$ ist die Anzahl aller Knoten, die sich unterhalb des Knotens u befinden.

Definition 8.4.1 (Fortsetzung):

Es sei α eine Zahl aus dem reellen Intervall $(0, ½]$. Ein Knoten u eines Binärbaums heißt α-gewichtsbalanciert, wenn gilt

$$\alpha \le \frac{|V_{UB_{links}}| + 1}{|V_{UB_{links}}| + |V_{UB_{rechts}}| + 2} \le 1 - \alpha$$

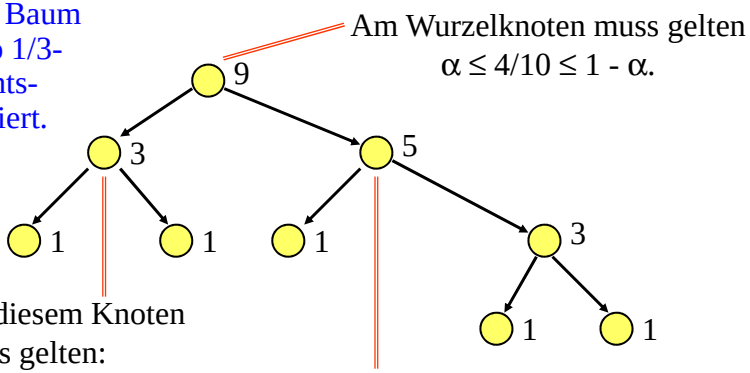
Den Ausdruck $\beta(u) = \frac{|V_{UB_{links}}| + 1}{|V_{UB_{links}}| + |V_{UB_{rechts}}| + 2}$

nennt man auch die Gewichts- oder Wurzelbalance von u.

Die Anzahlen der Knoten in den Unterbäumen werden hier jeweils um 1 erhöht, weil die Unterbäume leer sein können. Wenn ein Knoten α-gewichtsbalanciert ist und $0 \le \alpha' \le \alpha \le ½$ gilt, dann ist der Knoten auch α'-gewichtsbalanciert.

Beispiel 8.4.2: An jedem Knoten sei die Zahl der Knoten in dem Unterbaum, dessen Wurzel er ist, angegeben. Bestimme für folgenden Baum ein geeignetes α.

Dieser Baum ist also 1/3-gewichts-balanciert.



Am Wurzelknoten muss gelten $\alpha \le 4/10 \le 1 - \alpha$.

An diesem Knoten muss gelten: $\alpha \le 2/4 \le 1 - \alpha$.

An diesem Knoten muss gelten: $\alpha \le 2/6 \le 1 - \alpha$.

Blätter sind stets 1/2-gewichts-balanciert.

Folgerung 8.4.3: Symmetrie der Balance

Wenn

$$\alpha \le \frac{|V_{UB_{links}}| + 1}{|V_{UB_{links}}| + |V_{UB_{rechts}}| + 2} \le 1 - \alpha$$

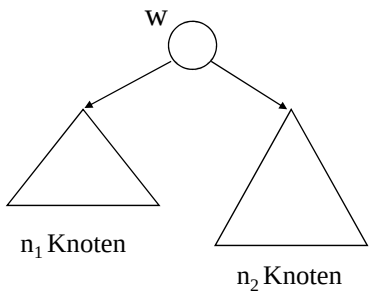
gilt, dann gilt auch

$$\alpha \le \frac{|V_{UB_{rechts}}| + 1}{|V_{UB_{links}}| + |V_{UB_{rechts}}| + 2} \le 1 - \alpha$$

Der Beweis ist einfach: Seien $|V_{UB_{links}}| = n_1$ und $|V_{UB_{rechts}}| = n_2$, dann folgt die Aussage folgt sofort aus

$\alpha \le (n_1+1)/(n_1+n_2+2) = 1- (n_2+1)/(n_1+n_2+2) \le 1 - \alpha$. ■

Betrachte einen Binärbaum B mit n Knoten und der Wurzel w:



Die Gesamtzahl der Knoten im Baum ist $n = n_1 + n_2 + 1$.

Für α-gewichtsbalancierte Bäume B gilt dann an der Wurzel w für i = 1, 2:

$$\alpha \le \frac{n_i + 1}{n + 1} \le 1 - \alpha, \text{ d.h., } n_i \le (1 - \alpha)\cdot(n + 1) - 1 = (1 - \alpha)\cdot n - \alpha$$

Dies gilt auch für das nächste Level, d.h.:

Anzahl der Knoten in einem Unterbaum zwei Level tiefer

$\le (1 - \alpha)\cdot n_i - \alpha$

$\le (1 - \alpha)\cdot((1 - \alpha)\cdot n - \alpha) - \alpha = (1 - \alpha)^2\cdot n - \alpha\cdot((1 - \alpha) + 1)$

Analog weiter einsetzen! Dies ergibt (beachte $\alpha > 0$):

Anzahl der Knoten in einem Unterbaum k Level tiefer

$\le (1-\alpha)^k\cdot n - \alpha\cdot((1-\alpha)^{k-1} + (1-\alpha)^{k-2} + (1-\alpha)^1 + (1-\alpha)^0)$

$= (1-\alpha)^k\cdot n - \alpha\cdot(1 - (1 - \alpha)^k) / (1 - (1 - \alpha))$

$= (1-\alpha)^k\cdot n - (1 - (1 - \alpha)^k)$

$< (1-\alpha)^k\cdot n$

Wenn $(1-\alpha)^k\cdot n < 2$ geworden ist, gibt es höchstens noch einen Knoten und dieser muss dann ein Blatt sein.

Aus $(1-\alpha)^k \cdot n < 2$ folgt mit $z := 1/(1-\alpha)$:

$n < 2 \cdot z^k$, d.h., $\log_2(n/2) < k \cdot \log_2(z) = -k \cdot \log_2(1-\alpha)$

Suche das kleinste k, für das diese Ungleichung zutrifft:
$k = 1 - (\log_2(n) - 1) / \log_2(1-\alpha) \in O(\log(n))$.

Dieses (reelle) k ist eine obere Schranke für die Tiefe des Baums (minus 1). Somit haben wir gezeigt:

Satz 8.4.4
Die Tiefe eines α-gewichtsbalancierten Baums mit n Knoten ist höchstens $2 - (\log(n) - 1) / \log_2(1-\alpha)$,
d.h., die Tiefe liegt stets in $O(\log(n))$.

Je mehr α sich der Zahl 0.5 nähert, um so mehr nähert sich die Tiefe dem minimalen Wert $\lceil \log_2(n+1) \rceil$ (vgl. 8.2.16).

Kann man die Operationen FIND, INSERT und DELETE auf solchen α-gewichtsbalancierten Bäumen so ausführen, dass diese Operationen schnell durchführbar sind (z.B. in O(log(n)) Schritten) und dass nach Ausführung jeder Operation der entstandene Baum wieder ein α-gewichtsbalancierter Baum ist?

Ja, das geht tatsächlich.

Wir führen dies hier jedoch nicht durch, sondern verweisen auf die Literatur über Datenstrukturen.

An Stelle der gewichtsbalancierten Bäume untersuchen wir nun die höhenbalancierten Bäume genauer, die besonders angenehme Eigenschaften haben.

Für das Folgende beachten Sie, dass hier "Höhe" und "Tiefe" synonym verwendet werden (8.2.4). T bezeichnet die Tiefe.

Definition 8.4.5:

Ein binärer Baum heißt **AVL-Baum** (oder **höhenbalancierter** Baum), wenn für jeden Knoten u gilt:

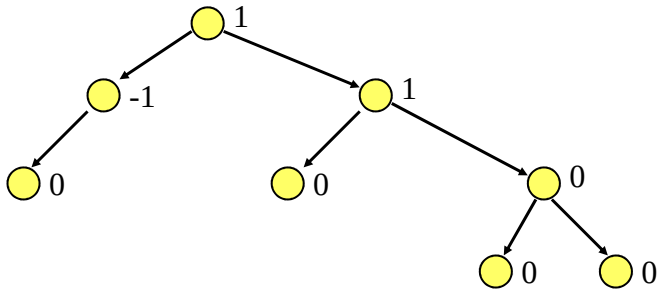
$T(UB_{rechts}) - T(UB_{links}) =$
("Höhe" des rechten Unterbaums von u -
"Höhe" des linken Unterbaums von u) $\in \{-1, 0, 1\}$,

d.h., für jeden Knoten unterscheiden sich die Höhen (= Tiefen) seiner Unterbäume höchstens um 1.

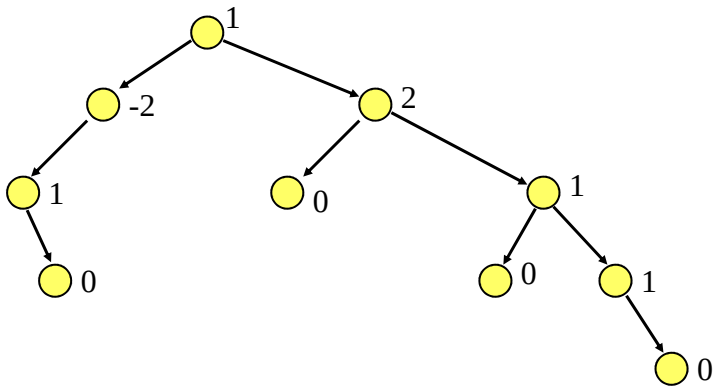
$T(UB_{rechts}) - T(UB_{links})$ heißt die Höhenbalance von u, oft auch Balancefaktor oder kurz Balance von u genannt.

AVL-Bäume wurden benannt nach ihren beiden Erfindern Adelson-Velski und Landis.

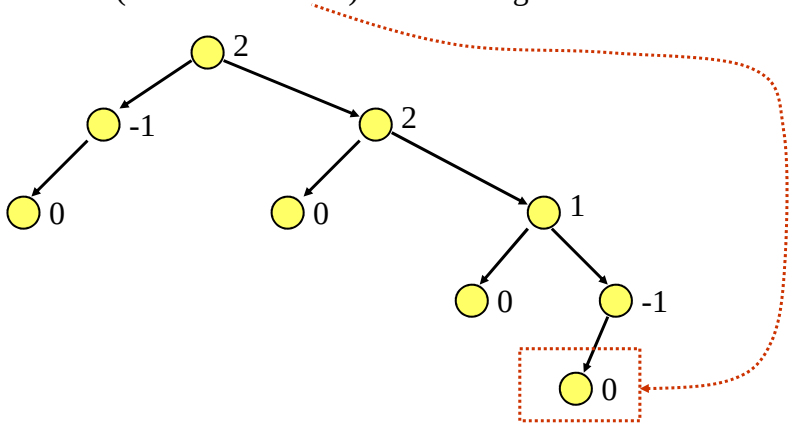
Beispiel: Der bereits im Beispiel 8.4.2 betrachtete Baum ist höhenbalanciert, d.h. ein AVL-Baum. Wir tragen die Höhenbalancen neben jedem Knoten ein:



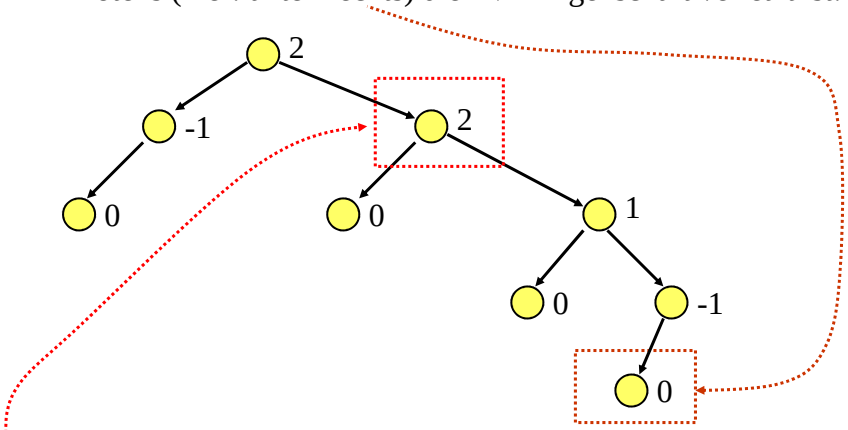
Beispiel: Dagegen ist folgender Baum *nicht* höhenbalanciert:



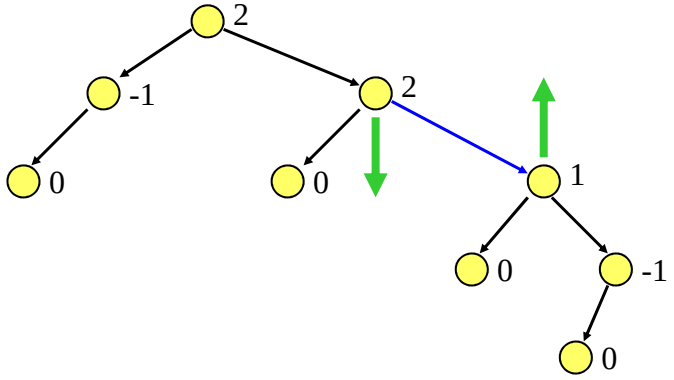
Betrachte nun einen AVL-Baum, in dem nur wegen eines Knotens (hier: unten rechts) die AVL-Eigenschaft verletzt ist:



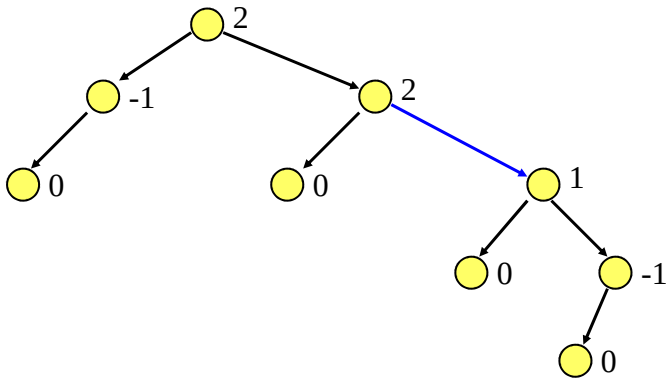
Betrachte nun einen AVL-Baum, in dem nur wegen eines Knotens (hier: unten rechts) die AVL-Eigenschaft verletzt ist:



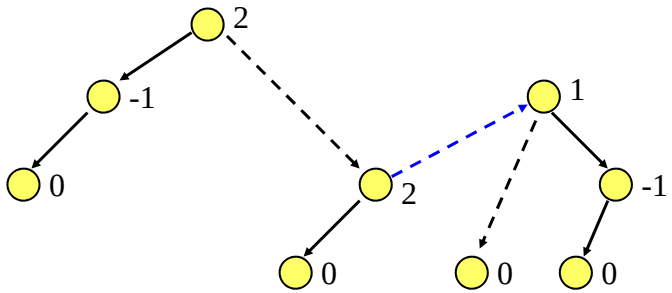
Dann kann man durch eine leichte Korrektur an der untersten Verletzungs-Position die AVL-Eigenschaft wieder herstellen:



Dies nennt man eine "Links-Rotation" (= Drehung der blauen Kante nach links, also gegen den Uhrzeigersinn).

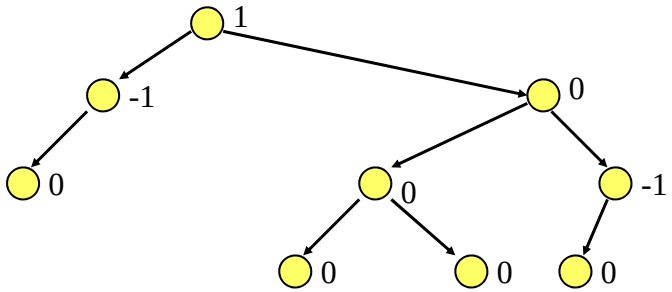


Nur die drei gestrichelten Kanten werden verzerrt.



Nun muss man die drei gestrichelten Kanten neu orientieren, wobei die Suchbaum-Eigenschaft erhalten bleiben muss,

und schon liegt wieder ein AVL-Baum vor:



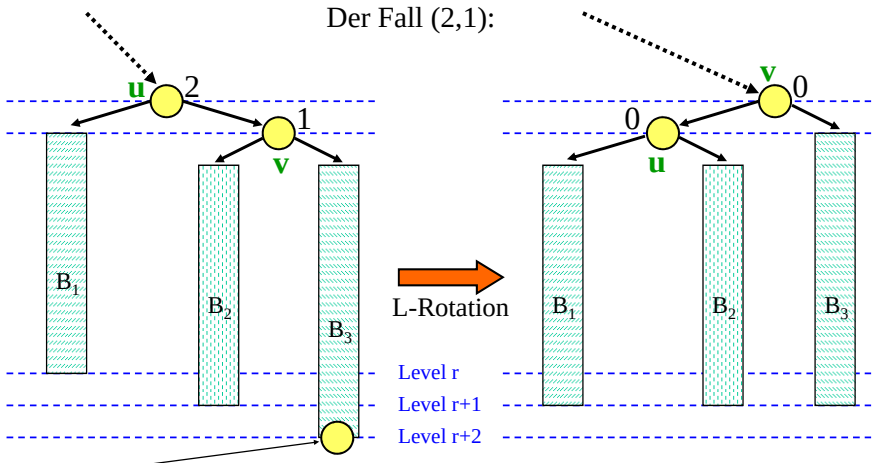
Durch Ausprobieren stellt man fest, dass man mit vier Rotationsarten auskommt, um alle solche Fälle zu korrigieren.

Einfügen in einen AVL-Baum. Dies geschieht wie bei einem beliebigen Suchbaum stets in einem Blatt. Dabei kann durch den neu eingefügten Knoten die AVL-Eigenschaft der Höhenbalancierung verletzt werden.

In diesem Fall führen wir eine Rotation an dem untersten Knoten, der nicht mehr höhenbalanciert ist, durch. Dadurch wird die AVL-Eigenschaft wieder hergestellt (siehe unten).

Wir untersuchen die möglichen Fälle, die zu einer Verletzung führen. Da durch einen Knoten das Level höchstens um 1 steigen kann, muss an einem Knoten u die Balance 2 oder -2 entstehen. Dann muss aber der direkte Nachfolgeknoten von u, der auf dem Weg zum eingefügten Blatt liegt, jetzt 1 oder -1 geworden sein (wäre er 0, so hätten sich die Tiefe dieses Unterbaums un ddie Balance von u nicht geändert). Somit gibt es die vier Fälle (2,1), (2,-1), (-2,1) und (-2,-1).

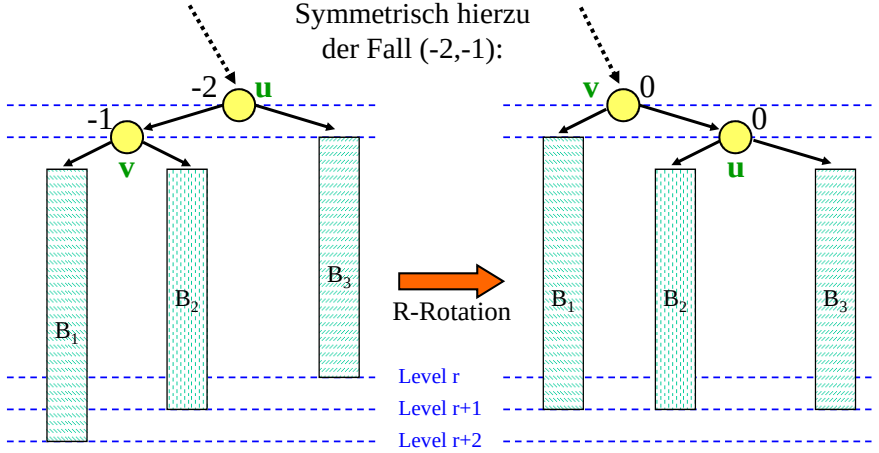
Der Fall (2,1):



Situation an der untersten Verletzungsstelle **u — v**: Balancen 2 und 1.

⇒ Links-Rotation durchführen (3 Zeiger umhängen, Balancen 0 und 0).

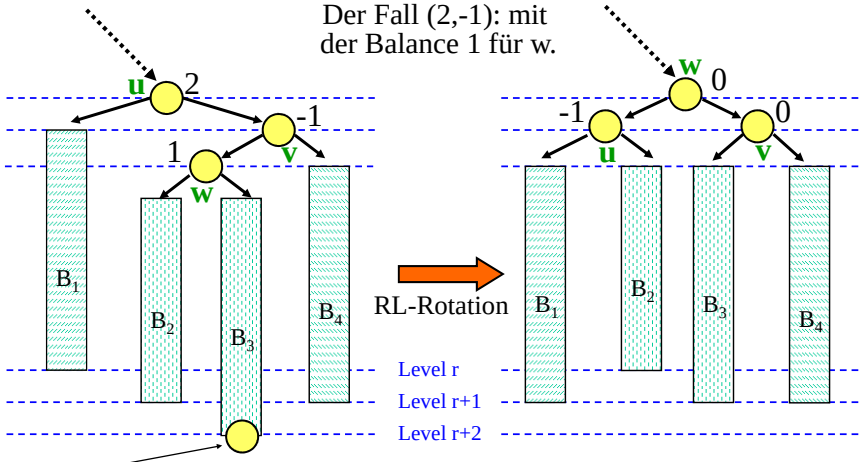
Symmetrisch hierzu der Fall (-2,-1):



Situation an der untersten Verletzungsstelle **u — v**: Balancen -2 und -1.

⇒ Rechts-Rotation durchführen (3 Zeiger umhängen, Balancen 0 und 0).

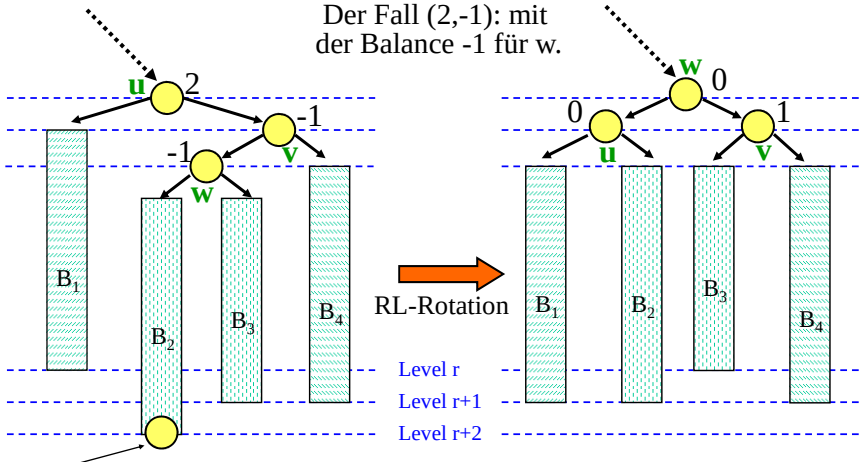
Der Fall (2,-1): mit der Balance 1 für w.



Situation an der untersten Verletzungsstelle **u — v**: Balancen 2 und -1.

⇒ Rechts-Links-Rotation durchführen (4 Zeiger umhängen).

Der Fall (2,-1): mit der Balance -1 für w.



Situation an der untersten Verletzungsstelle **u — v**: Balancen 2 und -1.

⇒ ebenfalls Rechts-Links-Rotation durchführen (4 Zeiger umhängen).

Der Fall (-2,1) ist symmetrisch und wird mit einer Links-Rechts-Rotation (LR-Rotation) gelöst.

Zusammenfassung zum Einfügen eines Elements s:

- Füge ein neues Blatt mit Inhalt s und Höhenbalance 0 in den Suchbaum ein, siehe 8.2.12.
- Setze von diesem Blatt aufsteigend die Höhenbalancen (eventuell bis zur Wurzel hinauf) neu.
- Wird dabei einmal die neue Höhenbalance 0 eingetragen oder wurde die Balance der Wurzel bearbeitet, so ist man fertig.
- Wird dabei einmal der Wert 2 oder -2 angenommen, so führe man die zugehörige Rotation durch; ebenfalls fertig.

Beachten Sie:

(1) Bei jedem Einfügen muss *höchstens eine Rotation* durchgeführt werden, um die AVL-Eigenschaft wieder herzustellen. (Beim Löschen gilt dies nicht, siehe unten.)

(2) Die "Rotation" ist eine Drehung um den Mittelpunkt einer Kante. Logisch betrachtet findet dabei eine vertikale Verschiebung von Knoten statt (sonst würde die Suchbaum-Eigenschaft verletzt werden).

Genauere Nachfrage:
Wie wird die Höhenbalance B(...) neu berechnet? Dieser Algorithmus wird auf der folgenden Folie vorgestellt.

Füge ein Blatt **v** mit Inhalt s und Höhenbalance B(**v**)=0 ein;
u := direkter Vorgängerknoten von **v**; Abbruch:=false;

```
while u ≠ null and not Abbruch loop
   if v = u.L then          -- v ist linkes Kind von u
      if  B(u) = 1  then B(u) := 0; Abbruch:=true;
      elsif  B(u) = 0  then B(u) := -1;
      else  "führe R/LR-Rotation durch;" Abbruch:=true;  end if;
   else                     -- v ist rechtes Kind von u
      if  B(u) = -1  then B(u) := 0; Abbruch:=true;
      elsif  B(u) = 0  then B(u) := 1;
      else  "führe L/RL-Rotation durch;" Abbruch:=true;  end if;
   end if;
   if not Abbruch then  v := u;
        u := direkter Vorgänger von u;  end if;
end loop;
```

(Um die richtige Rotation erkennen und durchführen zu können, benötigt man *den* Nachfolger von v, von dem man aufgestiegen ist. Selbst durchdenken!)

Das Löschen in einem AVL-Baum erfolgt zunächst wie beim Suchbaum: Finde den Knoten x mit Inhalt s. Hat x weniger als zwei Nachfolger, so lösche x und setze den Zeiger, der vom direkten Vorgänger auf x zeigte, auf den eventuell vorhandenen Nachfolger von x.
Hat x zwei Nachfolger, so ersetze s durch den Inorder-Vorgänger oder den Inorder-Nachfolger s', lösche den Knoten y, in dem s' stand, und biege die Kante, die auf y zeigte, auf den rechten Nachfolger von y (eventuell ist dieser null) um.

Ausgehend vom Vorgänger von x bzw. y müssen (zur Wurzel hin aufsteigend) die Höhenbalancen neu gesetzt werden. Im Gegensatz zum Einfügen, können nun mehrere Rotationen notwendig sein, bis die Wurzel erreicht ist oder bis man aus einem anderen Grund abbrechen kann. Der Algorithmus zur Berechnung der Höhenbalancen lautet ab dem Löschen, beginnend mit dem direkten Vorgänger des gelöschten Knotens (im Programm heißt anfangs der gelöschte Knoten "v"):

u := direkter Vorgängerknoten von **v**; Abbruch:=false;
if **u** ist nicht der Knoten, in dem s stand then **u**.L := **v**.R;
else **u**.R := **v**.R; end if; -- Knoten **v** wird hierdurch unerreichbar
while **u** ≠ null and not Abbruch loop
  if **v** = **u**.L then -- **v** ist linkes Kind von **u**
    if B(**u**) = -1 then B(**u**):=0; **v**:=**u**; **u**:=direkter Vorgänger von **u**;
    elsif B(**u**) = 0 then B(**u**) := 1; Abbruch := true;
    else *führe in Abhängigkeit von* B(**u**.R)∈{-1,0,1} *eine Rotation durch;* **v**:=**u**; **u**:=direkter Vorgänger von **u**; end if;
  else -- **v** ist rechtes Kind von **u**
    if B(**u**) = 1 then B(**u**):=0; **v**:=**u**; **u**:=direkter Vorgänger von **u**;
    elsif B(**u**) = 0 then B(**u**) := -1; Abbruch := true;
    else *führe in Abhängigkeit von* B(**u**.L)∈{-1,0,1} *eine Rotation durch;* **v**:=**u**; **u**:=direkter Vorgänger von **u**; end if;
  end if;
end loop;

Aufgabe: Führen Sie die kursiven Teile des Algorithmus im Detail aus.

Zusammenfassung:

FIND: Wie beim Suchbaum.

INSERT: Wie beim Suchbaum, aber die Balancen längs des Einfügepfades aufsteigend korrigieren. Dabei ist höchstens eine Rotation erforderlich.

DELETE: Wie beim Suchbaum, aber aufsteigend vom Inorder-Nachfolger- (oder -Vorgänger-) Knoten entlang des Pfads zur Wurzel die Höhenbalancen (evtl. mittels Rotationen) ändern.

8.4.6 Ergebnis: Jede der Operationen FIND, INSERT und DELETE erfordert höchstens O(log(n)) Schritte.

Der Grund hierfür: Ein AVL-Baum mit n Knoten kann höchstens die Tiefe 1,4404 · log(n) besitzen.

Diese Aussage beweisen wir im Folgenden.

Aufgabe: Man stelle fest, wie viele Knoten in einem AVL-Baum der Tiefe t maximal und minimal liegen können.
Maximal können es 2^t-1 Knoten sein (klar, 8.2.16).
Sei m_t die Anzahl der Knoten, die sich mindestens in einem AVL-Baum der Tiefe t befinden müssen, dann gilt: $m_0=0$, $m_1=1$ und für alle $t \geq 2$: $m_t = 1 + m_{t-1} + m_{t-2}$. (Wurzel plus linker Unterbaum und rechter Unterbaum, die Folge der Zahlen ist 0,1,2,4,7,12,20,...).
Die Lösung der Gleichung lautet: $m_t = F_{t+2} - 1$, wobei F_t die t-te Fibonacci-zahl ist. Dies ist durch Einsetzen in die Gleichung leicht nachzuweisen.

8.4.7 Die *Fibonaccizahlen* sind definiert durch:
$F_0=0$, $F_1=1$, $F_k = F_{k-1} + F_{k-2}$ für alle $k \geq 2$ (Folge: 0,1,1,2,3,5,8,13,21,34,...).
Man nehme an, F_k erfüllt eine Gleichung der Form $F_k = a \cdot x^k$, dann muss $x^k = x^{k-1} + x^{k-2}$ gelten, d.h., man muss die Gleichung $x^2 = x + 1$ lösen. Deren Lösungen sind c_1 und c_2 (siehe nächste Folie).
Da auch deren Linearkombinationen Lösungen darstellen, erhält man für die Fibonaccizahlen die Formel $F_k = a_1 \cdot c_1^k + a_2 \cdot c_2^k$. Mit den Anfangsbedingungen $F_0=0$ und $F_1=1$ ergibt sich $a_1 = -a_2 = f$, woraus man $F_k = f \cdot (c_1^k - c_2^k)$ erhält.
Wegen $|c_2| < 1$ ist F_k stets die nächste natürliche Zahl zu $f \cdot c_1^k$.

Einschub: Fibonaccizahlen

$F_0 = 0$, $F_1 = 1$ und $F_k = F_{k-1} + F_{k-2}$ für alle $k \geq 2$.

$$F_k = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right) =: f \cdot (c_1^k - c_2^k)$$

mit $c_1 = \left(\frac{1+\sqrt{5}}{2} \right) \approx 1.618035$,

$c_2 = \left(\frac{1-\sqrt{5}}{2} \right) \approx -0.618035$, $f = \frac{1}{\sqrt{5}} \approx 0.447213$

Es folgt: F_k = nächste natürliche Zahl zur Zahl $f \cdot c_1^k$.

Sei also ein AVL-Baum mit n Knoten der Tiefe t gegeben.

$n \geq m_t = F_{t+2} - 1 \approx f \cdot c_1^{t+2} - 1$

$\approx 0.447213 \cdot 1.618035 \cdot 1.618035 \cdot 1.618035^t - 1$

$\approx 1.17082 \cdot 1.618035^t - 1$

Es folgt mit $1/\log_2(1.618035) \approx 1.4404$:

$t \leq \log_2((n+1)/1.17082) / \log_2(1.618035)$

$\approx 1.4404 \cdot \log_2(n+1) - \log_2(1.17082) / \log_2(1.618035)$

$\approx 1.4404 \cdot \log(n)$ [Diese Abschätzung ist sehr genau.]

Satz 8.4.8: Ein AVL-Baum mit n Knoten besitzt höchstens die Tiefe $1.4404 \cdot \log(n)$.

Hinweis: Messungen ergaben, dass diese maximale Tiefe in der Praxis fast nie auftritt und sich die Tiefe der AVL-Bäume meist nur sehr wenig von log(n) unterscheidet. Aber: Der Fall der Tiefe $\approx 1.4404 \cdot \log(n)$ kann auftreten, siehe im Folgenden.

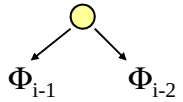
8.4.9 Fibonacci-Bäume

Φ_0 ist der leere Baum, Φ_1 ist der einknotige Baum.

Definition:

(1) Φ_0 und Φ_1 sind die Fibonaccibäume der Ordnung 0 und 1.

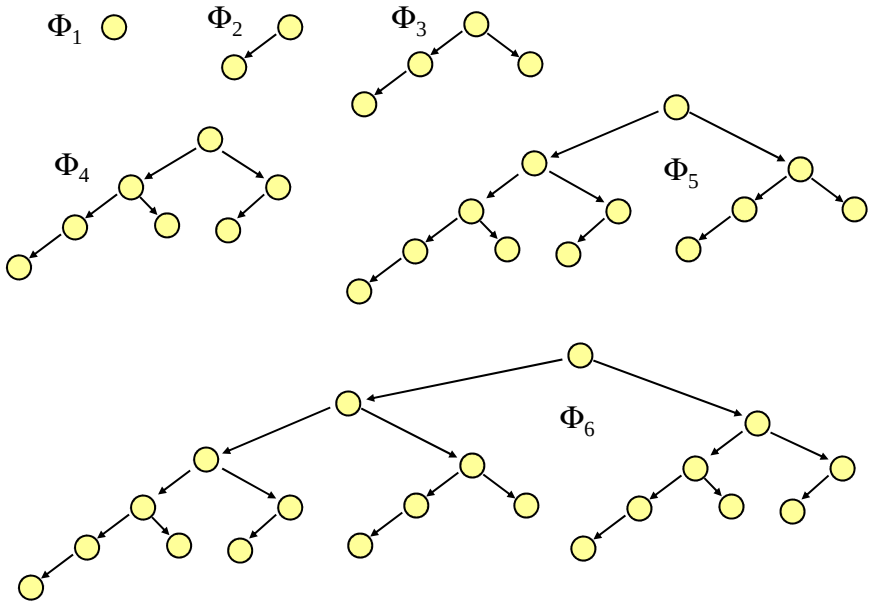
(2) Wenn Φ_{i-1} der Fibonaccibaum der Ordnung i-1 und Φ_{i-2} der Fibonaccibaum der Ordnung i-2 sind ($i \geq 2$), dann ist



der Fibonaccibaum Φ_i der Ordnung i.

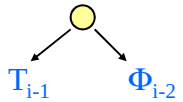
Der Fibonaccibaum Φ_i der Ordnung i ist der "dünnste" AVL-Baum der Tiefe i, also der AVL-Baum mit minimal vielen Knoten zu gegebener Tiefe i.

Φ_i besitzt genau $m_i = F_{i+2} - 1$ Knoten.



8.4.10 Ist jeder AVL-Baum zugleich gewichtsbalanciert?

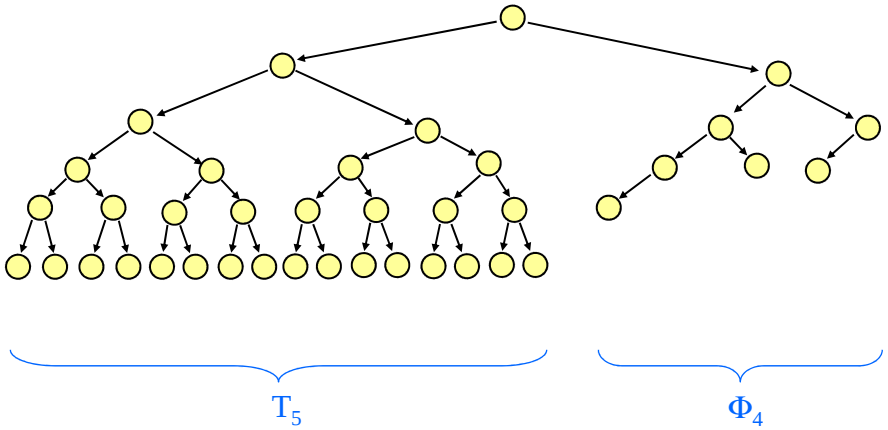
Nein! Betrachte hierzu folgenden AVL-Baum der Tiefe i:



wobei T_{i-1} der AVL-Baum der Tiefe i-1 mit maximal vielen Knoten ist. T_{i-1} besitzt $2^{i-1}-1$ und Φ_{i-2} besitzt $m_{i-2} = F_i - 1$ Knoten. (Ein Beispiel steht auf der nächsten Folie.)

Der Quotient $2^{i-1}/F_i$ wächst exponentiell in i. Daraus folgt, dass es für jede vorgegebene reelle Zahl $\alpha \in (0, ½]$ AVL-Bäume gibt, die *nicht* α-gewichtsbalanciert (siehe Definition 8.4.1) sind.

Der folgende Baum ist ein AVL-Baum. Man erkennt, dass seine Tiefe durch log(n)+2 beschränkt ist; seine 39 Knoten ($i = 6$ und $n = 39 = 1 + 2^5\text{-}1 + 8\text{-}1$) sind jedoch sehr unterschiedlich auf die beiden Unterbäume der Wurzel verteilt.



Abschließender Hinweis: Es werden häufig Bäume, die sehr gleichverzweigt sind und nur Blätter auf dem Level log(n) oder log(n)-1 besitzen, verwendet. Ihnen wollen wir einen Namen geben, nämlich "ausgeglichene" Bäume:

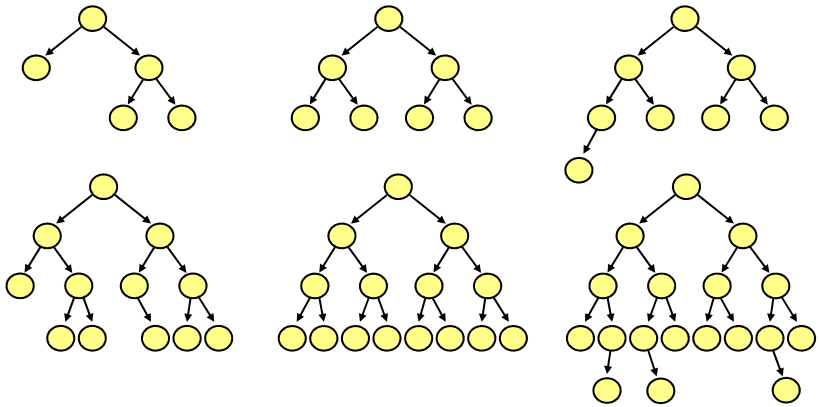
Definition 8.4.11:

Ein nicht-leerer, k-närer Baum (siehe Definition 8.2.3) heißt ausgeglichener Baum (der Tiefe r), wenn es eine natürliche Zahl r gibt, so dass jeder Knoten, der mindestens einen null-Zeiger enthält, das Level r-1 oder r besitzt und wenn es mindestens ein Blatt der Tiefe r gibt.

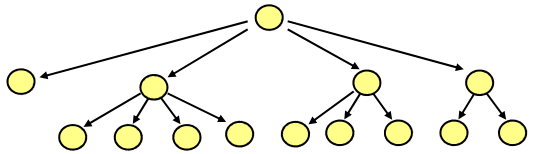
Man kann diese Definition leicht von k-nären auf beliebige geordnete Bäume übertragen. Die B-Bäume des nächsten Abschnitts 8.5 bilden solch ein Beispiel.

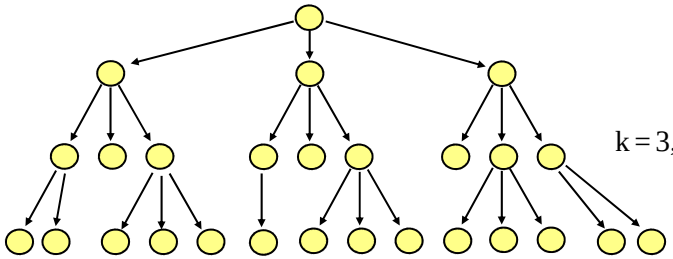
Sofern es Knoten mit null-Zeigern in verschiedenen Leveln gibt, ist in einem ausgeglichenen Bäumen das Level r-1 komplett mit Knoten besetzt. Überschüssige Knoten können sich dann nur noch auf dem nächsten Level r befinden.

Beispiele für $k = 2$ (also binäre Bäume) und für $r = 3, 4$ und 5:



Beispiele für $k > 2$ (null-Zeiger wurden nicht eingetragen, wodurch zu jeder Skizze mehrere k-näre Bäume gehören können):





8.5 B-Bäume (für externe Speicherung)

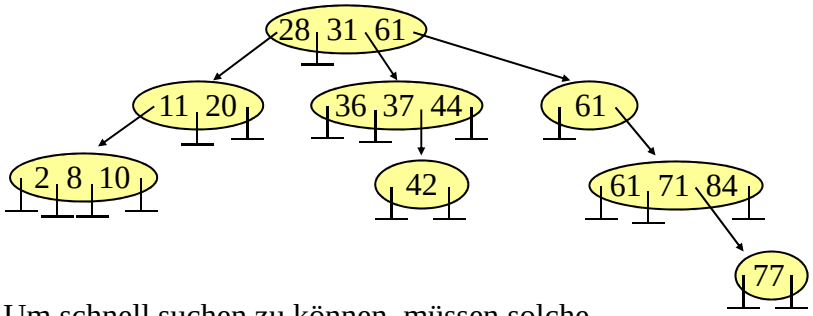
AVL-Bäume eignen sich gut als Darstellung von Mengen, wenn sich die gesamte Menge im Hauptspeicher befindet. Liegen die Elemente dagegen auf einem Hintergrundspeicher, so muss man bei jedem Übergang zu einem Kindknoten auf diesen externen Speicher zugreifen. Heute ist der Hintergrund- speicher meist eine Festplatte mit einer Zugriffszeit von im Mittel 5 Millisekunden. Hat man einen AVL-Baum der Tiefe 20 (dies entspricht einer Menge mit 1 Million Elementen), so werden alleine 100 Millisekunden für den Zugriff benötigt, während die durchgeführten 20 Vergleiche weniger als eine Millisekunde erfordern. Der Rechner verbringt seine Zeit daher zu über 99% mit Warten und ist nach rund 101 Millisekunden mit der Suche fertig.

Ziel ist daher eine Datenstruktur, die mit möglichst wenigen externen Zugriffen die gesuchten Daten findet bzw. einfügt bzw. löscht.

Nahe liegend ist eine gute Mischung aus baumartiger und sequenzieller Suche: Man holt bei jedem Zugriff - sagen wir - 500 Werte in den Hauptspeicher und grenzt den Bereich, wo der gesuchte Schlüssel zu finden ist, nicht wie beim binären Baum um den Faktor 2, sondern um den Faktor 500 ein.

Bei einem Datenbestand von 1 Million Werten sind dann nur noch drei Zugriffe erforderlich. Zusammen mit dem dreimaligen (internen) Durchlaufen der 500 Werte braucht man jetzt nur noch eine Gesamtlaufzeit von unter 20 Millisekunden, wobei 15 Millisekunden aus Warten bestehen.

Idee: Wir betrachten Bäume, in deren Knoten sich nicht nur *ein* Schlüssel, sondern ein sortiertes k-Tupel von Schlüsseln befindet. Beim Suchen durchläuft man dieses k-Tupel und folgt je nachdem, zwischen welchen Elementen der gesuchte Schlüssel liegt, einem Zeiger in den nächsten Unterbaum. Beispiel:



Um schnell suchen zu können, müssen solche Bäume die Suchbaumeigenschaft erfüllen.

Definition 8.5.1: Allgemeine Suchbaumeigenschaft

Gegeben sei ein geordneter Baum. In jedem Knoten steht ein nicht-leeres sortiertes Tupel von Schlüsseln einer geordneten Menge. Für jeden Knoten u muss gelten:
Sind $s_1, s_2, ..., s_k$ die sortierten Schlüssel von u ($s_1 \le s_2 \le ... \le s_k$), so besitzt u genau k+1 Kinder und es gilt:
Alle Schlüssel im Unterbaum links vom ersten Kind sind kleiner als s_1, alle Schlüssel im Unterbaum rechts des ersten Kindes sind größer oder gleich s_1 und kleiner als s_2, alle Schlüssel im Unterbaum rechts des i-ten Kindes sind größer oder gleich s_i und kleiner als s_{i+1} (für $i = 1, 2, ..., k-1$), und alle Schlüssel im Unterbaum rechts des k-ten Kindes sind größer oder gleich s_k.
Ein geordneter Baum mit dieser Eigenschaft heißt (allgemeiner) Suchbaum.

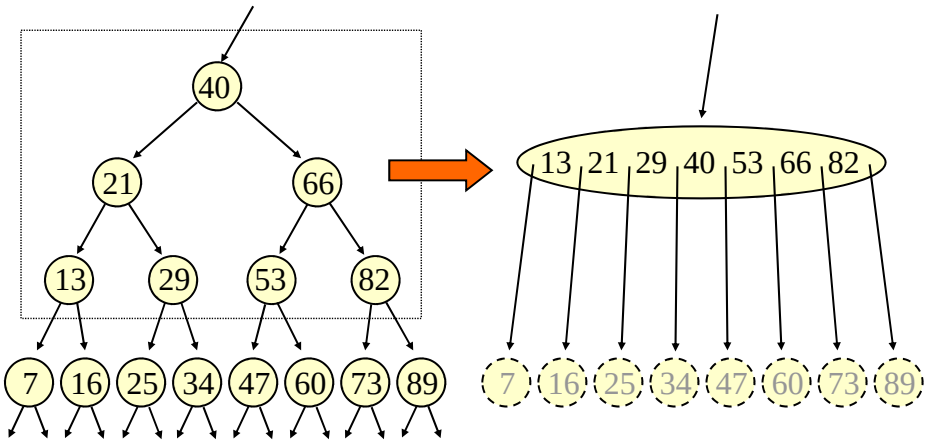
Hinweis: Wenn gleiche Schlüssel zugelassen sind, so wird man diese wie üblich im rechten Unterbaum eines Schlüssels ablegen (siehe Zahl 61 in der Abbildung auf der vorletzten Folie). Bei den nun zu definierenden B-Bäumen kann man diese Eigenschaft jedoch nicht garantieren, weil durch Einfüge- oder Löschoperationen gleiche Schlüssel auch in den linken Unterbaum verschoben werden können. Wir werden diesen Fall daher verbieten und ihn am Ende des Abschnitts gesondert betrachten.
In Anwendungsfällen informiere man sich zuvor genau, ob gleiche Schlüssel erlaubt sind und wie sie behandelt werden.

Wir werden in diesem Abschnitt 8.5 nur B-Bäume mit Inhalten behandeln, bei denen alle Schlüssel verschieden sind!

Definition 8.5.2:

Es sei m ≥ 1 eine natürliche Zahl. Ein geordneter nicht-leerer Baum B, in dem jeder Knoten eine sortierte Folge von Schlüsseln aus einer geordneten Menge enthält, heißt **B-Baum** der Ordnung m, wenn für alle Knoten von B gilt:

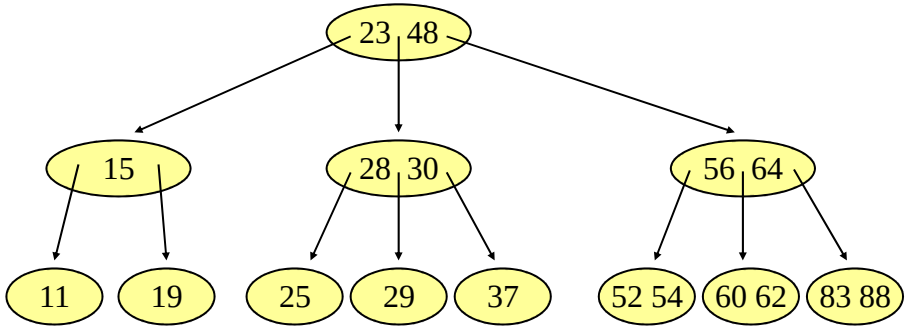
1. Jeder Knoten enthält höchstens 2m Schlüssel.
2. Die Wurzel enthält mindestens einen Schlüssel.
3. Jeder andere Knoten enthält mindestens m Schlüssel.
4. Ein Knoten mit k Schlüsseln besitzt entweder genau k+1 Kinder oder kein Kind.
5. Alle Blätter (= Knoten ohne Kinder) besitzen das gleiche Level.
6. B erfüllt die allgemeine Suchbaumeigenschaft.



Ausschnitt aus einem binären Suchbaum.

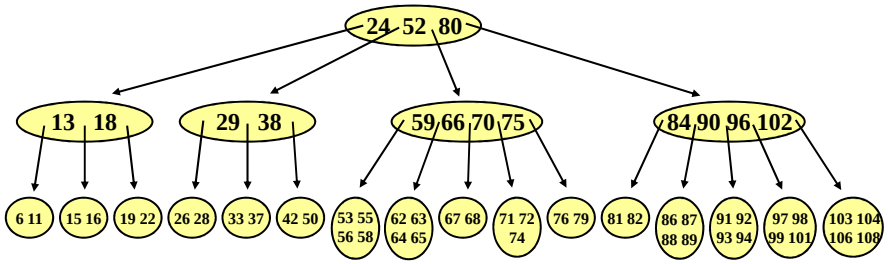
Zusammenfassung zu einem B-Baum-Knoten, dessen Inhalt linear (oder mit Intervall-schachtelung) durchlaufen wird.

Beispiel für einen B-Baum der Ordnung 1
Man beachte, dass alle Blätter das gleiche Level besitzen.



Hinweis: B-Bäume der Ordnung 1 heißen auch 2-3-Bäume.

Beispiel für einen B-Baum der Ordnung 2:



Übliche Ordnungen liegen in der Praxis zwischen 128 und 4096.
(Begründen Sie diese Größenordnung!)

Tiefe t eines B-Baums der Ordnung m mit n Knoten:

$$\log_{2m+1}(n) \le t \le \log_{m+1}(n) + 1$$

Grund: Jeder Knoten (außer der Wurzel) besitzt mindestens m+1 und höchstens 2m+1 Kinder.

Hinweis: Die Bezeichnungen sind in der Literatur unterschiedlich. Oft verwendet man auch "2m" oder "2m+1" als die Ordnung des B-Baums. Vergewissern Sie sich daher stets über die zugrunde liegenden Definitionen.

In vielen Anwendungen speichert man die Daten zwar im B-Baum, möchte aber die eigentlichen Inhalte beim Löschen und bei manchen Implementierungen auch beim Einfügen nicht im Baum verschieben. Dann legt man alle Daten in die Blätter des Baums und errichtet hierüber einen B-Baum aus Zeigern. Da ein B-Baum nur an der Wurzel wächst und schrumpft (siehe später), wird eine solche Struktur bei B-Bäumen automatisch aufrecht erhalten.
Einen B-Baum, bei dem alle Daten nur in den Blättern liegen, nennt man B*-Baum.

Wir betrachten nun die drei Operationen Suchen, Einfügen und Löschen.

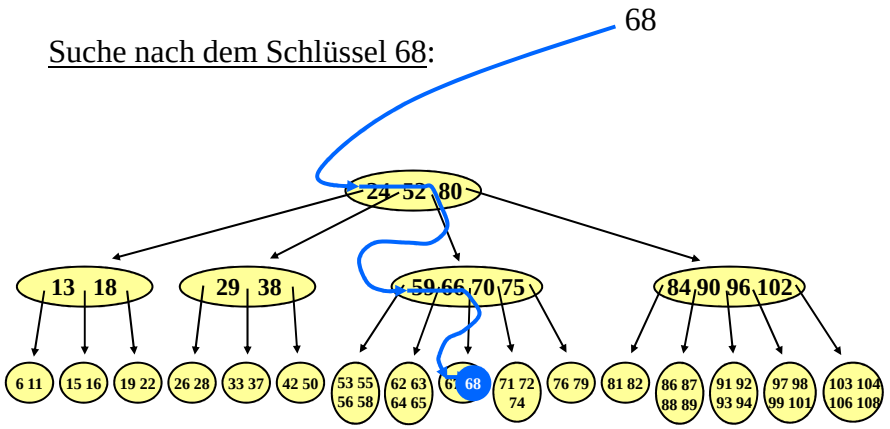
8.5.3 Suchen in einem B-Baum

Das Suchen erfolgt wie oben angegeben: Man durchlaufe das sortierte Tupel des Knotens und folge dem "richtigen" Zeiger entsprechend der Suchbaumeigenschaft. Der Zeitaufwand ist proportional zur Tiefe des Baums. Man muss jedoch noch die sortierten Listen der Schlüssel in jedem betrachteten Knoten durchlaufen. Dies kostet mindestens m und höchstens 2m Vergleiche; bei binärer Suche genügen log(2m) Vergleiche, wenn die Schlüssel in einem array abgelegt sind. Maximale Zahl der Vergleiche insgesamt:

$$2m \cdot (\log_{m+1}(n)+1) \quad \text{bzw.} \quad \log(2m) \cdot (\log_{m+1}(n)+1).$$

Suchen erfordert also $O(\log(2m) \cdot \log_{m+1}(n))$ Schritte.

Suche nach dem Schlüssel 68:



Für n Elemente im Baum gilt offenbar:
Zahl der Vergleiche ≤ 2m·Tiefe des Baumes ≤ 2m · $\log_{m+1}(n)$.
Beispiel: Für m = 512 und eine Wertemenge bis 134 Millionen Elemente sind dies 6m = 3072 Vergleiche und 3 Zugriffe auf den Speicher. Bei binärer Suche auf der Schlüsselmenge jedes Knotens genügen 3·log(2m) = 30 Vergleiche. Wesentlich ist aber, dass man mit nur 3 Zugriffen auf den Speicher auskommt!

8.5.4 Einfügen in einen B-Baum

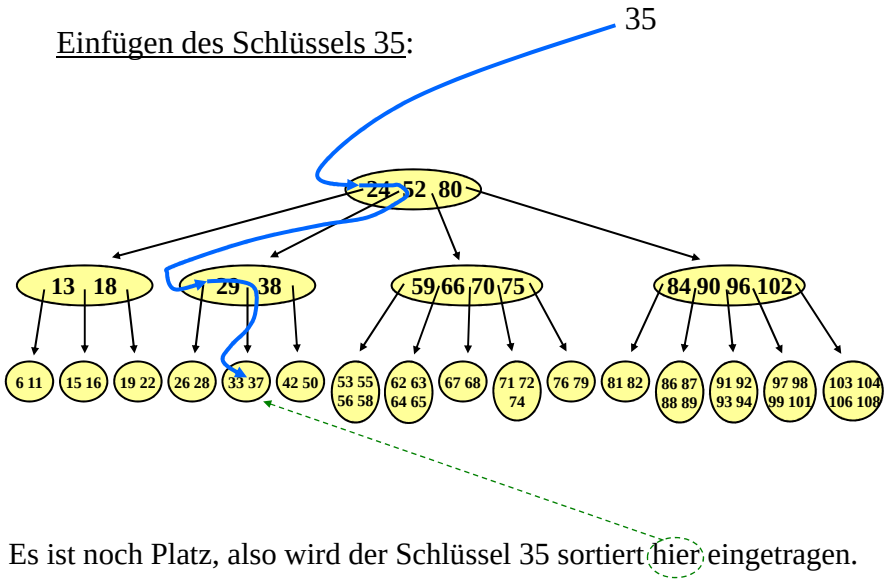
Beim Einfügen ermittelt man zunächst das Blatt, in das der neue Schlüssel s einzutragen ist. Sind weniger als 2m Schlüssel in diesem Blatt, so fügt man den neuen Schlüssel s sortiert ein und ist fertig. Anderenfalls liegt ein Überlauf im Knoten vor und das Blatt muss in zwei Blätter mit je m Schlüsseln aufgespalten ("gesplittet") werden, wobei der mittlere der 2m+1 Schlüssel an den Elternknoten weitergereicht und dort eingetragen wird.

Eventuell muss auch der Elternknoten aufgespalten werden usw. Hierbei kann von unten nach oben ein Aufspalten bis zur Wurzel erfolgen. Wird die Wurzel aufgespalten, so wird eine neue Wurzel mit genau einem Schlüssel erzeugt.

Der *Aufwand* liegt wie bei der Suche in $O(\log(2m) \cdot \log_{m+1}(n))$.

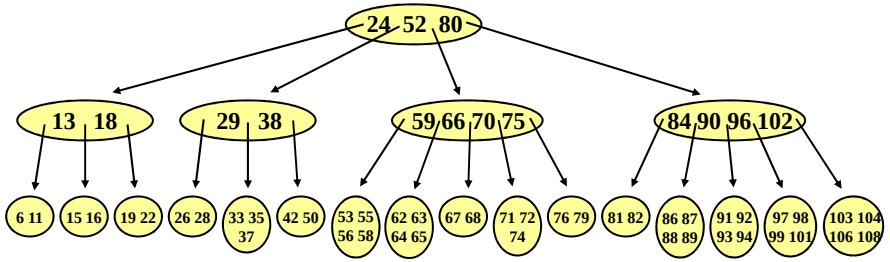
Frage an Sie: Stimmt dies? Man muss doch den Schlüssel s noch einsortieren, was bei einem array bis zu 2m Schritte erfordern kann. Wie löst man dieses Problem effizient?

Einfügen des Schlüssels 35:

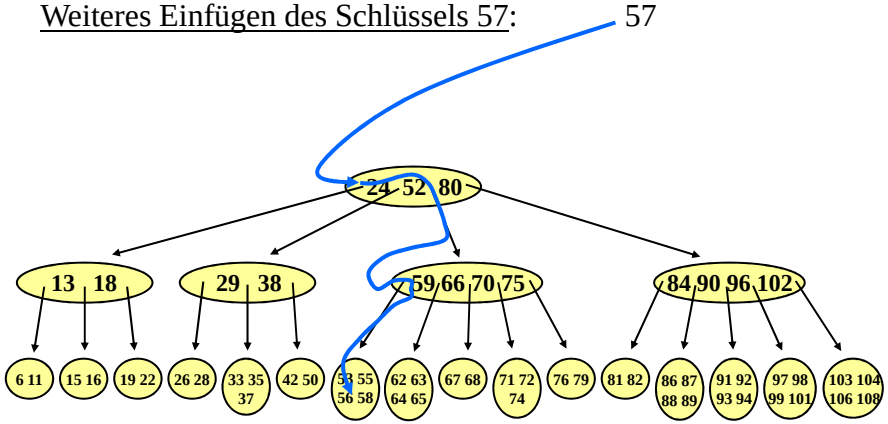


Es ist noch Platz, also wird der Schlüssel 35 sortiert hier eingetragen.

Schlüssels 35 ist nun eingefügt:

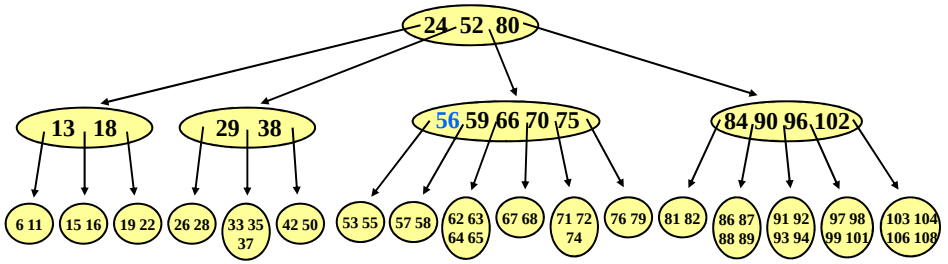


Weiteres Einfügen des Schlüssels 57: 57



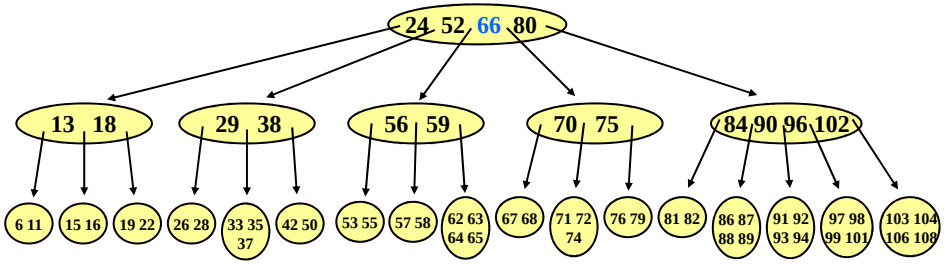
Beim Einfügen der 57 läuft der Knoten über, da er jetzt fünf Schlüssel enthält. Also wird er in zwei Knoten aufgespalten mit den Inhalten "53 55" bzw. "57 58". Der mittlere Wert "56" wird zum Elternknoten hinauf gereicht.

Weiteres Einfügen des Schlüssels 57:



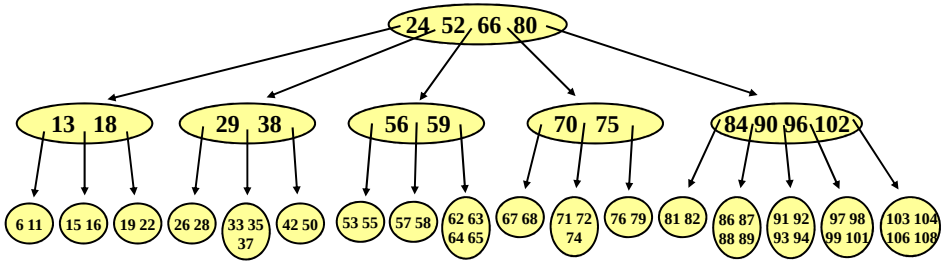
Beim Einfügen der 56 läuft aber nun der Elternknoten über. Also wird dieser in zwei Knoten aufgespalten mit den Inhalten "56 59" bzw. "70 75". Der mittlere Wert "66" wird zu seinem Elternknoten (dies ist hier die Wurzel) hinauf gereicht.

Ergebnis des Einfügens von Schlüssel 57:



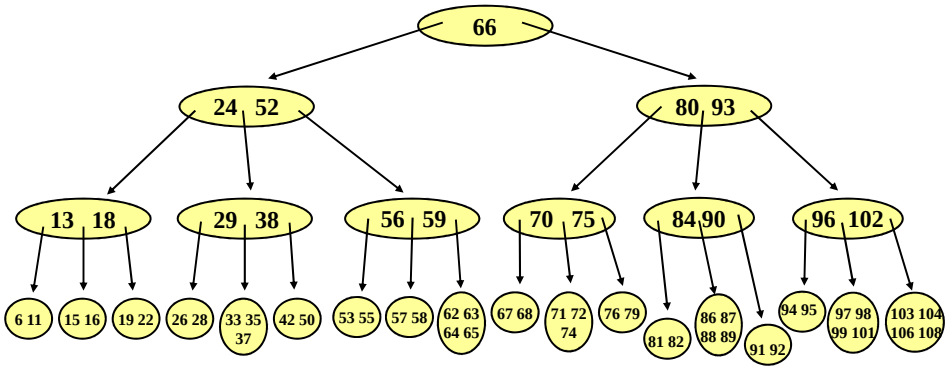
Die Wurzel darf (wie jeder Knoten in einem B-Baum der Ordnung 2) bis zu 4 Schlüssel besitzen, folglich ist dieser Baum das Ergebnis, nachdem der Schlüssel 57 eingefügt wurde.

Weiteres Einfügen des Schlüssels 95:



Die 95 muss in den mit "91 92 93 94" beschrifteten Knoten eingetragen werden. Dieser läuft über und wird in zwei Knoten mit den Inhalten "91 92" und "94 95" aufgespalten; der Schlüssel "93" wird nach oben gereicht. Der Elternknoten wird auch aufgespalten und reicht den Schlüssel "93" weiter nach oben. Auch die Wurzel läuft nun über, wird gespalten und reicht den Schlüssel 66 weiter. Dieser wird neue Wurzel des Baumes. Man erhält also folgenden B-Baum der Ordnung 2 (mit einer um 1 größeren Tiefe):

Ergebnis nach Einfügen der Schlüssel 35, 57 und 95:



Man beachte, dass die Blätter hierbei stets auf dem gleichen Level liegen. Ein B-Baum kann nur an der Wurzel wachsen und schrumpfen, AVL-Bäume nur an den Blättern.

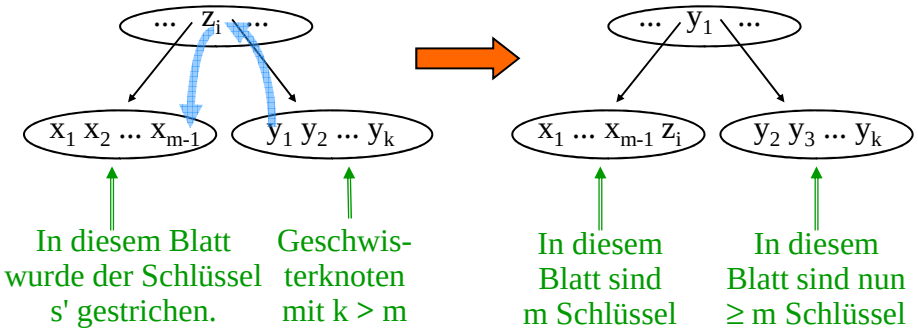
8.5.5 Löschen in einen B-Baum

Beim Löschen wird zunächst der Knoten, in dem der gesuchte Schlüssel s steht, ermittelt ("FIND"). Ist dieser Knoten kein Blatt, so wird der Schlüssel s durch seinen Inorder-Vorgänger oder -Nachfolger s' ersetzt (wie man diesen findet, siehe 8.2.13); bei B-Bäumen steht dieser immer in einem Blatt, also wird auf jeden Fall ein Schlüssel s' (oder s) in einem Blatt gelöscht.

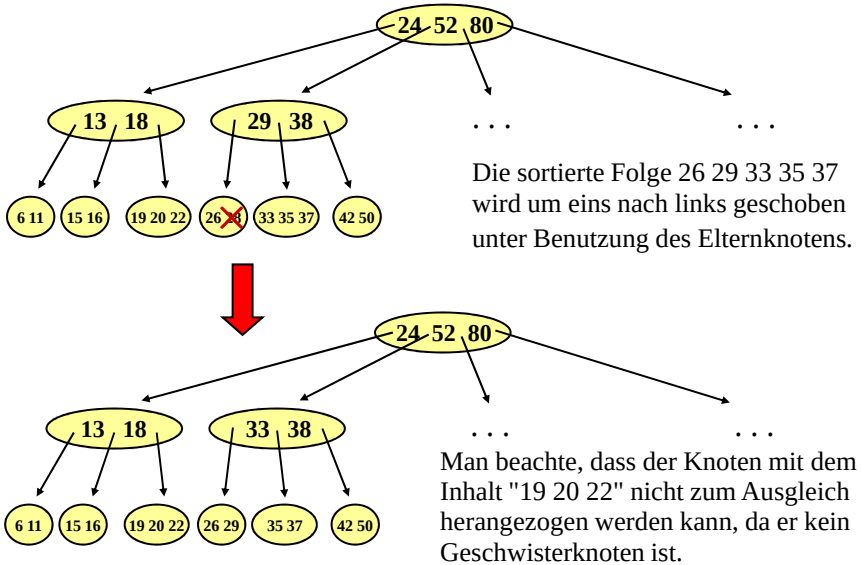
Fall 1: Besitzt das Blatt mindestens m+1 Schlüssel, so wird der Schlüssel s bzw. s' gelöscht und man ist fertig.

Fall 2: Besitzt das Blatt genau m Schlüssel (es liegt hier ein "Unterlauf" vor), so prüft man, ob ein Geschwisterknoten mit mindestens m+1 Schlüsseln existiert. In diesem Fall schiebt man den Schlüssel des Elternknotens in das untergelaufene Blatt und den kleinsten Schlüssel des Geschwisterknotens an dessen Stelle.

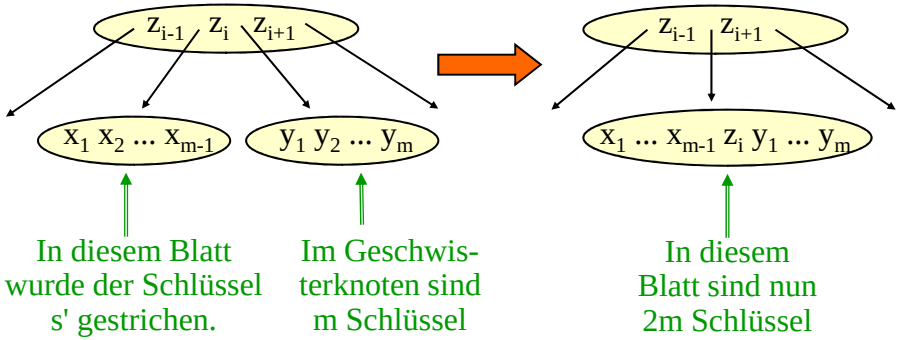
Hierbei werden nur 2 Schlüssel verschoben. Skizze:



Beispiel zu Fall 2: Lösche den Schlüssel 28



Fall 3: Das Blatt besitzt m Schlüssel und die Geschwisterknoten besitzen ebenfalls m Schlüssel. Dann wird das Blatt mit einem Geschwisterknoten verschmolzen unter Hinzunahme des zugehörigen Schlüssels im Elternknoten. Hat der Knoten zwei Geschwisterknoten, so wähle man zufällig einen der Geschwisterknoten für das Verschmelzen.



Fall 3 (Fortsetzung): In diesem Fall wird die Zahl der Schlüssel im Elternknoten verringert. Wende daher rekursiv auf den Elternknoten die Fälle 1 bis 3 an.
Hierbei kann jedes Mal Fall 3 auftreten und schließlich ein Schlüssel aus der Wurzel entfernt werden. Wenn die Wurzel dann noch mindestens einen Schlüssel enthält, ist man fertig. Besaß die Wurzel aber nur einen Schlüssel, so wird die Wurzel leer und der einzige, neu entstandene verschmolzene Knoten unter ihr wird zur neuen Wurzel des Baumes. Genau in diesem Fall wird die Tiefe des Baumes um 1 verringert.

Da ein B-Baum höchstens die Tiefe $\log_{m+1}(n)+1$ besitzt, benötigt auch das Löschen nur $O(\log(2m) \cdot \log_{m+1}(n))$ Schritte.

Frage an Sie: Stimmt dies? Man muss doch aus zwei arrays eventuell ein array machen, was mindestens m Schritte allein für das Kopieren erfordert. Wie löst man dieses Problem effizient?

8.5.6 Speicherplatzausnutzung durch B-Bäume

Ein B-Baum ist stets zu mindestens 50% gefüllt. Wie viel Speicherplatz wird aber tatsächlich ausgenutzt? Man wird 75% erwarten, jedoch zeigt eine theoretische Analyse, dass es im Mittel ca. 69% sind.
In der Praxis wurde durch Messungen festgestellt, dass B-Bäume im Mittel sogar nur zu knapp 67% gefüllt sind. Man sollte daher das Einfügen von Schlüsseln so implementieren, dass das Aufspalten von Knoten möglichst lange vermieden wird (z.B., indem man die Geschwisterknoten einbezieht, wie es beim Löschen geschieht).
Beim Löschen von Schlüsseln in inneren Knoten sollte man (wie bei anderen Bäumen) zufällig bestimmen, ob man den Inorder-Nachfolger oder -Vorgänger verwendet. Überlegen Sie, ob es einen Einfluss auf den Speicherplatz hat, wenn man immer nur den Inorder-Nachfolger wählt?!

8.5.7 Zur Implementierung:

Für die Implementierung kann man in jedem Knoten ein
- array[1..2*m] für die Schlüssel, ein
- array[1..2*m] für die zugehörigen Inhalte und ein
- array[0..2*m] für die Zeiger auf die Kinder

mitführen. Zusätzlich ist eine natürliche Zahl "Anzahl" zu speichern, die die aktuelle Zahl der Schlüssel angibt, sowie eine Boolesche Variable für die Eigenschaft "Blatt".

Die zugehörige Datentypdefinition bitte selbst einfügen:

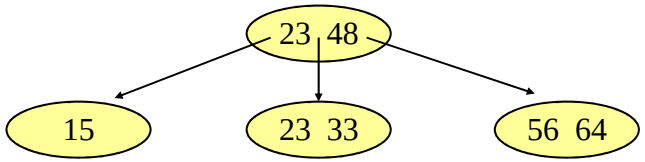
….

Alternative: In der Praxis nennt man einen B-Baum-Knoten auch "Seite". Zeiger werden oft durch Ref_ oder durch Ptr_ (von "pointer") bezeichnet. m sei hier als Konstante vorgegeben. In einem Knoten stehen bis zu 2m Schlüssel mit den durch die Schlüssel eindeutig charakterisierten Inhalten. SchlüsselTyp sei der Typ der Schlüssel, InhaltTyp sei der Typ der Inhalte. Einen Schlüssel, einen Inhalt und einen Zeiger auf den Knoten mit den nächstgrößeren Schlüsseln fassen wir als "Item" zusammen. Dann bilden bis zu 2m Items plus ein Zeiger auf das linkeste Kind sowie die Boolesche Eigenschaft "Blatt" und die aktuelle "Anzahl" der gespeicherten Schlüssel den gesamten Knoten. Mögliche Datentypdefinition für einen B-Baum:

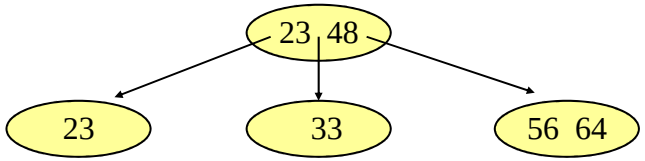
```
type Seite;
type Ref_Seite is access Seite;
type Item is  record Schlüssel: SchlüsselTyp;
                     Inhalt: InhaltTyp;
                     Zeiger: Ref_Seite;
                   end record;
type Seite is record Anzahl: Natural;
                     Blatt: Boolean;
                     linkeSeite: Ref_Seite;
                     Items: array (1..2*m) of Item;
                   end record;
```


8.5.8 Gleiche Schlüssel

Wenn gleiche Schlüssel auftreten dürfen, dann lässt sich die allgemeine Suchbaumeigenschaft mit den bisher vorgestellten Operationen nicht gewährleisten. Beispiel: Betrachte den folgenden B-Baum der Ordnung m=1:



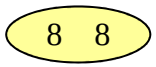
Wird nun "15" gelöscht, so wird vom Knoten "23 33" der linke Schlüssel "23" in den Elternknoten und dessen linker Schlüssel "23" in das Blatt geschoben. Ergebnis:



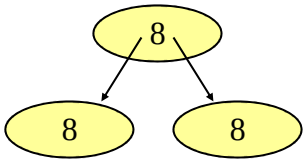
Da "23" nun links von "23" des Elternknotens steht, ist die allgemeine Suchbaumeigenschaft verletzt.

Dies lässt sich bei B-Bäumen prinzipiell nicht verhindern. Fügt man in einen B-Baum nacheinander den gleichen Schlüssel (z.B. 8) ein, so tritt der Elternschlüssel zwangsläufig sowohl im linken wie im rechten Unterbaum auf (die Ordnung sei hier erneut 1):

B-Baum nach Einfügen der ersten "8"



B-Baum nach Einfügen der zweiten "8"



B-Baum nach Einfügen der dritten "8"

Wir müssen also die Suchbaumeigenschaft abschwächen:

Definition: Abgeschwächte Suchbaumeigenschaft

Gegeben sei ein geordneter Baum. In jedem Knoten steht ein nicht-leeres sortiertes Tupel von Schlüsseln einer geordneten Menge. Für alle Knoten u muss gelten:
Sind $s_1, s_2, ..., s_k$ die sortierten Schlüssel von u ($s_1 \le s_2 \le ... \le s_k$), so besitzt u genau k+1 Kinder und es gilt:
Alle Schlüssel im Unterbaum links vom ersten Kind sind kleiner oder gleich s_1, alle Schlüssel im rechten Unterbaum des ersten Kindes sind größer oder gleich s_1 und kleiner oder gleich s_2, alle Schlüssel im rechten Unterbaum des i-ten Kindes sind größer oder gleich s_i und kleiner oder gleich s_{i+1} (für $i = 1, 2, ..., k-1$), und alle Schlüssel im rechten Unterbaum des k-ten Kindes sind größer oder gleich s_k.

[Einen geordneten Baum mit dieser Eigenschaft bezeichnet man oft auch als (allgemeinen) Suchbaum.]

Diese Definition erzwingt, dass zu jedem Schlüssel (genauer: zu jedem Item in 8.5.7) eine Boolesche Variable mitzuführen ist, die angibt, ob sich im linken Unterbaum gleiche Schlüssel befinden. Diese Variable ist beim erstmaligen Auftreten eines Schlüssels false. Sie wird in einem Elternknoten auf true gesetzt, wenn dieser Knoten beim Einfügen aufgespalten wurde und hierbei der Schlüssel in den linken Unterbaum verschoben und durch den gleichen Schlüssel ersetzt wurde oder wenn beim Löschen ein Ausgleich über den Elternknoten mit einem gleichen Schlüssel erfolgte. Andererseits muss sie auf false zurückgesetzt werden, falls alle gleichen Schlüssel im linken Unterbaum gelöscht wurden.

Kriterium: Diese Boolesche Variable muss zu einem konkreten Schlüssel s genau dann den Wert true besitzen, wenn s nicht in einem Blatt steht und wenn s gleich seinem Inorder-Vorgänger ist.

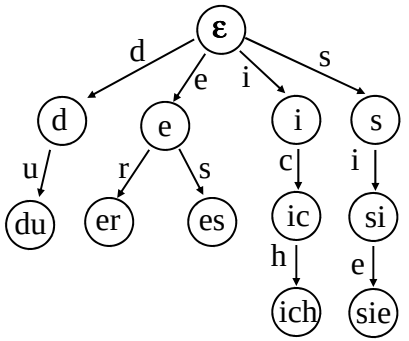
Hierdurch müssen nun alle Algorithmen für die Operationen modifiziert werden. Unangenehm ist, dass sich beim Einfügen und Löschen die Zahl der Schritte erhöht, um die Informationen über gleiche Schlüssel zu erfassen und zu speichern.

Man kann dies vermeiden zum Beispiel durch:
- Man speichere jeden Schlüssel im B-Baum nur einmal und lege für jeden Schlüssel eine lineare Liste an, auf deren Anfang vom B-Baum-Knoten aus verwiesen wird.
- Man verwende B*-Bäume, bei denen die unterste Schicht, in der alle Schlüssel und Inhalte stehen, doppelt verkettet ist. (Dann sind die Inorder-Vorgänger und -Nachfolger in dieser Liste stets benachbart.)

8.6 Digitale Suchbäume

8.6.1 Zeichenweises Aufbauen: Sind die Schlüssel Wörter über einem Alphabet (und dies ist oft der Fall), so kann man sie zeichenweise (digit per digit, also "digital") lesen und hiermit gleichzeitig einen Baum durchlaufen, dessen Kanten mit den zulässigen Alphabetzeichen markiert sind. Dies kann entweder vollständig geschehen oder man verkürzt die Kantenfolge so, dass die restliche Zeichenfolge eindeutig auf den Schlüssel schließen lässt.
Hier gibt es mehrere einfache Ansätze, von denen wir drei vorstellen werden.

Baum mit vollständigen Pfaden (von vorne nach hinten durchlaufener Schlüssel). Die Schlüssel lauten "ich", "du", "er", "sie", "es".



Wird ein Wort, z.B. "ich" gesucht, dann folgt man zuerst der Kante "i", dann der Kante "c" und dann der Kante "h". Trifft man hierbei auf einen "null"-Zeiger (z.B. wenn man "ihr" sucht) oder befindet sich das gesuchte Wort am Ende nicht im erreichten Knoten, so endet die Suche erfolglos.

Dies ist ein Ausschnitt davon, wie man alle Wörter über einem Alphabet Σ als unendlichen Baum darstellt. Wir demonstrieren dies für den Fall Σ = {0, 1}.

Definition: Gegeben sei ein Alphabet Σ mit m = |Σ| Elementen. Ein Baum, dessen Kanten mit Zeichen aus Σ markiert sind, bzw. in dem die Selektoren für die Nachfolger jedes Knotens Zeichen aus Σ sind, und dessen Knoteninhalte Wörter über Σ sind, heißt "**trie**" (gesprochen wie das deutsche "trei"). In der Regel verlangt man zusätzlich, dass die Markierung längs des Weges von der Wurzel zu einem Knoten u ein Teil von oder gleich dem Knoteninhalt von u ist.

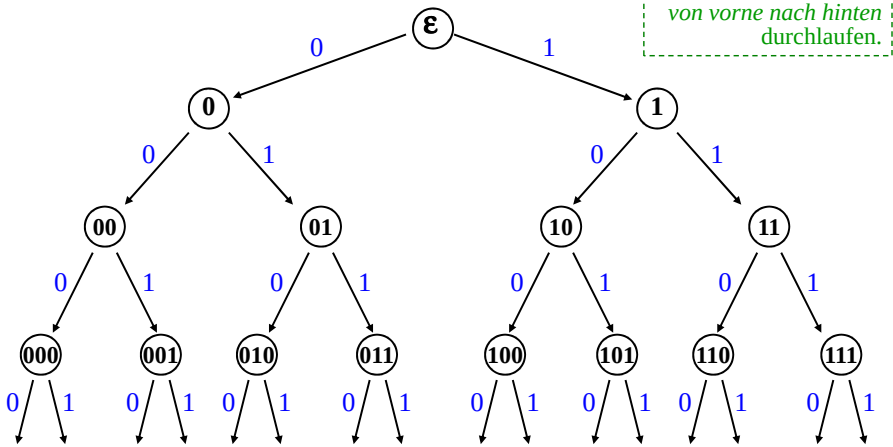
In einem trie hat jeder Knoten maximal m Nachfolger.

Tries dienen dazu, Mengen von Wörtern darzustellen; sie werden vor allem bei der schnellen Suche nach Wörtern eingesetzt.

Das Wort "trie" wurde als Mittelteil von "retrieve" (= etwas wiederfinden) künstlich geschaffen.

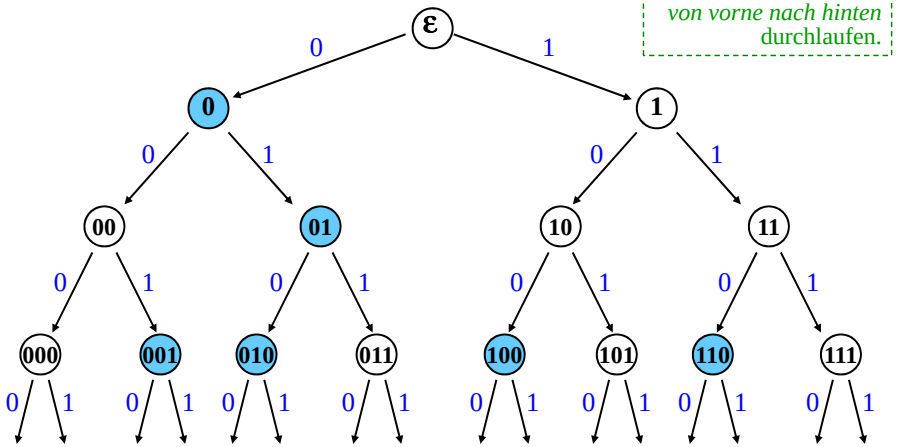
8.6.2 Darstellung aller Wörter aus $\Sigma^* = \{0, 1\}^*$.

Schlüssel zeichenweise *von vorne nach hinten* durchlaufen.



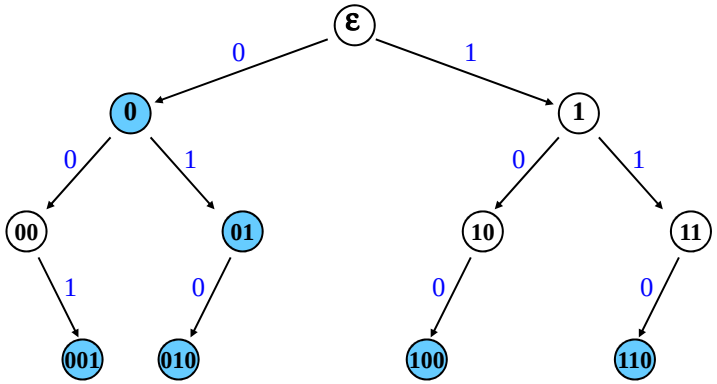
0 = linkes Kind, 1 = rechtes Kind. Erreicht man einen Knoten durch die Zeichenfolge s_1, s_2, s_3, ..., s_k, so ist $s_1s_2s_3...s_k$ der Inhalt des erreichten Knotens.

Schlüssel zeichenweise *von vorne nach hinten* durchlaufen.



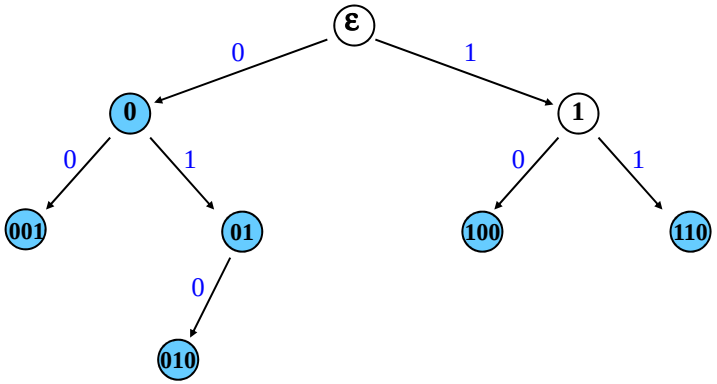
Die Menge {0, 01, 001, 010, 100, 110} im unendlichen Baum.

Ansatz 1 "Standard-Trie":



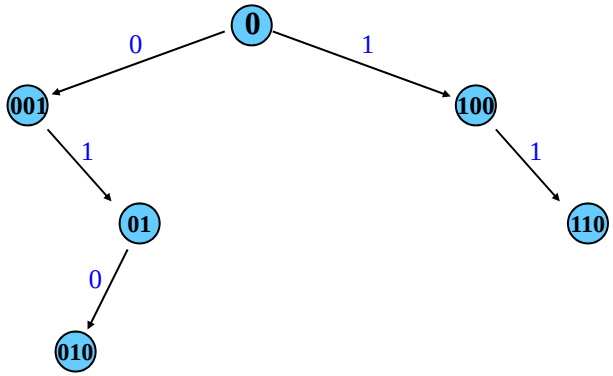
Die Menge {0, 01, 001, 010, 100, 110} als endlicher Baum mit vollständigen Pfaden. Leere Unterbäume sind nicht dargestellt.

Ansatz 2 "Komprimierter Trie":



Idee: Ziehe lineare Ketten im Baum zusammen, sofern sie keine Elemente der Menge enthalten. Beispiel oben: Die Menge {0, 01, 001, 010, 100, 110} als Baum mit verkürzten Pfaden.

Ansatz 3: "Kompakter Trie"



Idee: Schiebe Knoteninhalte nach oben (ziemlich egal, wie, aber nicht an anderen Elementen der Menge vorbei). Noch knappere Darstellung der Menge {0, 01, 001, 010, 100, 110} als Baum.

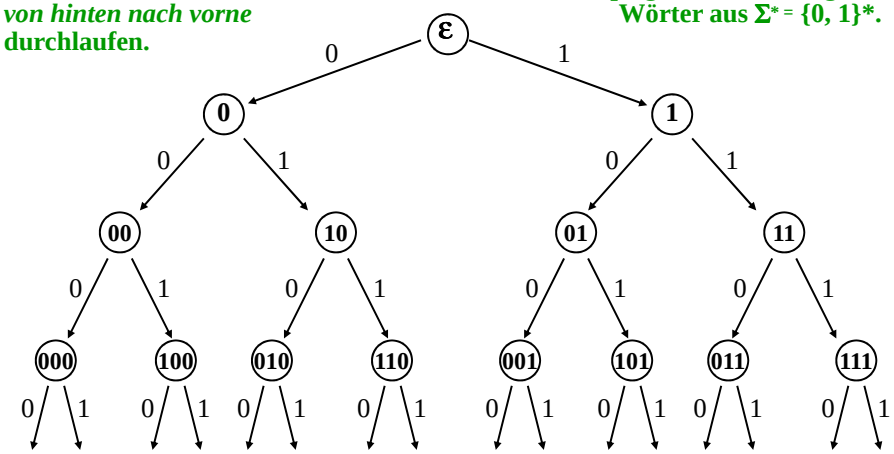
Hinweis (3 Folien lang): In der Praxis durchläuft man die Zeichenfolge $s_1 s_2 ... s_k$ manchmal auch von hinten nach vorne.
Grund: Die Inhalte der Knoten kann man auch als binär dargestellte Zahlen interpretieren. Also: Einen Schlüssel s kann man als Zahl zur Basis m auffassen, wobei wir wegen der möglichen führenden Nullen eine etwas modifizierte Zahldarstellung wählen müssen:
$s \in \Sigma^* = \{0, 1, ..., m-1\}^*$, m ist die Zahl der Alphabetzeichen, wir fassen jedes Zeichen s_i als die Zahl $0 \le s_i \le m-1$ auf und interpretieren s als die folgende natürliche Zahl:

$$s = (1+s_1)\cdot m^{k-1} + (1+s_2)\cdot m^{k-2} + ... + (1+s_{k-1})\cdot m + (1+s_k) - 1.$$

(Ist das k eine feste Zahl, dann kann man auch die gewohnte Darstellung zur Basis m verwenden.) Wenn dann als Schlüssel $s = s_1 s_2 ... s_k$ eine Zahl zu dieser Zahldarstellung gegeben ist, so erreicht man von der Wurzel aus den Knoten, der zu s gehört, indem man s+1 bildet, dann mit geeigneter Modulo-Bildung die Zeichen s_k, s_{k-1}, ..., s_1 *von hinten nach vorne* berechnet und den hierdurch definierten Pfad im Baum aufbaut. Auf diese Weise durchläuft man die Zeichenfolge rückwärts, wozu folgender Baum (im Falle m = 2) gehört:

Schlüssel zeichenweise *von hinten nach vorne* durchlaufen.

Gespiegelte Darstellung der Wörter aus $\Sigma^* = \{0, 1\}^*$.



0 = linkes Kind, 1 = rechtes Kind. Erreicht man einen Knoten durch die Zeichenfolge s_1, s_2, s_3, ..., s_k, so ist $s_k s_{k-1} ... s_2 s_1$ der Inhalt des erreichten Knotens.

Wenn das k bekannt ist, errechnet man die Zeichen "x"
von hinten nach vorn wie folgt:

```
z := "Wurzel des Baums"; zahl := s + 1;
for i := 1 to k do
        x := zahl mod m - 1;  zahl := (zahl - m) div m;
        z := "x-tes Kind von z, sofern existiert"
od;
"Untersuche den Inhalt von z"
```

Kennt man k dagegen zuvor nicht:

```
z := "Wurzel des Baums"; zahl := s + 1; k := 0;
while zahl > 1 do
        x := zahl mod m - 1;  zahl := (zahl - m) div m;
        z := "x-tes Kind von z, sofern existiert"; k := k+1
od;
"Untersuche den Inhalt von z"
```


8.6.3 Drei Ansätze für digitale Suchbäume (hier über {0,1})

Digitaler Suchbaum über {0,1} mit allen Zwischenknoten:

Ansatz 1: Gegeben sind Schlüssel als 0-1-Folgen. Für jeden solchen Schlüssel durchlaufe man den (binären) 0-1-Baum entsprechend der Binärfolge des Schlüssels und platziere ihn genau in dem zum Schlüssel gehörenden Knoten.

FIND, INSERT und DELETE verhalten sich nun wie bei binären Suchbäumen und dauern O(k) Schritte, wenn der Schlüssel genau k Zeichen enthält (k ist zugleich die Tiefe des Baums).

Vorteile dieser Struktur, falls man auf Maschinenebene programmiert oder die Schlüssel in Binärdarstellung vorliegen.

Nachteil: Dies kostet oft viel zusätzlichen Speicherplatz für die Zwischenknoten.

Digitaler Suchbaum über {0,1}, komprimiert (ohne lineare Anteile):

Ansatz 2: Schlüssel = 0-1-Folge.
Durchlaufe den (binären) 0-1-Baum entsprechend.
Prüfe an jedem Knoten, wie lang der lineare Teil sein könnte.
Dies führt dann zum Knoten mit dem gehörenden Schlüssel.

FIND, INSERT und DELETE: wie bei Ansatz 1 im schlechtesten Fall, aber oft schneller.

Vorteile, falls man auf Maschinenebene programmiert oder die Schlüssel in Binärdarstellung vorliegen. Kompaktere Darstellung des Baums.

Nachteil: Ständig durch Vergleich des restlichen Schlüssels mit dem Knoteninhalt testen, ob ein linearer Anteil vorliegt und wie lang dieser ist.

Digitaler Suchbaum über {0,1}, kompakt:

Ansatz 3: Schlüssel = 0-1-Folge.
Durchlaufe den 0-1-Baum mit den Zeichen des Schlüssels und platziere den Schlüssel in den ersten freien Knoten.

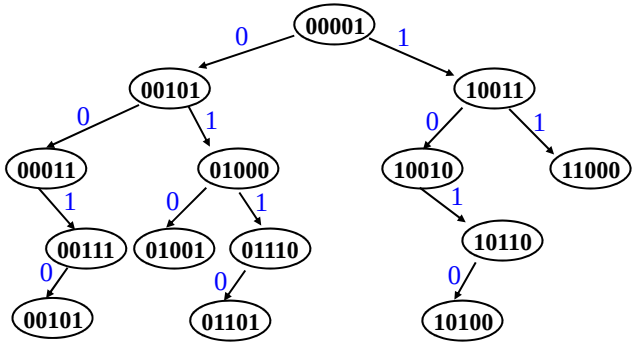
FIND, INSERT und DELETE wie bei binärem Suchbaum, da dieses Vorgehen dem bei binären Bäumen entspricht.

Vorteil: Zahl der Knoten = Zahl der Elemente, d.h., der Baum hat minimal viele Knoten.

Nachteil: Ständig ganzen Schlüssel auf Gleichheit testen.

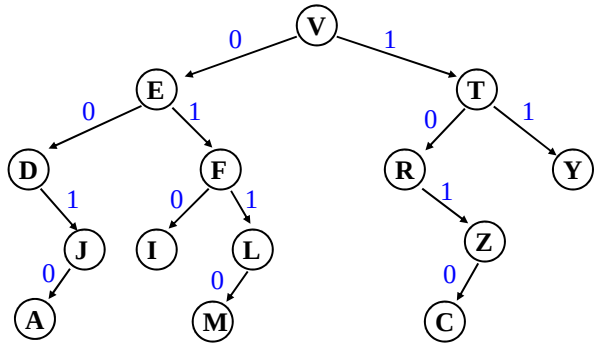
Verallgemeinerbar auf beliebige Alphabete, aber evtl. viel zusätzlicher Speicher nötig (wegen hohem Ausgangsgrad; oder man muss die Liste der Zeiger auf Nachfolger selbst wieder in einem Baum verwalten).

Vorgehen mit Ansatz 3: Codiert man Zeichen eines beliebigen Alphabets binär, so kann man entsprechend der Reihenfolge, in der die Schlüssel eingelesen werden, mit ihnen einen Binärbaum aufbauen und jeden Schlüssel in das jeweils erreichte Blatt legen. Beispiel: Folgende 14 Schlüssel sollen in einen Baum eingefügt werden: 00001, 10011, 00101, 10010, 01000, 01001, 00011, 01110, 00111, 11000, 01101, 10110, 00101, 10100. Ergebnis:



Sind beispielsweise die Zeichen {A, C, D, E, F, I, J, L, M, R, T, V, Y, Z} wie angegeben als 0-1-Folgen der Länge 5 kodiert, so haben wir faktisch einen digitalen Suchbaum für diese Menge erzeugt. Schreibt man statt der 0-1-Folgen die zugehörigen Zeichen in die Knoten, so entsteht aus obigem Baum:

Codierung und Reihenfolge:
V: 00001
T: 10011
E: 00101
R: 10010
F: 01000
I: 01001
D: 00011
L: 01110
J: 00111
Y: 11000
M: 01101
Z: 10110
A: 00101
C: 10100



8.6.4 Zur Implementierung

In der Praxis muss man erkennen, wo ein Schlüssel endet. In Programmiersprachen benutzt man "Trennzeichen" (*delimiter* wie z.B.: " ", ";", "(", ")",); wir verwenden hier das Zeichen '#'.

Die Schlüssel 1, 10 und 101 werden also als
1#, 10# und 101# dargestellt,
wenn man einen digitalen Baum aufbaut.

Die Implementierung dieses Ansatzes sollte nun klar sein (?). Hinweise zur möglichen Datenstruktur für digitale Bäume über {0,1} und zum Suchen und Aufbauen finden Sie auf den nächsten Folien. Das Löschen in einem digitalen Baum ist je nach Ansatz unterschiedlich aufwendig.

Durch Codierung nach {0,1}* kann man sich stets auf das Alphabet {0,1} beschränken.

Als Selektoren müssten wir 0 und 1 verwenden, was aber in Ada und anderen Programmiersprachen nicht zugelassen ist. Wir verwenden daher die Selektoren N0 und N1 für die Verweise auf die Nachfolger mit dem Selektor 0 bzw. 1.

Um Zahlen mitbehandeln zu können, führen wir die Funktion bit(s,k) ein. Für eine natürliche Zahl (oder einen Schlüssel als 0-1-Folge, aufgefasst als Zahl) s und eine natürliche Zahl k sei
bit(s,k) = if $s \geq 2^{k-1}$ then k-tes Bit von hinten in der Binärdarstellung von s else # fi.
bit(s,1) ist also die letzte Binärstelle von s, bit(s,2) die vorletzte usw. Vorne wird mit '#' aufgefüllt (nicht mit 0).

Nahe liegende Datenstruktur:

```
Maxlaenge: constant Positive := ... ;
type Stelle is (0,1,#);
type Schluesseltyp is array (1..Maxlaenge) of Stelle;
type Knotentyp;
type Ref_Knoten is access Knotentyp;
type Knotentyp is record
        Schluessel: Schluesseltyp := (others => #);
        N0, N1: Ref_Knoten;
        end record;
```


Suche oder Einfügen für Schlüssel s (für jeden der 3 Ansätze)

```
h1, h2: Ref_Knoten; i: Natural := Maxlaenge; Code: Schluesseltyp;
begin  h1 := "Verweis auf die Wurzel des trie"; h2 := null;
    "wandle s mittels bit(s,k) für k = 1, 2, ..., Maxlaenge in
        Schluesseltyp um und weise dies Code zu";
    while h1 /= null and then h1.Schluessel /= Code loop
        h2 := h1;
        if Code(i) = 0 then h1:=h1.N0;
        elsif Code(i) = 1 then h1:=h1.N1;
        else h1 := null; end if;
        i:= i-1;
    end loop;
    if h1 = null then "s nicht im Baum; h2 verweist auf den Knoten,
        an den ein Knoten mit Schlüssel s angehängt werden kann"
    else "h1 verweist auf Knoten mit dem Schlüssel s" end if;
end;
```

Für Ansatz 1 und 2 kann man dieses Programmstück etwas vereinfachen.

8.6.5 Suffix Tries

Definition: Gegeben sei ein Alphabet Σ mit m = |Σ| Elementen.
Es sei $w = w_1w_2w_3...w_k$ ein Wort der Länge k über Σ.

(1) Jedes Wort $w_iw_{i+1}...w_k$ mit $1 \le i \le k+1$ heißt Suffix von w.

(2) Es sei Suff(w) = { u | u ist Suffix von w } die Menge aller Suffixe von w. Ein komprimierter Trie (= ein Baum gemäß Ansatz 2) der Menge Suff(w) heißt Suffix Trie von w (oder *suffix tree* oder *Suffixbaum* oder *position tree* von w).

Suffixbäume werden bei der Mustererkennung verwendet.
Ohne Beweis notieren wir:
- Der Suffixbaum von w belegt nur O(k) Speicherplätze, wobei k die Länge von w ist.
- Es gibt Algorithmen, die den Suffixbaum von w in O(k) Schritten aufbauen (das ist nicht-trivial).

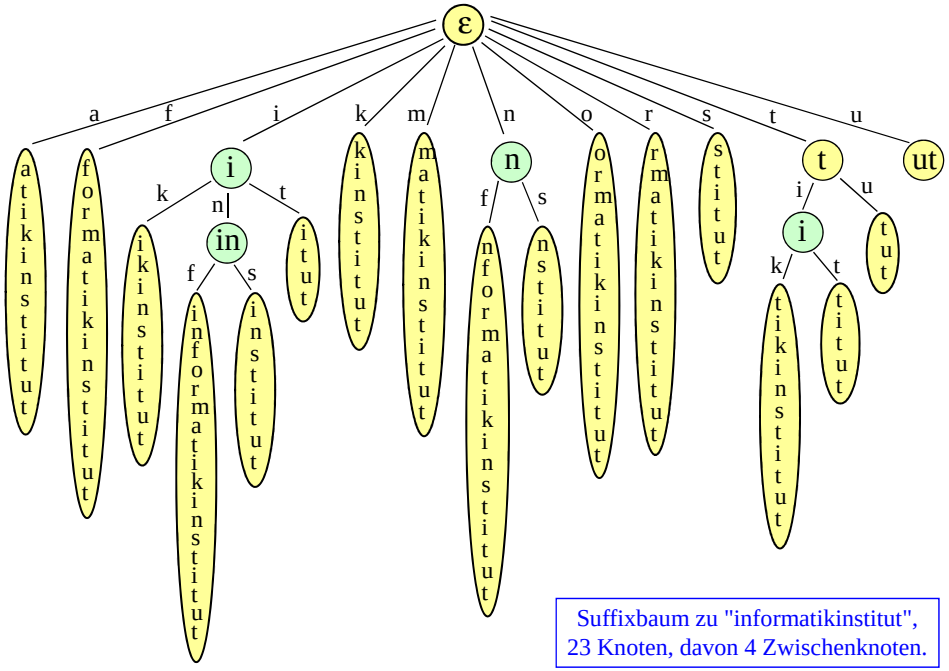
Beispiel: Betrachte das Wort w = "informatikinstitut".

Die Länge des Wortes w beträgt k =18.
Suff(w) = {ε, t, ut, tut, itut, titut, stitut, nstitut, institut, kinstitut, ikinstitut, tikinstitut, atikinstitut, matikinstitut, rmatikinstitut, ormatikinstitut, formatikinstitut, nformatikinstitut, informatikinstitut}.

Diese k+1 Wörter haben die Gesamtlänge
0+1+2+3+...+k=k·(k+1)/2=171.
Man würde erwarten, dass der Suffixbaum daher auch ungefähr 171 Knoten besitzt, was aber nicht zutrifft.

Aufgabe: Konstruieren Sie den Suffixbaum zu diesem Wort w.
Lösung: nächste Folie.

Suff(w)
ε
t
ut
tut
itut
titut
stitut
nstitut
institut
kinstitut
ikinstitut
tikinstitut
atikinstitut
matikinstitut
rmatikinstitut
ormatikinstitut
formatikinstitut
nformatikinstitut
informatikinstitut



Suffixbaum zu "informatikinstitut", 23 Knoten, davon 4 Zwischenknoten.

8.7 Datenstrukturen mit Historie

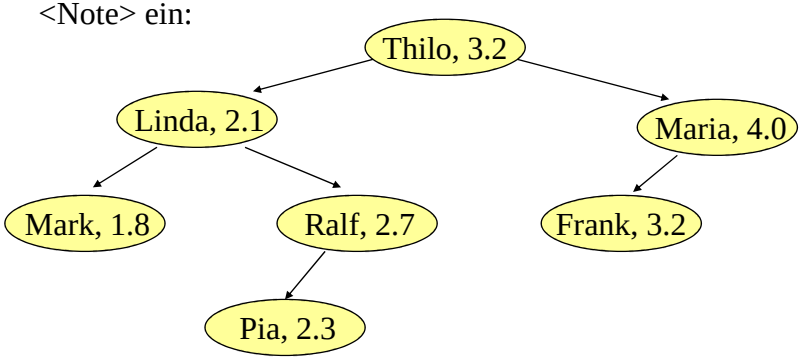
Dieser Abschnitt entstand nach einem Vortrag, den Professor Thomas Ottmann (Universität Freiburg) im Kolloquium der Universität Stuttgart hielt.

Entwicklungen und Vorgänge unterliegen der Zeit. So gibt es auch beim Aufbau einer konkreten Datenstruktur stets eine Historie, also einen zeitlichen Ablauf, in dem diese Struktur entsteht.

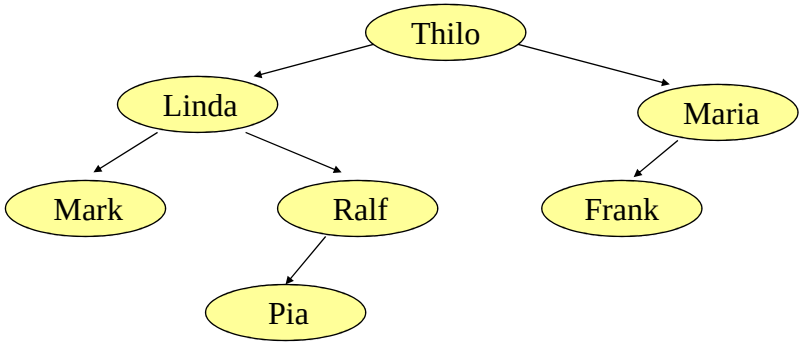
Wenn wir uns die zeitliche Reihenfolge merken möchten, so müssen wir die Datenstruktur und die verwendeten Operationen kennen. In diesem Kapitel 8 würden wir geordnete Bäume und die Operationen FIND, INSERT und DELETE wählen.

Wir betrachten ein Beispiel:

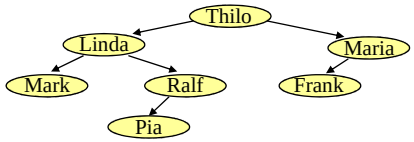
8.7.1 Beispiel "Teilnehmer und Ergebnisse einer Klausur":
Die Datenstruktur sei ein binärer Suchbaum. Die verwendeten Daten "(<Name>,<Note>)" bilden die Menge {(Tilo, 3.2), (Linda, 2.1), (Ralf, 2.7), (Maria, 4.0), (Mark, 1.8), (Pia, 2.3), (Frank, 3.2)}.
Diese 7 Elemente tragen wir in dieser Reihenfolge mittels INSERT in einen binären Suchbaum bzgl. des Schlüssels <Note> ein:



Wir möchten nun wissen, wer an der Klausur teilgenommen hat. Hierzu lassen wir die Noten weg und übermitteln den verbliebenen Baum.



Können wir diesem fertigen Produkt, also dem binären Baum, mehr als nur die Teilnehmer ansehen? **Ja!**



Erstens erkennen wir die Rangordnung der Klausurergebnisse mit einem Inorder-Durchlauf:
Mark, Linda, Pia, Ralf, Thilo, Frank, Maria.

Zweitens liefert uns der Baum eine Halbordnung der Eintragungsreihenfolge: Jeder Pfad von der Wurzel zu einem Blatt gibt die relative Reihenfolge beim Einfügen an.
So müssen Linda vor Ralf und Pia, Maria vor Frank, Linda vor Mark usw. eingefügt worden sein.

Folgerung: Die Datenstruktur speichert also nicht nur irgendwelche Fakten, sondern sie merkt sich zugleich Ausschnitte aus ihrer Historie und gewisse zusätzliche Inhalte.

Im hier betrachteten Fall sind die zusätzlichen Informationen nichts anderes als Relationen auf der Menge der Daten.

Idee und intuitive Formulierung: Es sei M die Menge der Daten, die in einer Datenstruktur D gespeichert sind. Eine Relation $R \subseteq M^k = M \times M \times ... \times M$ heißt *mit D verträglich*, wenn jede Beziehung $(m_1, m_2, ..., m_k) \in R$ aus der Datenstruktur D gefolgert werden kann.

Diese Formulierung ist nicht so genau, dass wir sie programmieren könnten. Es fehlt die Präzisierung, was es bedeutet, dass eine Beziehung aus einer Datenstruktur gefolgert werden kann. Hierzu müssten wir eine formale Logik über den Datenstrukturen definieren. Siehe hierzu Vorlesungen aus dem Gebiet "sichere und zuverlässige Systeme". Wir verfolgen daher eine andere Idee.

Eine Datenstruktur d_i entsteht aus der Menge der Daten M, indem zulässige Operationen auf Elemente aus M und die bis dahin gewonnene Datenstruktur d_{i-1} angewendet werden. Anfangs ist die Datenstruktur leer: d_0 = 'empty'.

Definition 8.7.2
- Es sei M eine Menge. Es sei D eine Menge von Datenstrukturen, deren Inhalte aus M seien. Es sei 'empty' eine spezielle Datenstruktur aus D (die "leere Struktur").
- Es sei Ω eine endliche Menge von Operationen.
- Der Einfachheit halber nehmen wir hier an, dass für jedes $\omega \in \Omega$ gilt: $\omega: M \times D \to D$.

Sei $d \in D$. Eine Folge $h = (\omega_1, m_1), (\omega_2, m_2), ..., (\omega_r, m_r)$ von Operationen ω_i und Elementen m_i heißt eine **Historie** der Länge r, falls gilt: $d = \omega_r(m_r, ...(\omega_2(m_2, \omega_1(m_1, \text{empty})))...)$.

Wir vollziehen also den Aufbau der Datenstruktur schrittweise nach. Wenn wir nacheinander die Operationen $\omega_1, \omega_2, ..., \omega_r$ mit den Elementen $m_1, m_2, ..., m_r$ durchführen, so erhalten wir ausgehend von d_0 nacheinander die Datenstrukturen $d_1, d_2, ..., d_r$:

d_0 = 'empty', $d_1 = \omega_1(m_1, \text{empty})$, $d_2 = \omega_2(m_2, d_1)$,
$d_3 = \omega_3(m_3, d_2)$, ... $d_i = \omega_i(m_i, d_{i-1})$ für alle i > 0, ... ,
und am Ende:
$d_r = \omega_r(m_r, d_{r-1}) = \omega_r(m_r, ...(\omega_2(m_2, \omega_1(m_1, \text{empty})))...)$.

Frage 1: Besitzt jede Datenstruktur oder ein gegebenes $d \in D$ mindestens eine Historie und wie findet man diese?

Frage 2: Wie viele verschiedene Historien (der Länge r oder $\leq r$) kann eine Datenstruktur $d \in D$ haben?

Frage 3: Wie und wie schnell kann man eine kürzeste Historie zu einem $d \in D$ finden?

Dies soll an Beispielen erläutert werden. Wir wählen als Menge M die ganzen Zahlen **Z** und als Menge D die Menge der binären Suchbäume mit ganzen Zahlen als Inhalten. 'empty' sei der leere Baum.

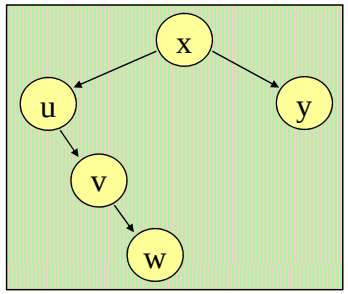
Weiterhin wählen wir als Menge der Operationen
Ω = {FIND, IN, DEL}.
Für $\omega \in \Omega$ sei ω: $M \times D \to D$ definiert durch:
- im Falle ω = FIND bleibt D unverändert (d.h.: $\omega(m,d) = d$),
- im Falle ω = IN wird das Element als Blatt eingefügt (insert),
- im Falle ω = DEL wird über den Inorder-Nachfolger gelöscht, vgl. 8.2.13 delete-Operation.

Wir betrachten nun einen binären Suchbaum und fragen, welche Historie zu ihm gehört. Dies ist im Allgemeinen nicht eindeutig. Daher interessiert uns die Anzahl aller Historien, die ein solcher Baum haben kann.

Beispiel 8.7.3 a

M = {u, v, w, x, y}

d =



Diese Datenstruktur d besitzt genau 4 Historien *(welche?)*, wenn man nur die Operation IN (= INSERT) verwenden darf.

Zwei Historien für dieses d lauten:
Historie 1: (IN,x), (IN,u), (IN,v), (IN,w), (IN,y)
Historie 2: (IN,x), (IN,u), (IN,y), (IN,v), (IN,w)

Man erkennt: Die Anzahl der Historien, die *nur* die INSERT-Operation verwenden, ist gleich der Zahl der toplogischen Sortierungen des Baums (→ 8.8.2). Obiges d besitzt also genau 4 topologische Sortierungen.

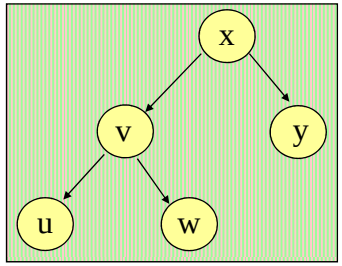
Verwendet man auch die Operation DEL (= DELETE), so besitzt jeder Suchbaum unendlich viele Historien. Zum Beispiel ist
(IN,w), (IN,x), (IN,y), (DEL,w), (IN,u), (IN,v), (IN,w)
eine Historie von d.

Hier kann man eine umfangreiche Theorie anschließen, wie sie etwa von Datenbanken bekannt ist. Man kann Regeln aufstellen und auf Historien "rechnen", wie man es auf arithmetischen Ausdrücken gewohnt ist. Doch sind diese Regeln kompliziert.
Eine einleuchtende Regel lautet: Wenn man etwas einfügt und anschließend sofort wieder löscht, so bleibt die Datenstruktur unverändert, also:
Für jedes $a \in M$ darf man die Teilfolge " (IN,a),(DEL,a) " aus jeder Historie entfernen (ohne dass sich die zugehörige Datenstruktur ändert).
Dies trifft aber im Allgemeinen nicht zu! Auch bei der hier benutzten Datenstruktur "Suchbaum" ist diese Regel nur richtig, wenn gleiche Elemente beim Einfügen stets im rechten Unterbaum als Blatt abgelegt werden und wenn beim Löschen stets der Inorder-Nachfolger verwendet wird.
Man kann dann nach vollständigen Regelsystemen suchen. Diese haben die Eigenschaft: Wenn d zwei verschiedene Historien h und h' besitzt, dann kann man h mit dem Regelsystem in h' überführen.

Beispiel 8.7.3 b

M = {u, v, w, x, y}

d' =



Historie 1: (IN,x), (IN,v), (IN,u), (IN,w), (IN,y)
Historie 2: (IN,x), (IN,v), (IN,y), (IN,w), (IN,u) usw.

Dieser Suchbaum d' besitzt genau 8 Historien, wenn man nur die Operation IN verwenden darf. Seine Entstehung ist also "unbestimmter" als die des Suchbaums d in Beispiel 8.7.3.a. Unbestimmheit bezeichnet man oft als Entropie.

Definition 8.7.4:

Es seien M, D, Ω und ω: M×D → D (für ω∈Ω) wie in Def. 8.7.2.
Für eine natürliche Zahl r und eine Struktur d∈ D sei die r-Entropie von d die Anzahl der Historien der Länge r, die zu d gehören.
Sofern es zu d nur endlich viele Historien gibt, bezeichnet man die Anzahl aller Historien als die Entropie von d.

Hinweis: Die Strukturen mit hoher Entropie sind also zugleich die "vergesslichen Strukturen", aus denen die eigene Entstehung nur ziemlich unvollkommen rekonstruiert werden kann.

Fallstudie 8.7.5:

Es seien M = **Z** die Menge der ganzen Zahlen, D = die Menge aller binären Suchbäume (mit Inhalten aus **Z**) wie in 8.2.10, Ω = {IN} eine einelementige Menge und IN: **Z**×D → D die übliche Einfügeoperation INSERT, die durch die Prozedur procedure Einfügen(Anker: in out Ref_BinBaum; s: Integer) in 8.2.12 exakt definiert ist (der neue Schlüssel s∈ **Z** wird hierbei in ein neues Blatt eingefügt).
Gegeben sei eine natürliche Zahl n und die Menge der ersten n natürlichen Zahlen $\underline{\underline{n}} := \{0, 1, 2, ..., n-1\} \subset \mathbf{Z}$. Es sei D_n die Menge der binären Suchbäume mit genau n Knoten und den n verschiedenen Knoteninhalten 0, 1, 2, ..., n-1. Bilde die Menge $S_n = \{i_1 i_2 ... i_n \mid i_j \in \underline{\underline{n}}, i_j \neq i_k \text{ für } j \neq k\}$. Dies ist genau die Menge der Historien aller binären Suchbäume mit n Knoten.
S_n ist die Menge aller *Permutationen* von n Zahlen (hier von 0 bis n-1); Näheres siehe Abschnitt 6.5.6.

Jeder Permutation (oder Anordnung) $i_1 i_2 ... i_n$ ist genau ein binärer Suchbaum zugeordnet, der durch Einfügen hieraus entsteht und zu dessen Historie sie gehört. Es gibt also eine (surjektive) Abbildung $\varphi: S_n \to D_n$.

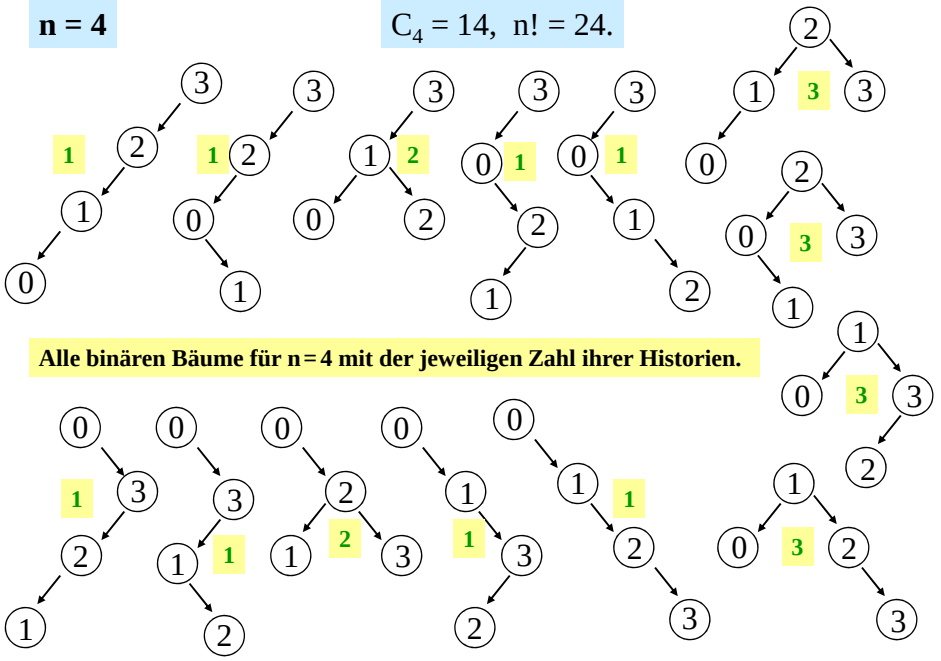
Formal genauer: Die Historie $h = (IN,i_1),(IN,i_2),...,(IN,i_n)$, die wir mit der Folge $i_1 i_2 ... i_n$ identifizieren (mit $i_j \neq i_k$ für $j \neq k$), liefert genau den binären Suchbaum $\varphi(h) \in D_n$.

Für die inverse Abbildung $\varphi^{-1}: D_n \to 2^{S_n}$ gilt dann:
$|\varphi^{-1}(d)|$ ist die Entropie des binären Suchbaums $d \in D_n$.
$(\varphi^{-1}(d) = \{ i_1 i_2 ... i_n \mid \varphi(i_1 i_2 ... i_n) = d\} \subset S_n .)$

Faustregel: Je gleichverzweigter der Suchbaum d ist, umso größer ist seine Entropie.

Dies beleuchten wir zunächst am Beispiel n = 4.

n = 4 $C_4 = 14$, n! = 24.



Alle binären Bäume für n = 4 mit der jeweiligen Zahl ihrer Historien.

Da S_n die Menge aller Permutationen der ordnung n ist und $|S_n| = n!$ ist, ergibt sich: Die Entropien stellen den Zusammenhang zwischen der Fakultät n! und der Catalanschen Zahl C_n her:

n! = Summe der Entropien aller binären Suchbäume.

Dies erkennt man auch am Beispiel $n = 4$.

Damit erhält man zugleich die "durchschnittliche Entropie" eines Suchbaums als den Wert $n!/C_n$.

Von dieser durchschnittlichen Entropie weichen die einzelnen Suchbäume mehr oder weniger stark ab. Die beiden Extremfälle sollen nun untersucht werden.

Wie viele Historien kann ein binärer Suchbaum mit n Knoten minimal und maximal besitzen?

Fall 1: Der Baum hat die Tiefe n. Er bildet dann eine lineare Liste, die nur *genau eine Historie* besitzt, nämlich die Folge der Zahlen von der Wurzel bis zum einzigen Blatt.

Fall 2: Für $n=2^k-1$ kann man den vollständig ausgeglichenen Baum mit genau 2^{k-1} Blättern betrachten (in 8.4.10 hatten wir diesen Baum mit T_k bezeichnet). Es sei E_k die Zahl der Historien eines solchen Baumes mit 2^k-1 Knoten. Dann gilt:

$E_1 = 1$ und

$$E_k = \binom{2^k-2}{2^{k-1}-1} \cdot E_{k-1} \cdot E_{k-1} \quad \text{für } k>1.$$

$E_1 = 1$ und

$$E_k = \binom{2^k-2}{2^{k-1}-1} \cdot E_{k-1} \cdot E_{k-1} \quad \text{für } k>1.$$

Begründung: $E_1 = 1$ ist klar, weil es nur einen Knoten gibt.
Es liege nun ein binärer Suchbaum mit 2^k-1 Knoten k>1 vor, dessen Wurzel den Inhalt z besitzt.
Dann muss z am Anfang der Historie stehen und man kann für die $2^{k-1}-1$ Inhalte des linken Unterbaums der Wurzel irgendeine beliebige Teilmenge der 2^k-2 Positionen in der Historie festlegen. Dies sind "(2^k-2) über $(2^{k-1}-1)$" Möglichkeiten. Auf den $2^{k-1}-1$ Positionen gibt es E_{k-1} Möglichkeiten der Zuordnung zu den Inhalten < z und ebenso viele Möglichkeiten für die Zuordnung der Inhalte > z zu den restlichen $2^{k-1}-1$ Positionen. Diese Auswahlen kann man unabhängig voneinander treffen, woraus sich die Rekursionsfomel für E_k ergibt.

E_k ist eine schnell wachsende Funktion. Die ersten Werte:

k = 1 und n = 1: $E_1 = 1$;	n! = 1.
k = 2 und n = 3: $E_2 = 2$;	n! = 6.
k = 3 und n = 7: $E_3 = 80$;	n! = 5040.
k = 4 und n=15: $E_4 = 21.964.800$;	n! = 1.307.674.368.000.
k = 5 und n=31: $E_5 \approx 7{,}5 \cdot 10^{22}$;	$n! \approx 8{,}222838654 \cdot 10^{33}$.

Vergleichswert: Der durchschnittliche Wert für die Entropie eines binären Suchbaumes mit n Knoten beträgt $n!/C_n$. Für $n = 2^k-1$ liegt der Wert E_k deutlich über diesem Wert. Beispiel:

Für $k = 5$ und $n = 31$ ist $C_n \approx 4^{31}/315.795$ und $n!/C_n \approx 5{,}63 \cdot 10^{17}$, d.h., im Durchschnitt hat jeder Suchbaum mit "nur" 31 Knoten rund $5{,}63 \cdot 10^{17}$ Historien. Der Wert für E_5 ist um etwas mehr als das 10^5-fache größer als die durchschnittliche Entropie.

Für $k > 2$ gilt: $(E_k)^2 > (2^k\text{-}1)!$, wie man mit Induktion unter Verwendung der Stirlingschen Formel beweist.

Aufgabe für Fortschrittenere und/oder an der Theorie Interessierte: Untersuchen Sie die Funktionswerte E_k selbst weiter und versuchen Sie, gute Abschätzungen für diese Werte zu finden.
Hinweis: "$(2^k\text{-}2)$ über $(2^{k-1}\text{-}1)$" hat die Form "2n über n", die wir bereits in 8.2.7 angenähert hatten.

Erinnerung: Die Zahl der Historien eines binären Suchbaums ist gleich der Zahl der topologischen Sortierungen dieses Baums (vgl. 8.7.3.a).

Allgemeine Aussagen:

Jeder binäre Suchbaum mit n Knoten, der eine lineare Liste ist (dessen Tiefe also genau n ist), besitzt genau eine Historie. (Es gibt genau 2^{n-1} solche "linearen" Suchbäume.)

Ein binärer Suchbaum mit $2^k\text{-}1$ Knoten besitzt höchstens E_k Historien.

Alle binären Suchbäume mit n Knoten zusammen besitzen genau $|S_n| = n!$ Historien.

Es gibt genau C_n verschiedene binäre Suchbäume mit n Knoten (Satz 8.2.7). Im Durchschnitt entfallen somit $n!/C_n$ Historien auf jeden binären Suchbaum. Diese Zahl wächst mindestens mit $(n/11)^n$, wie auf der folgenden Folie bewiesen wird.

Es gilt natürlich stets $E_k \geq (2^k\text{-}1)!/C_{2^k-1}$. (Beweis? Selbst versuchen.)

8.7.6: Mittlere Entropie bei binären Suchbäumen:
Die Zahl aller Historien wächst sehr viel schneller als die Zahl der binären Bäume. Genauer:

Mit der Stirlingschen Formel ($e \approx 2{,}718\,281\,828\,459\,045$)

$$n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\,\pi\,n} \qquad \text{erhält man:}$$

$$\frac{n!}{C_n} = \frac{n!}{\frac{1}{n+1}\binom{2n}{n}} \approx \ldots = \pi\sqrt{2}\cdot n\cdot(n+1)\cdot\left(\frac{n}{4e}\right)^n$$

$$\approx 4.442882938\cdot n\cdot(n+1)\cdot\left(\frac{n}{10.873127314}\right)^n$$

Diese Näherung ist auch für kleine Werte von n recht genau.

8.7.7: Mittlere Suchdauer und mittlere Tiefe binärer Bäume

Mittleres Level eines Knotens in einem binären Baum, wobei jeder binäre Baum mit seiner Entropie gewichtet wird
= Mittelwert der Suche in binären Bäumen bezogen auf alle möglichen Eingabefolgen aus n verschiedenen Elementen
= (Summe über das Level aller Knoten von allen binären Bäumen mit n Knoten mal ihrer Entropie)/$(n\cdot n!)$
$\approx 1{,}3863\cdot\log(n) - 1{,}8456$ (= MS_n, *siehe 8.2.21).*

Da sich die mittlere Suche auf alle möglichen Eingabefolgen bezieht, wurde hierbei also jeder Baum so oft gezählt, wie er Eingabefolgen als Historie besitzt.
Daher ist der Wert $1{,}3863\cdot\log(n) - 1{,}8456$ nicht das mittlere Level eines Knotens ML_n, wenn man zufällig einen binären Baum und in ihm zufällig einen Knoten auswählt.

Wir beenden hiermit den Exkurs über die Entropie von Bäumen, die sich auf andere Datenstrukturen übertragen lässt. Die Untersuchung ergab zugleich eine Klärung über die auf Folgen (Historien) bezogene mittlere Suchdauer bei binären Bäumen, die deutlich kleiner ist als das mittlere Level eines beliebigen Knotens in einem beliebigen binären Baum.
Interessanterweise sind AVL-Bäume sowie gewichts-balancierte Bäume Strukturen mit einer hohen Entropie. Bei einer hohen Entropie erreicht man im Mittel nach der Anwendung einer Operation schneller eine äquivalente Struktur, als wenn eine geringe Entropie vorliegt. Daher lassen sich AVL-Bäume und andere entropiereiche Strukturen leicht (mit O(log(n)) als Aufwand) beim Ein- oder Ausfügen in eine gleichartige Struktur überführen, während beispielsweise lineare Listen beim Ein- oder Ausfügen einen Aufwand von O(n) erfordern.
Man kann nun je nach Anwendungsproblem nach Strukturen suchen, die ihre eigene Historie mitspeichern oder rasch vergessen und unter dieser Nebenbedingung sehr effizient bearbeitet werden können.
Hier gibt es noch viel Unbekanntes zu entdecken.

Anhang zu Kapitel 8:

8.8 Weitere Definitionen zu Graphen

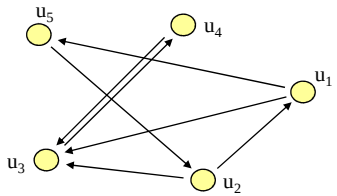
8.9 Sonstiges

8.8 Weitere Definitionen zu Graphen

Graphen wurden in Abschnitt 3.8 auf 41 Folien ausführlich erläutert. Gehen Sie bitte jene Folien nochmals genau durch. Die Begriffe ungerichteter Graph, gerichteter Graph, adjazent bzw. benachbart, inzident, (induzierter) Teilgraph, Grad, Weg, Kreis, Länge von Wegen, erreichbar, azyklisch, starke und schwache Zusammenhangskomponente, Adjazenzmatrix, Adjazenzliste (mit zugehöriger Datenstruktur), Inzidenzliste, transitive Hülle und Graphdurchlauf sollten Ihnen vertraut sein. (DAG = directed acyclic graph = gerichteter azyklischer Graph.)

Auf den folgenden Folien ergänzen wir jene Definitionen um Begriffe, die zum Teil in diesem Kapitel aufgetreten sind. Wir werden sie in den Übungen vertiefen. Hiermit werden zugleich die Graphalgorithmen in Kapitel 11 vorbereitet.

8.8.1 Erinnerung zur Darstellung von Graphen:
Graphen kann man durch ihre *Adjazenzmatrix* A, durch *Adjazenzlisten* oder durch *Inzidenzlisten* darstellen (siehe 3.8.5 g).



Diesen Graphen stellen wir als Adjazenzmatrix und als Adjazenzliste dar.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Adjazenzmatrix A

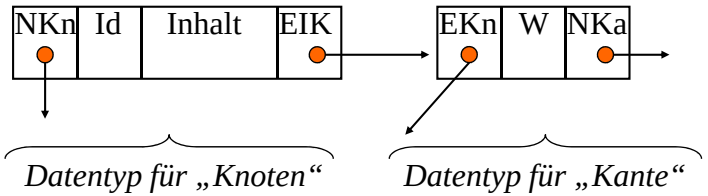
$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

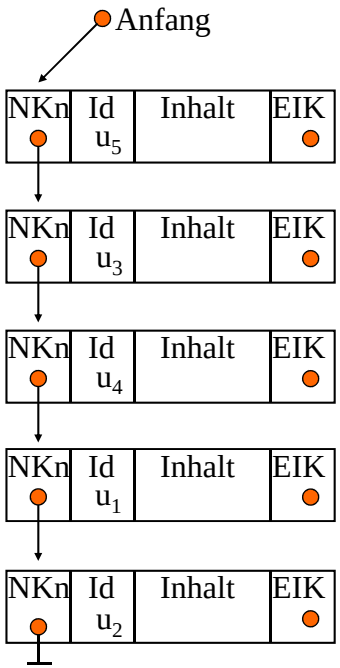
Erweiterte Adjazenzmatrix A'

Darstellung als Adjazenzliste (siehe 3.8.6)

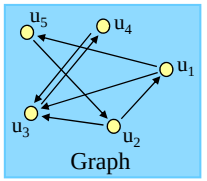
Jeder *Knoten* erhält einen Identifikator Id (z.B. einen Bezeichner oder eine natürliche Zahl) und einen Inhalt.
Die Knoten werden in einer Liste zusammengefasst und besitzen neben Id und Inhalt weitere Komponenten:
- Verweis auf den **N**ächsten **K**noten in der Liste "NKn",
- Verweis auf die **E**rste **I**nzidente **K**ante "EIK".

Jede von einem Knoten ausgehende *Kante* muss enthalten: den "Endknoten der Kante" (EKn), ihren Wert W ("weight") und einen Verweis auf die nächste Kante "NKa".





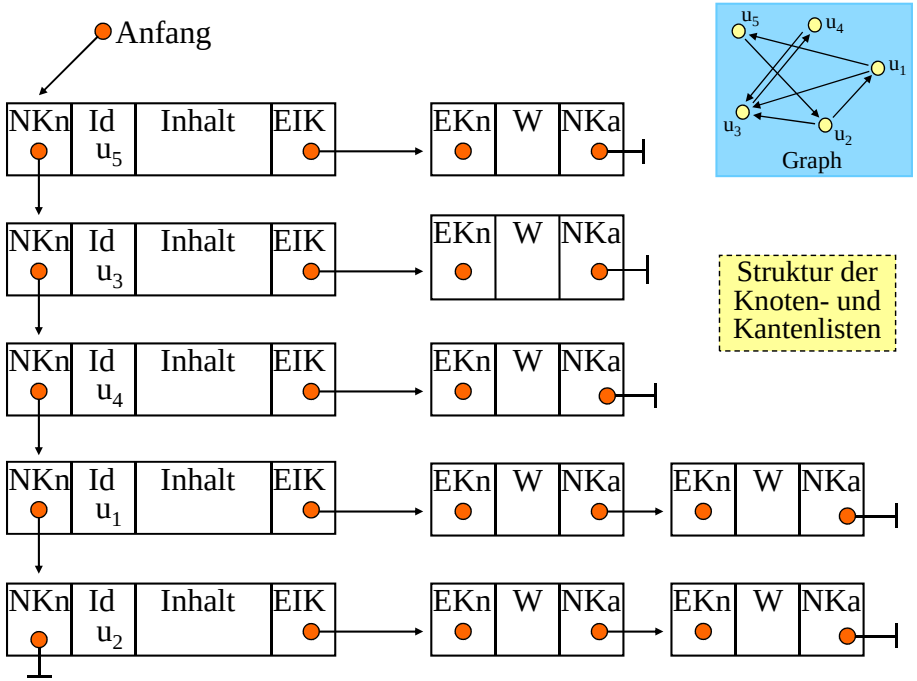
Beispiel

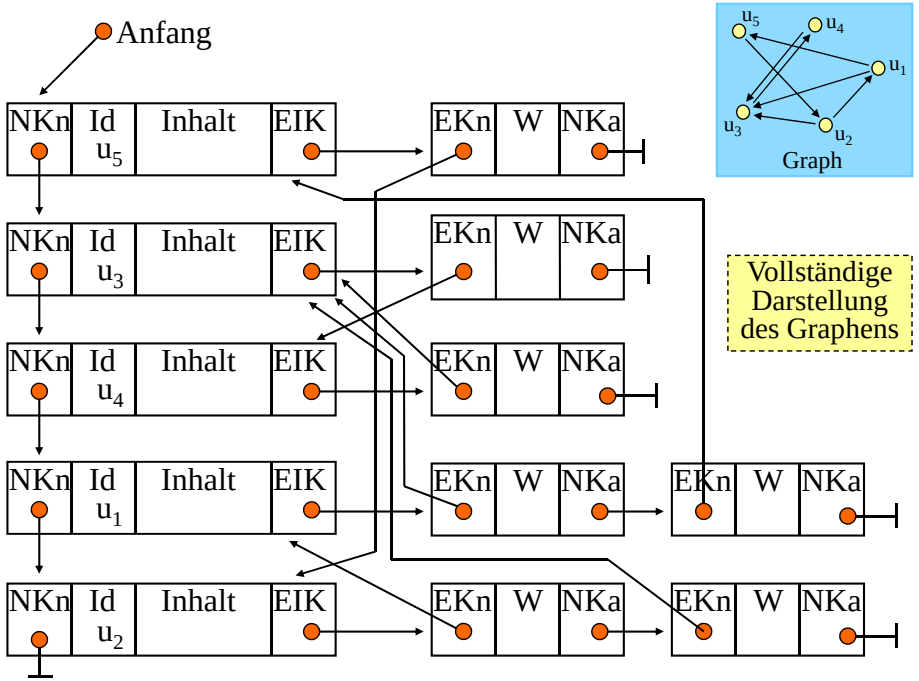


Zunächst werden die Knoten in beliebiger Reihenfolge in einer Liste gespeichert (siehe links).

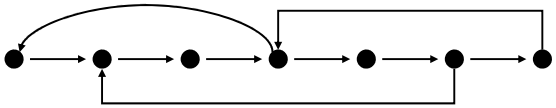
Danach (siehe nächste Folien) werden die Kanten über die EIK-Listen und dann die Endknoten eingetragen.

Wir fügen hier keine Werte für die Komponenten „Inhalt“ und „W“ ein.

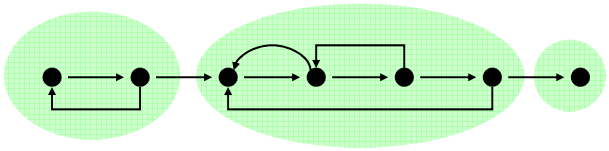




Erinnerung: Zusammenhang im gerichteten Fall



Dieser Graph ist stark zusammenhängend.



Dieser Graph besitzt drei starke Zusammenhangskomponenten.

Vgl. hierzu 3.8.14.

Definition 8.8.2: Topologische Sortierung von DAGs

[*Erinnerung an 3.8.13:* Zu einem Graphen $G = (V, E)$ heißt
$G^* = (V, E^*)$ die transitive Hülle, wenn im ungerichteten Fall gilt
$E^* = \{\{u,v\} \mid u \neq v$ und es gibt einen Weg von u nach v in $G\}$
bzw. im gerichteten Fall gilt
$E^* = \{(u,v) \mid u \neq v$ und es gibt einen Weg von u nach v in $G\}$.
Stets gilt natürlich $E \subseteq E^*$.
Warshall-Algorithmus, um E^* zu berechnen: siehe 6.5.5.]

Es sei $G = (V,E)$ gerichtet. Eine Abbildung $ord: V \to \mathbb{N}$ mit
$\forall\, u, v \in V$ mit $u \neq v$ gilt: $(u,v) \in E^* \Rightarrow ord(u) < ord(v)$
heißt topologische Sortierung von G.
Man ordnet also die Knoten so an, dass jeder von u aus erreichbare Knoten eine höhere Nummer als u bekommt.

Genau jeder azyklische gerichtete Graph („DAG“) besitzt eine topologische Sortierung; sie ist nicht eindeutig (selbst am Beispiel klar machen!). Es genügt, $ord: V \to \{1, 2, ..., |V|\}$ zu betrachten.

Definition 8.8.3: Markierte oder gewichtete Graphen

In Anwendungen sind Graphen meist "markiert" oder "gewichtet", d.h., ihre Knoten und/oder ihre Kanten sind mit Werten ("Markierung" oder "Gewicht") aus einer Wertemenge W bzw. W' versehen: $\mu: V \to W$ und $\delta: E \to W'$.

Man schreibt $G = (V, E, \mu)$ bzw. $G = (V, E, \delta)$ bzw. $G = (V, E, \mu, \delta)$.
Meist sind die Markierungen ganze oder reelle Zahlen. Oft hat man mehr als nur zwei solche Abbildungen.

Kantenmarkierungen $\delta: E \to \mathbb{R}^{\geq 0}$ mit
$\mathbb{R}^{\geq 0}$ = Menge der nichtnegativen reellen Zahlen
bezeichnet man auch als Entfernungen (oder Distanzen).

[Die Summe aller Entfernungen nennt man das Gewicht des Graphens, siehe unten bei "minimaler Spannbaum".]

Definition 8.8.4: Länge von Wegen bzgl. δ, Abstand

Sei $G = (V, E, \delta)$ ein Graph. Die reellwertige Abbildung $\delta: E \to \mathbb{R}$ wird auf die Menge der Wege fortgesetzt durch
$\delta((u_0, u_1, ..., u_k)) := \delta((u_0, u_1)) + \delta((u_1, u_2)) + ... + \delta((u_{k-1}, u_k))$.
Dieser Wert heißt die Länge des Weges $(u_0, u_1, ..., u_k)$.

Sei $\delta: E \to \mathbb{R}^{\geq 0}$ (= Menge der nichtnegativen reellen Zahlen), dann bezeichnet man für den Graphen $G = (V, E, \delta)$ die (kürzeste) Entfernung oder den Abstand oder die Distanz eines Knotens u zum Knoten v den Wert $\delta: V \times V \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ mit

$\delta(u,v) = 0$, für $u = v$,
$\delta(u,v) = \text{Min}\,\{\delta((u_0, u_1, ..., u_k)) \mid u = u_0, v = u_k\}$, sofern es mindestens einen Weg von u nach v gibt,
$\delta(u,v) = \infty$, falls es keinen Weg von u nach v gibt.

Statt $\delta(u,v)$ schreibt man oft $dist(u,v)$ oder $d(u,v)$.
Die maximale Distanz in einem Graphen bezeichnet man auch als den Durchmesser des Graphen.

8.8.5: Kürzeste Wege

Die Aufgabe, den Abstand dist(u,v) vom Knoten u zum Knoten v und einen Weg von u nach v mit der Länge dist(u,v) zu ermitteln, bezeichnet man als das Kürzeste-Wege-Problem.

Hierbei unterscheidet man die Probleme:

SSSP = single source shortest paths
= alle kürzesten Wege, die von einem gegebenen Knoten zu jedem anderen Knoten führen,

SPSP = single pair shortest path
= kürzester Weg zwischen zwei gegebenen Knoten u und v

APSP = all pair shortest paths
= alle kürzesten Wege zwischen allen Paaren (u,v) von Knoten

Definition 8.8.6: (minimaler) Spannbaum, Gewicht

Sei $G=(V,E)$ ein zusammenhängender gerichteter oder ungerichteter Graph. Ein (gerichteter bzw. ungerichteter) Teilgraph $B=(V,E_B)$, der ein Baum ist, heißt Spannbaum oder aufspannender oder spannender Baum von G (beachte: in B kommen alle Knoten des Graphens vor).

Es sei $G=(V,E,\delta)$ ein Kanten-markierter Graph mit $\delta: E \to \mathbb{R}$. Die Summe aller seiner Entfernungen $\delta(G)$

$$\delta(G) := \sum_{e \in E} \delta(e)$$

heißt das Gewicht des Graphen G.

Ein Spannbaum $B=(V,E_B,\delta)$ des Graphen $G=(V,E,\delta)$ heißt minimaler Spannbaum (engl.: minimal spanning tree) von G, wenn $\delta(B) \le \delta(B')$ für alle Spannbäume B' von G gilt.

Definition 8.8.7: Hamiltonsche und Eulersche Wege

Ein Weg $(u_0, u_1, ..., u_{n-1})$ in einem Graphen G mit n Knoten heißt Hamiltonscher Weg, wenn er jeden Knoten genau einmal enthält. Ein Weg $(u_0, u_1, ..., u_{n-1}, u_0)$ heißt Hamiltonscher Kreis, wenn $(u_0, u_1, ..., u_{n-1})$ ein Hamiltonscher Weg ist.

Ein Weg $(u_0, u_1, ..., u_k)$ heißt Eulerscher Weg, wenn jede Kante des Graphen genau einmal in ihm vorkommt. Er heißt Eulerscher Kreis, wenn zusätzlich $u_k = u_0$ ist.

Die Bestimmung Hamiltonscher Wege ist schwierig, diejenige Eulerscher Wege dagegen leicht, siehe unten.

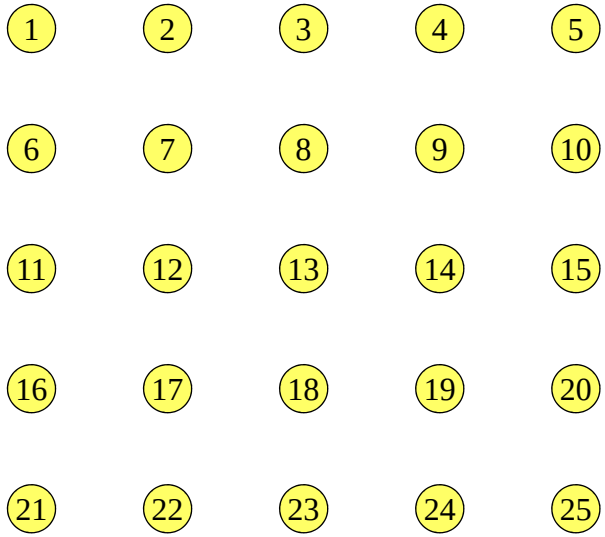
8.8.8 Einige Fragen und ihre Lösungen.

Wir wollen folgende Aussagen erläutern.

1. Die Zahl der verschiedenen (doppelpunktfreien) Wege zwischen zwei festen Knoten u und v in einem Graphen mit n Knoten und höchstens 2n Kanten kann bereits exponentiell wachsen. Wir geben ein Beispiel mit mehr als $4^{\sqrt{n}}$ solchen Wegen an („Gitter“).

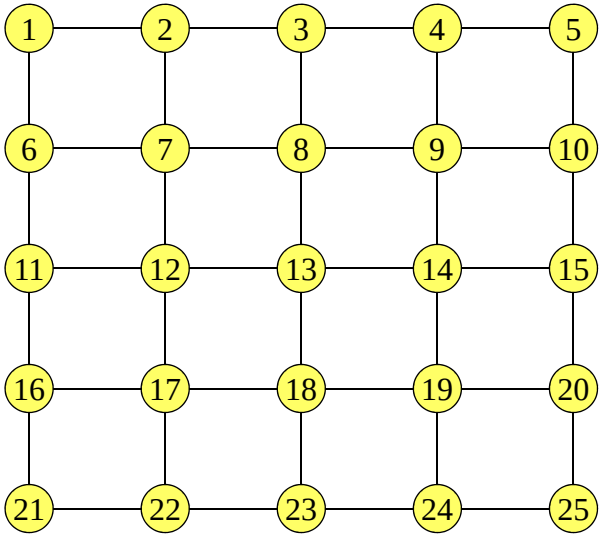
2. Hamiltonsche Wege/Kreise sind nicht leicht zu finden.
   (Manche von Ihnen werden später beweisen, dass dieses Problem „NP-hart“ ist und daher nach heutiger Meinung nur mit exponentiellem Aufwand gelöst werden kann.)

3. Eulersche Wege/Kreise sind dagegen leicht zu finden.
   (Wir erläutern dies nur an einem Beispiel. Den Beweis des Satzes sollten Sie selbst versuchen oder in einem Lehrbuch nachlesen.)



Wir betrachten im Folgenden diese Knotenmenge {1, 2, 3, ... , 25}.

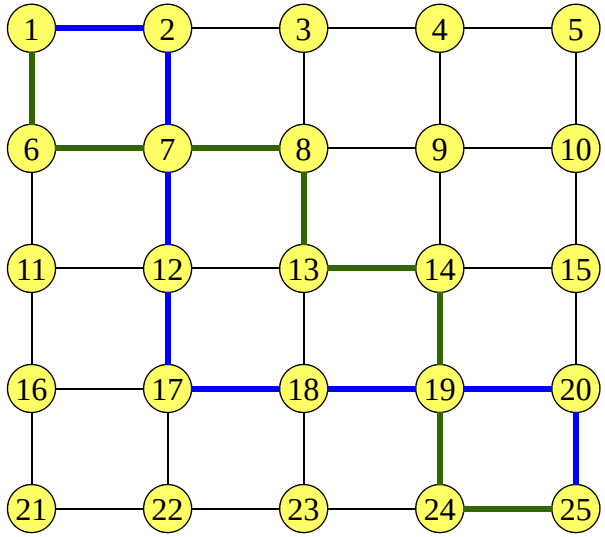
zu 1. Berechne die Anzahl von Wegen zwischen zwei Knoten



Als Beispiel wählen wir ein ungerichtetes „Gitter“.

25 Knoten, 40 Kanten.

Wie viele verschiedene Wege gibt es von Knoten 1 nach Knoten 25?



25 Knoten, 40 Kanten.

Es sind *mindestens*

$$\binom{10}{5} = (10\cdot9\cdot8\cdot7\cdot6) / (1\cdot2\cdot3\cdot4\cdot5) = 252 \text{ Wege.}$$

Allgemein: Wenn ein k×k-Gitter mit $k^2 = n$ vielen Knoten gegeben ist, so gibt es mindestens $\binom{2k}{k}$ doppelpunktfreie Wege zwischen den beiden Knoten 1 und k^2.

Warum? Man kann genau k-mal eine Kante nach rechts („r“) und genau k-mal eine Kante nach unten („u“) gehen. Die Reihenfolge ist beliebig. Dies gibt eine Folge der Länge 2k bestehend aus je k Buchstaben „r“ und „u“. Man kann aus den 2k Positionen beliebig k Stellen für den Buchstaben „r“ auswählen (auf die restlichen Stellen kommt dann der Buchstabe „u“). Man erhält genau „2k über k“ Möglichkeiten. Dies ist aber nur ein Bruchteil der tatsächlichen Möglichkeiten, da man ja auch nach links und nach oben gehen kann.

Die Zahl der doppelpunktfreien Wege wächst mindestens exponentiell mit $\sqrt{n}$. Denn es ist:

$$\binom{\mathbf{2k}}{\mathbf{k}} = \frac{(2k)!}{k! \cdot k!}$$

Mit der Stirlingschen Formel (6.5.6.3) für die Fakultät

$$k! \approx \left(\frac{k}{e}\right)^k \cdot \sqrt{2\cdot\pi\cdot k}$$

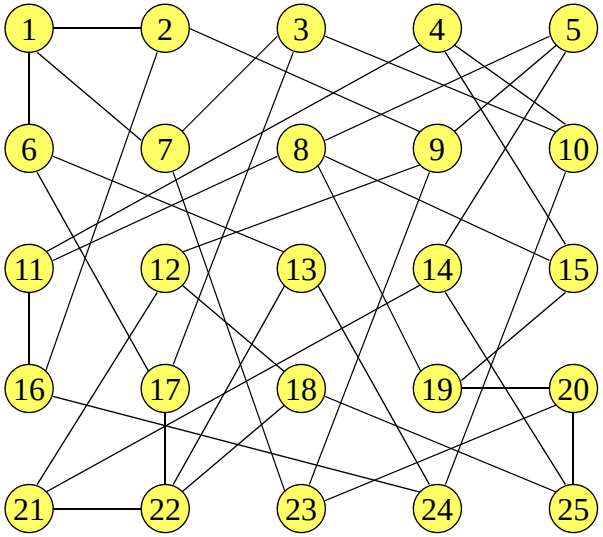
mit e = 2,71828182845904...
und π = 3,14159265358979...

folgt hieraus durch Einsetzen und Ausrechnen:

$$\binom{\mathbf{2k}}{\mathbf{k}} \approx \frac{\mathbf{4^k}}{\sqrt{\boldsymbol{\pi}\cdot\mathbf{k}}}$$

mit $k^2 = n$.

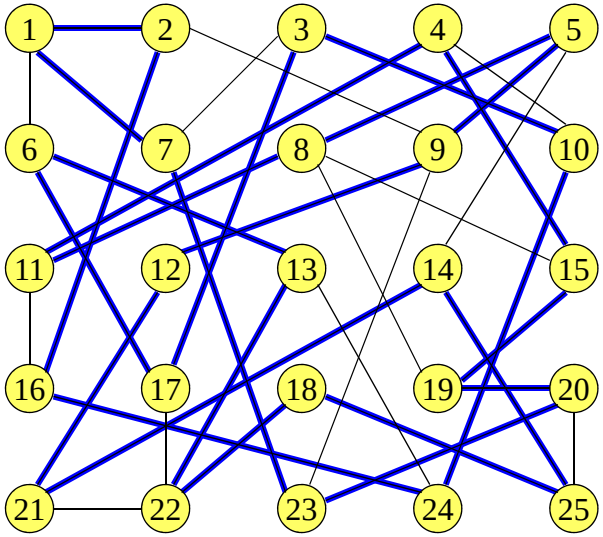
zu 2. Finde einen Weg, der jeden Knoten genau einmal besucht



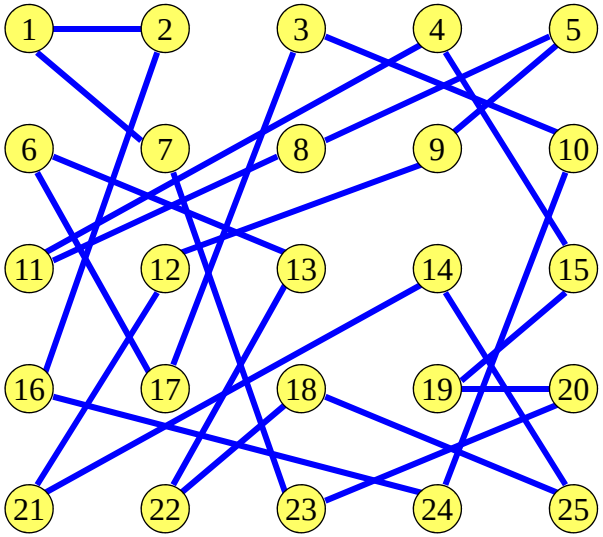
Knoten: 25

Kanten: 39

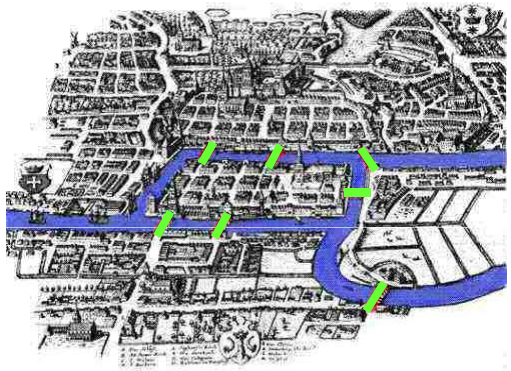
Gibt es in diesem Graphen einen Hamiltonschen Kreis? **Ja.**



Kreis: 1, 2, 16, 24, 10, 3, 17, 6, 13, 22, 18, 25, 14, 21, 12, 9, 5, 8, 11, 4, 15, 19, 20, 23, 7, 1

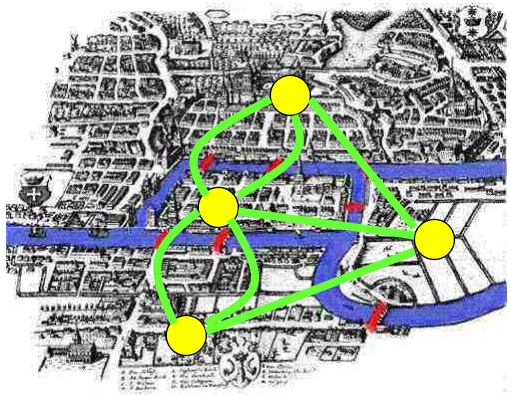


zu 3. Eulersche Kreise: Das Königsberger Brückenproblem (1736)

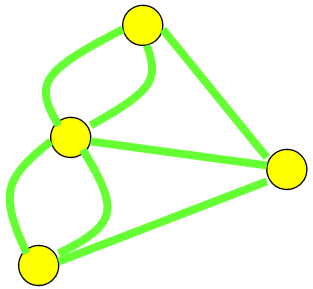


Kann man einen Rundgang durch Königsberg machen, so dass man jede der 7 Brücken genau einmal überquert und am Ende wieder am Startpunkt ankommt? Dieses Problem gilt als der Beginn der Graphentheorie.

Konstruiere hierzu einen Graphen (mit Mehrfachkanten)

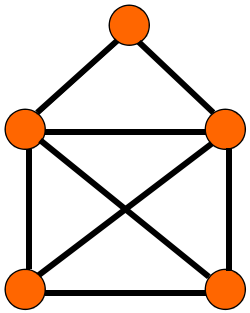


Satz: Ein zusammenhängender ungerichteter Graph besitzt genau dann einen Eulerschen Kreis, wenn alle seine Knoten einen geraden Grad haben. (Dies gilt auch für Graphen mit Mehrfachkanten.)



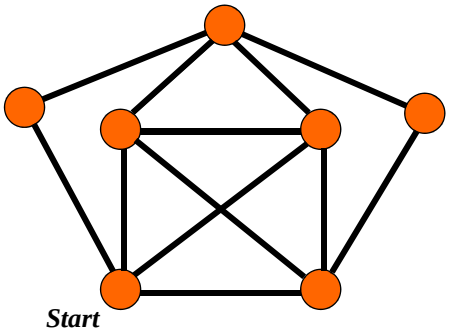
Eulers Beweis ergab: Es gibt keinen Rundgang durch Königsberg, auf dem jede Brücke genau einmal benutzt wird.

Zusatz: Haben genau zwei Knoten einen ungeraden Grad, so hat der Graph einen Eulerschen Weg, aber keinen Eulerschen Kreis.



Beispiel: "Dies ist das Haus des Nikolaus".

Durch Hinzunahme eines oder zweier Knoten mit geeigneten Kanten erhalten wir hieraus einen Graphen mit einem Eulerschen Kreis. Jeder Knoten erhält einen geraden Grad.



8.8.9 Durchsuchen eines Graphens

Gesucht ist ein Algorithmus, der an irgendeinem Knoten u beginnt und dann alle von u aus erreichbaren Knoten und Kanten besucht.
Im ungerichteten Fall wird hierbei genau die zu u gehörige Zusammenhangskomponente G(u) = (Z(u), E'(u)) durchlaufen, siehe 3.8.9. Im gerichteten Fall wird der von u aus erreichbare Teil der schwachen Zusammenhangskomponente (speziell: alle Knoten v mit (u,v)∈ E*, siehe transitive Hülle) besucht. Werden auf diese Weise nicht alle Knoten des Graphens erreicht, so muss man dieses Vorgehen mit irgendeinem bisher noch nicht besuchten Knoten fortsetzen. Vergleiche hierzu 3.8.11 und 12.

Wir fassen das bereits Bekannte zusammen und ergänzen es. Zum Durchlauf speziell von Bäumen siehe 8.2.8.

Ausgehend von einem Knoten u werden die Knoten und Kanten meist nach zwei Strategien aufgesucht:

Tiefensuche (**DFS** = depth first search): Erreicht man einen Knoten v, so wird ab hier rekursiv weiter gesucht, indem man allen von v ausgehenden Kanten folgt. Stößt man hierbei auf einen bereits besuchten Knoten, so wird in dieser Richtung nicht weiter gesucht (Abbruch der Rekursion).

Breitensuche (**BFS** = breadth first search): Man durchläuft den Graphen, ausgehend von u, schalenförmig gemäß des Abstandes, d.h., man besucht zunächst alle Knoten v mit dem Abstand dist(u,v) = 1, dann alle Knoten v mit dem Abstand dist(u,v) = 2 usw.

Tiefensuche (**DFS** = depth first search) umgangssprachlich.

Gerichteter Fall :

```
procedure besuche(u) is
begin  "bearbeite den Knoten u;"  markiere u als besucht;
       for alle v aus der Menge der Nachfolger S(u) loop
           if v noch nicht besucht then besuche(v); end if;
       end loop;
end besuche;
```

Im ungerichteten Fall ersetzt man die Menge S(u) durch die Menge der Nachbarn N(u), siehe 3.8.5e.

Wir betrachten im Folgenden nur den gerichteten Fall. Die Tiefensuche läuft im Prinzip wie im Algorithmus GD aus 3.8.7 ab. Zur Datenstruktur siehe 8.8.1 bzw. 3.8.6.

Rekursives Ada-Programm zur *Tiefensuche*, gerichteter Fall.

```
procedure DFS (Anfang: in NextKnoten) is
p: NextKnoten;                    -- Datentypen siehe am Ende von 3.8.6
procedure besuche (u: in NextKnoten) is
   edge: NextKante; v: NextKnoten;
   begin  u.Besucht := true; edge := u.EIK;
      while  edge /= null  loop  v := edge.EKn;
         if not v.Besucht then besuche(v); end if;
         edge := edge.NKa;  end loop;
   end besuche;
begin p := Anfang;
   while p /= null loop  p.Besucht := false;  p:=p.NKn;  end loop;
   p := Anfang;
   while p /= null loop  if not p.Besucht then besuche(p); end if;
      p := p.NKn;  end loop;
end DFS;
```


Iteratives Programm zur *Tiefensuche*, gerichteter Fall. Zu den Kellern vgl. Abschnitte 4.3.3 und 4.4.3 (das generische Paket "Stack" sei hier sichtbar).

```
procedure DFS (Anfang: in NextKnoten) is
p, v: NextKnoten; KK: new Stack (item => NextKnoten); edge: NextKante;
begin p := Anfang;
    while p /= null loop  p.Besucht := false; p:=p.NKn;  end loop;
    p := Anfang ;
    while p /= null loop
    if not p.Besucht then Push(p,KK);
        while not Isempty(KK) loop
            u := Top(KK); Pop(KK); u.Besucht := true; edge := u.EIK;
            while  edge /= null  loop  v := edge.EKn;
                if not v.Besucht then push(v,KK); end if;
                edge := edge.NKa;
            end loop;
        end loop;
    end if;
    p := p.NKn;
    end loop:
end DFS;
```


Iteratives Programm zur Breitensuche, gerichteter Fall: Wie Tiefensuche, aber ersetze den Keller durch eine Schlange! (blau und kursiv = Unterschied zu DFS)

```
procedure BFS (Anfang: in NextKnoten) is
p, v: NextKnoten; "KK: new Schlange (NextKnoten);" edge: NextKante;
begin p := Anfang;
    while p /= null loop  p.Besucht := false; p:=p.NKn;  end loop;
    p := Anfang ;
    while p /= null loop
    if not p.Besucht then Enter (p,KK);
        while not Isempty(KK) loop
            u := First(KK); Remove(KK); u.Besucht := true; edge := u.EIK;
            while  edge /= null  loop  v := edge.EKn;
                if not v.Besucht then Enter(v,KK); end if;
                edge := edge.NKa;
            end loop;
        end loop;
    end if;
    p := p.NKn;
    end loop:
end BFS;
```

Tiefensuche mit der Adjazenzmatrix A als Darstellung:

```
procedure DFS_Adj is                                   -- n ist hier global
A: array (0..n-1, 0..n-1) of Integer;
Besucht: array(0..n-1) of Boolean;
procedure besuche(j: in 0..n-1) is
    begin  Besucht(j) := true;
        for k in 0..n-1 loop
            if A(j,k) /= 0 and not Besucht(k)
            then besuche(k); end if;
        end loop;
    end besuche;
begin  ...          -- die Adjazenzmatrix A möge aufgebaut worden sein
    for i in 0..n-1 loop  Besucht(i) := false;  end loop;
    for i in 0..n-1 loop
        if not Besucht(i) then besuche(i); end if; end loop;
end DFS_Adj;
```


Zur Zeitkomplexität (n = Zahl der Knoten, m= Zahl der Kanten):

Adjazenzlistendarstellung: Jeder Durchlauf besucht jede Kante genau einmal. Jeder Knoten wird so oft besucht, wie Kanten auf ihn verweisen, mindestens aber einmal. Also ist die Zeitkomplexität linear O(n+m).

Adjazenzmatrix: Da eine Matrix angelegt werden muss, beträgt der Aufbau des Graphens $\Theta(n^2)$ Schritte, auch wenn er nur relativ wenige Kanten besitzt. In der Prozedur wird bei jedem Aufruf von "besuche" die for-k-Schleife n Mal durchlaufen; weil "besuche" mindestens n Mal aufgerufen werden muss, ergibt sich eine Zeitkomplexität von $O(n^2)$.

Überlegen Sie, wie viele Schritte die Verfahren für einen vollständigen Graphen K_n und für einen Graphen mit n isolierten Knoten (also ein Graph ohne Kanten) benötigen.

8.9 Sonstiges

8.9.1: Anregung für analytisch Interessierte

Wie kann man einer Rekursionsgleichung ansehen, welche Lösungen sie hat? (siehe Herleitung von Satz 8.2.21)

Das lässt sich nicht pauschal beantworten. Denn schließlich ist dieses Problem nicht entscheidbar. Manchmal reicht es aber bereits, wenn man die Gleichung umformt (erweitern, einsetzen, in irgendeinem Sinne vereinfachen usw.).

Oft ist es hilfreich, die "Ableitung" oder "Differenzen" zu betrachten, also F(n) - F(n-1), oder die "zweiten Differenzen" (F(n) - F(n-1)) - (F(n-1) - F(n-2)) = F(n) - 2 F(n-1) + F(n-2).

Dies liefert einen Hinweis, wie die Lösung aussehen könnte, und man kann dann mit einem Lösungansatz, den man in die Gleichung einsetzt, versuchen, eine Lösung aufzuspüren.

Wir betrachten als Beispiel die Rekursionsformel für F(n):

F(n) = (n+1)/n·F(n-1) + (2n-1)/n.

Einfaches Umformen liefert:

F(n) = F(n-1) + 1/n·F(n-1) + (2n-1)/n, d.h.:

F(n) - F(n-1) = 1/n·F(n-1) + (2n-1)/n .

Hier sieht man wenig, weil der Wert F(n-1) noch auf der rechten Seite stehen geblieben ist.

Daher versuchen wir nun, die zweiten Differenzen zu berechnen.

Aus obiger Formel für F(n)-F(n-1) folgt:

F(n-1) - F(n-2) = 1/(n-1)·F(n-2) + (2n-3)/(n-1).

Man subtrahiere die letzte Formel von der davor:

(F(n) - F(n-1)) - (F(n-1) - F(n-2))

= 1/n·F(n-1) + (2n-1)/n - 1/(n-1)·F(n-2) - (2n-3)/(n-1)

= 1/n·F(n-1) - 1/(n-1)·F(n-2) + 1/(n·(n-1)).

Der Term 1/n·F(n-1) - 1/(n-1)·F(n-2) erinnert nun an die Rekursionsformel, denn dort steht (ersetze n durch n+1):

1/(n+1)·F(n) = 1/n·F(n-1) + ...

Also gilt:
$1/n \cdot F(n-1) = 1/(n-1) \cdot F(n-2) + (2n-3)/(n \cdot (n-1))$ bzw.
$1/n \cdot F(n-1) - 1/(n-1) \cdot F(n-2) = (2n-3)/(n \cdot (n-1))$.
Dies setzt man oben ein:
$(F(n) - F(n-1)) - (F(n-1) - F(n-2))$
$= 1/n \cdot F(n-1) - 1/(n-1) \cdot F(n-2) + 1/(n \cdot (n-1))$
$= (2n-3)/(n \cdot (n-1)) + 1/(n \cdot (n-1))$
$= (2n-2)/(n \cdot (n-1)) = 2/n$

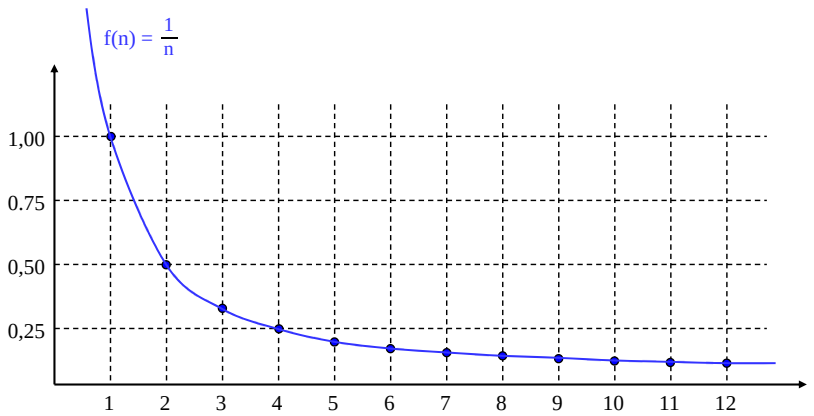
Es entsteht ein überraschend einfacher Ausdruck. Nun erinnern wir uns an die Stammfunktionen aus der Analysis: 1/n ist die Ableitung des Logarithmus ln(n). Also müsste die "Ableitung" ln(n) und damit die Funktion F von der Form $n \cdot \ln(n)$ sein. Da keine kontinuierliche Funktion herauskommen wird, sollte man die harmonische Funktion H(n) anstelle von ln(n) nehmen, d.h., das Ergebnis könnte die Form $F(n) = a \cdot n \cdot H(n) + b \cdot n + c$ (mit Konstanten a, b, c) haben.

Dies setzt man nun in die ursprüngliche Gleichung
$F(n) = (n+1)/n \cdot F(n-1) + (2n-1)/n$ ein:
$a \cdot n \cdot H(n) + b \cdot n + c$
$= (n+1)/n \cdot (a \cdot (n-1) \cdot H(n-1) + b \cdot (n-1) + c) + (2n-1)/n$
Multiplikation mit n liefert:
$a \cdot n^2 \cdot H(n) + b \cdot n^2 + c \cdot n$
$= (n+1) \cdot a \cdot (n-1) \cdot H(n-1) + b \cdot (n^2-1) + c \cdot (n+1) + (2n-1)$, also
$a \cdot n^2 \cdot H(n-1) + a \cdot n = a \cdot (n^2-1) \cdot H(n-1) - b + c + (2n-1)$
$a \cdot H(n-1) + a \cdot n = c - b + (2n-1)$.
Man sieht, dass diese Gleichung für Konstanten a, b und c nicht zu erfüllen ist. Man braucht offenbar im Ansatz ein weiteres Glied $d \cdot H(n)$. Neuer Ansatz:
$F(n) = a \cdot n \cdot H(n) + b \cdot n + c + d \cdot H(n)$.
Wegen der Nebenbedingung F(0)=0 muss c=0 sein. Erneut einsetzen:

$a \cdot n \cdot H(n) + b \cdot n + d \cdot H(n)$
$= (n+1)/n \cdot (a \cdot (n-1) \cdot H(n-1) + b \cdot (n-1) + d \cdot H(n-1)) + (2n-1)/n$

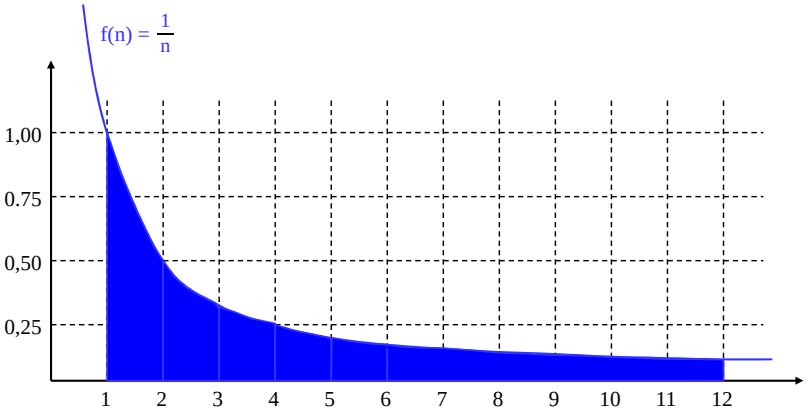
Multiplikation mit n liefert:
$a \cdot n^2 \cdot H(n) + b \cdot n^2 + d \cdot n \cdot H(n)$
$= a \cdot n^2 \cdot H(n-1) + a \cdot n + b \cdot n^2 + d \cdot n \cdot H(n)$
$= (n+1) \cdot a \cdot (n-1) \cdot H(n-1) + b \cdot (n^2-1) + d \cdot (n+1) \cdot H(n-1) + (2n-1)$
$= a \cdot (n^2-1) \cdot H(n-1) + b \cdot (n^2-1) + d \cdot n \cdot H(n-1) + d \cdot H(n-1) + (2n-1)$.
Umformen:
$a \cdot H(n-1) + a \cdot n + b \cdot n^2 + d \cdot n \cdot H(n)$
$= b \cdot (n^2-1) + d \cdot n \cdot H(n-1)) + d \cdot H(n-1)) + (2n-1)$ wird zu
$a \cdot H(n-1) + a \cdot n + d = -b + d \cdot H(n-1) + (2n-1)$.
Wenn dies lösbar ist, so muss a = d und a = 2 sein; es verbleibt:
$2 = -b - 1$, d.h., b = -3. Somit haben wir eine Lösung gefunden:
$F(n) = 2 \cdot n \cdot H(n) - 3 \cdot n + 2 \cdot H(n)$, also die gleiche Lösung wie im Beweis zu Satz 8.2.21.

8.9.2: Veranschaulichung der harmonischen Funktion 8.2.20



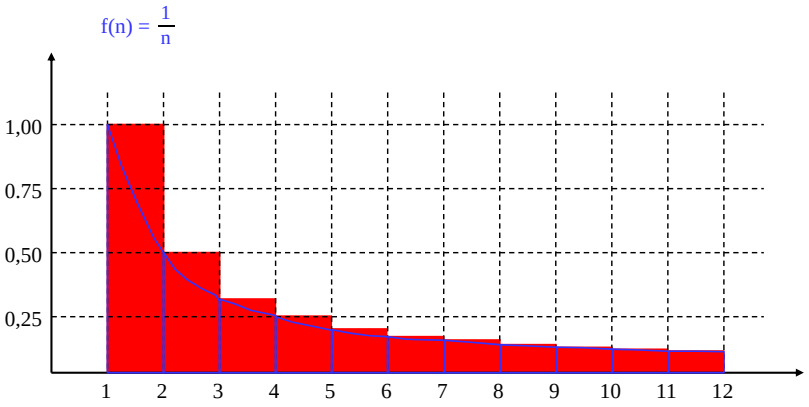
Eine ausführlichere Untersuchung steht in Abschnitt 1.5.2.

Abschätzung der harmonischen Funktion



Der natürliche Logarithmus ln(n) ist die Fläche unterhalb der Kurve von 1 bis n.

Abschätzung der harmonischen Funktion

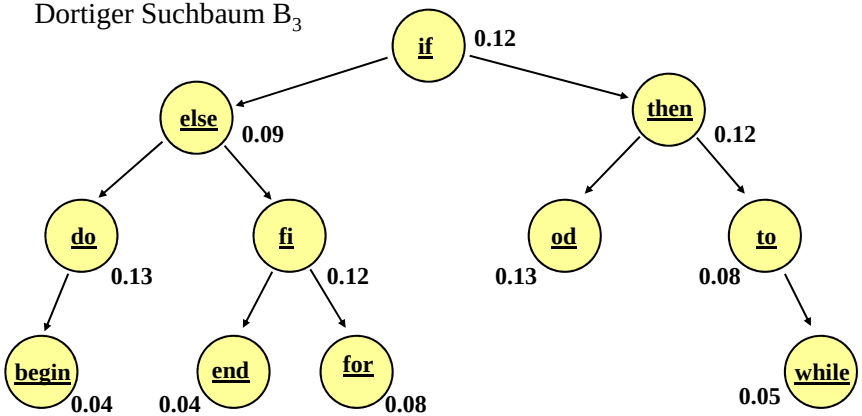


Der natürliche Logarithmus ln(n) ist die Fläche unterhalb der Kurve von 1 bis n.
Die harmonische Funktion H(n-1) ist durch die rote Fläche von 1 bis n gegeben.

Man sieht: $H(n) - 1 \le \ln(n) \le H(n-1)$ für alle $n \ge 1$. Die kleinen über der blauen Kurve liegenden roten Flächenstücke summieren sich zur Eulerschen Konstanten C = 0,5772156649... auf.

8.9.3: Lösung des Beispiels 8.3.3

Dortiger Suchbaum B_3



Gewichtete mittlere Suchdauer von B_3: 2.76.
Dieser Wert ist tatsächlich optimal.

Zugehöriges Ada-Programm in Laufzeit $\Theta(n^3)$:

```
with Ada.Text_IO; use Ada.Text_IO;  with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Ada.Float_Text_Io; use Ada.Float_Text_Io;

procedure Opt_Suchbaum_Inf2 is
  N: constant Integer := 11;
  subtype Index is Integer range 0..N+1;
  S, G: array (Index, Index) of Float := (others => (others => 0.0));
  R: array (Index, Index) of Index := (others => (others => 0));
  P: array (Index) of Float;      -- wird eingelesen bzw. in Prozedur "Beispiel" gefüllt
  Min: Float;  K_Min, J: Index;
  procedure Ausdruck is                                  --  hier wird die Matrix R gedruckt
    begin New_Line; Put("Matrix R"); New_Line;
    for I in 1..N loop
      New_Line;  for J in 1..N loop Put(R(I,J),4); end loop;
    end loop;
  end;
  procedure Beispiel is        -- das Beispiel 8.3.3 aus der Vorlesung
  begin  P(0) := 0.0;  P(12) := 0.0;
    P(1) := 0.04;  P(2) := 0.13;  P(3) := 0.09;  P(4) := 0.04;  P(5) := 0.12;  P(6) := 0.08;
    P(7) := 0.12;  P(8) := 0.13;  P(9) := 0.12;  P(10) := 0.08;  P(11) := 0.05;
  end;
```

```
begin
   Beispiel;
   for i in 1..N loop S(i+1,i) := 0.0; S(i,i) := P(i); G(i,i) := P(i); R(i,i) := i; end loop;
   for diff in 1..N-1 loop
      for i in 1..N-diff loop
         j := i + diff;  G(i,j) := G(i,j-1) + P(j);  min := S(i+1,j);  k_min := i;
         for k in i+1..j loop
            if  S(i,k-1) + S(k+1,j) < min  then
               min := S(i,k-1) + S(k+1,j);  k_min := k;
            end if;
         end loop;
         S(i,j) := min + G(i,j);  R(i,j) := k_min;
      end loop;
   end loop;
   New_Line; Put("Minimale Suchdauer: "); Put(S(1,N),3,4,0); New_Line;
   Ausdruck;
end;
```

Das schnellere Programm, in dem die Monotonie der Wurzeln ausgenutzt wird, unterscheidet sich in den Grenzen für k in der innersten Schleife und bei der Initialisierung von min und k_min.

Das Ada-Programm liefert als minimale Suchdauer: 2.76.
Als zugehörige Matrix der Wurzeln R wurde ausgedruckt:

```
Matrix R
   1   2   2   2   3   3   5   5   5   5   7
   0   2   2   2   3   3   5   5   5   7   7
   0   0   3   3   5   5   5   5   7   7   7
   0   0   0   4   5   5   5   7   7   7   7
   0   0   0   0   5   5   6   7   7   7   7
   0   0   0   0   0   6   7   7   8   8   8
   0   0   0   0   0   0   7   8   8   8   8
   0   0   0   0   0   0   0   8   8   9   9
   0   0   0   0   0   0   0   0   9   9  10
   0   0   0   0   0   0   0   0   0  10  10
   0   0   0   0   0   0   0   0   0   0  11
```

Liefert: Wurzel ist das 7. Element, also "if".
Links im Bereich 1..6 ist das 3. Element "else" die Wurzel;
rechts im Bereich 8..11 ist es das 9. Element "then";
usw. So kann man den Baum aus dieser Matrix R konstruieren.

9. Hashing

Ziel dieses 9. Kapitels:

Beim Hashing werden die Elemente einer Menge nicht wie bei einem Suchbaum sortiert und durch Vergleiche wiedergefunden, sondern man berechnet mit Hilfe einer "Hashfunktion" aus dem Schlüssel einen Index (eine "Adresse"), unter dem der Schlüssel in einer "Hash"-Tabelle gespeichert wird oder zu finden ist.

In diesem Kapitel lernen Sie, welche Eigenschaften solch eine Hashfunktion besitzen muss, welche Funktionen in der Praxis eingesetzt werden, wie man durch Freihalten von Speicherplatz im Mittel eine konstante Such- und Einfügezeit erreicht, wie man effizient löscht und wie man den Speicherbereich in linearer Zeit dynamisch vergrößern kann. Zugleich werden Ihnen die hierfür benötigten Parameter (Tabellengröße, Auslastungsgrad, Kollisionsstrategien, Zyklenlänge) und ihre Bedeutung für Anwendungen vermittelt.

9.1 Einführung

9.1.1. Grundidee: Schlüssel sollen durch einen einzigen Zugriff auf eine Tabelle (mit maximal p Einträgen) gefunden werden.

Gegeben sei eine Menge möglicher Schlüssel S und die Tabellengröße, dies ist eine natürliche Zahl p << |S|.

Es ist eine Abbildung f: S → {0, 1, ..., p-1} zu konstruieren, sodass es in einer zufällig ausgewählten n-elementigen Teilmenge $B = \{b_1, ..., b_n\} \subseteq S$ (mit $n \leq p$) im Mittel nur wenige Elemente $b_i \neq b_j$ mit $f(b_i) = f(b_j)$ gibt.

Die drei Operationen Suchen, Einfügen und Löschen (siehe Anfang von Kap. 8) müssen sehr "effizient" realisiert werden:

- Entscheide, ob $s \in S$ in B liegt (und gib an, wo). FIND
- Füge einen Schlüssel s in B ein. INSERT
- Entferne einen Schlüssel s aus B. DELETE

Für die Abbildung f: S → {0, 1, ..., p-1} müssen wir daher mindestens folgendes fordern:

- Sie muss surjektiv sein.
- Sie muss "gleichverteilt" sein, d.h., für jedes $0 \leq m < p$ sollte die Menge $S_m = \{s \in S \mid f(s) = m\}$ ungefähr |S|/p Elemente enthalten.
- Sie muss schnell berechnet werden können.

Solch eine Abbildung f heißt Schlüsseltransformation oder **Hashfunktion**. (Genaueres siehe 9.4.1.)

Diese Funktion verstreut die möglichen Schlüssel über den Indexbereich {0, 1, ..., p-1}. Daher der Name:

Hashing = (über eine Tabelle) gestreute Speicherung

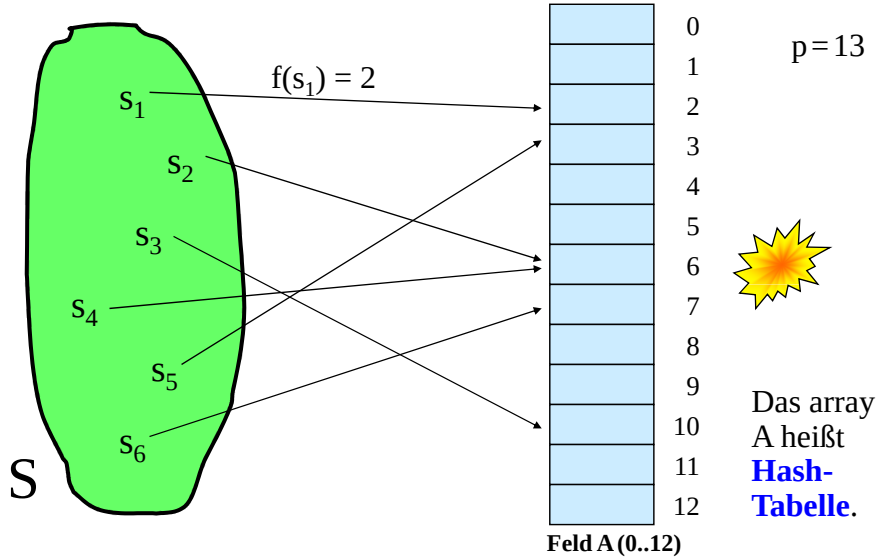
Es sei f: S → {0, 1, ..., p-1} eine solche Abbildung. Es sei
A: array (0..p-1) of <Datentyp zur Menge S>
ein Feld, in dem Teilmengen $B = \{b_1, ..., b_n\} \subseteq S$ (mit $n \leq p$) gespeichert werden können.

Jedes Element $s \in S$ wird unter der Adresse f(s) gespeichert, d.h., nach dem Speichern sollte $A(f(s)) = s$ gelten.

Um festzustellen, ob ein Schlüssel s in der jeweiligen Teilmenge liegt, braucht man nur zu prüfen, was in A(f(s)) steht. Doch es entstehen Probleme, wenn in der konkreten Teilmenge B mehrere Elemente mit gleichem f-Wert ("Kollisionen") enthalten sind.

Wie sieht es mit den Operationen INSERT und DELETE aus? Wir schauen uns zunächst eine Skizze und dann ein Beispiel an.

Es sollen 6 Elemente s_1 bis s_6 gespeichert werden:



Konflikt: Hier ist $f(s_2) = f(s_4) = 6$. Was nun?

9.1.2 Fallstudie mit dem Beispiel "modulo p"

$S = \Sigma^*$ = die Menge aller Folgen über einem t-elementigen Alphabet $\Sigma = \{\alpha_0, \alpha_1, ..., \alpha_{t-1}\}$.

Weiterhin sei p eine natürliche Zahl, p>1.

Eine nahe liegende Codierung $\varphi: \Sigma \to \{0, 1, ..., t\text{-}1\}$ lautet: Bilde jedes Zeichen α_i auf seinen Index i ab, also $\varphi(\alpha_i) = i$, und wähle als Abbildung $f: \Sigma^* \to \{0, 1, ..., p\text{-}1\}$ die Summe dieser Indizes für ein Teilwort, z. B. für das Anfangswort der Länge q (für ein q mit $0 < q \le r$):

$$f(\alpha_{i_1}\alpha_{i_2} ...\alpha_{i_r}) = \Big(\sum_{j=1}^{q} \varphi(\alpha_{i_j})\Big) \underline{\text{mod}}\ p.$$

Wir demonstrieren dies am lateinischen Alphabet, wobei wir nur die großen Buchstaben A, B, C, ... verwenden. Als Codierung φ wählen wir die Position des Buchstabens im Alphabet, und als Menge B die Menge der Monatsnamen:

a	φ(a)
A	1
B	2
C	3
D	4
E	5
F	6
G	7
H	8
I	9

a	φ(a)
J	10
K	11
L	12
M	13
N	14
O	15
P	16
Q	17
R	18

a	φ(a)
S	19
T	20
U	21
V	22
W	23
X	24
Y	25
Z	26

Abzubildende Menge B:

B = {JANUAR, FEBRUAR, MAERZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER}.

Wir berechnen nun $f(\alpha_{i_1}\alpha_{i_2} ...\alpha_{i_r}) = \Big(\sum_{j=1}^{q} \varphi(\alpha_{i_j})\Big) \underline{\text{mod}}\ p.$

für alle Wörter aus B. Zum Beispiel muss man für das Wort JANUAR als erstes die zu q gehörende Summe

$\varphi(J) = 10$ (im Falle q = 1),

$\varphi(J) + \varphi(A) = 10 + 1 = 11$ (im Falle q = 2),

$\varphi(J) + \varphi(A) + \varphi(N) = 10 + 1 + 14 = 25$ (im Falle q = 3),

$\varphi(J) + \varphi(A) + \varphi(N) + \varphi(U) = 46$ (im Falle q = 4) usw.

ermitteln. Wir listen zunächst diese Summen in der folgenden Tabelle für verschiedene q auf; hierbei kann man auch Buchstaben weglassen und z.B. nur den ersten und dritten Buchstaben betrachten ("1.+3."); später müssen wir diese Werte modulo p (siehe übernächste Tabelle, die nur drei der sechs Spalten aus der anderen Tabelle benutzt) nehmen.

Wir erhalten für q = 1, 2, 3, 4 und für "1. und 3.", "2. und 3." die Werte:

Monatsname	q=1	q=2	q=3	q=4	1.+3.	2.+3.
JANUAR	10	11	25	46	24	15
FEBRUAR	6	11	13	31	8	7
MAERZ	13	14	19	37	18	6
APRIL	1	17	35	44	19	34
MAI	13	14	23	23	22	10
JUNI	10	31	45	54	24	35
JULI	10	31	43	52	22	33
AUGUST	1	22	29	50	8	28
SEPTEMBER	19	24	40	60	35	21
OKTOBER	15	26	46	61	35	31
NOVEMBER	14	29	51	56	36	37
DEZEMBER	4	9	9	14	4	5

Wir verwenden nur die Spalten "q=2", "q=3" und "2.+3.", wählen als p die Zahlen 17 und 22 und erhalten:

Monatsname	q = 2 p=17	q = 3 p=17	2.+3. p=17	q = 2 p=22	q = 3 p=22	2.+3. p=22
JANUAR	11	8	15	11	3	15
FEBRUAR	11	13	7	11	13	7
MAERZ	14	2	6	14	19	6
APRIL	0	1	0	17	13	12
MAI	14	6	10	14	1	10
JUNI	14	11	1	9	1	13
JULI	14	9	16	9	21	11
AUGUST	5	12	11	0	7	6
SEPTEMBER	7	6	4	2	18	21
OKTOBER	9	12	14	4	2	9
NOVEMBER	12	0	3	7	7	15
DEZEMBER	9	9	5	9	9	5

In den Tabellen haben wir bereits die genannte Modifikation für f benutzt, indem wir Teilmengen der Indizes $\{1, 2, ..., r\}$ ausgewählt und die zugehörigen Buchstabenwerte aufsummiert haben. Dies waren die Teilmengen

$\{1, 3\}$, bezeichnet durch 1.+3. sowie
$\{2, 3\}$, bezeichnet durch 2.+3.

Die Abbildungen, die in den Spalten angegeben sind, sind untereinander nicht "besser" oder "schlechter", sondern sie sind nur von unterschiedlicher Qualität für unsere spezielle Menge B der Monatsnamen. Wir wählen nun irgendeine dieser Funktionen und fügen mit ihr die Monatsnamen in eine Tabelle (= in ein array A = in eine "Hashtabelle" A) mit p Komponenten ein, nummeriert von 0 bis p-1.

Für die Abbildung wählen wir willkürlich q=2 und p=22 und berechnen also die Hashfunktion

$f(\alpha_{i_1}\alpha_{i_2} ...\alpha_{i_r}) = \big(\varphi(\alpha_{i_1}) + \varphi(\alpha_{i_2}) \big)$ mod 22.

Zum Beispiel ist dann f(JANUAR) = (10+1) mod 22 = 11 und f(OKTOBER) = (15+11) mod 22 = 4. Alle Werte dieser Abbildung finden Sie in der entsprechenden Spalte für q=2 und p=22 auf der vorletzten Folie.

Die Monatsnamen tragen wir in ihrer jahreszeitlichen Reihenfolge nacheinander in das Feld A ein. Die Menge $S = \Sigma^*$ ist unendlich; doch nehmen wir hier an, dass die tatsächlich benutzten Wörter der Menge S höchstens die Länge 20 haben (kürzere Wörter werden durch Zwischenräume, deren φ-Wert 0 sei, hinten aufgefüllt) und deklarieren daher die Hashtabelle:

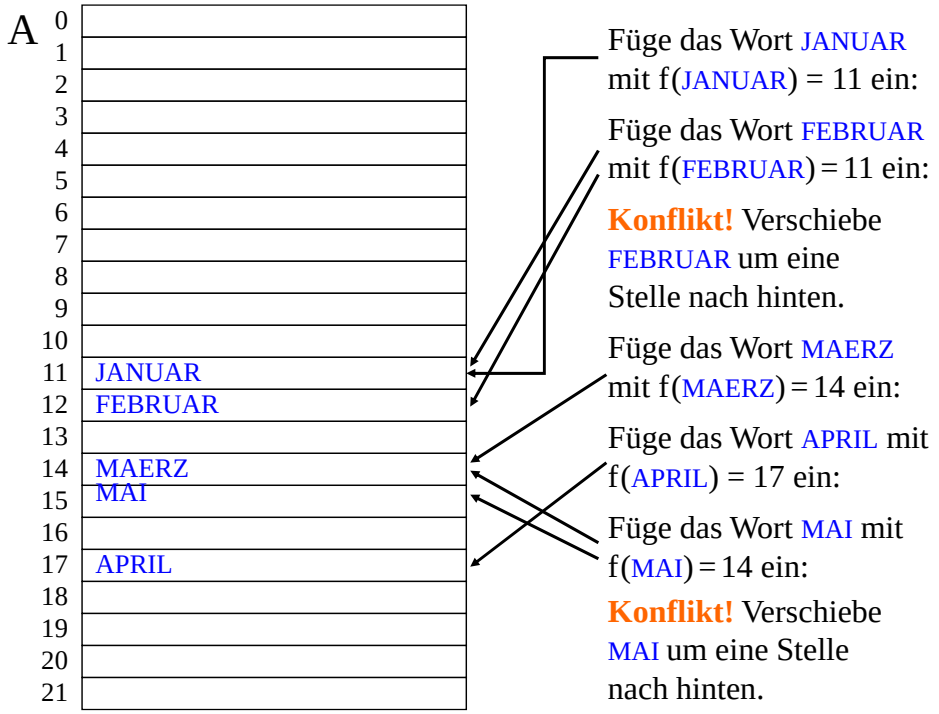
A: array (0..p-1) of String(20);

Es wird sicher auch folgender Fall auftreten:

Wir möchten einen Schlüssel s mit Hashwert f(s) = k in die array-Komponente A(k) eintragen; dort befindet sich jedoch bereits ein Schlüssel (es tritt ein "Konflikt" ein). Wir werden verschiedene Konfliktstrategien betrachten. Die einfachste ist sicherlich: Speichere den Schlüssel s an der Stelle A(k+1); ist diese ebenfalls besetzt, so versuche es mit A(k+2) usw., wobei man von der Stelle A(p-1) nach A(0) übergeht (das Feld wird also als zyklisch aufgefasst).

Diese Vorgehensweise wird nun auf den nächsten Folien demonstriert; hierbei tritt bereits beim zweiten Schlüssel ein Konflikt auf.

Ein weiteres Beispiel finden Sie in Abschnitt 9.6.



Füge das Wort JANUAR mit f(JANUAR) = 11 ein:

Füge das Wort FEBRUAR mit f(FEBRUAR) = 11 ein:

Konflikt! Verschiebe FEBRUAR um eine Stelle nach hinten.

Füge das Wort MAERZ mit f(MAERZ) = 14 ein:

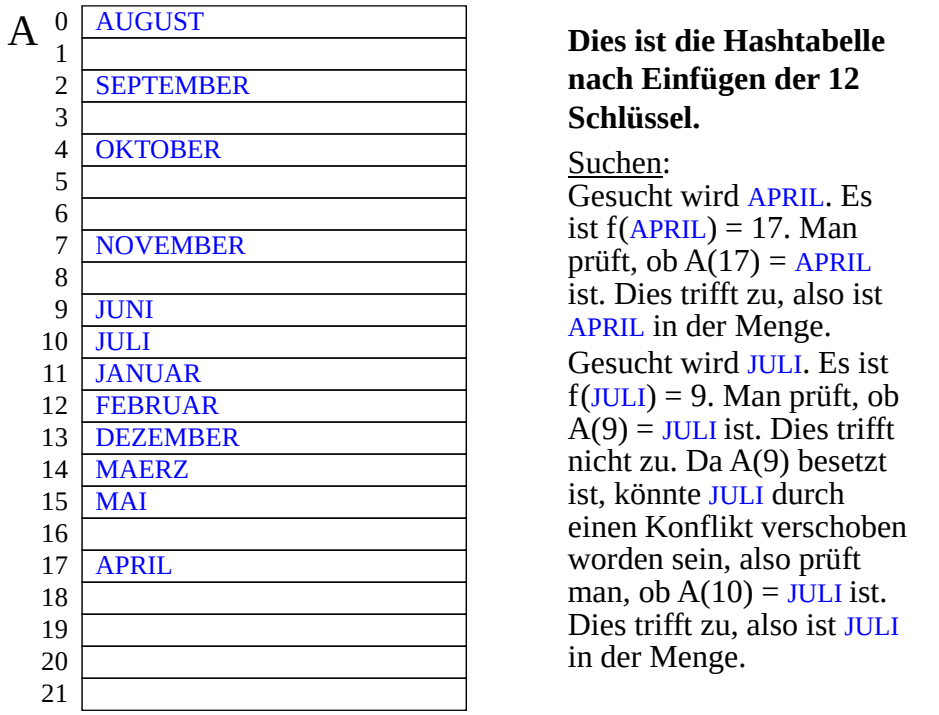
Füge das Wort APRIL mit f(APRIL) = 17 ein:

Füge das Wort MAI mit f(MAI) = 14 ein:

Konflikt! Verschiebe MAI um eine Stelle nach hinten.

Füge nun weiterhin die Wörter JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER ein; ihre zugehörigen f-Werte lauten: 9, 9, 0, 2, 4, 7, 9.

Es entstehen erneut **Konflikte** bei JULI und DEZEMBER.
JULI muss um eine Stelle verschoben werden.
DEZEMBER muss man sogar um 4 Stellen bis Index 13 verschieben.



Dies ist die Hashtabelle nach Einfügen der 12 Schlüssel.

Suchen:
Gesucht wird APRIL. Es ist f(APRIL) = 17. Man prüft, ob A(17) = APRIL ist. Dies trifft zu, also ist APRIL in der Menge.
Gesucht wird JULI. Es ist f(JULI) = 9. Man prüft, ob A(9) = JULI ist. Dies trifft nicht zu. Da A(9) besetzt ist, könnte JULI durch einen Konflikt verschoben worden sein, also prüft man, ob A(10) = JULI ist. Dies trifft zu, also ist JULI in der Menge.

Gesucht wird DEZEMBER. Es ist f(DEZEMBER) = 9. Man prüft, ob A(9) = DEZEMBER ist, dann für A(10) usw., bis man entweder auf DEZEMBER oder auf einen leeren Eintrag stößt.

Gesucht wird JURA. Es ist f(JURA) = 9. Man prüft, ob A(9) = JURA ist. Dies trifft nicht zu, also geht man zu A(10) usw., bis man auf den leeren Eintrag A(16) trifft, dort bricht die Suche ab und JURA ist nicht in der Menge der Monatsnamen.

Wie löscht man? (Später!)

Wie viele Vergleiche braucht man, um einen Schlüssel zu finden, der in der Menge liegt?

JANUAR:	1 Vergleich
FEBRUAR:	2 Vergleiche
MAERZ:	1 Vergleich
APRIL:	1 Vergleich
MAI:	2 Vergleiche
JUNI:	1 Vergleich
JULI:	2 Vergleiche
AUGUST:	1 Vergleich
SEPTEMBER:	1 Vergleich
OKTOBER:	1 Vergleich
NOVEMBER:	1 Vergleich
DEZEMBER:	5 Vergleiche
insgesamt	19 Vergleiche

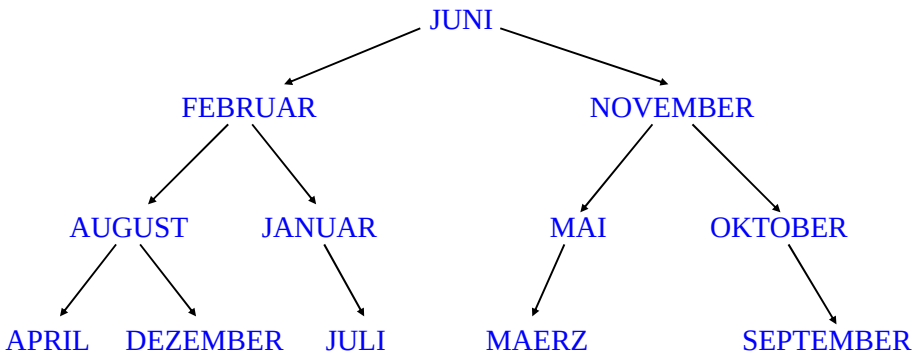
Im Mittel braucht man also $19/12 \approx$ **1,6 Vergleiche**, falls der Schlüssel in der Menge ist (erfolgreiche Suche).

Wie viele Vergleiche braucht man für einen Schlüssel, der **nicht** in der Menge liegt? Gehe jede Komponente des Feldes hierzu durch (f soll gleichverteilt sein, siehe Anfang von Kapitel 9.1):

0:	2 Vergleiche	11:	6 Vergleiche
1:	1 Vergleich	12:	5 Vergleiche
2:	2 Vergleiche	13:	4 Vergleiche
3:	1 Vergleich	14:	3 Vergleiche
4:	2 Vergleiche	15:	2 Vergleiche
5:	1 Vergleich	16:	1 Vergleich
6:	1 Vergleich	17:	2 Vergleiche
7:	2 Vergleiche	18:	1 Vergleich
8:	1 Vergleich	19:	1 Vergleich
9:	8 Vergleiche	20:	1 Vergleich
10:	7 Vergleiche	21:	1 Vergleich
	Gesamt:		55 Vergleiche

Im Mittel braucht man also 55/22 = **2,5 Vergleiche**, falls der gesuchte Name **nicht** in der Menge ist (erfolglose Suche).

Vergleich mit einem ausgeglichenen Suchbaum (siehe 8.4.11):



Mittlere Anzahl der Vergleiche für Elemente, die in der Menge sind: $(1+2+2+3+3+3+3+4+4+4+4+4)/12 \approx$ **3,1 Vergleiche**.
Falls das Element **nicht** in der Menge ist (13 null-Zeiger): im Mittel $49/13 \approx$ **3,8 Vergleiche**.
(Wir haben hier vernachlässigt, dass man an jedem Knoten eigentlich zwei Vergleiche durchführt: auf "Gleichheit" und auf "Größer".)

Zeitbedarf: Die Hashtabelle ist deutlich günstiger. Man muss aber die Berechnung der Abbildung f hinzu zählen, die allerdings nur einmal je Wort durchgeführt wird.

Speicherplatz: Wir benötigen 22 statt 12 Bereiche für die Elemente der Schlüsselmenge S. Dafür sparen wir die Zeiger des Suchbaums. Es hängt also vom Platzbedarf ab, den jedes Element aus S braucht, um abschätzen zu können, ob sich diese Tabellendarstellung mit der Abbildung f lohnt.

Sie ahnen es schon: Hashtabellen sind in der Regel deutlich günstiger als Suchbäume. Allerdings darf man die Tabelle nicht zu sehr füllen, da dann die Suchzeiten, insbesondere für Wörter, die *nicht* in der Tabelle sind, stark anwachsen. Erfahrungswert: Mindestens **20%** der Stellen sollten ständig frei bleiben (vgl. Abschnitt 9.4).

9.2 Hashfunktionen

9.2.1 Aufgabe:
n Elemente einer (sehr großen) Menge S sollen in einem Feld array (0..p-1) of ... gesucht und dort in irgendeiner Reihenfolge eingefügt und gelöscht werden können.

Es sei |S| > p (sonst ist die Aufgabe ohne Konflikte durch irgendeine injektive Zuordnung zu lösen).

Benutze hierfür eine Funktion f: S $\rightarrow \{0, 1, ..., p\text{-}1\}$, genannt *Hashfunktion*, die
- surjektiv ist (d.h., jede Zahl von 0 bis p-1 tritt als Bild auf),
- die Elemente jeder Teilmenge B $\subseteq$ S möglichst gleichmäßig über die Zahlen von 0 bis p-1 verteilt und
- schnell berechnet werden kann.

In der Praxis verwendet man meist das

9.2.2 Divisionsverfahren

1. Fasse den gegebenen Schlüssel s als Zahl auf (jedes Datum ist binär dargestellt und kann daher prinzipiell als Zahl aufgefasst werden).
2. Bilde den Rest der Division durch die Zahl p
   f(s) = s mod p.

Diese Restbildung hatten wir in 9.1 benutzt.

(p wählt man meist als Primzahl, um Abhängigkeiten bei der Modulo-Bildung zu verringern; bei quadratischer Kollisionsstrategie, siehe unten, maximiert man hierdurch die Zykellänge.)

Ein anderes Verfahren verwendet eine "möglichst irrationale" Zahl z zwischen 0 und 1.

9.2.3 Multiplikationsverfahren:
(p ist die Größe der Hashtabelle)

1. Fasse wiederum den gegebenen Schlüssel s als Zahl auf.
2. Multipliziere diese Zahl mit z und betrachte nur den Nachkommateil, d.h., die Ziffernfolge nach dem Dezimalpunkt:
   $g(s) = s \cdot z - \lfloor s \cdot z \rfloor$
   (dies ist eine reelle Zahl größer gleich 0 und kleiner 1).
3. Erweitere dies auf das Intervall [0..p) und bilde den ganzzahligen Anteil:
   $f(s) = \lfloor p \cdot g(s) \rfloor$.

Beispiel: Seien p = 22 und z = 0,624551.

Dann gilt für s = 34: $s \cdot z = 21{,}234734$, $g(s) = 0{,}234734$.

f(s) = ganzzahliger Anteil von $p \cdot g(s) = \lfloor 5{,}164148 \rfloor = 5$.

Hinweise:
1. Die Zahl $|c_2| = 0.618\,033\,988\,749\,894\,848\,204$ ... gilt als gut geeignete Zahl z (zu c_2 siehe Fibonaccizahlen, 8.4.7).
2. Es kann auch $g(s) = \lceil s \cdot z \rceil - s \cdot z$ benutzt werden.

9.2.4: Wenn Zeichenfolgen als Schlüsselmenge $S = \Sigma^*$ vorliegen, wählt man gerne ein Teilfolgenverfahren (hier bzgl. der Division vorgestellt; analog: bzgl. der Multiplikation):

1. Codiere die Buchstaben: $\varphi: \Sigma \to \{0, 1, ..., t\text{-}1\}$, z.B. ASCII.
2. Wähle fest eine Folge von Indizes $i_1 i_2 ... i_q \in \{1, 2, 3, ..., r\}^*$.
3. Wähle als Hashfunktion $f: \Sigma^* \to \{0, 1, ..., p\text{-}1\}$

$$f(\alpha_1\alpha_2 ...\alpha_r) = \Big(\sum_{j=1}^{q} \varphi(\alpha_{i_j}) \Big) \text{ mod } p$$

oder verwende allgemein eine gewichtete Summe mit irgendwelchen speziell gewählten Zahlen $x_1, x_2, ..., x_q$:

$$f(\alpha_1\alpha_2 ...\alpha_r) = \Big(\sum_{j=1}^{q} x_j \cdot \varphi(\alpha_{i_j}) \Big) \text{ mod } p.$$

Definition 9.2.5: Eine Hashfunktion $f: S \to \{0, 1, ..., p\text{-}1\}$ heißt perfekt *bzgl. einer Menge* $B \subseteq S$ (mit $|B| \le p$), wenn f auf der Menge B injektiv ist, wenn also für alle Elemente $b_i \neq b_j$ aus B stets $f(b_i) \neq f(b_j)$ gilt.

Wenn man einen unveränderlichen Datenbestand hat (etwa gewisse Wörter in einem Lexikon oder die reservierten Wörter einer Programmiersprache), so lohnt es sich, eine Hashtabelle mit einer perfekten Hashfunktion einzusetzen, da dann die Entscheidung, ob ein Element b in der Tabelle vorkommt, durch eine einzige Berechnung f(b) und einen weiteren Vergleich getroffen werden kann.

Durch Ausprobieren lassen sich solche perfekten Funktionen finden. Siehe Spalte 2.+3. und p = 17 in der großen Tabelle in 9.1. Suchen Sie z. B. eine perfekte Hashfunktion für B = {JANUAR, ... , DEZEMBER} und p = 15 (⇒ nächste Folie).

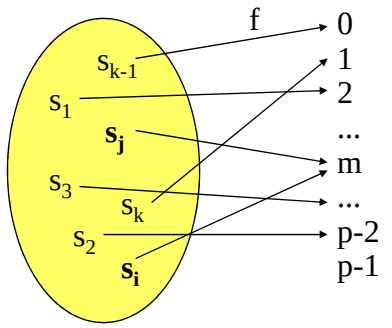
Eine Lösung für B = {JANUAR, ... , DEZEMBER} und p = 15 lautet:

$$f(\alpha_1\alpha_2 ...\alpha_r) = \big(7\,\varphi(\alpha_1) + 5\,\varphi(\alpha_2) + 2\,\varphi(\alpha_3)\big) \text{ mod } 15.$$

Dieses f ist tatsächlich injektiv:

JANUAR	13
FEBRUAR	11
MAERZ	1
APRIL	3
MAI	9
JUNI	8
JULI	4
AUGUST	6
SEPTEMBER	10
OKTOBER	5
NOVEMBER	7
DEZEMBER	0

9.2.6: Wahrscheinlichkeit für einen Konflikt.
In eine Tabelle mit p Komponenten sollen k Elemente aus S mit Hilfe einer Hashfunktion $f: S \to \{0, 1, ..., p\text{-}1\}$ eingetragen werden. Die Hashfunktion f soll die Elemente aus S möglichst gleichmäßig auf die p Zahlen abbilden. Wie groß ist die Wahrscheinlichkeit, dass sich unter k verschiedenen Elementen mindestens zwei Elemente s_i und s_j befinden mit $f(s_i) = f(s_j)$?



Berechne die Wahrscheinlichkeit, dass unter k verschiedenen Elementen mindestens zwei Elemente s_i und s_j sind mit $f(s_i) = f(s_j)$. Dies ist 1 minus der Wahrscheinlichkeit, dass alle k Elemente auf verschiedene Werte abgebildet werden:

$$1 - (1 - \frac{1}{p}) \cdot (1 - \frac{2}{p}) \cdot \ldots \cdot (1 - \frac{k-1}{p})$$

$$\approx 1 - \prod_{i=1}^{k-1} e^{-\frac{i}{p}} = 1 - e^{-\frac{k(k-1)}{2p}}$$

Beachte hierbei: $(1-i/p) \approx e^{-\frac{i}{p}} = 1 - \frac{i}{p} + \frac{\left(\frac{i}{p}\right)^2}{2!} - \frac{\left(\frac{i}{p}\right)^3}{3!} + - \ldots$.

Wann beträgt die Wahrscheinlichkeit 50%, dass mindestens zwei Schlüssel auf den gleichen Wert abgebildet werden?

$1 - e^{-\frac{k(k-1)}{2p}} = ½$ liegt vor bei

$\ln(½) = -\frac{k(k-1)}{2p}$, d.h., es gilt ungefähr

$k \approx \sqrt{p \cdot 2 \cdot \ln(2)}$ mit $2 \cdot \ln(2) \approx 1.386$ und $\sqrt{2 \cdot \ln(2)} \approx 1.1777$.

Satz 9.2.7

Trägt man gleichverteilte Schlüssel nacheinander in eine Hashtabelle der Größe p ein, so muss man nach $1.1777 \cdot \sqrt{p}$ Schritten mit 50% Wahrscheinlichkeit damit rechnen, dass erste "Kollisionen" eingetreten sind, dass also zwei verschiedene Schlüssel an der gleichen Stelle einzutragen sind.

Hinweis:
Diese Aussage gilt natürlich nicht nur für Hashtabellen.

Bekannt ist das "Geburtstagsparadoxon": Wie groß ist die Wahrscheinlichkeit, dass sich unter k Personen mindestens zwei Personen mit gleichem Geburtstag befinden?
Da hier $p = 365$ (oder 366) und $2p = 730$ ist, lautet die Antwort:

$$\approx 1 - e^{-\frac{k(k-1)}{730}}$$

Soll die Wahrscheinlichkeit 50% sein, so ist k so zu wählen, dass

$$½ = e^{-\frac{k(k-1)}{730}}$$

gilt, d.h., $k \approx 1.1777 \cdot 19.105 \approx 22.5$. Wenn also nur 23 Personen zusammen sind, so ist die Wahrscheinlichkeit, dass zwei von ihnen am gleichen Tag Geburtstag haben, bereits über 50%.

9.2.8: Wie häufig werden Kollisionen auftreten?

Hierzu wählen wir zufällig k Schlüssel $s_1, s_2, \ldots, s_k$ und betrachten die Folge der Hashwerte $(f(s_1), f(s_2), \ldots, f(s_k))$. Wie groß ist die Wahrscheinlichkeit, dass ein Wert j in dieser Folge nicht auftritt $(0 \le j \le p-1)$?

Wegen der angenommenen Eigenschaften der Funktion h sollte jeder Index j mit gleicher Wahrscheinlichkeit 1/p auftreten. Folglich tritt j bei k Berechnungen mit der Wahrscheinlichkeit $(1-1/p)^k$ *nicht* auf, d.h., mit der Wahrscheinlichkeit $1 - (1-1/p)^k$ kommt j mindestens einmal vor. Es gilt wegen $(1-x/k)^k \to e^{-x}$:

$$(1-1/p)^k = (1 - (k/p)/k)^k \approx e^{-\frac{k}{p}} = e^{-\lambda}$$

mit $\lambda = k/p$ = "Auslastungsgrad" der Tabelle.

Jeder Index wird also ungefähr mit der Wahrscheinlichkeit $1-e^{-\lambda}$ vorkommen. Ist $k=p/2$, d.h., $\lambda=½$, so werden ungefähr $p\cdot(1-e^{-\lambda}) = p\cdot(1-e^{-½}) \approx p\cdot 0.3905$ verschiedene Indizes auftreten; die verbleibenden $p/2 - p\cdot 0.3905 = 0.1095\cdot p$ Berechnungen führen also zu sofortigen Konflikten; also in rund 22% der Fälle tritt eine Kollision (gleiche Werte $f(s_i) = f(s_j)$) bereits bei der Berechnung der Hashfunktion auf, wenn man die Hashtabelle nur bis zur Hälfte füllt.

Ist $\lambda=1$, so wird jeder Index mit der Wahrscheinlichkeit $(1-e^{-1}) \approx 0.63212...$ besucht. Fügt man also p Elemente nacheinander in eine Hashtabelle der Größe p ein, so werden nur rund 63,2% verschiedene Tabellen-Indizes beim Ausrechnen der Hashfunktion berechnet. Es treten also $0.3689\cdot p$ Kollisionen durch gleiche Werte der Hashfunktion auf.

Die Zahl der tatsächlichen Kollisionen ist natürlich größer, da im Falle einer Kollision durch das Verschieben von Elementen weitere Kollisionen entstehen können.

9.3 Techniken beim Hashing

Man unterscheidet beim Hashing zwei Techniken: das geschlossene Hashing mit angefügten Überlaufbereichen und das offene Hashing, das alle Schlüssel im vorgegebenen Array unterbingt.

9.3.1 Geschlossenes Hashing

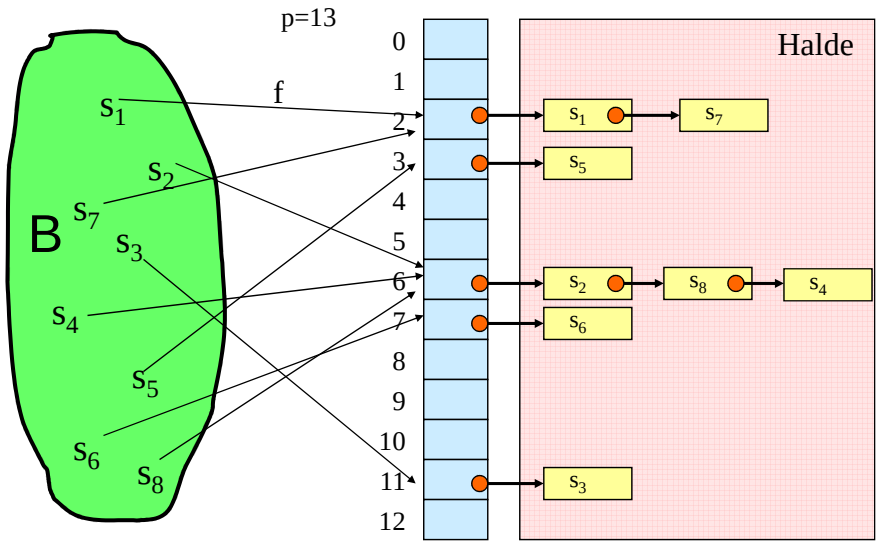
Beim *geschlossenen* Hashing lässt man keine Korrekturen des Hashwertes zu, sondern die Hash-Funktion führt zu einem Index, über den man zu einer Datenstruktur gelangt, an der sich der gesuchte Schlüssel befindet, befinden müsste oder an der er einzufügen ist. In der Regel werden alle gespeicherten Schlüssel, die den gleichen Hash-Wert besitzen, in eine lineare Liste oder in einen Suchbaum eingefügt.

Als Beispiel betrachten wir auf der nächsten Folie den Fall, dass alle Schlüssel, die den gleichen f-Wert haben, in einer linearen Liste gespeichert werden. In der Hashtabelle A steht an der Stelle A(i) der Zeiger auf die Liste der Schlüssel s mit $f(s)=i$.

Auf diese Weise entstehen keine Kollisionen in der Hashtabelle, sondern dieses Problem wird in die Verwaltung der p Listen verlagert.
(Hinweis: In der Praxis ist dies die "Haldenverwaltung".)

Die Hashtabelle ist somit nur eine Zugriffsstruktur für viele gleichartige Speicherstrukturen, die relativ klein gehalten werden können. Man spricht auch von externen Hashtabellen oder von Hashing mit externer Kollision.

Skizze: "Externe" Hashtabelle



Überlaufprobleme müssen mit der Haldenverwaltung gelöst werden!

Zeitaufwand beim geschlossenen Hashing: Das Suchen ("FIND") erfordert einen Zugriff auf das Feld A(f(s)) und anschließend muss die jeweilige Datenstruktur durchsucht werden.

Auch Einfügen ("INSERT") und Löschen ("DELETE") laufen bis auf den ersten Zugriff wie bei den zugrunde liegenden Datenstrukturen ab.

Vorteil des geschlossenen Hashings: Man muss keine feste obere Grenze für die Menge B der Schlüssel vorgeben. Vor allem wenn viel gelöscht wird, kann man die schnellen Algorithmen der jeweiligen Datenstrukturen einsetzen. Insbesondere bei großen Datenbeständen lohnen sich Suchbäume.

Nachteil: Das Verfahren hängt von der Verwaltung der Listen (Zugriffszeiten, Garbage Collection) ab. (Offenes Hashing ist oft effizienter, siehe im Folgenden).

9.3.2 Offenes Hashing

Beim *offenen* Hashing wird der tatsächliche Index, unter dem der Schlüssel s später steht, erst mit dem Eintragen, also ggf. nach dem Durchlaufen einer Kollisionsstrategie, bestimmt. In diesem Fall befinden sich alle Schlüssel im Speicherbereich, den der Indexraum vorgibt (also im array (0..p-1)), und es gibt keine Überlaufbereiche.

Die Bezeichnungen „offen“ und „geschlossen“ sind anfangs verwirrend, da sie aus der Indexzuordnung abgeleitet wurden und sich nicht auf die Struktur des Speicherbereichs beziehen. Der Index bleibt also nach der Berechnung des Hashwertes "noch offen" und wird erst festgelegt, wenn eine freie Stelle gefunden wurde.

Wir zeigen nun: Das Suchen geht in der Regel schnell, sofern mindestens 20% der Komponenten des array frei gehalten werden.
Das Einfügen ist nicht schwierig.
Problem ist das Löschen.

9.3.3 Datentypen für das offene Hashing festlegen

(Die Booleschen Werte brauchen wir erst später; in der Praxis speichert man den Hashwert f(s) des Schlüssels s zusätzlich ab; in der Regel lässt man die Komponente "Inhalt" weg, wenn der Schlüssel auf den eigentlichen Speicherort verweist.)

```
type EintragTyp is record
        belegt: Boolean;  geloescht: Boolean;
        kollision: Boolean;  behandelt: Boolean;
        Schluessel: Schluesseltyp;
        Inhalt: InhaltTyp;
end record;

type Hashtabelle is array(0..p-1) of EintragTyp;
```

FIND: Der Suchalgorithmus lautet dann: ...

INSERT: Der Einfügealgorithmus lautet dann: ...

9.3.4 Einfüge-Algorithmus (für offenes Hashing)

```
A: Hashtabelle; i, j: Integer;              -- p sei global bekannt
k: Integer :=0;                             -- k gibt die Anzahl der Schlüssel in A an
for i in 0..p-1 loop  A(i).besetzt:=false; A(i).kollision:=false;
      A(i).geloescht:=false; A(i).behandelt:=false; end loop;
while "es gibt noch einen einzutragenden Schlüssel s" loop
  if k < p then
      k := k+1; j := f(s);        -- j ist der Index in A für s
      if A(j).geloescht or not A(j).besetzt then  A(j).besetzt := true;
            A(j).Schluessel := s; A(j).Inhalt := ...;
      else A(j).kollision := true; "Starte eine Kollisionsstrategie";
      end if;
  else "Tabelle A ist voll, starte eine Erweiterungsstrategie für A";
  end if;
end loop;
```

9.3.5 Das Suchen erfolgt ähnlich:

Um einen Eintrag mit dem Schlüssel s zu finden, berechne f(s) und prüfe, ob in A(f(s)) ein Eintrag mit dem Schlüssel s steht. Falls ja, ist die Suche erfolgreich beendet, falls nein, prüfe A(f(s)).kollision. Ist dieser Wert false, dann ist die Suche erfolglos beendet, anderenfalls berechne mit der verwendeten Kollisionsstrategie (s.u.) eine neue Stelle j und prüfe erneut, ob der Schlüssel s gleich A(j).Schlüssel ist; falls ja, ist die Suche erfolgreich beendet, falls nein, prüfe den Wert von A(j).kollision. Ist dieser Wert false, dann ist die Suche erfolglos beendet, anderenfalls berechne mit der verwendeten Kollisionsstrategie eine neue Stelle j usw.

Wir wenden uns nun den Kollisionen und ihrer Behandlung zu.

Definition 9.3.6: Sei f: S $\rightarrow \{0, 1, ..., p\text{-}1\}$ eine Hashfunktion.

Gilt f(s) = f(s') für zwei einzufügende Schlüssel s und s', so spricht man von einer Primärkollision. In diesem Fall muss der zweite Schlüssel s' an einer Stelle A(i) gespeichert werden, für die $i \neq f(s')$ gilt.

Ist f(s') = i und befindet sich an der Stelle A(i) ein Schlüssel s'' mit $f(s'') \neq i$, so spricht man von einer Sekundärkollision, d.h., die erste Kollision beim Eintragen von s' wird durch einen Schlüssel s'' verursacht, der vom Hashwert her nicht an die Position i gehört und selbst durch Kollision hierhin gelangt ist.

Wenn man eine Strategie zur Behandlung von Kollisionen festlegt, so kann man sich gegen die Primärkollisionen kaum wehren, aber man kann versuchen, die Sekundärkollisionen klein zu halten.

Beispiel: Sei wiederum
B = {JANUAR, FEBRUAR, MAERZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER} mit
$f(\alpha_1\alpha_2 ...\alpha_r) = \big(\varphi(\alpha_1) + \varphi(\alpha_2) \big) \bmod 14$.
Drei Schlüssel werden in der Reihenfolge JANUAR, FEBRUAR, OKTOBER eingegeben. Es gilt:
f(JANUAR) = 11, f(FEBRUAR) = 11, f(OKTOBER) = 12.

Wir tragen JANUAR in der Komponente A(11) ein.
Der Schlüssel FEBRUAR führt zu einer Primärkollision. Die Strategie sei: *Gehe von der Stelle j zur nächsten Stelle j+1.*
Dann wird FEBRUAR an der Stelle A(12) gespeichert.
Der Schlüssel OKTOBER gehört in die Stelle A(12), doch hier steht ein Schlüssel, der dort nicht hingehört, sondern durch eine Kollision hierhin verschoben wurde. Folglich führt OKTOBER zu einer Sekundärkollision. (OKTOBER wird dann an der Stelle A(13) eingetragen.)

Definition 9.3.7: Kollisionsstrategien ("Sondieren")

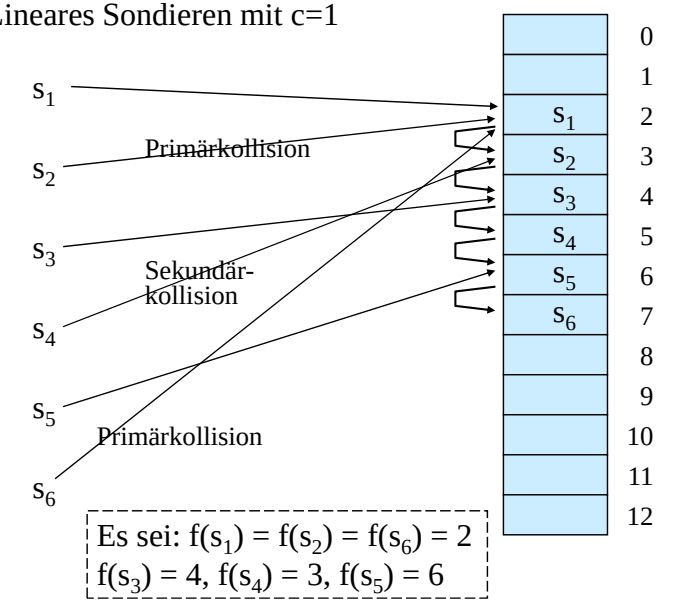
Der Schlüssel s soll stets unter dem Indes f(s) gespeichert werden. Ist diese Komponente bereits besetzt, so versuche man, den Schlüssel s unter dem Index G(s,i) einzufügen. Es sei i die Zahl der versuchten Zugriffe. c ist eine fest gewählte Konstante mit ggT(c,p) = 1; man kann stets c=1 wählen.
Übliche Funktionen G sind:

$G(s,i) = (f(s)+i \cdot c) \bmod p$	heißt "lineare Fortschaltung" oder "lineares Sondieren" oder "lineares Hashing".
$G(s,i) = (f(s)+i^2) \bmod p$	heißt "quadratische Fortschaltung" oder "quadratisches Sondieren".

Beispielskizze:

Lineares Sondieren mit c=1

p=13



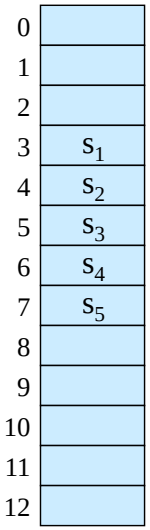
Das lineare Sondieren ist ein leicht zu implementierendes Verfahren, das sich in der Praxis bewährt hat. *Nachteil* ist die sog. "Clusterbildung", die durch Sekundärkollissionen hervorgerufen wird (siehe auch das vorige Beispiel):
Die Wahrscheinlichkeit, auf eine bereits durchlaufene Kollisionskette zu stoßen, wird im Laufe der Zeit größer als die Wahrscheinlichkeit, auf Anhieb eine freie Stelle zu finden. Dadurch müssen die Kollisionsketten immer nachvollzogen werden: Es bilden sich sog. Cluster, siehe Erläuterung unten.
Der *Vorteil* des linearen Sondierens liegt darin, dass man bei dieser Kollisionsstrategie Werte aus der Hashtabelle wieder löschen kann. (Selbst überlegen und Übungen. Das prinzipielle Vorgehen wird unten erläutert.)

Das quadratische Sondieren ist ebenfalls leicht zu implementieren. Sein *Vorteil* ist, dass die Sekundärkollisionen und damit die Clusterbildungen reduziert werden. (Bei Primärkollisionen müssen natürlich alle Versuche erneut nachvollzogen werden.)

Der *Nachteil* des quadratischen Sondierens besteht darin, dass der Aufwand, um Werte aus der Hashtabelle wieder zu löschen, unzumutbar groß ist. Man markiert daher die gelöschten Elemente als "gelöscht", belässt sie aber weiter in der Tabelle und entfernt alle gelöschten Elemente erst nach einiger Zeit gemeinsam (siehe 9.5).

Um nicht in kurze Zyklen bei der Kollisionsstragegie zu gelangen, sollte man beim quadratischen Sondieren unbedingt verlangen, dass die Größe der Hashtabelle p eine Primzahl ist. Dies wird in Satz 9.3.10 begründet.

9.3.8 Clusterbildung bei linearem Sondieren



Es möge die nebenstehende Situation mit dem Cluster A(3) bis A(7) entstanden sein.

Es soll nun ein weiterer Schlüssel s eingefügt werden. Ist f(s) einer der Werte 3, 4, 5, 6 oder 7, so wird s bei linearem Sondieren mit c=1 in A(8) gespeichert.

Die Wahrscheinlichkeit, dass im nächsten Schritt A(8) belegt wird, ist daher 6/13, während für jede andere Stelle nur die Wahrscheinlichkeit 1/13 gilt.

Cluster haben also eine hohe Wahrscheinlichkeit, sich zu vergrößern. Genau dieser Effekt wird in der Praxis beobachtet.

Die Clusterbildungen beruhen auf den Sekundärkollisionen. Diese werden beim quadratischen Sondieren reduziert.

Als Beispiel betrachten wir erneut B = {JANUAR, FEBRUAR, MAERZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER} mit p=22.

Als Abbildung verwenden wir dieses Mal die Hashfunktion

$f(\alpha_1\alpha_2 ...\alpha_r) = (2\varphi(\alpha_1) + \varphi(\alpha_2)) \bmod 17$.

A

0	FEBRUAR
1	APRIL
2	
3	
4	JANUAR
5	
6	AUGUST
7	JUNI
8	JULI
9	SEPTEMBER
10	MAERZ
11	MAI
12	
13	NOVEMBER
14	DEZEMBER
15	
16	OKTOBER

Quadratisches Sondieren: Wir fügen die Wörter ein JANUAR, FEBRUAR, MAERZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER .

Die zugehörigen f-Werte lauten: 4, 0, 10, 1, 10, 7, 7, 6, 9, 7, 9, 13.

Tragen Sie die Wörter in die zunächst leere Tabelle ein. Es ergibt sich am Ende die nebenstehende Tabelle.

9.3.9: Länge von Zyklen bei Kollisionsstrategien

Wir müssen uns nun überzeugen, dass bei den Kollisions-strategien keine zu kurzen Zyklen durchlaufen werden. Beim linearen Sondieren ist dies gewährleistet: Wenn c und p teilerfremd sind (ggT(c,p) = 1), dann durchläuft die Folge der Zahlen $(f(s)+i\cdot c) \bmod p$ (für $i = 0, 1, 2, ...$) alle Zahlen von 0 bis p-1, bevor eine Zahl erneut auftritt.

Wir wollen nun zeigen, dass beim quadratischen Sondieren keine "kurzen" Zyklen auftreten, sofern p eine Primzahl ist. (Wir nehmen hier an, dass p > 2 und somit ungerade ist.)

Wir fragen daher: Wann tritt in der Folge der Zahlen

$(f(s)+i^2) \bmod p$ für $i = 0, 1, 2, 3, ...$

erstmals eine Zahl wieder auf?

Wenn eine Zahl erneut auftritt, so muss es zwei Zahlen i und j geben mit $i \neq j$, $i \geq 0$, $j \geq 0$ und

$(f(s)+i^2) \bmod p = (f(s)+j^2) \bmod p$,

d.h. $(i^2-j^2) \bmod p = (i+j)\cdot(i-j) \bmod p = 0$.

Wenn p eine Primzahl ist, dann muss (i-j) oder (i+j) durch p teilbar sein. Wir nehmen an, dass wir höchstens p Mal das quadratische Sondieren durchführen, d.h., dass $0 \leq i \leq p-1$ und $0 \leq j \leq p-1$ gelten. Dann ist $-p < (i-j) < p$ und wegen $i \neq j$ kann daher p nicht (i-j) teilen. Also muss p die Zahl (i+j) teilen. Das geht aber nur, wenn mindestens eine der beiden Zahlen größer als die Hälfte von p ist. Also gilt:

Satz 9.3.10 (Zykluslänge beim quadratischen Sondieren) Wenn p eine Primzahl ist, dann kann beim quadratischen Sondieren frühestens nach (p+1)/2 Schritten eine Zahl erneut auftreten.

Für $i = (p+1)/2$ und $j = (p-1)/2$ ist $(i+j)\cdot(i-j) \bmod p = 0$, da $i+j=p$ und $i-j=1$ gilt. Die in Satz 9.3.10 genannte Länge eines Zyklus von $(p+1)/2$ Schritten tritt somit für *alle* Zahlen (auch für die Primzahlen) tatsächlich auf.
Hinweis: Bei einer Primzahl p kommen die "komplementären Zahlen" $(-i^2) \bmod p$ beim quadratischen Sondieren nicht vor. Modifiziert man daher das quadratische Sondieren so, dass zwischen $i^2 \bmod p$ und $(i+1)^2 \bmod p$ immer $(-i^2) \bmod p$ eingeschoben wird, so erreicht man die volle Zykluslänge p, sofern $p \bmod 4 = 3$ ist (ohne Beweis).

Übungsaufgabe:
Beweisen Sie diese Aussage und schreiben Sie eine Prozedur für diese Vorgehensweise.

Tritt beim linearen oder beim quadratischen Sondieren eine Primärkollision (= zwei verschiedene Schlüssel haben den gleichen Hashwert) auf, so wird für das Einfügen des jeweils letzten Schlüssels die gesamte Kette der Kollisionen, die die früheren Schlüssel mit gleichem Hashwert durchlaufen haben, ebenfalls durchlaufen.

Will man diesen Effekt vermeiden, so muss man eine zweite Hashfunktion g hinzunehmen, die möglichst unabhängig von f ist, d.h., für f und g sollte auf jeden Fall gelten:

$S_{m,n} = \{s \in S \mid f(s) = m \text{ und } g(s) = n\}$ enthält für alle m und n ungefähr $|S|/p^2$ Elemente.

Dies führt zu "Doppel-Hash"-Kollisionsverfahren:

Definition 9.3.11: Kollisionsstrategien (Fortsetzung)

Es seien f und g zwei unterschiedliche Hashfunktionen. Sei i die Zahl der Zugriffe (beginnend mit $i = 0$). Die Kollisionsstrategie

$D(s,i) = (f(s) + i\cdot g(s)) \bmod p$ heißt "Doppel-Hash-Verfahren".

Es seien $f_1, f_2, f_3, f_4, \ldots$ eine Folge von möglichst unterschiedlichen Hashfunktionen. Die Kollisionsstrategie

$M(s,i) = f_i(s)$ heißt "Multi-Hash-Verfahren".

Hinweis: In der Praxis hat man mit Doppel-Hash-Strategien gute Erfahrungen gemacht.

9.3.12 Löschen in Hashtabellen (DELETE)
Dieses bildet das Hauptproblem beim offenen Hashing.

Der einfachste Weg ist es, das Löschen durch Setzen eines Booleschen Wertes zu realisieren: Wenn der Eintrag mit dem Schlüssel s gelöscht werden soll, so suche man seine Position j auf und setze A(j).geloescht := true.
Beim Einfügen behandelt man dieses Feld A(j) dann wie eine freie Stelle (nicht aber beim Suchen!).

Nachteil: Wenn oft gelöscht wird, dann ist die Tabelle schnell voll und muss mit gewissem Aufwand reorganisiert werden, vgl. Abschnitt 9.5. Dennoch ist dieses Vorgehen in der Praxis gut einsetzbar.

Hat man sich jedoch für das lineare Sondieren entschieden, dann kann man das Löschen korrekt durchführen: Man sucht den Eintrag A(j) mit dem zu löschenden Element auf und geht dann die Einträge A(j+c), A(j+2c), A(j+3c) solange durch, bis man auf eine freie Komponente stößt. In dieser Kette kopiert man alle Einträge um c, 2c, 3c usw. Stellen zurück, aber niemals über eine Komponente k hinaus, für deren Schlüssel s gilt f(s)=k.

Details: selbst überlegen! Siehe auch Übungen.

9.4 Analyse der Hashverfahren

Für die Analyse der Laufzeiten beim Suchen und Einfügen benötigt man Annahmen. Diese fordern meist die Gleichverteilung der Schlüssel und die Unabhängigkeit von Ereignissen.

9.4.1 Annahmen:

1. Die Hashfunktion f ist gleichverteilt über die Schlüsselmenge, sie bevorzugt oder benachteiligt dort keine Bereiche.
2. Jeder Schlüssel ist beim Suchen und beim Einfügen gleichwahrscheinlich.
3. Erfolgt beim Einfügen eines Schlüssels eine Kollision, so werden bis zu dessen Eintrag in eine freie Stelle nur paarweise verschiedene Stellen besucht.

Wie lange dauert es unter diesen Annahmen im Mittel, einen Schlüssel in eine Hashtabelle einzufügen, in der bereits k von p Stellen belegt sind?

Setze

$\mathbf{w_i}$ = Wahrscheinlichkeit dafür, dass für dieses Einfügen genau i Vergleiche durchgeführt werden ($1 \le i \le k+1$).

Wegen der Annahmen gilt: Mit der Wahrscheinlichkeit k/p trifft man beim ersten Mal auf eine belegte Stelle, mit der Wahrscheinlichkeit (k-1)/(p-1) beim zweiten Mal, mit (k-2/(p-2) beim dritten Mal usw. So erhalten wir die Formeln:

$w_1 = 1 - k/p$

$w_2 = (k/p)\cdot(1 - (k-1)/(p-1))$, allgemein:

$w_i = (k/p)\cdot(k-1)/(p-1)\cdot(k-2)/(p-2)\cdot\ldots\cdot(k-i+2)/(p-i+2)\cdot(1 - (k-i+1)/(p-i+1))$

$$= \frac{k\cdot(k-1)\cdot(k-2)\cdot\ldots\cdot(k-i+2)}{p\cdot(p-1)\cdot(p-2)\cdot\ldots\cdot(p-i+2)} \left(1 - \frac{k-i+1}{p-i+1}\right)$$

Dann lautet die mittlere Zahl der Vergleiche $\mathbf{E_{k+1}}$ beim Einfügen eines (k+1)-ten Schlüssels in eine Hashtabelle der Größe p:

$$\mathbf{E_{k+1}} = \sum_{i=1}^{k+1} i\cdot w_i = \ldots = \frac{p+1}{p+1-k} = \frac{1}{1-\mu} \qquad \text{mit } \mu = k/(p+1)$$

Es ist $\mu \approx$ "Auslastungsgrad" $\lambda = k/p$.

(Dieses Ergebnis lässt sich nicht allzu schwer herleiten. Versuchen Sie es einmal!)

Satz 9.4.2 Unter den Annahmen 9.4.1 gilt:

Um den (k+1)-ten Schlüssel in eine Hashtabelle der Größe p einzufügen, werden im Mittel $\frac{p+1}{p+1-k} = \frac{1}{1-\mu}$ Vergleiche (hier: mit $\mu = k/(p+1)$) benötigt.

Einige Funktionswerte für $E_{k+1} = \frac{p+1}{p+1-k}$

μ	E_{k+1}	μ	E_{k+1}	μ	E_{k+1}
0,1	1,11	0,5	2,00	0,85	6,67
0,2	1,25	0,6	2,50	0,88	8,33
0,3	1,43	0,7	3,33	0,90	10,00
0,4	1,67	0,8	5,00	0,95	20,00

9.4.3: Wie lange dauert bei "idealen Hashfunktionen" die *erfolgreiche Suche* im Mittel und wie lange die *erfolglose Suche*, wenn k der p Komponenten belegt sind?

Die erfolglose Suche entspricht dem Einfügen eines neuen Schlüssels; sie benötigt also E_{k+1} Vergleiche.

Es sei S_k die Zahl der erforderlichen Vergleiche, um einen Schlüssel zu finden, der in einer Hashtabelle der Größe p mit dem Auslastungsgrad k/p steht.

Die erfolgreiche Suche kann man dann durch folgende Formel beschreiben:

$$S_k = \frac{1}{k} \sum_{i=0}^{k-1} E_{i+1} \ ,$$

denn der gesuchte Schlüssel muss in einem der Schritte 1, 2, 3, ..., k in die Tabelle eingefügt worden sein und wegen Annahme 2 können wir den Mittelwert der Zahl der Vergleiche nehmen. Durch Auswerten dieser Formel erhält man:

$$S_k \approx \frac{1}{\mu} \cdot \ln\left(\frac{1}{1-\mu}\right)$$

(Dieses Ergebnis soll evtl. in den Übungen ausgerechnet werden.)

Satz 9.4.4 Unter den Annahmen 9.4.1 gilt:

Für die erfolgreiche Suche nach einem Schlüssel in einer Hashtabelle der Größe p werden im Mittel $S_k \approx \frac{1}{\mu} \cdot \ln\left(\frac{1}{1-\mu}\right)$ Vergleiche benötigt ($\mu = k/(p+1)$).

Hinweis: Es muss natürlich die erfolgreiche Suche S_k kleiner als die erfolglose Suche E_{k+1} sein. Dies trifft auch zu.

Denn es gilt $\frac{1}{\mu} \cdot \ln\left(\frac{1}{1-\mu}\right) \leq \frac{1}{1-\mu}$ wegen

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \geq 1 + x \text{ mit } x = \ln\left(\frac{1}{1-\mu}\right) \text{ für } 1 > \mu \geq 0.$$

Experimente haben ergeben, dass Doppel-Hash-Verfahren recht gut den Wert S_k annähern. Wie wirken sich Kollisionen aus?

Es sei $Slin_k$ *die mittlere Suchzeit für die erfolgreiche Suche* bei der linearen Kollisionsstrategie, dann kann man mit einigem Aufwand beweisen (ohne nähere Erläuterung hier):

$$Slin_k \approx \frac{1 - \frac{\mu}{2}}{1 - \mu}$$

Einige Werte zu S_k und $Slin_k$ mit $\mu = k/(p+1)$:

μ	S_k	$Slin_k$
0,50	1,39	1,50
0,75	1,85	2,50
0,80	2,01	3,00
0,90	2,56	5,50
0,95	3,15	10,50
0,99	4,65	50,50

Für die Praxis, die meist mit linearem Sondieren arbeitet, folgt hieraus: Man begrenze den Auslastungsgrad möglichst auf 80%.

9.5 Rehashing

Was muss man tun, wenn der Auslastungsgrad über 80% hinausgeht oder gar den Wert 1 erreicht? Dann muss man die Hashtabelle verlängern, also p durch eine Zahl p' > p ersetzen, hierfür eine *neue Hashfunktion* festlegen und die neue Hashtabelle aus der alten Tabelle, in der die Schlüssel in den Komponenten von 0 bis p-1 standen, errechnen.

Diesen Vorgang der Umorganisation innerhalb der bestehenden Hashtabelle bezeichnen wir als "Rehashing". Dieses Verfahren wird auch verwendet, wenn man Schlüssel, statt sie zu löschen, nur als "gelöscht" markiert, wodurch im Laufe der Zeit der Auslastungsgrad zu groß und eine Umorganisation mit dem gleichen p notwendig wird (Rehashing mit p = p').

9.5.1 Beispiel für p = 7 und p' = 13

	0
	1
MAE	2
JAN	3
FEB	4
APR	5
MAI	6

p = 7

Wörter (in dieser Reihenfolge): JAN, FEB, MAE, APR, MAI.

Alte Hashfunktion (lin. Sond.):
$f(\alpha_1\alpha_2 \ldots\alpha_r) = (\varphi(\alpha_1) + 2\,\varphi(\alpha_3))$ mod 7.

0	
1	
2	
3	
4	
5	MAE
6	APR
7	
8	FEB
9	MAI
10	
11	JAN
12	

p' = 13

Alle Wörter müssen übertragen werden.

Neue Hashfunktion:
$f'(\alpha_1\alpha_2 \ldots\alpha_r) = (\varphi(\alpha_1)+\varphi(\alpha_3))$ mod 13.
Lineares Sondieren.

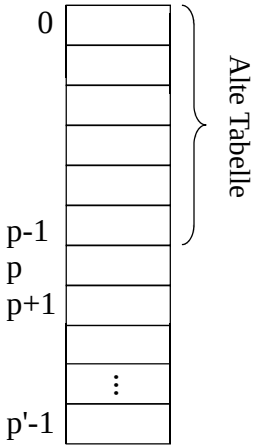
Alte Tabelle von oben nach unten durchgehen und umsortieren! Wir fangen also nun mit MAE an, dann JAN usw.

9.5.2 Rehashing:

Verlängere die Tabelle von p auf p' Komponenten. Durchlaufe diese Tabelle von 0 bis p-1 (= alte Tabelle!) und sortiere dabei die hier stehenden Schlüssel in die erweiterte Tabelle ein.

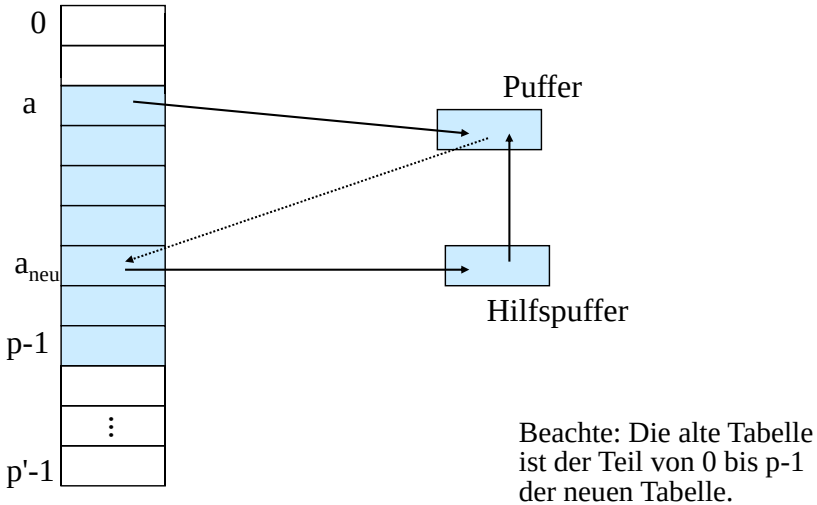
Fall 1: Verwende hierzu eine zusätzliche Tabelle mit p Komponenten. Doch genau dies wollen wir nicht, da hier zusätzlicher Speicherplatz anfällt.

Fall 2: Verwende nur die neue Tabelle und eventuell wenige Zusatzvariablen. Dies ist das eigentliche Rehashing, wie es im Folgenden vorgestellt wird.



Neue Tabelle mit p' Stellen

Durchlaufe die neue Tabelle von 0 bis p-1. Bis Index a sei bereits alles bearbeitet. Der Schlüssel an der Stelle a möge die neue Hashadresse a_{neu} haben. Vorgehensweise:



Beachte: Die alte Tabelle ist der Teil von 0 bis p-1 der neuen Tabelle.

Man kommt also mit zwei Zusatzvariablen "Puffer" und "Hilfspuffer" aus, in denen die gerade betrachteten oder die weiter zu verschiebenden Elemente zwischengespeichert werden. Nun zur Programmierung:

Erinnerung: (siehe 9.3.1)

```
type EintragTyp is record
        belegt: Boolean;  geloescht: Boolean;
        kollision: Boolean;  behandelt: Boolean;
        Schluessel: Schluesseltyp;
        Inhalt: InhaltTyp;
end record;

type Hashtabelle is array(0..p-1) of EintragTyp;

A: Hashtabelle;
```


Programm für das Rehashing

Bevor die Hashtabelle erstmals benutzt wird, wurde gesetzt:

```
for j in 0..p-1 loop  A(j).belegt:=false; A(j).kollision:=false;
        A(j).geloescht:=false; A(j).behandelt:=false; end loop;
```

Während der Verwendung der Tabelle A wurden A(j).besetzt, A(j).kollision und A(j).geloescht eventuell verändert.

Erforderliche Variablen:
Puffer, Hilfspuffer: EintragTyp;

Berechne p' als neue Größe der Hashtabelle A. Diese besitzt dann also die Grenzen von 0 bis p'-1.

Die verwendete Hashfunktion f ' möge zwei Parameter haben: den Schlüssel und die Anzahl i der Zugriffe (= die Anzahl der bisherigen Kollisionen).

Vorgehensweise: Führe (1) bis (3) von a = 0 bis a = p-1 durch.

(1) *A(a).belegt and not A(a).gelöscht and not A(a).behandelt:*
kopiere A(a) in den Puffer und setze A(a).belegt auf false.

(2) Berechne in diesem Fall mit der neuen Hashfunktion f ' den neuen Index a_{neu}, wohin das Element des Puffers hingehört.

(3) Unterscheide hierzu folgende Fälle:
not $A(a_{neu})$.belegt or $A(a_{neu})$.geloescht: Kopiere den Puffer nach $A(a_{neu})$; setze $A(a_{neu})$.belegt und $A(a_{neu})$.behandelt auf true. Erhöhe den Index a. Weiter bei (1).

not $A(a_{neu})$.behandelt and $A(a_{neu})$.belegt: Kopiere $A(a_{neu})$ in den Hilfspuffer; kopiere den Puffer nach $A(a_{neu})$; setze $A(a_{neu})$.behandelt und $A(a_{neu})$.belegt auf true. Kopiere dann den Hilfspuffer in den Puffer. Weiter bei (2).

Sonst, d.h.: *$A(a_{neu})$.behandelt:* Setze $A(a_{neu})$.kollision := true, berechne den Index a_{neu} neu, weiter bei (3).

Dieses Vorgehen erfordert nur O(p') Schritte. (Warum?)

Programmstück in Ada zum Rehashing (man beachte, dass man in Ada ein array nicht einfach verlängern kann; man muss also mit Tricks arbeiten oder in einer anderen Sprache)

```
-- a durchläuft die Adressen von 0 bis p-1,
-- i zählt die Zahl der auftretenden neuen Kollisionen,
-- aneu gibt die Adresse an, wohin der Eintrag in der neuen
--     Tabelle gehört.

for a in 0..p-1 loop
  if A(a).geloescht then "lösche den Eintrag A(a)"
  elsif  A(a).belegt and not A(a).behandelt  then
      Puffer := A(a);  Puffer.belegt := true;
      A(a).belegt := false;
      i := 0;
  -- nun f' auf Puffer anwenden und Adresse aneu berechnen,
  -- auf Kollisionen mit schon behandelten Einträgen achten.
```


(Programmstück in Ada zum Rehashing, Fortsetzung)

```
    while Puffer.belegt loop
        aneu := f'(Puffer.schluessel, i);   i := i+1;
        if  not A(aneu).belegt or A(aneu).geloescht  then
            A(aneu) := Puffer;
            A(aneu).geloescht := false; A(aneu).belegt := true;
            A(aneu).behandelt := true; A(aneu).kollision:=false;
            Puffer.belegt:= false;
        elsif  not A(aneu).behandelt  then
            Hilfspuffer := A(aneu); Hilfspuffer.belegt := true;
            A(aneu) := Puffer; A(aneu).behandelt := true;
            Puffer := Hilfspuffer;
            i := 0;
        else  A(aneu).kollision := true;   end if;
    end loop;
  end if;
end loop a;
```


Abschnitt 9.6 ist nur zum Üben; er wird in der Vorlesung übersprungen.

9.6 Noch ein Beispiel

Auf den folgenden Folien ist nochmals ein Beispiel für das lineare Sondieren angegeben, wobei die zusätzlichen Boole-schen Werte miteingetragen sind. Setzen Sie dieses Beispiel fort, indem Sie Elemente wieder explizit löschen bzw. ein Rehashing mit größerem p' durchführen. Machen Sie sich hieran die Schwierigkeiten beim quadratischen Sondieren klar, insbesondere das Problem eines auftretenden unend-lichen Zyklus.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0							
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							

p = 13

B = {JANUAR, FEBRUAR, MAERZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER},

Hashfunktion f: Nummer des ersten plus Nummer des zweiten Buchstabens, modulo 13.

JANUAR	-	-	-	-	11	?
FEBRUAR	-	-	-	-	11	?
MAERZ	-	-	-	-	1	?
APRIL	-	-	-	-	4	?
MAI	-	-	-	-	1	?
JUNI	-	-	-	-	5	?
JULI	-	-	-	-	5	?
AUGUST	-	-	-	-	9	?
SEPTEMBER	-	-	-	-	11	?
OKTOBER	-	-	-	-	0	?
NOVEMBER	-	-	-	-	3	?
DEZEMBER	-	-	-	-	9	?

- = false
+ = true

Hashfunktion f = Nummer des ersten plus Nummer des zweiten Buchstabens, modulo 13.

Nun tragen wir diese Werte nacheinander in die Hashtabelle mit p = 13 Komponenten ein, lineare Kollisionstrategie, c = 1. Das Feld für "Inhalt" interessiert nicht und wird stets "?" gesetzt.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0							
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11	JANUAR	+	-	-	-	11	?
12							

FEBRUAR
erzeugt eine Kollision an der Stelle 11 und wird dann an der Stelle 12 eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0							
1	MAERZ	+	-	-	-	1	?
2							
3							
4	APRIL	+	-	-	-	4	?
5							
6							
7							
8							
9							
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	-	-	11	?

MAI
erzeugt eine Kollision mit der Stelle 1 und wird dann in Stelle 2 eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0							
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	-	-	1	?
3							
4	APRIL	+	-	-	-	4	?
5	JUNI	+	-	-	-	5	?
6							
7							
8							
9							
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	-	-	11	?

JULI
erzeugt eine Kollision an der Stelle 5 und wird dann in die Stelle 6 eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0							
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	-	-	1	?
3							
4	APRIL	+	-	-	-	4	?
5	JUNI	+	-	+	-	5	?
6	JULI	+	-	-	-	5	?
7							
8							
9	AUGUST	+	-	-	-	9	?
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	-	-	11	?

SEPTEMBER
erzeugt eine Kollision an den Stellen 11 und 12 und wird dann in der Stelle 0 eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0	SEPTEMBER	+	-	-	-	11	?
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	-	-	1	?
3							
4	APRIL	+	-	-	-	4	?
5	JUNI	+	-	+	-	5	?
6	JULI	+	-	-	-	5	?
7							
8							
9	AUGUST	+	-	-	-	9	?
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	+	-	11	?

OKTOBER
erzeugt eine Kollision an den Stellen 0, 1 und 2 und wird dann in die Stelle 3 eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0	SEPTEMBER	+	-	+	-	11	?
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	+	-	1	?
3	OKTOBER	+	-	-	-	0	?
4	APRIL	+	-	-	-	4	?
5	JUNI	+	-	+	-	5	?
6	JULI	+	-	-	-	5	?
7							
8							
9	AUGUST	+	-	-	-	9	?
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	+	-	11	?

NOVEMBER
erzeugt eine Kollision an den Stellen 3, 4, 5 und 6 wird dann in die Stelle 7 eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0	SEPTEMBER	+	-	+	-	11	?
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	+	-	1	?
3	OKTOBER	+	-	+	-	0	?
4	APRIL	+	-	+	-	4	?
5	JUNI	+	-	+	-	5	?
6	JULI	+	-	+	-	5	?
7	NOVEMBER	+	-	-	-	3	?
8							
9	AUGUST	+	-	-	-	9	?
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	+	-	11	?

DEZEMBER
erzeugt eine Kollision an der Stelle 9 wird dann in die Stelle 10 eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0	SEPTEMBER	+	-	+	-	11	?
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	+	-	1	?
3	OKTOBER	+	-	+	-	0	?
4	APRIL	+	-	+	-	4	?
5	JUNI	+	-	+	-	5	?
6	JULI	+	-	+	-	5	?
7	NOVEMBER	+	-	-	-	3	?
8							
9	AUGUST	+	-	+	-	9	?
10	DEZEMBER	+	-	-	-	9	?
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	+	-	11	?

Ergebnistabelle

Überlegen Sie sich:

Wie sähe eine "optimale" Reihenfolge der Eintragungen aus, so dass in der Ergebnistabelle minimal viele Kollisions-Bits auf true gesetzt sind?

Welche Tabelle erhält man bei einer quadratischen Kollisionsstrategie?

Berechnen Sie die mittlere Zugriffszeit bei den verschiedenen Tabellen. Trifft es hier zu, dass die quadratische Strategie besser als die lineare ist?

Hinweis: Schön wäre es, wenn auch die folgenden Operationen leicht ausführbar wären.

- Gib die Elemente von B_1 geordnet aus. SORT
- Vereinige B_1 und B_2. UNION
- Bilde den Durchschnitt von B_1 und B_2. INTERSECTION
- Entscheide, ob B_1 leer ist. EMPTINESS
- Entscheide, ob $B_1 = B_2$ ist. EQUALITY
- Entscheide, ob $B_1 \subseteq B_2$ ist. SUBSET

Überlegen Sie sich, welche dieser Operationen mit Hilfe des Hashings mit welchem Aufwand (Zeit und Platz) durchgeführt werden können. Welche Zusatzinformationen sollte der Datentyp "Hashtabelle über Schlüsseldatentyp" besitzen und wie würde er in Ada spezifiziert und implementiert?

10. Sortieren

Ziele des 10. Kapitels:

Um Dinge schneller wiederfinden zu können oder um den Überblick zu bewahren, legt man Dokumente in geordneter Form ab. Hierzu muss der Datenbestand sortiert werden.

In diesem Kapitel lernen Sie die wichtigsten Sortierverfahren kennen und welche Eigenschaften, insbesondere welche Zeit- und Platz-Komplexität sie besitzen. Da man bei sequenziellem Vorgehen mindestens n·log(n) Vergleiche benötigt, werden auch deutlich schnellere parallel arbeitende Sortierverfahren vorgestellt, die allerdings recht viele Bausteine erfordern.

Zugleich wiederholen wir aus Kapitel 7 den Nachweis der Korrektheit für einige dieser Verfahren (vor allem in den Übungen).

10.1 Überblick

Suchen und Sortieren machen einen Großteil aller Verwaltungstätigkeiten im Rechner aus, wo Daten ständig abgelegt und schnell wiedergefunden werden müssen. Das Sortieren nutzt die Anordnung der Schlüsselmenge direkt aus und ermöglicht ein schnelles Auffinden: O(log(n)) durch binäre Suche oder O(log(log(n))) durch Interpolationssuche bei großen Datenbeständen, siehe Abschnitt 6.5.1 und Abschnitt 8.1.

Wir klären als erstes den Begriff "Sortieren" und leiten dann eine *untere* Schranke für die Zahl der Vergleiche her, die bei einem Sortierverfahren, das ausschließlich auf Vergleichen basiert, erforderlich ist. Sodann diskutieren wird die gängigen Sortierverfahren und deren Komplexität.

Definition 10.1.1: Sortierte Folgen

Gegeben sei eine endliche oder unendliche Menge mit totaler Ordnung $A = \{a_1, ..., a_s\}$ oder $A = \{a_1, a_2, a_3, a_4, ...\}$ mit $a_1 < a_2 < a_3 < a_4 < ...$.

(1) Eine Folge $v = v_1v_2...v_n$ mit $v_i \in A$ (d.h., $v \in A^*$) heißt (aufsteigend) geordnet genau dann, wenn gilt $v_1 \le v_2 \le ... \le v_n$.
(Speziell ist jede leere oder einelementige Folge geordnet.)

(2) Eine Folge $v = v_1v_2...v_n$ mit $v_i \in A$ (d.h., $v \in A^*$) heißt invers geordnet oder absteigend geordnet $\Leftrightarrow$ $v_n \le v_{n-1} \le ... \le v_1$.
(Speziell ist jede leere oder einelementige Folge invers geordnet.)

Die Sortieraufgabe lautet: Ordne eine Folge von n Elementen so um, dass sie sortiert ist. Wir präzisieren dies nun.

Definition 10.1.2: Wiederholung *Permutationen* (siehe 6.5.6)

Es sei n eine natürliche Zahl.
Eine bijektive Abbildung $\pi: \{1, 2, ..., n\} \to \{1, 2, ..., n\}$ heißt Permutation der Ordnung n.

Eine Permutation ist also "nur" eine Anordnung der ersten n Zahlen oder allgemein eine Anordnung von n verschiedenen Elementen.
Erinnerung 1: Es gibt es genau n! verschiedene Permutationen der Ordnung n.
Erinnerung 2: Da π bijektiv ist, existiert auch die inverse Abbildung $\pi^{-1}: \{1, 2, ..., n\} \to \{1, 2, ..., n\}$, definiert durch $[\pi^{-1}(i) = j \Leftrightarrow \pi(j) = i]$. π^{-1} ist ebenfalls eine Permutation.
Erinnerung 3: Permutationen π schreibt man in der Form $(\pi_1, \pi_2, \pi_3, ..., \pi_n)$, meist ohne Kommata, und meint hiermit $\pi(k) = \pi_k$ für $k = 1, ..., n$. Z.B. bezeichnet (4,1,3,2) bzw. (4 1 3 2) die Permutation $\pi(1) = 4$, $\pi(2) = 1$, $\pi(3) = 3$, $\pi(4) = 2$.

Definition 10.1.3: Die Sortieraufgabe lautet:
Finde zu einer beliebigen Folge $v = v_1 v_2 ... v_n \in A^*$ eine Permutation π der Ordnung n mit: $v_{\pi(1)} v_{\pi(2)} ... v_{\pi(n)}$ ist sortiert (oder invers sortiert, je nach Anwendung).
Ein Algorithmus, welcher $v_1 v_2 ... v_n$ in die sortierte Folge $v_{\pi(1)} v_{\pi(2)} ... v_{\pi(n)}$ überführt, heißt Sortierverfahren.

Beispiel: Seien $A = \{1, 2, 5, 8\}$, 5 8 2 5 1 8 = $v_1 v_2 ... v_n \in A^*$ eine Folge mit $n = 6$ und $v_1 = 5, v_2 = 8, v_3 = 2, v_4 = 5, v_5 = 1$ und $v_6 = 8$.

Die geordnete Folge $v_{\pi(1)} v_{\pi(2)} ... v_{\pi(n)}$ lautet 1 2 5 5 8 8; man kann also π folgendermaßen festlegen:
$\pi(1) = 5$, $\pi(2) = 3$, $\pi(3) = 1$, $\pi(4) = 4$, $\pi(5) = 2$, $\pi(6) = 6$;
denn dann gilt
$v_{\pi(1)} v_{\pi(2)} v_{\pi(3)} v_{\pi(4)} v_{\pi(5)} v_{\pi(6)} = v_5 v_3 v_1 v_4 v_2 v_6 = 1\ 2\ 5\ 5\ 8\ 8$.
$\pi^{-1}(1) = 3$, $\pi^{-1}(2) = 5$, $\pi^{-1}(3) = 2$, $\pi^{-1}(4) = 4$, $\pi^{-1}(5) = 1$, $\pi^{-1}(6) = 6$ gibt an, wohin das i-te Element der Folge verschoben wird.

Meist hängt ein Sortieralgorithmus ab von Fragen wie:

- Wie sind die Daten gegeben, zu welchen Mengen gehören sie?
- Wo liegen die Daten? (Hauptspeicher, Platten, Bänder, ...?)
- Zulässige Operationen? (Vergleichen, vertauschen ...?)
- Gibt es Elemente mit gleichem Schlüssel? ($\Rightarrow$ Stabilität.)
- Nur Zugriffsstruktur oder Gesamtdaten sortieren?
- Darf man zusätzlichen Speicher verwenden oder nicht?
- Sequenzielles, paralleles, verteiltes Sortieren?
- Effizienz im Mittel oder auch im schlechtesten Fall?
- Erkennen oder Beachten von Vorsortierungen?

10.1.4: Jedes Element v_i der zu sortierenden Folge $v = v_1 v_2 ... v_n$ ist in der Praxis meist ein umfangreiches Dokument. Die Folge wird in der Regel nur nach *einem* Ordnungskriterium sortiert.

Fall 1: Dieses Kriterium wird durch eine Funktion $g: A \rightarrow M$ beschrieben, wobei M eine geordnete Menge ist (A braucht in diesem Fall gar nicht sortierbar zu sein). $g(v_i) = k_i$ heißt dann der Schlüssel von v_i, der in der Regel im record v_i enthalten ist. (Wir können also stets Fall 2 annehmen.)

Fall 2: Das Ordnungskriterium bezieht sich auf eine oder mehrere Komponenten des Dokuments. Den Vektor dieser Komponenten, nach denen zu sortieren ist, bezeichnen wir als Schlüssel. Man verlangt oft, dass dieser ein Element eindeutig beschreibt, doch lassen wir hier auch verschiedene Elemente mit gleichem Schlüssel zu.

In der Regel sortiert man nur die Schlüssel einer Folge.

Beispiel: Folgende Folge aus Name und Alter
(Beier, 23), (Zahn, 40), (Fuhr, 30), (Horn, 41), (Beier, 30), (Horn, 17)
wird zur Folge
(Beier, 23), (Beier, 30), (Fuhr, 30), (Horn, 41), (Horn, 17), (Zahn, 40),
sofern nach dem Namen sortiert wird. Selbstverständlich hat man bei *gleichen Schlüsseln* eine Wahlfreiheit. Statt dieser Reihenfolge hätte man auch die Folge
(Beier, 30), (Beier, 23), (Fuhr, 30), (Horn, 17), (Horn, 41), (Zahn, 40)
als korrektes Ergebnis erhalten können. Die erste Folge hat den Vorteil, dass dort die relative Anordnung von Elementen mit gleichem Schlüssel erhalten blieb (ein solches Verfahren nennt man "stabil").
Dagegen hätte man die Folge
(Horn, 17), (Beier, 23), (Beier, 30), (Fuhr, 30), (Zahn, 40), (Horn, 41),
erhalten können, falls nach dem Alter sortiert wird.

■

Wird also eine Folge v nach mehreren Komponenten (Schlüsseln) nacheinander geordnet, so kann man die Folge zunächst nach dem am wenigsten relevanten Kriterium (im Beispiel: nach dem Alter) und dann schrittweise nach dem nächstwichtigeren Kriterium sortieren. Hierfür muss ein Sortierverfahren "stabil" sein.

Definition 10.1.5:

Ein Sortierverfahren *Sort* heißt stabil, wenn *Sort* die Reihenfolge von Elementen mit gleichem Schlüssel nicht verändert, d.h.: Wenn $Sort(v_1 v_2 ... v_n) = v_{\pi(1)} v_{\pi(2)} ... v_{\pi(n)}$ ist, so gilt für alle $i < j$ mit $v_i = v_j$ stets $\pi^{-1}(i) < \pi^{-1}(j)$ [d.h., stand in der ursprünglichen Folge v_i vor v_j, so gilt dies auch für die sortierte Folge, vgl. 10.1.3].

Sort heißt invers stabil, wenn die Reihenfolge der Elemente mit gleichem Schlüssel von *Sort* gespiegelt wird.

Beispiel: Ein stabiles Sortierverfahren wird aus der Folge
(Beier, 23), (Zahn, 40), (Fuhr, 30), (Horn, 41), (Beier, 30), (Horn, 17)
beim Sortieren nach dem Alter die Folge
(Horn, 17), (Beier, 23), (Fuhr, 30), (Beier, 30), (Zahn, 40), (Horn, 41)
liefern und das anschließende Sortieren nach dem Namen ergibt:
(Beier, 23), (Beier, 30), (Fuhr, 30), (Horn, 17), (Horn, 41), (Zahn, 40).
Wir erhalten also die Reihenfolge, die wir erwarten würden: Die Liste ist nach dem Namen sortiert und gleiche Namen sind aufsteigend nach dem Alter angeordnet.
Prüfen Sie Definition 10.1.5 hieran nochmals genau nach. ■

Wir haben bereits die Sortierverfahren Bubble Sort (4.4.4 und 7.3.3), Baumsortieren (3.7.7 und 8.2.22) und Quicksort (7.3.4) kennen gelernt.
Frage an Sie: Welche dieser drei Verfahren sind stabil?

Begriffsbeschreibung:

Ein weiteres Kriterium für ein Sortierverfahren lautet umgangssprachlich: Ein Sortierverfahren *Sort* heißt ordnungsverträglich
⇔ Je geordneter die zu sortierende Folge bereits ist, umso schneller arbeitet *Sort*.

Wie kann man "geordnet sein" präzisieren? Dies geschieht hier durch die "Inversionszahl".

Definition 10.1.6:

Für eine Permutation $\pi: \{1, 2, ..., n\} \to \{1, 2, ..., n\}$ heißt
$$\Im(\pi) = \left| \{ (i,j) \mid i < j \text{ und } \pi(i) > \pi(j) \} \right|$$
die Inversionszahl (oder der Fehlstand) von π.

Analog: Für eine Folge $v = v_1 v_2 ... v_n \in A^*$ (A sei eine geordnete Menge) heißt $\Im(v) = \left| \{ (i,j) \mid i < j \text{ und } v_i > v_j \} \right|$
die Inversionszahl (oder der Fehlstand) von v.

Wegen $\Im(v_1 v_2 ... v_n) + \Im(v_n v_{n-1} ... v_1) = \left| \{ (i,j) \mid i < j \} \right|$ gilt der Hilfssatz 10.1.7: $\Im(v_1 v_2 ... v_n) + \Im(v_n v_{n-1} ... v_1) = ½ \cdot n \cdot (n-1)$.

Wegen $\Im(1\ 2\ 3\ ...\ n) = 0$ folgt $\Im(n\ n\text{-}1\ ...\ 2\ 1) = ½ \cdot n \cdot (n-1)$.

Zeigen Sie: $\Im(\pi) = \Im(\pi^{-1})$ für alle Permutationen π.

10.1.8: *Präzisierung der Ordnungsverträglichkeit:*

Ein Sortierverfahren *Sort* soll ordnungsverträglich heißen, wenn es folgendes Kriterium erfüllt:
Wenn *Sort* für zwei Folgen $v = v_1 v_2 ... v_n$ und $w = w_1 w_2 ... w_n$ die Permutationen π_1 bzw. π_2 realisiert und wenn die $\Im(\pi_1) \le \Im(\pi_2)$ gilt, dann benötigt *Sort* zum Sortieren von v nicht mehr Zeit als zum Sortieren von w.

Wir betrachten die Operation "*benachbartes Vertauschen*". Diese Operation überführt eine Folge

$$v_1 v_2 \dots v_{i-1} v_i v_{i+1} v_{i+2} \dots v_n$$

in die Folge

$$v_1 v_2 \dots v_{i-1} v_{i+1} v_i v_{i+2} \dots v_n \quad \text{(für ein } 0<i<n\text{)}.$$

Hierfür gilt:

$$\left| \Im(v_1 v_2 \dots v_{i-1} v_i v_{i+1} v_{i+2} \dots v_n) - \Im(v_1 v_2 \dots v_{i-1} v_{i+1} v_i v_{i+2} \dots v_n) \right| = 1.$$

Wegen 10.1.7 haben wir somit gezeigt:

Hilfssatz 10.1.9:
Ein Sortierverfahren, das ausschließlich mit der Operation "benachbartes Vertauschen" arbeitet, benötigt im schlechtesten Fall mindestens $\frac{1}{2} \cdot n \cdot (n-1)$ Schritte.

Wir betrachten die Operation "*beliebiges Vertauschen*". Diese überführt eine Folge

(für $1 \le i \le j \le n$) $\quad v_1 v_2 \dots v_{i-1} v_i v_{i+1} \dots v_{j-1} v_j v_{j+1} \dots v_n$

in die Folge $\quad v_1 v_2 \dots v_{i-1} v_j v_{i+1} \dots v_{j-1} v_i v_{j+1} \dots v_n$.

Hierfür gilt:

$$0 \le \left| \Im(v_1 \dots v_i \dots v_j \dots v_n) - \Im(v_1 \dots v_j \dots v_i \dots v_n) \right| \le 2(j-i)-1 \le 2n-3,$$

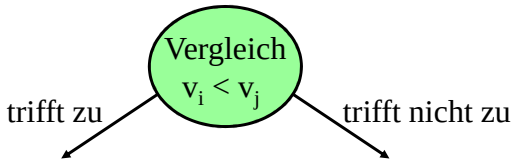
wobei der größte Wert 2n-3 nur erreicht wird, wenn das kleinste Element am Ende und das größte am Anfang stand und genau diese beiden vertauscht wurden. Es folgt:

Folgerung 10.1.10: Ein Sortierverfahren, das ausschließlich mit der Operation "Vertauschen" arbeitet, benötigt im schlechtesten Fall mindestens $n \cdot (n-1)/(4n-6) > \frac{1}{4} \cdot n$ Schritte.
(Man kann diese Schranke noch verschärfen. Versuchen Sie, eine größere untere Schranke zu finden.)

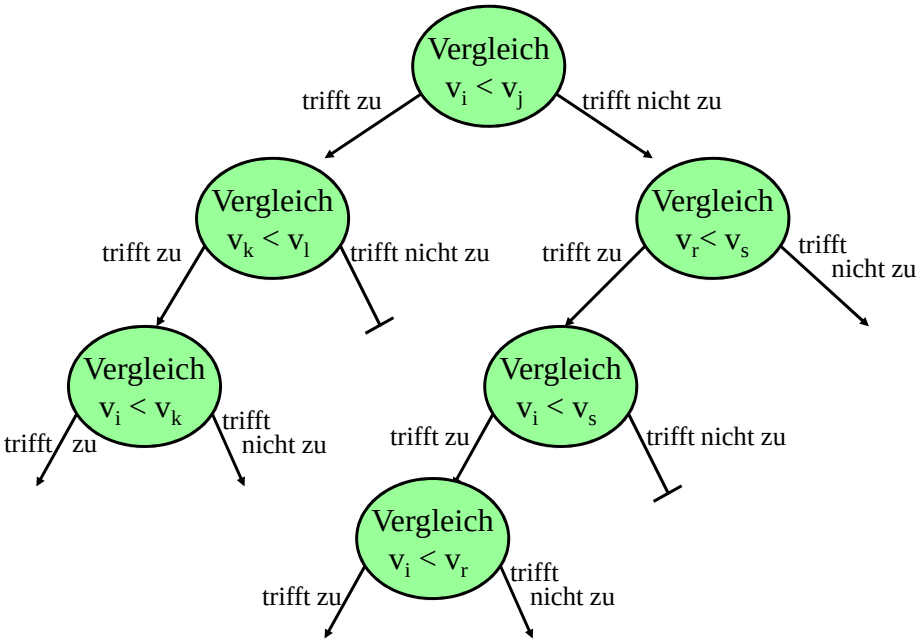
In der Regel weiß man nicht, ob man zwei Elemente einer Folge vertauschen soll. Diese Entscheidung wird durch einen Vergleich getroffen: Falls $v_i < v_j$ und $i > j$ ist, dann vertauscht man diese beiden Elemente in der Folge.

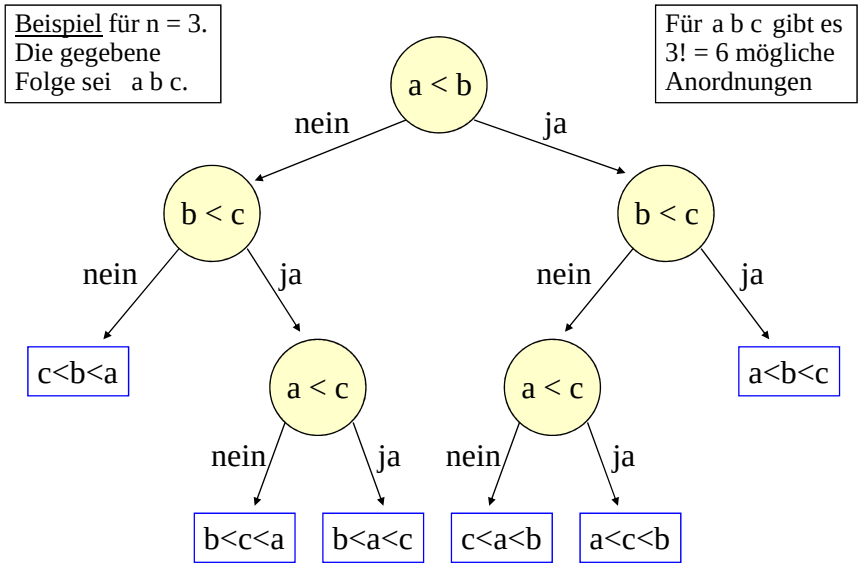
Wird das Vertauschen durch vorher gehende Vergleiche gesteuert, so dauert das Sortierverfahren mindestens so lange wie die Anzahl der hierfür erforderlichen Vergleiche.

Eine Folge von Vergleichen bildet einen binären Baum, der aus den Knoten und Kanten



aufgebaut ist.





Die sortierte Folge ist jeweils blau umrandet als Blatt angegeben.

Hat man alle Informationen zum Sortieren gewonnen, dann stellen genau die null-Zeiger in diesem Baum (= alle blau umrandeten Aussagen in obigem Beispiel) die Permutationen dar, die zur Sortierung gehören. Da es n! Permutationen der Ordnung n gibt, muss der "Baum der Vergleiche" daher mindestens n! null-Zeiger besitzen.

Durch Vergleiche entsteht immer ein binärer Baum. Ein binärer Baum mit m-1 Knoten besitzt genau m null-Zeiger. Also muss der Baum der Vergleiche

mindestens n!-1

Knoten besitzen.
Die Länge des längsten Weges, also die Tiefe dieses Baums gibt die Zahl der erforderlichen Vergleiche im worst case an. Die Tiefe eines binären Baums mit k Knoten ist aber mindestens log(k+1), siehe Folgerung 8.2.16. Daher gilt:

Satz 10.1.11: Ein Sortierverfahren, das ausschließlich auf Vergleichen zweier Elemente beruht, benötigt im worst case

mindestens $\log(n!) = n \cdot \log(n) - 1.4404 \cdot n + O(\log(n))$

Schritte.

Beweis: Nach der *Stirlingschen Formel* gibt es zu jedem n ein d mit $0 < d < 1$, so dass gilt (mit $e = 2.718281828459...$):

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\ \ e^{\frac{1}{12}d} \qquad \text{(siehe 6.5.6.3)}$$

Durch Logarithmieren erhält man hieraus:
$\log(n!) \approx n \cdot \log(n) - n \cdot \log(e) \approx n \cdot \log(n) - 1.4404 \cdot n$ und von $2\pi n$ und $d/12$ verbleibt noch ein additiver Anteil $O(\log(n))$.

10.1.12 Überblick über die üblichen Sortiermethoden:

Austauschen:
- a. Benachbartes Austauschen (Bubble sort, Shaker sort)
- b. Shellsort
- c. Quicksort

Einfügen:
- a. Einfügen in Listen (Insertion sort)
- b. Baumsortieren (mit binären Bäumen, AVL-Bäumen, ...)
- c. Fachverteilen (und radix exchance)

Aussuchen / Auswählen:
- a. Minimumsuche (minimum sort)
- b. Heapsort (normal, bottom up, ultimativ)

Mischen:
- Merge sort und diverse Varianten

Streuen und Sammeln (bucket sort)

Sortiermethoden	Zeitaufwand	zusätzl. Platz
Austauschen		
a. Benachb. Austauschen	$\frac{1}{2}\cdot n^2$	konstant
b. Shellsort	$O(n^{\frac{3}{2}})$	$\le \log(n)$
c. Quicksort (im Mittel !)	$1.3863\cdot n\cdot\log(n)$	$2\log(n)$
Einfügen		
a. Einfügen in Listen	$\frac{1}{2}\cdot n^2$	konstant
b. Baumsortieren (AVL-Bäume)	$\le 1.4404\cdot n\cdot\log(n)$	O(n)
c. Fachverteilen (im Mittel)	$O(n\cdot\log(n))$	O(n)
Aussuchen / Auswählen		
a. Minimumsuche	$\frac{1}{2}\cdot n^2$	konstant
b. Heapsort	$\approx 2n\cdot\log(n)$	konstant
c. Bottom-up Heapsort	$\le 1.5\cdot n\cdot\log(n)+O(n)$	konstant
Mischen		
Verschmelzen (Merge sort)	$O(n\cdot\log(n))$	n
Streuen und Sammeln	O(n)	O(n)

10.2 Sortieren durch Austauschen

Die beiden wichtigsten Vertreter Bubble Sort und Quicksort wurden bereits in 4.4.4, 7.3.4 und 7.3.4 vorgestellt; sie sind auf den nächsten Folien nochmals als Programm ausformuliert.

Aufwandsabschätzung für Bubble Sort:
Platzbedarf: konstant (3 zusätzliche Variablen)
Zeitbedarf: worst case: $\frac{1}{2}\cdot n\cdot(n-1)$ Vergleiche,
best case: n, falls das Feld bereits sortiert ist,
im Mittel sind $\frac{1}{4}\cdot n\cdot(n-1)$ Vergleiche zu erwarten **(wirklich?)**.

Man kann das Feld abwechselnd aufwärts und abwärts durchlaufen (dies nennt man Shaker Sort). Diese Variante bringt aber in der Praxis keine Verbesserung des Laufzeitverhaltens.

10.2.1 Bubble Sort für einen "Vektor" A mit dem Indextyp 1..n:

```
procedure BubbleSort (A: in out Vektor) is
Weiter: Boolean := True;  H: Integer;
begin
    while Weiter loop
        Weiter := False;
        for i in 1..n-1 loop
            if A(i) > A(i+1) then Weiter := True;
                H := A(i); A(i) := A(i+1); A(i+1) := H;
            end if;
        end loop;

    end loop;
end BubbleSort;
```

Beachte: Durch die Variable "Weiter" wird Bubble Sort ordnungsverträglich.

Beachte:
Wegen A(i) > A(i+1)
ist Bubble Sort stabil.

Den bereits sortierten Teil muss man natürlich nicht immer wieder sortieren:

Bubble Sort um den Faktor 2 beschleunigt:

```
procedure BubbleSort (A: in out Vektor) is
Weiter: Boolean := True;  H: Integer;  K: Integer := n-1;
begin
    while Weiter loop
        Weiter := False;
        for i in 1..K loop
            if A(i) > A(i+1) then Weiter := True;
                H := A(i); A(i) := A(i+1); A(i+1) := H;
            end if;
        end loop;
        K := K - 1;
    end loop;
end BubbleSort;
```

Beachte: Durch die Variable "Weiter" wird Bubble Sort ordnungsverträglich.

Beachte:
Wegen A(i) > A(i+1)
ist Bubble Sort stabil.

10.2.2 Quicksort: Aus 7.3.4 übernehmen wir mit den Datentypen
type Index is 1..n; ... A: array (Index) of Integer; ... das Programm:

```
procedure Quicksort(L, R: Index) is     -- A ist global, A(L..R) wird sortiert
i, j: Index; p, h: Integer;             -- p wird das Pivot-Element
begin
  if L < R then
     i := L; j := R; p := A((L+R)/2);       -- man kann p auch anders wählen
     while i <= j loop                      -- die Indizes i und j laufen aufeinander zu
          while A(i) < p loop i := i+1; end loop;
          while A(j) > p loop j := j-1; end loop;
          if i <= j then h:=A(i); A(i):=A(j); A(j):=h;
                         i := i+1; j := j-1;  end if;
     end loop;                              -- auch bei Gleichheit A(i)=p oder A(j)=p vertauschen!
     if (j-L) < (R-i) then Quicksort(L, j); Quicksort(i, R);
     else Quicksort(i, R); Quicksort(L, j); end if;
  end if;
end Quicksort;
... Quicksort(1,n); ...                 -- Aufruf des Sortierverfahrens
```


Beachte:

Ein Verfahren heißt deterministisch, wenn es in jeder Situation höchstens eine Folgesituation gibt (vgl. 2.2.2)

Es heißt determiniert, wenn man es eine Abbildung realisiert (d.h., zu jeder Eingabe erzeugt das Verfahren höchstens eine Ausgabe).

Quicksort ist ein nichtdeterministisches Verfahren, da man p willkürlich wählen kann.

Quicksort ist aber ein determiniertes Verfahren, da stets die sortierte Folge erzeugt wird.

Der Nichtdeterminismus wirkt sich nur auf die Laufzeit, nicht aber auf die Korrektheit aus.

10.2.3 *Platzbedarf von Quicksort*

Quicksort ist ein "in-place"-Verfahren, d.h., der eigentliche Algorithmus arbeitet nur auf dem Speicherplatz des Feldes A. Dennoch braucht Quicksort zusätzlichen Platz. In 7.3.4 wurde erläutert, warum wegen der Rekursion sogar mit einem hohen Platzbedarf zu rechnen ist. Man muss das Quicksort-Programm daher abändern, dass man den Stack für die Rekursion selbst verwaltet und nutzlos gewordene Information sofort entfernt und nicht im Stack stehen lässt. Es ist klar, dass man auf diese Weise die Tiefe des Stacks auf log(n) Paare beschränken kann.

Aufgabe: Versuchen Sie, eine Lösung für dieses Problem anzugeben, d.h., Sie sollen die Prozedur Quicksort so abwandeln, dass die rekursive Tiefe des Aufruf-Stacks stets durch 2·log(n) beschränkt bleibt.
(Hinweis: Eine Lösung finden Sie im Buch Ottmann/Widmayer.)

10.2.4 *Zeitbedarf von Quicksort* (siehe 8.2.22):

Zeitbedarf: worst case: $\approx \frac{1}{2} \cdot n^2$ Vergleiche, wenn das Pivot-Element stets das kleinste oder das größte Element des Teilfelds ist.
best case: $n \cdot \log(n)$, wenn das Pivot-Element stets das mittelste der Elemente des Teilfelds ("Median") ist.
average case: Es ist mit $1{,}3863 \cdot n \cdot \log(n) - 1{,}8456 \cdot n$ Vergleichen im Mittel zu rechnen. Begründung:

Man kann das Aufteilen des Feldes in zwei Teilfelder mit dem Aufbau eines binären Suchbaums vergleichen, wobei das Pivot-Element in die Wurzel kommt und aus den beiden Teilfeldern rekursiv der linke und rechte Unterbaum aufgebaut werden. Quicksort verhält sich daher im Mittel genau wie das Baumsortieren mit binären Suchbäumen (Satz 8.2.21). Die formalen Berechnungen liefern das gleiche Ergebnis, siehe Lehrbücher.

Zeitbedarf von Quicksort im Mittel (Fortsetzung):

Für den Zeitbedarf ist die "gute Wahl" des Pivot-Elements p entscheidend. Gebräuchlich ist folgende Variante: Statt das Element p irgendwie fest zu wählen (z.B.: p:=A((L+R)/2) oder p:=A(L) oder ...) nimmt man das mittlere von drei Elementen, z.B. von A(L), A(R) und A((L+R)/2).
Mit dieser "Mittelwert aus 3"-Variante erhält man einen deutlich besseren mittleren Zeitbedarf, nämlich im Mittel höchstens

$1.188 \cdot n \cdot \log(n) - 2{,}255 \cdot n$ Vergleiche.

Diese Variante wird daher gern in der Praxis verwendet.

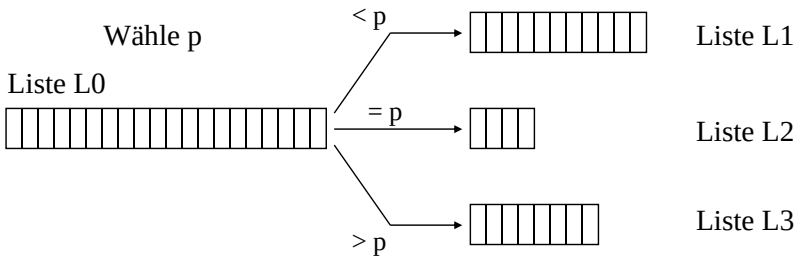
Aufgabe: Erweitern Sie unsere Prozedur um diese Variante. Machen Sie Messungen mit zufällig erzeugten Daten und verifizieren Sie hiermit den Laufzeitgewinn, der ab einem gewissen n eintritt. (Bestimmen Sie dieses n experimentell.)

Zeitbedarf von Quicksort im Mittel (Fortsetzung):

Eine andere Variante besteht darin, die Zerlegung in drei Teilfelder vorzunehmen (sog. "Dreiwege-Split"):
Alle Elemente in A(L..j) sind echt kleiner als p,
alle Elemente in A(i..R) sind echt größer als p und
alle Elemente in A(j+1..i-1) sind gleich p, wobei dieser Bereich nie leer ist. Da man ihn nicht bei der Rekursion berücksichtigen muss, verringert sich die Rekursionstiefe und damit die Laufzeit. Treten in einer Folge viele Schlüssel mehrfach auf, so lässt sich hierdurch das Sortieren beschleunigen.

Aufgabe: Erweitern Sie unsere Prozedur auch um diese Variante. Die Schwierigkeit liegt darin, alle Schlüssel, die gleich p sind, ohne Zusatzaufwand in der Mitte des zu sortierenden Teilfelds zu platzieren. Auch hier sollten Sie dann Messungen mit zufällig erzeugten Daten vornehmen und prüfen, ab welchem Prozentsatz gleicher Schlüssel sich dieses Vorgehen lohnt.

Natürlich ist Quicksort nicht auf Felder beschränkt. Man kann eine Liste L0 durchlaufen und hierbei alle Elemente, die größer, kleiner oder gleich p sind, in drei verschiedene Listen L1, L2, L3 einfügen. L1 und L3 werden rekursiv weitersortiert und anschließend in der Reihenfolge L1, L2, L3 aneinandergehängt.



Dies kostet zusätzlichen Speicherplatz, vermeidet aber die beiden Zeiger i und j. Die Abfrage auf " = p ?" lohnt sich oft nicht, d.h., man fragt auf " ≤ p ?" ab und spart die Liste L2. Suchen Sie nach weiteren Varianten dieses "divde-and-conquer"-Ansatzes.

Hinweise:

Historisch gesehen gibt es weitere Sortierverfahren, die auf dem Austauschen beruhen. Am bekanntesten ist Shellsort, bei dem die Folge in äquidistante Teilfolgen unterteilt wird und als erste Phase in diesen eine Sortierung erfolgt (z.B. ein Bubble Sort Durchlauf). Die äquidistanten Abstände werden dann für eine zweite Phase verringert und diese Verringerung wird fortgesetzt, bis der Abstand gleich 1 ist, d.h. eine Sortierung der bis dahin entstandenen schon gut vorsortierten Folge stattfindet. Dieses Vorgehen setzt in den einzelnen Phasen ordnungsverträgliche Sortierverfahren voraus.
Denken Sie sich selbst "schnelle" heuristische Verfahren aus. Der Fantasie sind hier kaum Grenzen gesetzt. Führen Sie aber auf jeden Fall Messungen durch (denn nur selten erfüllen sich die erhofften Beschleunigungen).

10.3 Sortieren durch Einfügen

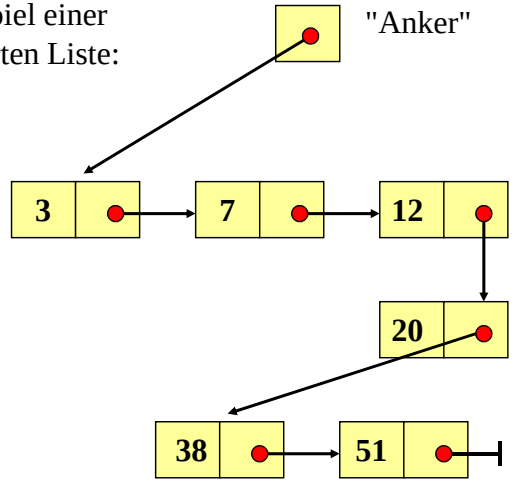
Wenn die Elemente einer Folge durch Zeiger (sortierte Listen, Suchbäume) dargestellt werden, so wird man die einzelnen Elemente der Folge nacheinander in diese Struktur einfügen und dabei die Struktur stets wiederherstellen.

Einfachster Fall: Einfügen in eine sortierte Liste.
Eine Liste heißt sortiert, wenn für alle Elemente der Liste gilt:
p.Inhalt ≤ p.next.Inhalt. Hierbei ist p ein Zeiger, der auf das jeweilige Element der Liste verweist.

Erinnerung: Datenstruktur für eine Liste (vgl. z. B. 3.5.2.0):

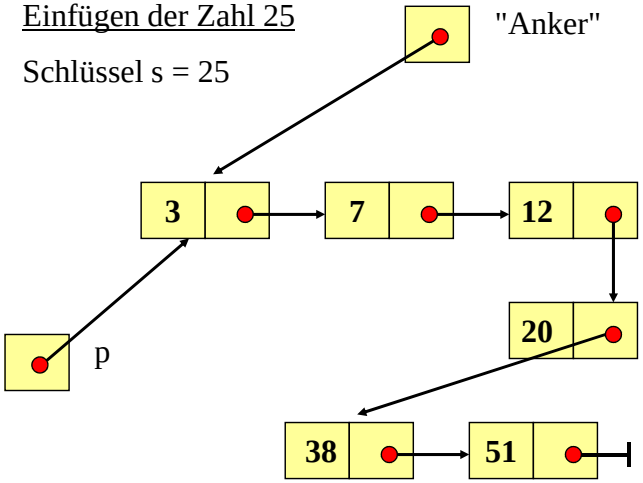
```
type Zelle;
type Ref_Zelle is access Zelle;
type Zelle is record Inhalt: Integer; Next: Ref_Zelle; end record;
```


Beispiel einer sortierten Liste:



Einfügen der Zahl 25

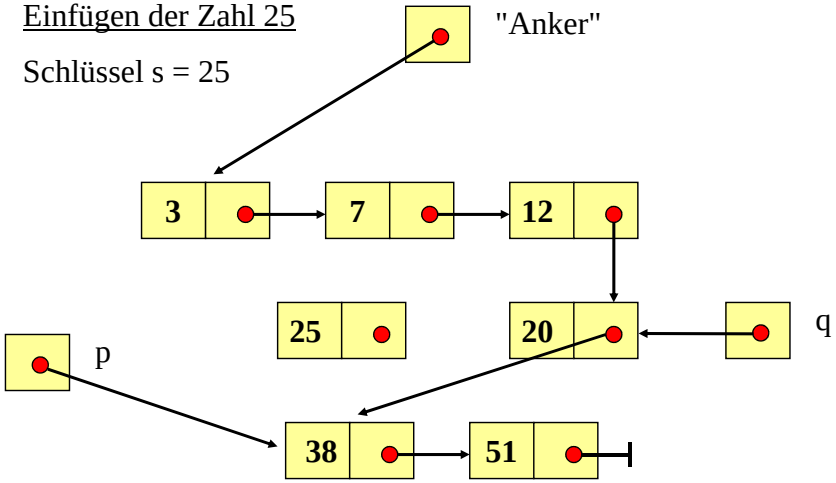
Schlüssel s = 25




```
p := Anker;
while p /= null and then s > p.Inhalt loop p:=p.Next; end loop;
```


Einfügen der Zahl 25

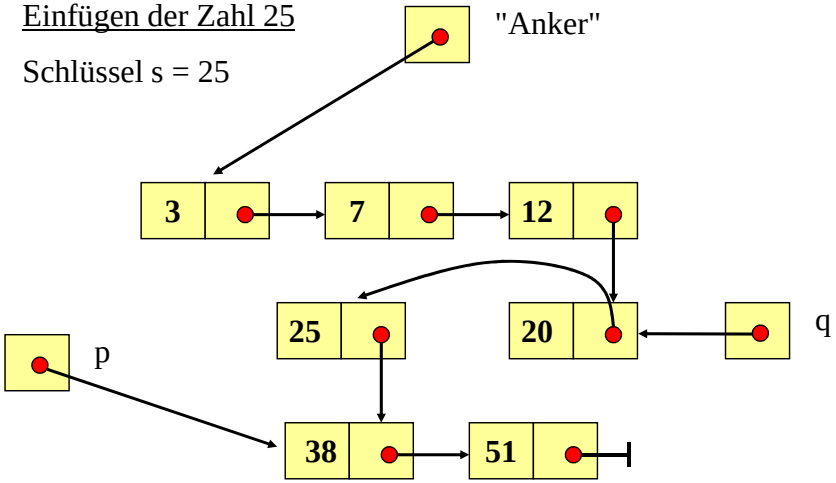
Schlüssel s = 25




```
p := Anker;
while p /= null and then s > p.Inhalt loop p:=p.Next; end loop;
```

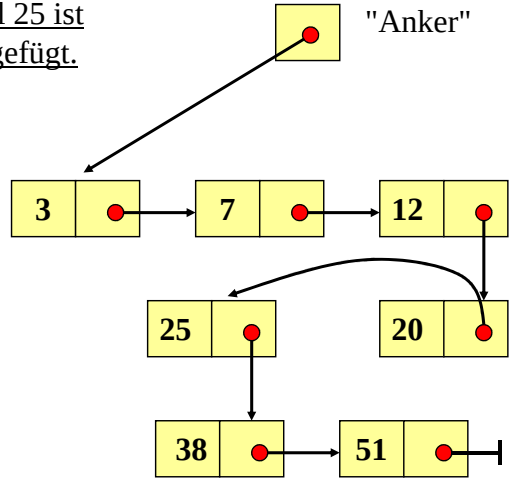
Einfügen der Zahl 25

Schlüssel s = 25



p := Anker;
while p /= null and then s > p.Inhalt loop p:=p.Next; end loop;

Die Zahl 25 ist nun eingefügt.



Prozedur zum Einfügen eines *Elementes in eine sortierte Liste:*

```
procedure Einf (Anker: in out Ref_Zelle; s: in Integer) is
p, q: Ref_Zelle;
begin  p := Anker; q := null;
       while p /= null and then s > p.Inhalt loop
           q := p;  p := p.Next;  end loop;
       if  q = null  then
           if p = null then Anker := new Zelle'(s,null);
           else Anker := new Zelle'(s, Anker); end if;
       else  q.Next := new Zelle'(s,p);  end if;
end Einf;
```


Diese Prozedur kann man vereinfachen zu (bitte selbst nachprüfen!):

```
procedure Einf (Anker: in out Ref_Zelle; s: in Integer) is
p, q: Ref_Zelle;
begin  p := Anker; q := new Zelle'(s, Anker);
           while p /= null and then s > p.Inhalt loop
               q := p;  p := p.Next;  end loop;
           q.Next := new Zelle'(s, p);
end Einf;
```

10.3.1 *Sortieren von n Elementen durch Einfügen in eine Liste:*

while not End_of_File loop Get(v); Einf(Anker, v); end loop;

Zahl der Vergleiche im Mittel und im schlechtesten Fall: $O(n^2)$.

10.3.2 *Sortieren durch Einfügen in einen Baum:* siehe 8.2.22.

Wenn man beliebige binäre Suchbäume verwendet, so können diese im schlechtesten Fall zu einer Liste entarten und daher beträgt im worst case die Zeitkomplexität $O(n^2)$.

Benutzt man anstelle beliebiger Suchbäume AVL-Bäume, so ist deren Höhe durch $1.4404 \cdot n \cdot \log(n)$ nach Satz 8.4.8 beschränkt. Daher ist die Anzahl der Vergleiche für das Sortieren mit Bäumen auch im schlechtesten Fall durch $1.4404 \cdot n \cdot \log(n) + O(n)$ nach oben begrenzt.

In der Praxis kann man bei der Verwendung von beliebigen Suchbäumen im Mittel mit $1.3863 \cdot n \cdot \log(n) + O(n)$, bei der Verwendung von AVL-Bäumen im Mittel mit $n \cdot \log(n) + O(n)$ rechnen (siehe Satz 8.2.21 und Hinweis nach Satz 8.4.8).

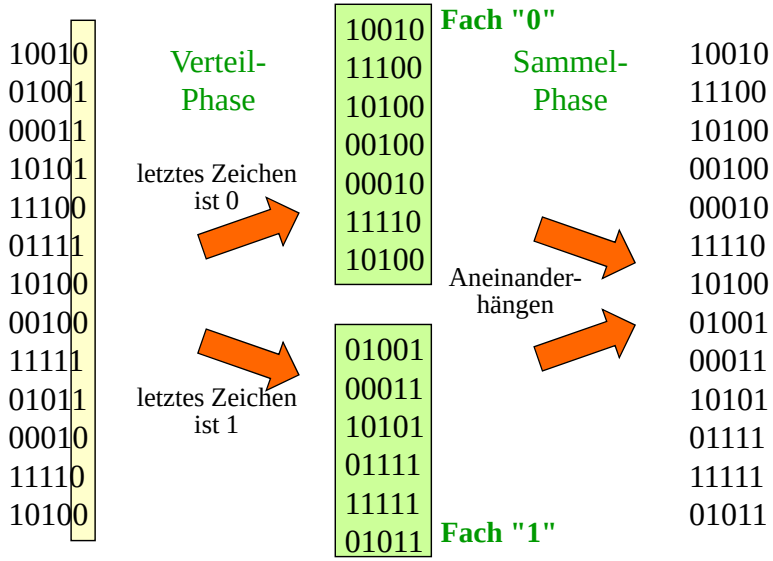
10.3.3 *Sortieren durch Fachverteilen*

Sind die Schlüssel Wörter über einem endlichen Alphabet $A = \{a_1, a_2, ..., a_s\}$, kann man die zu sortierenden Elemente zunächst bzgl. des letzten Zeichens an s Listen anfügen ("Verteilphase"). Diese Listen hängt man zu einer Liste aneineinander ("Sammelphase"). Dann fügt nun alle Elemente bzgl. des vorletzten Zeichens an s Listen an ("Verteilphase") usw. Liegt die Länge jedes Schlüssels in der Größenordnung von log(n), so ergibt sich ein $O(n \cdot \log(n))$-Sortierverfahren.

Das Anfügen an die s Listen entspricht dem Ablegen in s verschiedene Fächer, weshalb dieses Verfahren als "Fach-verteilen" bezeichnet wird.

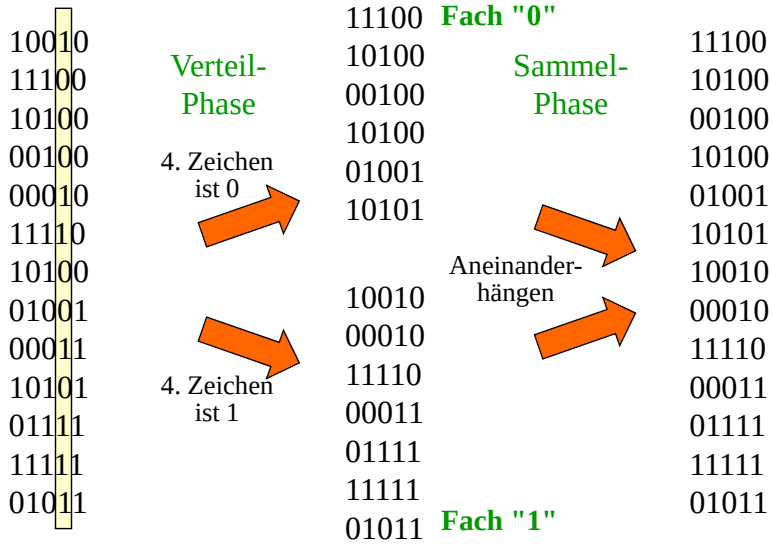
Wir erläutern das Verfahren nur an einem Beispiel mit $s = 2$. Die Programmierung ist nicht schwierig.

Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



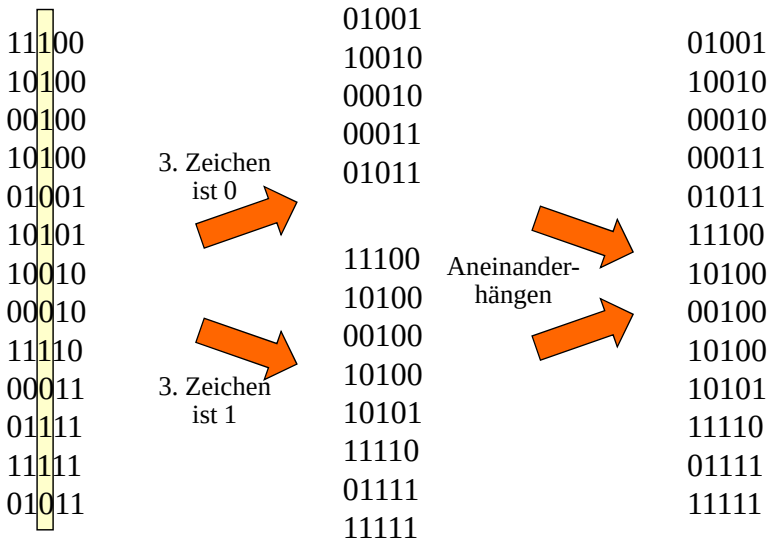
Nun iterativ weiter durch alle Stellen der Schlüssel.

Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5

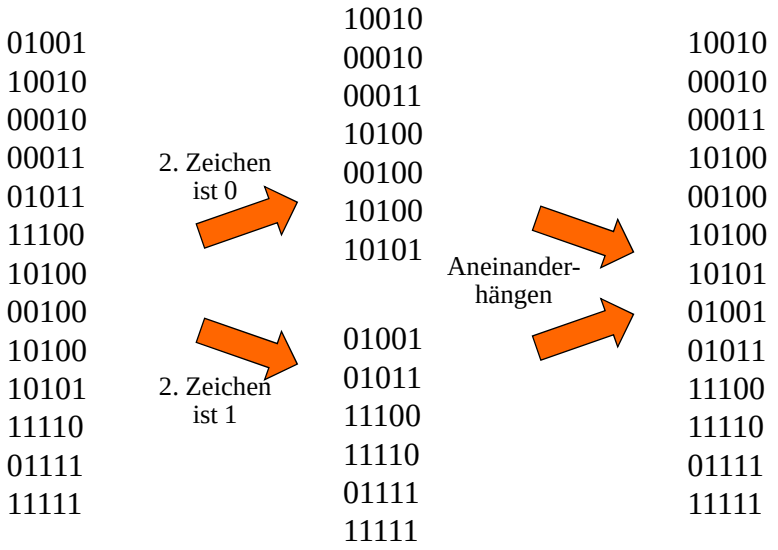


Nun iterativ weiter durch alle Stellen der Schlüssel.

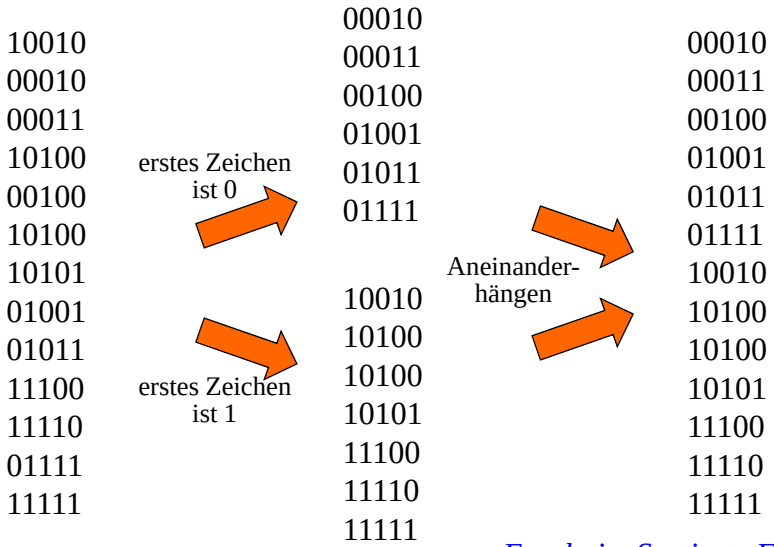
Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Ergebnis: Sortierte Folge.

Man beachte: Die Sortierung jeder Phase muss stabil sein.
Hat man s Zeichen (z.B. $s = 26$ für das Alphabet oder $s = 128$ für den ASCII-Zeichensatz), dann muss man s solche "Fächer" bereitstellen. In der Regel organisiert man die Fächer als Listen, an die man hinten (in einem Schritt) den jeweils nächsten gelesenen Schlüssel anhängt; danach werden die s Listen (in s Schritten) aneinandergehängt und die nächste Verteilphase kann beginnen.
Das Sortieren erfordert insgesamt $s \cdot n$ Vergleiche. Es eignet sich besonders gut für das Sortieren von binärer Information (z.B. in der Systemprogrammierung) und in allen Fällen, in denen $s \leq \log(n)$ ist. Nachteilig ist, dass man in der Regel das Doppelte an Speicherplatz benötigt. Man kann aber auch einen Austausch wie bei Quicksort vornehmen, so dass die 0-en oben und die 1-en unten zu stehen kommen (Vorsicht wegen der erforderlichen Stabilität!); im Falle $s > 2$ bietet sich auch ein bucket sort an, siehe 10.6.

10.4 Sortieren durch Aussuchen/Auswählen

Vorgehen:
Wähle das kleinste Element aus, stelle es an die erste Stelle und mache genauso mit den restlichen Elementen weiter.

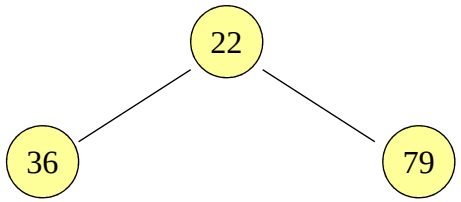
10.4.1 "Minimum sortieren":

```
for i in 1..n-1 loop
    min := A(i); pos := i;          -- finde das kleinste Element von A(i) bis A(n)
    for j in i+1..n loop
        if A(j) < min then min:=A(j); pos := j; end if;
    end loop;                       -- das kleinste Element steht an Position pos
    A(pos) := A(i); A(i) := min;    -- nun steht das kleinste Element an Position i
end loop;
```

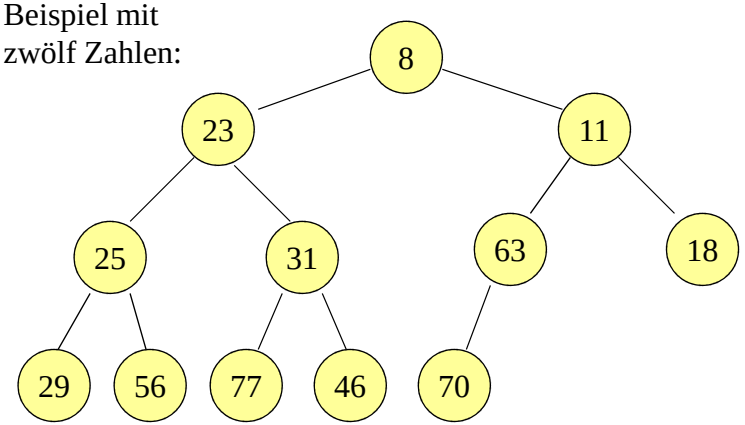
Zahl der Vergleiche stets ½·n·(n-1) Schritte: $\Theta(n^2)$
Platzaufwand 4 zusätzliche Speicherplätze: $O(1)$

Überlegung:
Könnte man die Information "Minimum sein" besser anordnen?

Ja, in einem binären Baum. Betrachte einen Knoten mit zwei Nachfolgern. Schreibe in den Elternknoten das Minimum der drei Knoten.



Beispiel mit zwölf Zahlen:



Als Feld levelweise aufgeschrieben:

8	23	11	25	31	63	18	29	56	77	46	70
1	2	3	4	5	6	7	8	9	10	11	12

Besonderheit dieses Baums: Auf jedem Pfad von der Wurzel zu einem Blatt sind die Elemente aufsteigend geordnet.

8	23	11	25	31	63	18	29	56	77	46	70
1	2	3	4	5	6	7	8	9	10	11	12

Die Bedingung "Der Inhalt eines Knotens ist nicht größer als der Inhalt jedes Nachfolgeknotens" lässt sich präzisieren durch $A(i) \le A(2i)$ und $A(i) \le A(2i+1)$. Folgen oder Felder mit dieser Eigenschaft nennen wir "Heap" (meist übersetzt mit "Haufen", es handelt ich aber um gut geordnete Haufen; sie haben nichts mit der Halde zu tun, die die mit new eingeführten Daten aufnimmt und die im Englischen ebenfalls "heap" heißt).

Definition 10.4.2: Heap-Eigenschaft

Eine Folge oder ein array A(1), A(2), A(3), ..., A(n) heißt ein (aufsteigender) **Heap**, wenn für jedes i gilt:

$A(i) \le A(2i)$ und $A(i) \le A(2i+1)$,

wobei natürlich nur solche Ungleichungen betrachtet werden, bei denen 2i bzw. 2i+1 nicht größer als n sind.

Eine Folge oder ein array A(1), A(2), A(3), ..., A(n) heißt ein absteigender **Heap**, wenn für jedes i gilt:

$A(i) \ge A(2i)$ und $A(i) \ge A(2i+1)$,

wobei natürlich nur solche Ungleichungen betrachtet werden, bei denen 2i bzw. 2i+1 nicht größer als n sind.

10.4.3 Heapsort:

Gegeben sei ein Feld A(1), A(2), A(3), ..., A(n) mit Elementen aus einer geordneten Menge.

1. Wandle dieses Feld in einen absteigenden Heap um, so dass anschließend gilt: $A(i) \ge A(2i)$ und $A(i) \ge A(2i+1)$ für alle i (sofern $2i \le n$ bzw. $2i+1 \le n$ ist).
2. Für j von n abwärts bis 2 wiederhole:
   Vertausche A(1) und A(j).
   (Nun verletzt A(1) in der Regel die Heapeigenschaft.)
   Wandle das Feld A(1..j-1) ausgehend von der Wurzel so um, dass wieder ein absteigender Heap entsteht.

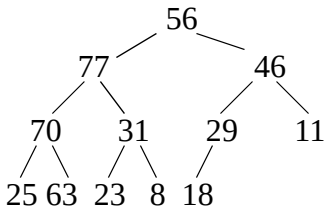
Im Folgenden beschreiben wir das Umwandeln in einen Heap (1.) und die Wiederherstellung der Heap-Eigenschaft (2.).

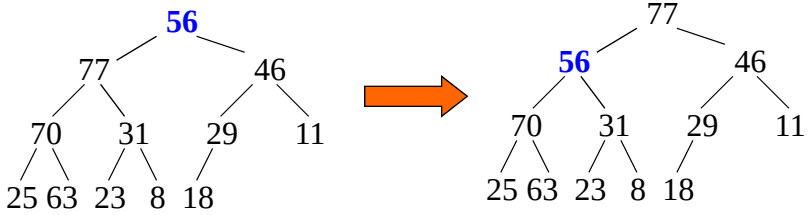
Die zentrale Prozedur ist die Herstellung der Heap-Eigenschaft in dem Teil des Feldes A, das mit dem Index "links" beginnt und mit dem Index "rechts" endet, unter der Annahme, dass höchstens beim Index "links" die Heap-Eigenschaft verletzt ist.

Hierfür vergleicht man den Inhalt des Elements A(links) mit den Inhalten der beiden Nachfolgeknoten und lässt gegebenenfalls den Inhalt von A(links) durch Vertauschen mit dem größeren der beiden Nachfolger-Inhalten "absinken".

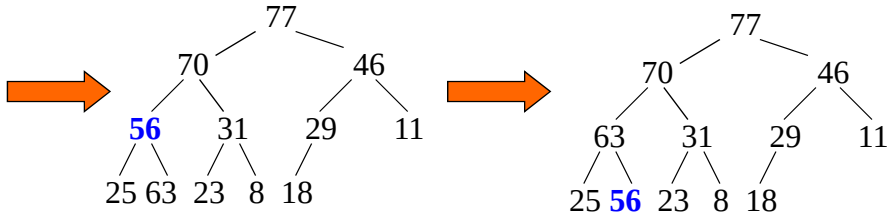
Betrachte ein Beispiel (mit links = 1 und rechts = 12) :

Die Heap-Eigenschaft ist hier nur bei "56" verletzt.





Vorgehen: Vergleiche 56 mit 77 und 46. Das Maximum ist 77, daher werden 77 und 56 vertauscht und bei 56 weitergemacht.



10.4.4: Prozedur für das Absinken

```
procedure sink (links, rechts: 1..n) is          -- A und n sind global
i, j: Natural;  weiter: Boolean := true;  v: <Elementtyp>;
begin  v := A(links);  i := links;  j := i+i;
    while  (j <= rechts) and weiter  loop
        if  j = rechts  then
            if A(j) > v then A(i) := A(j); i := j; end if;
            weiter := false;
        elsif A(j) < A(j+1) then
                if v < A(j+1) then  A(i) := A(j+1); i := j+1;
                else weiter := false; end if;
        else    if v < A(j) then A(i) := A(j); i := j;
                else weiter := false; end if;
        end if;
        j := i+i;
    end loop;
    A(i) := v;
end sink;
```

```
-- v wird erst am Ende explizit eingetragen.
-- i gibt am Ende die aktuelle Position an, an
-- der v = A(links) einzufügen ist.
-- Im Inneren der Schleife werden bis zu zwei
-- Vergleiche zwischen Elementen durchgeführt.
```


10.4.5: Prozedur für Heapsort

```
procedure heapsort is                            -- A und n sind global
 procedure sink ... begin .... end sink;         -- siehe oben 10.4.4
 x: <Elementtyp>;
 begin                                           -- wandle A in Heap um
    for k in reverse 1..(n/2) loop sink(k, n); end loop;
    for k in reverse 2..n loop                   -- Sortierphase
        x := A(1); A(1) := A(k); A(k) := x;      -- vertausche A(1) mit A(k)
        sink(1, k-1);  end loop;                 -- Wurzel absinken lassen
 end heapsort;
```

Hinweis: Es lassen sich noch einige Umspeicherungen vermeiden, indem man das Wechselspiel zwischen v (in 'sink') und x optimiert. Dies ändert aber nichts an der Zahl der (Element-) Vergleiche.

Wie viele Vergleiche benötigt Heapsort?

1. Aufbau des Heaps (for k in reverse 1..n/2 loop sink (k, n); end loop;)

Für k von n/2 bis n/4:	maximal jeweils 2 Vergleiche
für k von n/4 bis n/8:	maximal jeweils 4 Vergleiche
für k von n/8 bis n/16:	maximal jeweils 6 Vergleiche ....
für k von $n/2^i$ bis $n/2^{i+1}$	maximal jeweils 2i Vergleiche

für i = 1, 2, ..., log(n)-1.

Aufsummieren ergibt maximal 2·n Vergleiche:

$2\cdot n/4 + 4\cdot n/8 + 6\cdot n/16 + 8\cdot n/32 + ... + (2\cdot\log(n))\cdot 1$

$= n\cdot(2 - 2\cdot(\log(n)+1)/n) = 2\cdot n - 2\cdot\log(n) - 2 < 2\cdot n$ Vergleiche.

Der Aufbau des Heaps erfolgt also in linearer Zeit.

Dies beweist man genauso wie die entsprechende Formel $\sum_{j=1}^{k} j\cdot 2^{j-1}$ in Abschnitt 6.5.1. Es gilt:

$1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + ... + m/2^m = 2 - (m+1)/2^{(m-1)}$.

2. Sortierphase (for k in reverse 2..n loop ... sink(1, k-1); end loop;)

Für k von n bis n/2:	maximal 2·log(n) Vergleich
für k von n/2 bis n/4:	maximal 2·log(n)-1 Vergleiche
für k von n/4 bis n/8:	maximal 2·log(n)-2 Vergleiche ....
für k von $n/2^{i-1}$ bis $n/2^i$	maximal 2·(log(n)-i+1)

Vergleiche (i=1, 2, ..., log(n))

Aufsummieren ergibt maximal 2·n·log(n) Vergleiche:

$\log(n)\cdot n + (\log(n)-1)\cdot n/2 + (\log(n)-2)\cdot n/4 + (\log(n)-3)\cdot n/8 + ... + 2$

$< \log(n)\cdot n + \log(n)\cdot n/2 + \log(n)\cdot n/4 + \log(n)\cdot n/8 + ... + \log(n)\cdot n/n$

$= n\cdot\log(n)\cdot(1 + 1/2 + 1/4 + 1/8 + ... + 1/2^i + ...)$

$= n\cdot\log(n)\cdot(2 - 1/n) = 2\cdot n\cdot\log(n) - \log(n) < 2\cdot n\cdot\log(n)$.

Insgesamt ergeben sich für Heapsort im worst case maximal

$(2 \cdot n - 2 \cdot \log(n) - 2) + (2 \cdot n \cdot \log(n) - \log(n)) =$

$\leq 2 \cdot n \cdot (\log(n)+1)$ Vergleiche.

(Dies kann man noch etwas besser abschätzen; versuchen Sie es!)
Der Mittelwert wird ebenfalls dicht bei $2 \cdot n \cdot \log(n)$ liegen, da Heapsort keine Vorsortierungen oder günstigen Konstellationen ausnutzt und da das in die Wurzel vertauschte Element klein ist, also meist weit nach unten im Baum absinkt. Auch der best case spart aus diesem Grund kaum Vergleiche. Daher folgt:

Satz 10.4.6:
Dieses normale Heapsort (aus dem Jahre 1962) benötigt im schlechtesten Fall höchstens $2 \cdot n \cdot (\log(n)+1)$ Vergleiche.

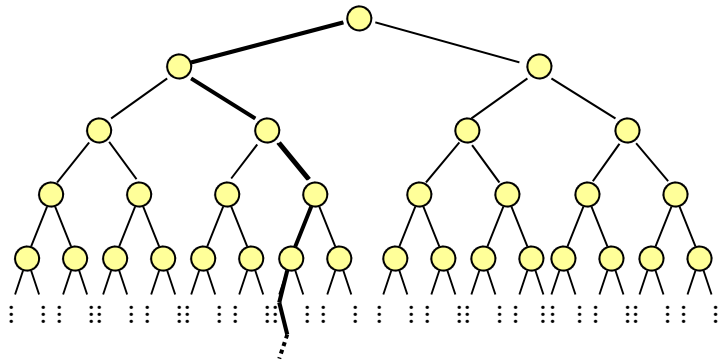
(Der average case liegt recht nahe beim schlechtesten Fall.)

Geht es nicht doch noch besser?

Ja, man kann den Faktor 2 noch verkleinern (allerdings kann er nicht kleiner als "1" werden, wie in Satz 10.1.11 gezeigt wurde.)

Beobachtung: Beim eigentlichen Sortieren wird das letzte Element mit dem ersten vertauscht. Die Prozedur "sink" wird dieses letzte Element in der Regel sehr weit absenken, da es ja zu den leichten Elementen gehört hat, die sich ganz unten im Baum befinden. Man sollte daher das Element nicht von oben nach unten absenken, sondern es von unten nach oben (also "bottom-up") aufsteigen lassen. Hierzu muss man aber die Stelle, an der es einzufügen ist, kennen. Genau diese Stelle ermittelt man durch die Berechnung des "Einsinkpfads".

10.4.7 Einsinkpfad: Dies ist der Weg, den ein Element, das in die Wurzel gesetzt wurde, nehmen muss, damit die Heap-Eigenschaft wiederhergestellt wird (vgl. 10.4.4).
Dieser Pfad ist unabhängig vom einzusortierenden Element. Er endet in einem Blatt des Baumes. Es genügt, den Index dieses Blattes zu bestimmen.



Ermittlung des Einsinkpfads:

```
starte mit der Wurzel;
while  noch nicht Blatt erreicht  loop
      gehe zum Nachfolgeknoten mit dem größeren Inhalt;
end loop;
```

Bei der Darstellung mit Feldern muss man nur den Index j des letzten Elements auf dem Einsinkpfad kennen. Die anderen Knoten auf diesem Pfad besitzen die Indizes j div 2, (j div 2) div 2, ((j div 2) div 2) div 2, ..., 1. (In binärer Darstellung muss man also nur die letzte Ziffer streichen.)

Die folgende Funktion berechnet den Index j des letzten Elements des Einsinkpfads, wobei man den Fall, dass der letzte Knoten keinen Geschwisterknoten besitzt, gesondert berücksichtigt ("if m = rechts then ...").

Ermittlung des Einsinkpfads:

```
function einsinkpfad (rechts: 1..n) return 1..n is
j: 1..n := 1; m: Natural := 2;
begin
    while  m < rechts  loop
        if  A(m) < A(m+1)  then  j := m+1;  else  j := m;  end if;
        m := j+j;
    end loop;
    if  m = rechts  then  j := rechts;  end if;
    return j;
end;
```


10.4.8 Bottom-up-Heapsort: (Carlsson 1987, Wegener 1993)

1. Ermittle den Index j des letzten Knotens auf dem Einsinkpfad.
2. Suche von j aus rückwärts entlang des Einsinkpfads die Stelle, wo das einzusortierende Element hingehört.
3. Füge es dort ein und schiebe alle darüber stehenden Elemente entlang des Einsinkpfads um eine Position in Richtung zur Wurzel.

Für die Programmierung benutzen wir die obige Funktion "einsinkpfad", die wir jedoch direkt in den Algorithmus integrieren. Weiterhin führen wir die Verschiebung von Punkt 3 bereits beim Berechnen von j durch, da man in der Regel den Einsinkpfad nur wenige Schritte zurücklaufen muss und hierdurch im Mittel ein doppeltes Durchlaufen vermieden wird.


```
procedure bottomupheapsort is                -- A und n sind global
procedure sink ... begin .... end sink;      -- wie früher
j, m: Natural;  x: <Elementtyp>;
begin                                        -- wandle A in einen Heap um
    for k in reverse 1..n/2 loop sink (k, n); end loop;
    for k in reverse 2..n loop               -- x = A(k) einsinken lassen
        x := A(k); A(k) := A(1);             -- rette A(1) nach A(k)
        j := 1; m:= 2;                       -- Suche den Index j
        while  m < k-1  loop
            if  A(m) < A(m+1)  then  A(j) := A(m+1); j := m+1;
            else  A(j) := A(m); j := m;  end if;
            m := j+j;
        end loop;
        if  m = k-1  then  A(j) := A(k-1); j := k-1;  end if;
-- Nun ist der Index j (= Ende des Einsinkpfads) bekannt und alle Inhalte auf dem
-- Einsinkpfad sind um eine Position in Richtung der Wurzel verschoben worden.
```



```
-- Die Elemente des Einsinkpfads müssen nun zurückgeschoben werden,
-- solange die Stelle, an die x gehört, noch nicht erreicht ist.
        while  (j > 1) and then (A(j) < x)  loop
            i := j / 2; A(j) := A(i); j := i;
        end loop;
-- Die Stelle j, an die x gehört, ist nun erreicht.
    A(j) := x;
    end loop k;
end bottomupheapsort;
```

Wie viele Vergleiche benötigt Bottom-up-Heapsort?

1. Aufbau des Heaps: Genauso wie beim normalen Heapsort
maximal $2 \cdot n - 2 \cdot \log(n) - 2 \leq 2 \cdot n$ Vergleiche.

2. Sortierphase

Hier benötigt man maximal für jedes k so viele Vergleiche, wie die doppelte Länge des Einsinkpfads ist, also rund $2 \cdot \log(k)$.
Dies führt zunächst nur auf genau die gleiche Abschätzung wie beim normalen Heapsort.
Aber: In den meisten Fällen wird man bereits nach etwas mehr als log(k) Vergleichen fertig sein.
Experimente bestätigen, dass der Faktor "2" vom normalen Heapsort im Mittel auf "1" sinkt; dies lässt sich auch beweisen.
Man kann aber Folgen konstruieren, bei denen man nur auf den Faktor 1.5 kommt. Daher gilt im worst case nur:

Bottom-up-Heapsort benötigt im worst case maximal $1.5 \cdot n \cdot \log(n)$ Vergleiche. Dieser Fall tritt aber fast nie auf, so dass man in der Praxis von $n \cdot \log(n) + O(n)$ Vergleichen ausgehen kann. Carlsson konnte zeigen, dass im Mittel höchstens $n \cdot \log(n) + 0.67 \cdot n$ Vergleiche benötigt werden.

10.4.9 Satz: Bottom-up-Heapsort benötigt
im schlimmsten Fall höchstens $1.5 \cdot n \cdot \log(n)$,
im Mittel höchstens $n \cdot \log(n) + 0.67 \cdot n$ Vergleiche.

Beachte: Heapsort und seine Varianten sind
garantierte $n \cdot log(n)$ - Verfahren.

Hinweis: Es gibt weitere Varianten, z.B. von McDiarmid and Reed 1998 oder von Katajainen 1998. Ziel ist es, die untere theoretische Schranke von Satz 10.1.11 zu erreichen. Der Faktor 1 (statt 1.5) wurde mit dem "ultimativen" Heapsort erreicht, das aber (wegen eines zu hohen linearen Anteils) bisher noch für die Praxis ungeeignet ist (siehe Algorithmik-Vorlesung).

Diskussion dieses Ergebnisses:

Mit Bottomup-Heapsort haben wir ein Verfahren gefunden, das schneller als Quicksort sein müsste. Dies trifft vor allem dann zu, wenn im Algorithmus der Vergleich zwischen zwei Elementen, also die Abfragen "A(m) < A(m+1)" und "A(j) < x", viel mehr Zeit kosten als alle anderen elementaren Aktionen; denn wir haben ja nur die Anzahl dieser Vergleiche gezählt. Diese Annahme gilt sicher, wenn die Schlüssel besonders lang (z.B. Texte) sind.

Sind die Schlüssel jedoch Zahlen vom Typ Integer, so dauert der Vergleich nicht länger als eine Zuweisung der Form "A(j) := A(i)". Dann kommen vor allem die Zugriffe auf den Hauptspeicher zum Tragen. Bei Quicksort erfordert das Umspeichern 4 Zugriffe, da zwei Werte ausgetauscht werden, bei Heapsort braucht man nur 2, da ein Wert verschoben wird. Allerdings werden bei Heapsort bei jedem Absinken mindestens log(k), insgesamt also ungefähr $n \cdot \log(n)$ Verschiebungen ausgeführt, während Quicksort mit weniger als $\frac{1}{2} \cdot n \cdot \log(n)$ Vertauschungen auskommt. Insgesamt ist bei Heapsort also mit $2 \cdot n \cdot \log(n)$, bei Quicksort dagegen mit weniger als $2 \cdot n \cdot \log(n)$ Zugriffen auf den Hauptspeicher zu rechnen. Daher wird Quicksort in der Praxis trotzdem oft etwas schneller als Heapsort sein. Bottom-up-Heapsort sollte aber schätzungsweise für $n > 50\,000$ besser abschneiden.

Auch die weiteren elementaren Anweisungen spielen eventuell noch eine Rolle. Für eine genaue (nicht uniforme) Abschätzung muss man nun die Implementierung und die Bearbeitungszeiten, die der verwendete Computer für die verschiedenen Anweisungen benötigt, kennen. Erfahrene Informatiker(innen) können dies gut einschätzen oder sie ermitteln - aufbauend auf den theoretischen Resultaten - die tatsächlichen Laufzeiten mit Hilfe von Testläufen oder aus Statistiken des Betriebssystems.

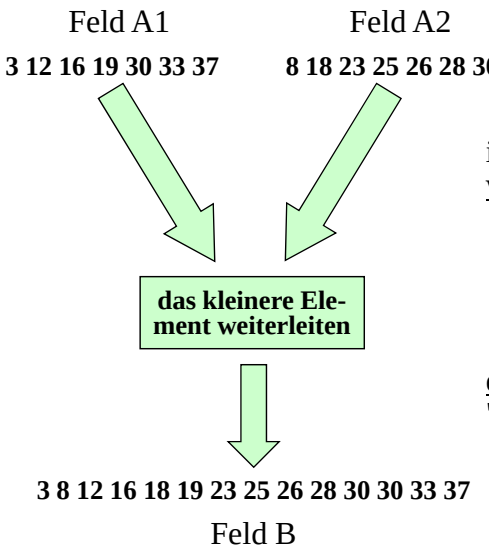
10.5 Mischen

Die bisherigen Sortierverfahren haben einzelne Elemente verglichen, die oft weit voneinander entfernt in der zu sortierenden Folge standen. Sie sind daher für riesige Datenbestände, die auf Hintergrundspeichern stehen, nur bedingt einsetzbar.

Daten werden von Hintergrundspeichern "stromartig" eingelesen, vergleichbar dem Lesen von Magnetbändern. Deshalb muss man immer möglichst viele Daten, die zusammenhängend verglichen und sortiert werden können, zusammenfassen und verarbeiten.

Hierfür ist vor allem das Zusammenmischen ("Verschmelzen") zweier bereits sortierter Datenströme geeignet.

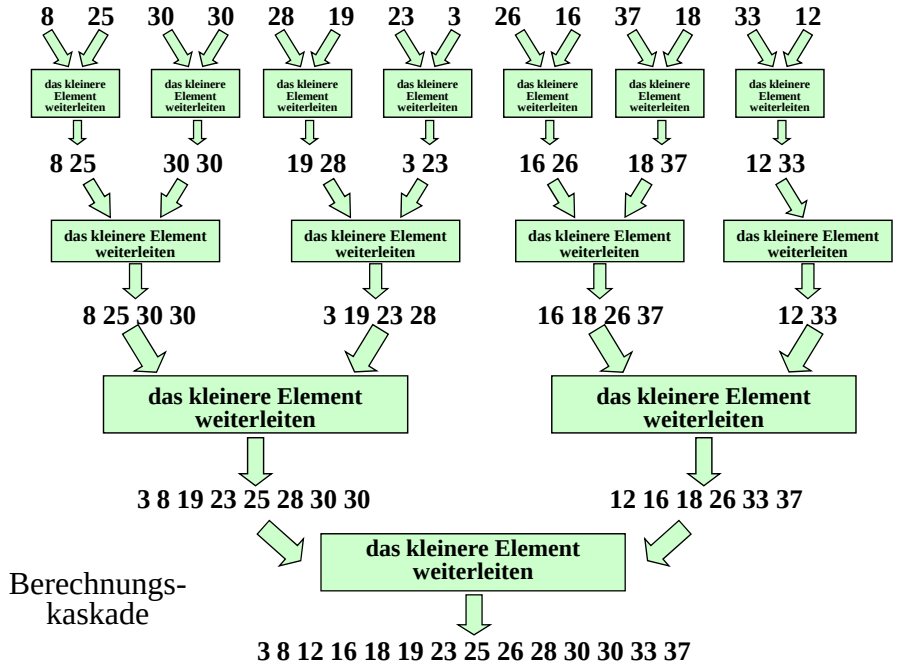
(Hinweis: Ob auf "≤" oder auf "<" abgefragt wird, ist im Folgenden für die Stabilität des Mischverfahrens wichtig.)



Verschmelzen zweier Folgen

```
i:= 1; j := 1; k := 0;
while i≤n and j≤n loop
  k := k+1;
  if A1(i) ≤ A2(j)
  then B(k) := A1(i); i := i+1;
  else  B(k) := A2(j); j := j+1;
  end if;
end loop;
"Füge den Rest an B an"
```

Der gesamte Sortierprozess ist auf der nächsten Folie dargestellt.



10.5.1: Verschmelzen zweier Folgen vom Typ Vektor

```
procedure Verschmelzen (A1, A2: in Vektor; B: in out Vektor;
                        LA1, RA1, LA2, RA2, LB: in Integer;
                        RB: out Integer) is
i, j, k: Integer;
begin i:= LA1; j := LA2; k := LB-1;        -- "L" = links, "R" = rechts
  while i≤RA1 and j≤RA2 loop               -- Verschmelze A1 und A2 zu B
    k := k+1;
    if A1(i) ≤ A2(j)  then B(k) := A1(i); i := i+1;
                      else  B(k) := A2(j); j := j+1; end if;
  end loop;                                -- nun muss der Rest einer Folge noch angefügt werden
  if i ≤ RA1  then
        for m in i..RA1 loop k:=k+1; B(k):=A1(m); end loop;
  else  for m in j..RA2 loop k:=k+1; B(k):=A2(m); end loop; end if;
  RB := k;                                 -- RB dient nur für Kontrollzwecke, kann entfallen
end Verschmelzen;
```


10.5.2: Sortieren durch Mischen

Es soll ein Feld A(1..n) durch Mischen sortiert werden.
Zuerst die "*Bottom-Up*-Denkweise", die zu einem Iterationsverfahren führt: Man verschmilzt zunächst je zwei Folgen der Länge 1 zur sortierten Folgen der Länge 2 (man muss hierfür n/2 mal "Verschmelzen" aufrufen),
dann verschmilzt man je zwei sortierte Folgen der Länge 2 zu sortierten Folgen der Länge 4 (hierfür muss man n/4 mal "Verschmelzen" aufrufen),
danach das Gleiche für sortierte Folgen der Länge 4, 8, 16 usw., bis zwei Folgen der Länge n/2 zu einer sortierten Folge der Länge n verschmolzen wurden.
Sofern n eine Zweierpotenz war, funktioniert dieses Verfahren bereits; im allgemeinen Fall muss man beim Verschmelzen unterschiedliche Längen von Folgen berücksichtigen (vgl. Beispiel).
Dieses Mischen heißt in der Literatur "straight mergesort".

Sortieren durch Mischen (Fortsetzung)

Nun die "*Top-Down*-Denkweise": Um ein Feld zu sortieren, sortiert man zuerst die linke Hälfte, dann die rechte Hälfte und verschmilzt die beiden sortierten Folgen. Das Ergebnis ist eine rekursive Prozedur.
Da dieses Vorgehen leichter zu verstehen und aufzuschreiben ist, wird die rekursive Version im Folgenden realisiert.
Der Ansatz ist einfach. Die Hauptschwierigkeit besteht hier in der präzisen Angabe der jeweiligen Teilfeld-Grenzen.
Wir verzichten auf volle Allgemeinheit und wollen "nur" das Feld A(L..R) sortieren. Sortierte Teilfelder A(x..y) und A(u..v) mischen wir in ein Hilfsfeld B(L..R) und schreiben danach das Ergebnis wieder nach A zurück.
Beachten Sie, dass die Abfrage "A(i) ≤ A(j)" die Stabilität des Mischens sichert (im Gegensatz zu "A(i) < A(j)").

10.5.3: Programm zum Sortieren durch Mischen ("Mergesort")

```
procedure Mergesort (L, R: in Integer) is   -- Es wird A(L..R) sortiert.
Mitte: Integer;  i, j, k: Integer;          -- Die Felder A und B sind global.
begin                                       -- Beachte: Mitte < R stets wegen L < R
if R > L then  Mitte := (L+R)/2;            -- Sortiere rekursiv zwei Hälften der Folge
   Mergesort(L, Mitte);  Mergesort(Mitte+1, R);
   i := L; j := Mitte+1; k := L-1;          -- Nun mischen von A nach B, wie in 10.5.1
   while i≤Mitte and j≤R loop  k := k+1;
      if A(i) ≤ A(j)  then B(k) := A(i); i := i+1;
                      else  B(k) := A(j); j := j+1; end if;
   end loop;                -- nun muss der Rest einer Folge noch angefügt werden
   if i ≤ Mitte  then
      for m in i..Mitte loop k:=k+1; B(k):=A(m); end loop;
   else for m in j..R loop k:=k+1; B(k):=A(m); end loop; end if;
   for m in L..R loop A(m) := B(m); end loop;    -- zurückkopieren nach A
end if;
end Mergesort;
```


Wir haben das Verfahren mit Feldern realisiert.

Es ist aber klar, dass man es auch leicht mit linearen Listen implementieren kann, wobei nur Zeiger umgesetzt werden müssen. Hierbei kann man alle Umspeicherungen vermeiden. Vor allem der Teil im Programm, der mit

-- nun muss der Rest einer Folge noch angefügt werden

beginnt, kann durch eine einzige Zeigersetzung erledigt werden.

Übungsaufgabe: Durchdenken Sie die Vor- und Nachteile eines solchen Verfahrens und entwerfen Sie ein Programm, für das die zu sortierenden Daten als einfach verkettete lineare Liste vorliegen.

10.5.4 Aufwandsabschätzung für das Mischen:

Wenn V(n) die maximale Zahl der Vergleiche zum Sortieren von n Elementen ist, dann gilt:

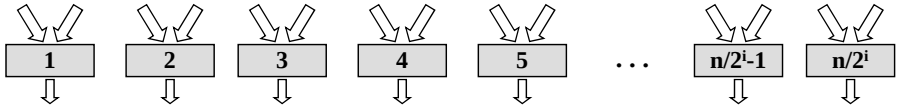
$V(1) = 0$ und für alle $n > 1$: $V(n) = 2 \cdot V(n/2) + n-1$.

Durch Einsetzen
$V(n) = 2 \cdot V(n/2) + n-1 = 2 \cdot (2 \cdot V(n/4) + n/2-1) + n-1$
$= 4 \cdot V(n/4) + n/2-1 + n-1 = ...$
erhält man als Lösung (n sei eine Zweierpotenz):

$V(n) = n \cdot \log(n) - n + 1$

Mergesort ist also ein garantiertes n·log(n)-Verfahren.
Die Zahl seiner Vergleiche kommt sehr dicht an die untere theoretische Grenze heran, siehe Satz 10.1.11.

Direkte Herleitung der Zeitkomplexität: Betrachte die i-te Schicht der baumartigen Berechnungskaskade mit $n/2^i$ Verarbeitungseinheiten:



Für i = 1, 2, ..., log(n):

Eingabe: $n/2^{i-1}$ Folgen, jede besitzt die Länge 2^{i-1}

Verarbeitung: für je zwei Folgen höchstens 2^i-1 Vergleiche

Ausgabe: $n/2^i$ Folgen, jede besitzt die Länge 2^i.

Hieraus erhält man die Gesamtzahl der Vergleiche durch Summation aller Schichten i = 1, 2, ..., log(n) (n = Zweierpotenz, also $2^{\log(n)} = n$):

$n/2\cdot(2-1) + n/4\cdot(4-1) + n/8\cdot(8-1) + n/16\cdot(16-1) +... + n/2^{\log(n)}\cdot(2^{\log(n)}-1)$

$= (n + n + ... + n) - n\cdot(1/2 + 1/4 + 1/8 + ... + 1/n)$

$= n\cdot\log(n) - n\cdot(1-1/n) = n\cdot\log(n) - n + 1 = V(n)$

Bei Mergesort sind *viele Umspeicherungen* erst beim Verschmelzen von A nach B und dann zurück nach A erforderlich. Wie hoch ist die Zahl der Speichervorgänge?

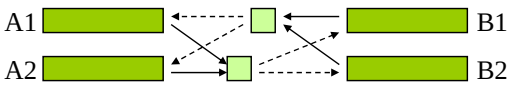
In jeder Rekursionstiefe werden alle Daten genau einmal nach B und wieder zurück transportiert. Folglich werden insgesamt stets $4\cdot n\cdot\log(n)$ Speicherzugriffe durchgeführt. (Dies setzt allerdings voraus, dass das Programm leicht modifiziert wird, dass also "if $A(i) \le A(j)$ then" nicht zu zusätzlichen Speicherzugriffen führt.)

Dies scheint viel zu sein. Aber auch bei Quicksort finden je Rekursionstiefe bis zu n/2 Vertauschungen statt, die je 4 Speicherzugriffe erfordern, weshalb Quicksort auch im günstigsten Falle, dass die Rekursionstiefe durch log(n) beschränkt bleibt, bis zu $2\cdot n\cdot\log(n)$ Umspeicherungen ausführt, und zwar für weit entfernt liegende Speicherplätze.

10.5.5 Varianten des Sortierens durch Mischen

Das obige Vorgehen bezeichnet man als "*Zwei-Phasen-Mischen*", da die Daten von Feld A nach B verschmolzen (erste Phase) und anschließend zurück nach A (zweite Phase) kopiert werden. Man spricht auch von "*2-Wege-Mischen*", da ein Weg nach B und einer zurück nach A führt.

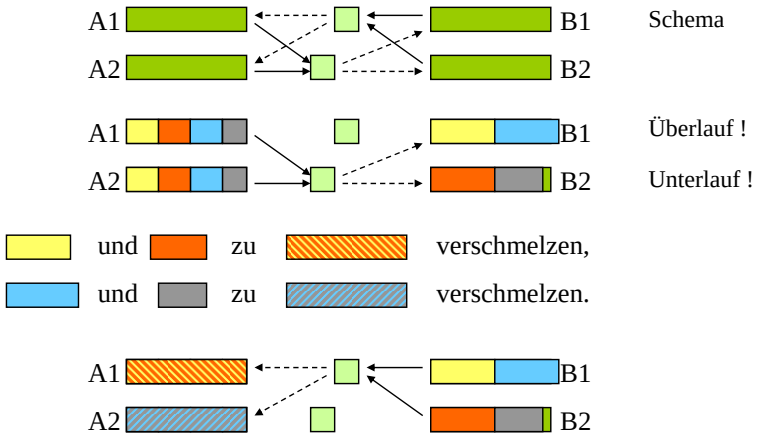
Natürlich kann man die Rolle der Ziel-Felder in jedem Durchgang ändern, d.h.: Man verschmilzt zwei sortierte Folgen von A1 und A2 abwechselnd auf die Felder B1 und B2 und anschließend zwei sortierte Folgen von B1 und B2 zurück nach A1 bzw. A2 usw. So spart man die Hälfte der Umspeicheroperationen. Man muss sich hierbei die jeweiligen Grenzen merken und bis zu 2n zusätzliche Speicherplätze bereitstellen.



Ein-Phasen-Mischen, wobei 4 Wege möglich sind.

Die durchgezogenen Pfeile bezeichnen das Zusammenführen zweier sortierter Folgen. Das sortierte Ergebnis des Verschmelzens wird abwechselnd über einen der gestrichelten Pfeile weitergeleitet.

Veranschaulichung: A1 und A2 sind Teilfelder; die Grenzen bereiten bei den letzten beiden zu verschmelzenden Blöcken zusätzliche Schwierigkeiten und können zum Über- oder Unterlauf führen. Wenn A1 das zu sortierende Feld am Anfang ist (also n Elemente besitzt), dann sollten A2, B1 und B2 ebenfalls n Speicherplätze besitzen; als erstes teilt man A1 zur Hälfte auf A1 und A2 auf und startet dann das Verfahren.

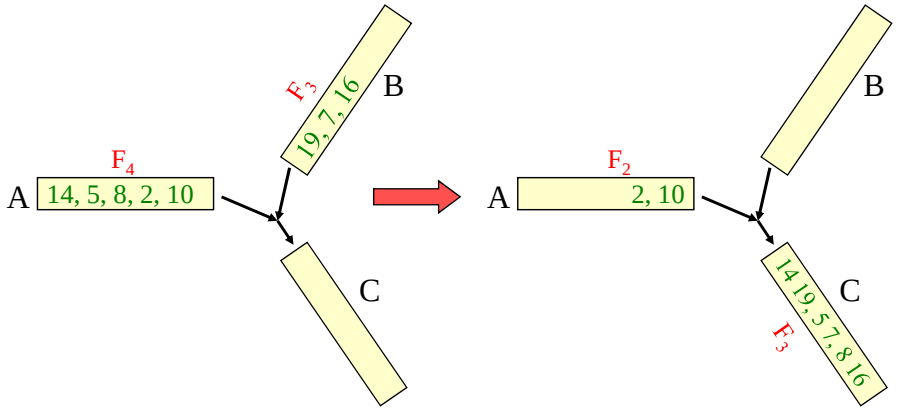


3-Wege-Mischen:

Man kann auch mit drei Feldern A, B und C arbeiten. Zuerst wird die zu sortierende Folge auf zwei Bänder A und B verteilt und zwar F_k sortierte Teilfolgen auf Band A und F_{k-1} sortierte Teilfolgen auf Band B (F_k ist die k-te Fibonaccizahl, siehe 8.4.7; falls die Anzahlen nicht "aufgehen", so fülle man mit "dummy"-Folgen auf die nächste Fibonaccizahl auf). Nun werden sortierte Teilfolgen von A und B nach C gemischt, bis das Feld B keine sortierte Folge mehr besitzt. Dann besitzen C genau F_{k-1} und A genau $F_k - F_{k-1} = F_{k-2}$ sortierte Teilfolgen. Danach werden die sortierten Teilfolgen von A und C nach B gemischt, bis das Feld A keine sortierte Teilfolge mehr besitzt (auf den Feldern B und C befinden sich nun F_{k-2} bzw. F_{k-3} sortierte Teilfolgen). Nun wird A das Ziel-Feld, d.h., es werden sortierte Teilfolgen von B und C nach A gemischt usw., bis am Ende auf einem der Bänder die sortierte Gesamtfolge steht. (Man wählt hier Fibonacci-Zahlen, weil die zu sortierenden Teilfolgen dann nicht zu kurz werden und weil man nicht ständig die Folgenlängen abfragen muss.)

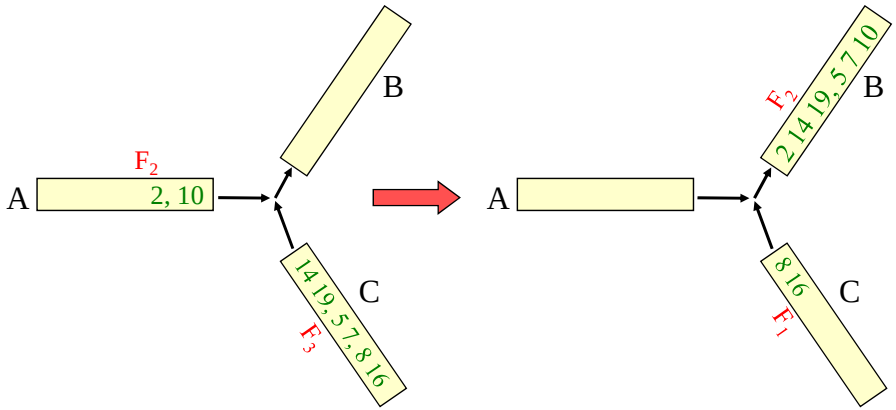
Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16

Dies sind $F_6 = 8$ Elemente. Verteile also $F_5 = 5$ und $F_4 = 3$ Elemente auf zwei Bänder A und B und mische diese auf Band C:



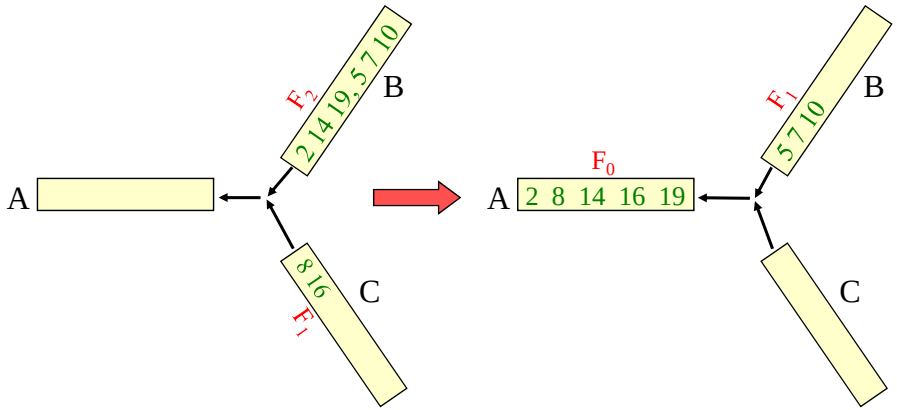
Verschmelze nun $F_4 = 3$ sortierte Teilfolgen von A mit $F_4 = 3$ sortierten Teilfolgen von B. Das Band B wird hierbei leer.

Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16



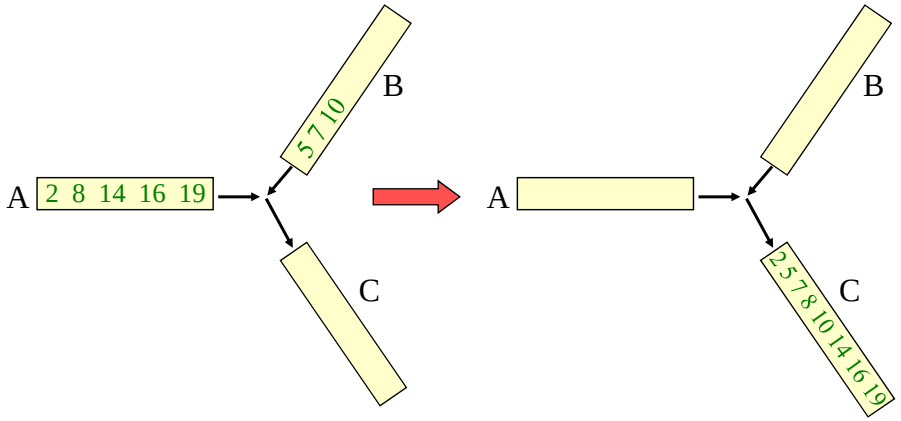
Verschmelze $F_3 = 2$ sortierte Teilfolgen von A mit $F_3 = 2$ sortierten Teilfolgen von C. Das Band A wird hierbei leer.

Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16



Verschmelze $F_2 = 1$ sortierte Teilfolge von B mit $F_1 = 1$ sortierten Teilfolge von C. Das Band C wird hierbei leer.

Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16



Verschmelze $F_1 = 1$ sortierte Teilfolge von A mit $F_0 = 1$ sortierten Teilfolge von B auf Band C. Fertig.

10.5.6 Natürliches Mischen:

Eine Teilfolge $a_i\, a_{i+1} \ldots a_j$ der Folge $a_1\, a_2 \ldots a_n$ heißt ein Lauf, wenn dies eine maximal lange nicht fallende Teilfolge ist, d.h., wenn $a_{i-1} > a_i \le a_{i+1} \le \ldots \le a_j > a_{j+1}$ gilt (wenn $i = 1$ oder $j = n$ ist, so entfallen die entsprechenden Ungleichungen am Rande).

Statt Teilfolgen der Länge 1, dann der Länge 2, 4, ... usw. zu mischen, kann man in jedem Durchgang die vorhandenen Läufe mischen, wodurch sich die Rekursionstiefe und damit die Zahl der Umspeicherungen verringert, die Zahl der Abfragen aber wegen des Tests, wo ein Lauf endet, insgesamt nicht geringer wird. Ein solches Ausnutzen der zufällig vorhandenen Teil-Sortierungen bezeichnet man als *natürliches Mischen*. Dadurch wird das Mischen mit seiner garantierten n·log(n)-Laufzeit auch zu einem ordnungsverträglichen Sortierverfahren.

Aufgabe: Programmieren Sie das "natürliche 3-Wege-Mischen". (Lästig ist das ständige Überprüfen, ob ein Lauf zu Ende ist.)

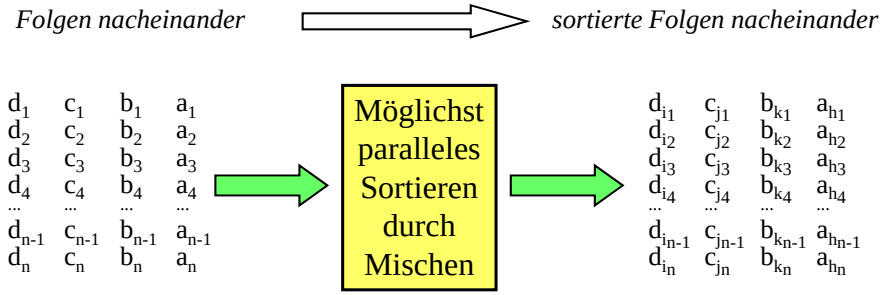
10.5.7 Parallelisieren

Hat man einen Baustein, der das Verschmelzen zweier Folgen vornimmt, so kann man in jeder Rekursionstiefe alle Operationen parallel ausführen (siehe die baumartige Kaskade zu Beginn von 10.5; in der obersten Zeile kann man alle n/2 Vergleiche parallel zueinander durchführen).

Man benötigt dann n/2 solcher Bausteine zum Mischen von Teilfolgen der Länge 1, man braucht n/4 dieser n/2 Bausteine für das Verschmelzen aller Teilfolgen der Länge 2 usw. Durch geschicktes Zusammenschalten kann man also die Kaskade mit n/2 solcher Bausteine realisieren.

Wie lange dauert dann das Sortieren? Die erste Schicht benötigt genau 2 "Takte", die nächste 4, die nächste 8 usw. bis zur letzten Schicht mit n Takten. Folglich kann man wegen $2+4+8+\ldots+n/4+n/2+n = 2n-2$ die n Daten mit diesem Parallelverfahren in 2n-2 Schritten sortieren. Der "Preis" hierfür sind die n/2 Bausteine und ihr Verschaltungsnetz.

Bringt dieses Verfahren trotzdem Vorteile, z.B. im Falle, dass man viele Zahlenfolgen a, b, c, d, ... sortieren möchte?



Faktisch bringt unser Mischverfahren leider kaum etwas, weil ein "Pipelining" zwar denkbar ist, aber wegen der sequenziellen Abarbeitung des letzten Schritts mit mindestens n/2 Schritten zu entsprechend langen Wartezeiten führt. Ein schnelles Verfahren, das das ständige Einschleusen der nächsten Folge in den Verarbeitungsprozess erlaubt, stellen wir in 10.7 vor.

10.6 Streuen und Sammeln

Manchmal liegen die zu sortierenden Werte $a_1\, a_2 \ldots a_n$ in einem festen Intervall [UNT, OB]: UNT ≤ a_i ≤ OB.
Der Einfachheit halber nehmen wir an, die a_i seien natürliche Zahlen und es seien UNT = 0 und OB = m-1.

Bucket sort: Verteile ("streue") die n Elemente A(1..n) auf m nacheinander angeordnete Fächer bucket(0..m-1) ("Eimer" genannt, daher "bucket sort"), hole sie anschließend in der Reihenfolge der Fächer wieder heraus und lege sie im Feld A ab.

Aufwand: O(n+m) für die Zeit- und die Platzkomplexität.

Einsatz: Vor allem, wenn m in der Größenordnung von n liegt.

Programmstück (fügen Sie die Listenbearbeitung selbst hinzu):


```
declare A: array (1..n) of Integer; k: 0..n;     -- n ist global
bucket: array (0..m-1) of "liste von Integer"; ...
begin
  for j in 0..m-1 loop bucket(j) := "leer"; end loop;
  for i in 1..n loop                              -- streuen
    "hänge A(i) an bucket(A(i)) an";
  end loop;
  k := 0;
  for j in 0..m-1 loop                            -- sammeln
    while "bucket(j) nicht leer" loop
        k := k+1; A(k) := "erstes Element von bucket(j)";
        "Entferne aus bucket(j) das erste Element"; end loop;
  end loop;
end;
```


10.7 Paralleles Sortieren

Das Sortieren von n Elementen kostet mindestens n·log(n) Vergleiche, wenn nur ein Prozessor vorhanden ist. Ein guter Rechner kann heute etwa 40 Millionen Vergleiche pro Sekunde durchführen, sofern man höchstens 64-stellige Zahlen als Schlüssel verwendet. Setzt man für die Umspeicherungen und dei weiteren Operationen den Faktor 10 an, so lassen sich mit einem Programm 4 Millionen Vergleiche einschl. der übrigen Operationen pro Sekunde durchführen. Will man maximal eine Minute auf das Ergebnis warten, so lassen sich also $240 \cdot 10^6$ Vergleiche ausführen. Berechne n so, dass $n \cdot \log(n) = 240 \cdot 10^6$ gilt ⇒ Es lassen sich etwa 10.000.000 Integer-Zahlen in einer Minute sortieren.

Solche Größenordnungen sind selten, so dass im Prinzip das Sortieren heute kein Problem mehr darstellen sollte. Allerdings entstehen Probleme, wenn eine Sortierung extrem schnell erfolgen muss, weil sicherheitskritische Systeme hiervon abhängen oder andere Gründe vorliegen.

Will man also schneller sortieren, so kann man entweder auf noch schnellere Rechner warten oder man kann eine Beschleunigung des Sortierens durch paralleles Vorgehen bewirken. Wir stellen hierzu zwei „einfache“ Verfahren vor. (Weitere Verfahren finden Sie z.B. in dem Buch von Ingo Wegener, "Effiziente Algorithmen für grundlegende Funktionen", Teubner-Verlag.)

Verfahren 1:

Lineare Kette mit n Prozessoren.

Verfahren 2:

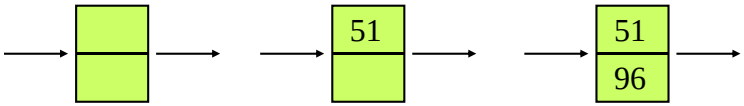
Divide and Conquer Verfahren mit $(1/4) \cdot n \cdot \log^2(n)$ Prozessoren.

10.7.1 Verfahren 1: Lineare Kette

Hierzu betrachten wir einen Prozessor, der

zwei Speicherplätze für Zahlen,
einen Eingang (links) und
einen Ausgang (rechts)

besitzt:



Grundbaustein, leer | Grundbaustein mit einer Zahl | Grundbaustein mit zwei Zahlen

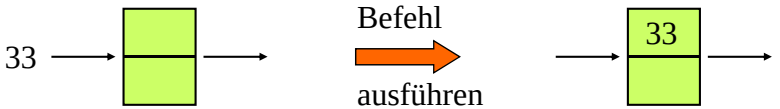
(Die Zahl darf auch unten stehen.)

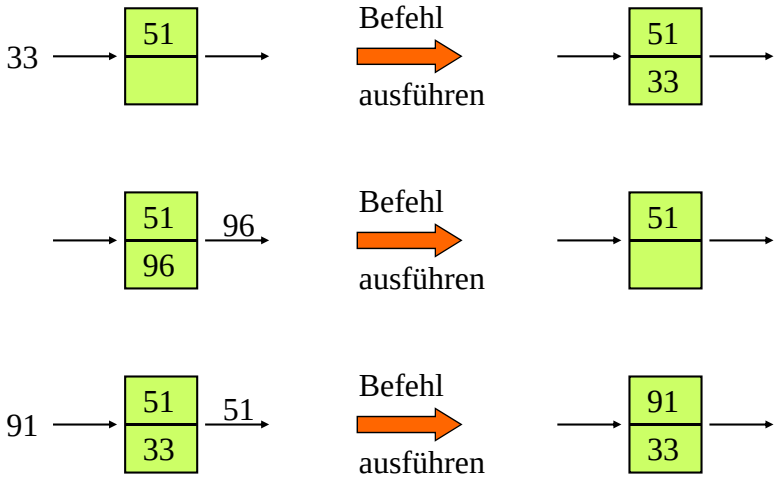
Der Grundbaustein kann nur folgenden Befehl ausführen:

1. Falls er zwei Zahlen besitzt, gibt er die größere der beiden nach rechts aus.
2. Falls eine Zahl von links kommt, legt er sie in einen seiner freien Speicherplätze.

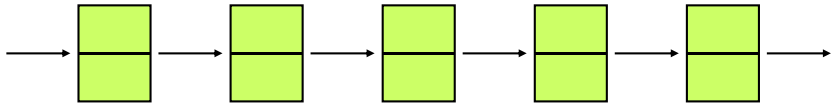
Die Befehlsteile 1. und 2. werden gleichzeitig ausgeführt!

Wenn die Bedingung in 1. und/oder in 2. zutrifft, dann muss der Befehl auch ausgeführt werden.

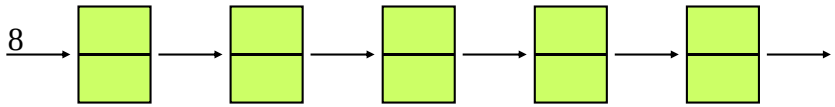


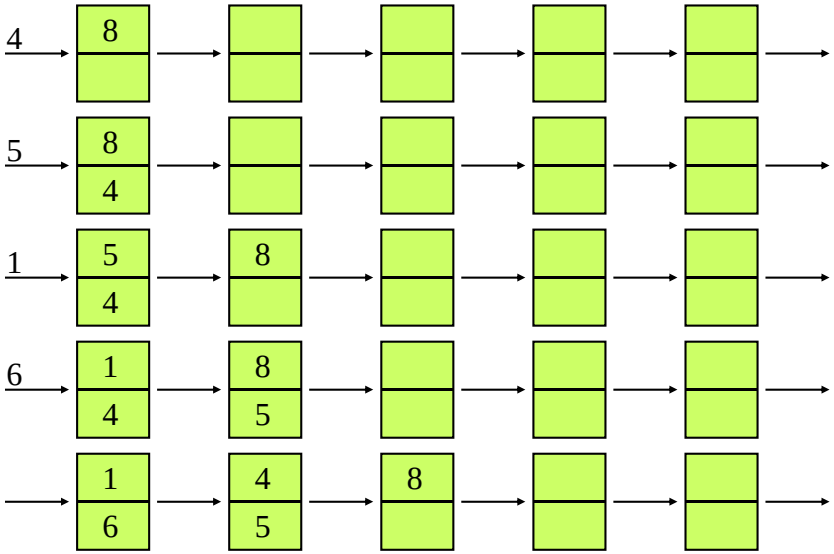


Nun koppeln wir n=5 solche Grundbausteine zu einer Kette aneinander:

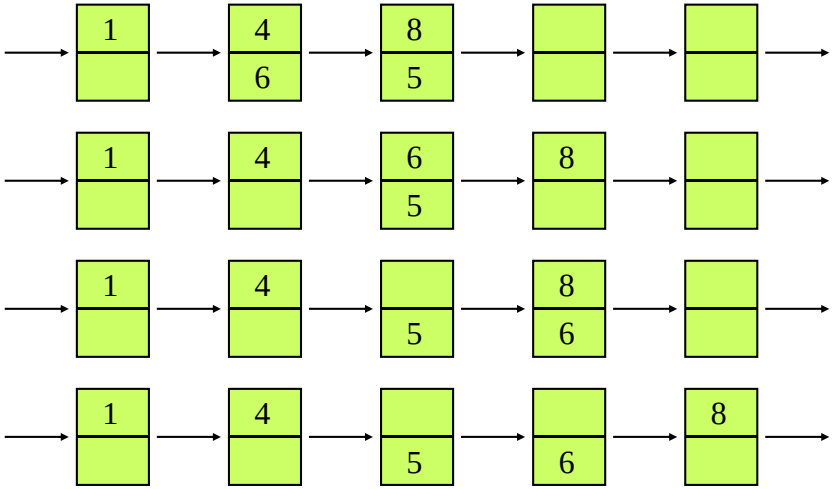


Wir verlangen, dass alle n Bausteine synchron arbeiten. Wir geben in n=5 Takten die Zahlen 8, 4, 5, 1, 6 von links ein:





Die Eingabe ist beendet. Die Kette arbeitet aber noch weiter, bis in jedem Baustein genau eine Zahl steht.



Nach 2n-1 Takten endet das Verfahren. Die Folge ist sortiert. Im 2n-ten Takt kann jeder Baustein seine Zahl (über einen weiteren Ausgang) ausgeben.

10.7.2 Wie viele Schritte benötigt dieses Verfahren?

Offensichtlich sind n Werte nach **2n-1** Schritten sortiert.

Hierfür benötigt man n Prozessoren.
Für große Werte von n ist dies eine deutliche Verbesserung gegenüber den bisherigen O(n·log(n))-Verfahren.

Hinweis: In 10.5.7 haben wir ein Verfahren skizziert, wie man n Werte schon mit n/2 Prozessoren parallel in 2n-2 Schritten sortieren kann. Jene Prozessoren hatten aber zwei Eingänge und waren komplizierter verschaltet.

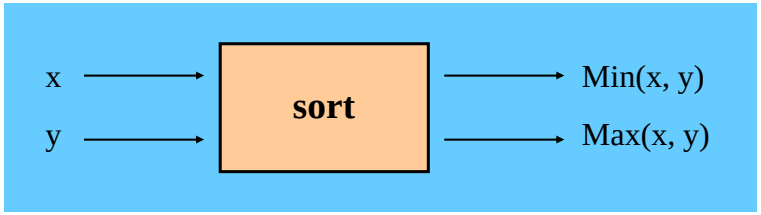
Gibt es bessere Verfahren? Möglichst solche, die viel schneller als in O(n) Schritten arbeiten - dafür dürfen sie dann auch mehr als O(n) Prozessoren besitzen ...

10.7.3 Verfahren 2: Divide and Conquer Ansatz

Der Divide-and-Conquer-Ansatz lautet:
Zerlege das Problem in z.B. zwei Teilprobleme, löse diese und setze aus den Teillösungen die Gesamtlösung zusammen.
(Quicksort und Mergesort sind typische Divide-and-Conquer-Verfahren.)

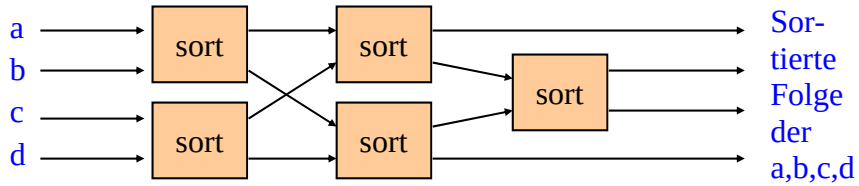
Wir gehen nun von einer allgemeinen Struktur aus, bei der die zu sortierenden Elemente nicht nacheinander, sondern parallel zueinander eingegeben werden. Dies bedeutet: wir müssen mit Ω(n) Grundbausteinen rechnen, um n Elemente zu sortieren.

10.7.4 Als Grundbaustein verwenden wir einen elementaren Sortierbaustein **sort** für zwei Werte:

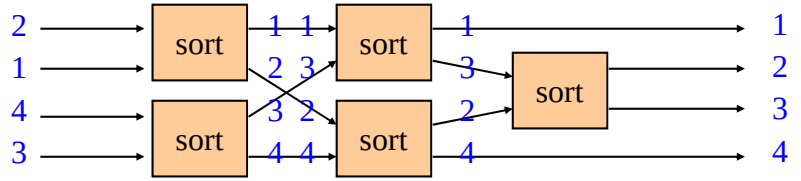


Wir werden nur diesen einen Grundbaustein, der zwei Werte sortiert, verwenden, um ein Netzwerk aufzubauen, mit dem wir riesige Mengen von Elementen in sehr kurzer Zeit sortieren können.

Hiermit kann man beispielsweise n = 4 Werte wie folgt sortieren:

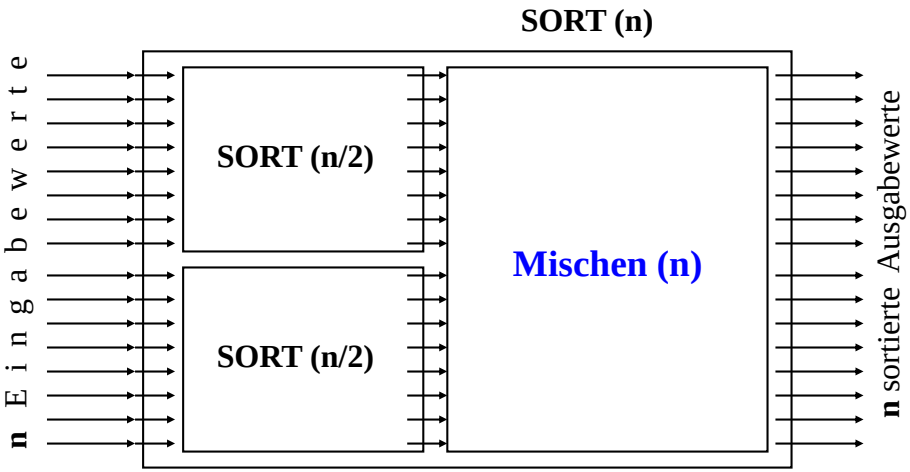


Beispiel:



4 Elemente werden in 3 Schritten mit 5 Bausteinen sortiert.

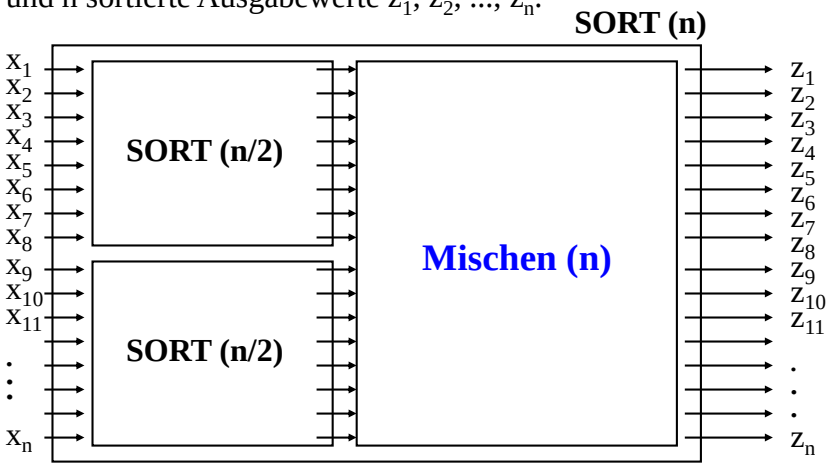
10.7.5 Divide and Conquer Ansatz für n Eingabewerte:



Genauer: Gegeben sind n Eingabewerte $x_1, x_2, ..., x_n$.
Wir erhalten 2 sortierte Folgen mit je m Elementen:

$a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ $(m = n/2)$

und n sortierte Ausgabewerte $z_1, z_2, ..., z_n$.



n Werte werden also sortiert, indem man annimmt, dass zweimal n/2 = m Werte bereits sortiert wurden, d.h., die Rekursion sorgt dafür, dass die Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ schon sortiert sind. Diese muss man "nur noch" zusammenmischen.
Wenn es einen "guten" Baustein **Mischen(n)** gibt, dann lässt sich hieraus auch ein "gutes" Sortierwerk für $n = 2^s$ Eingabewerte, für jede natürliche Zahl s, konstruieren.

In der Tat gibt es mehrere Techniken, um zwei sortierte Folgen *parallel* in relativ wenigen Schritten zu einer gemeinsamen sortierten Folge zu mischen.

Eine dieser Techniken nennt sich "odd-even-merge" (dtsch.: gerade-ungerad-Mischen); sie lässt sich ebenfalls rekursiv definieren. Diese Technik stellen wir nun vor.

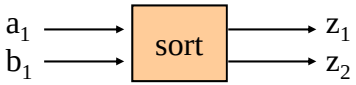
Definition 10.7.6: **odd-even-merge** "Mischen(2m)"
Gegeben: zwei sortierte Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ (für $m = 2^k$ für eine natürliche Zahl $k \geq 0$)
Ergebnis: die zugehörige sortierte Folge $z_1, z_2, ..., z_{2m}$.
Vorgehen zur Durchführung von "Mischen(2m)":
m = 1: Verwende den Baustein "**sort**"; fertig.
m > 1:
(1) Mische rekursiv die ungeraden Glieder der a- und der b-Folge sowie die geraden Glieder der a- und der b-Folge, d.h., mische die sortierten Folgen $a_1, a_3, a_5, ..., a_{m-1}$ und $b_1, b_3, b_5, ..., b_{m-1}$ zur sortierten Folge $u_1, u_2, u_3, ..., u_m$ und mische die sortierten Folgen $a_2, a_4, a_6, ..., a_m$ und $b_2, b_4, b_6, ..., b_m$ zur sortierten Folge $v_1, v_2, v_3, ..., v_m$.
(2) $z_1 = u_1$, $z_{2m} = v_m$ und führe parallel m-1 Vergleiche durch: $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ für i=1,2,..,m-1.

10.7.7 Erläuterung dieser Technik für m = 1, 2, 4.

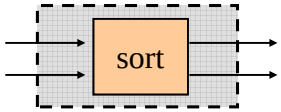
Wir konstruieren zunächst das Mischen für einige Werte von m.

Der Fall m=1 benötigt genau einen Sortierbaustein:

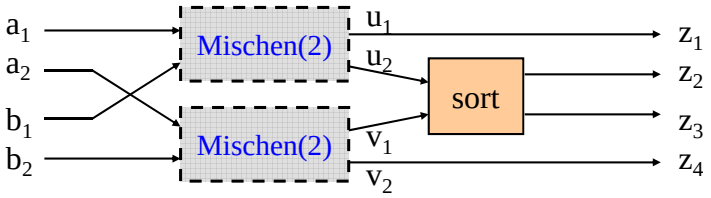
m = 1:



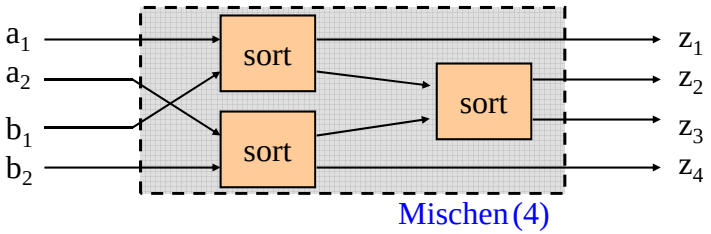
Dies ist der Baustein Mischen(2):



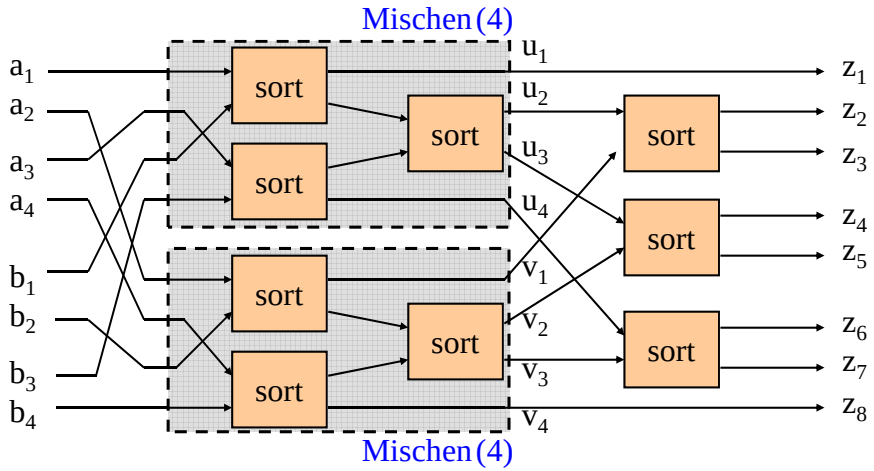
m = 2: (beachte, dass a_1, a_2 bzw. b_1, b_2 sortiert sind.)



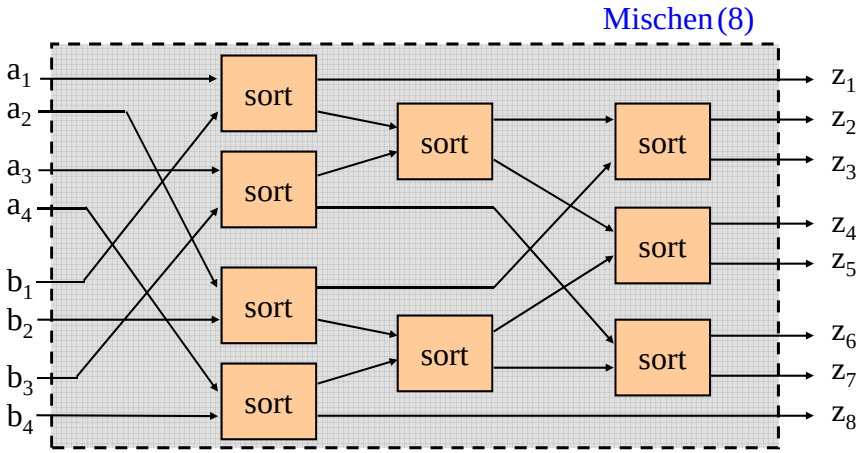
Einsetzen von Mischen(2) ergibt den Baustein Mischen(4):



m = 4:



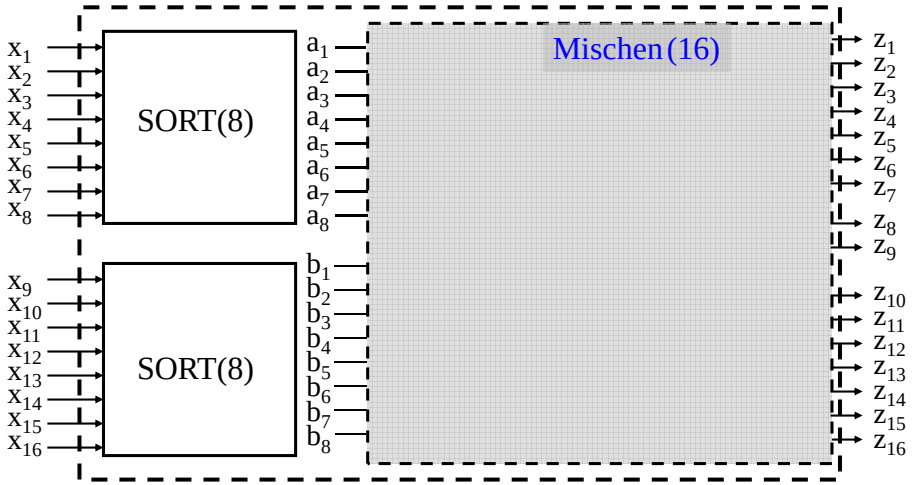
m = 4: Dies ergibt:



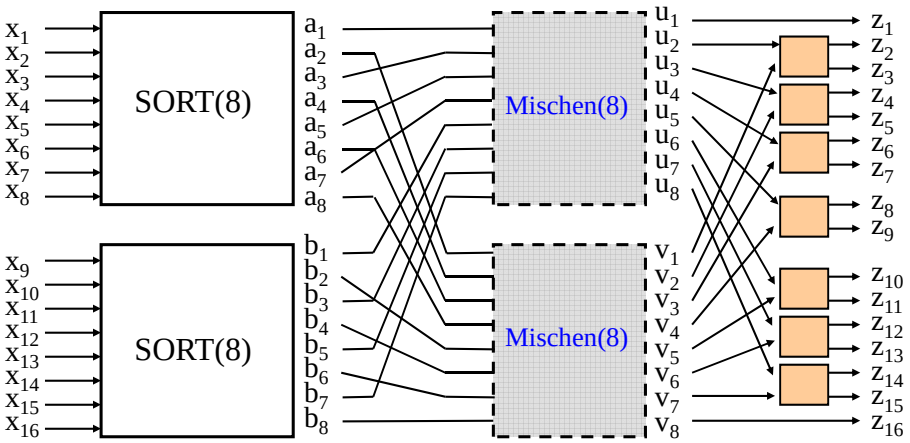
Beachte: a_1, a_2, a_3, a_4 bzw. b_1, b_2, b_3, b_4 sind sortierte Folgen.

10.7.8 Betrachte nun den Gesamtalgorithmus SORT für n = 16:

Rekursives Ersetzen für SORT(16) ergibt:

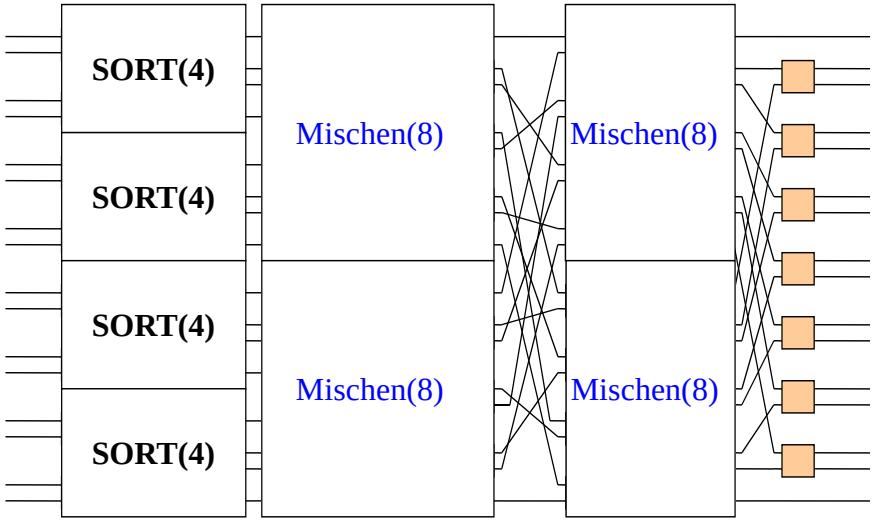


Einsetzen von Mischen(16) ergibt die Struktur:

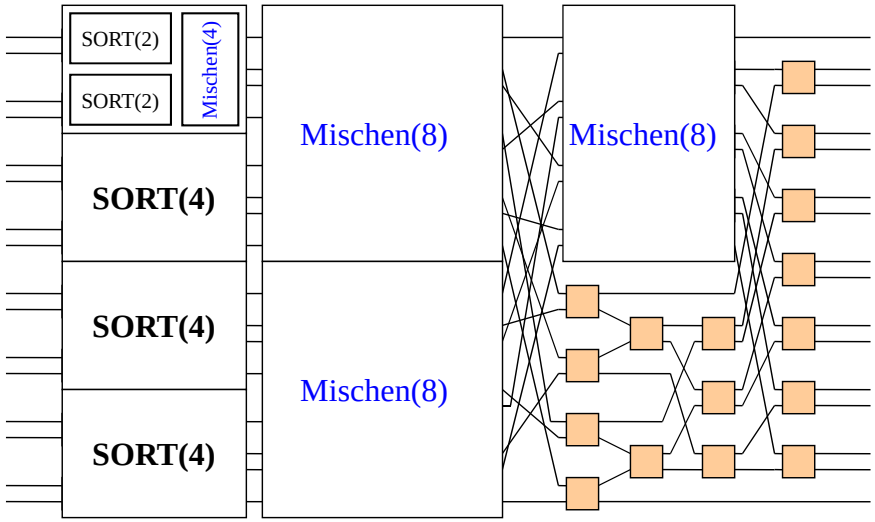


Setzt man alle kleineren schon konstruierten Bausteine ein, so erhält man den ausführbaren Algorithmus (selbst durchführen, vgl. dann letzte Folie in Kap. 10).

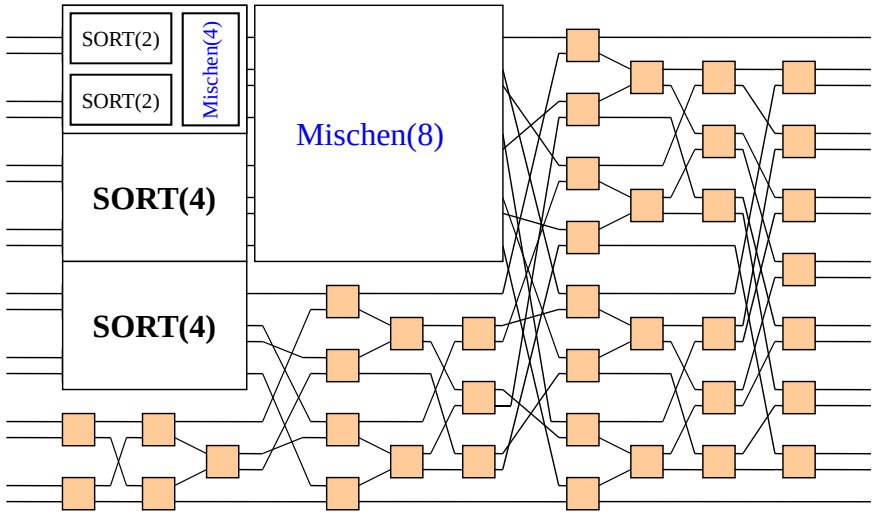
Weitere rekursive Ersetzung für SORT(16) und Weglassen der Ein-/Ausgangsbezeichnungen:



SORT(16)

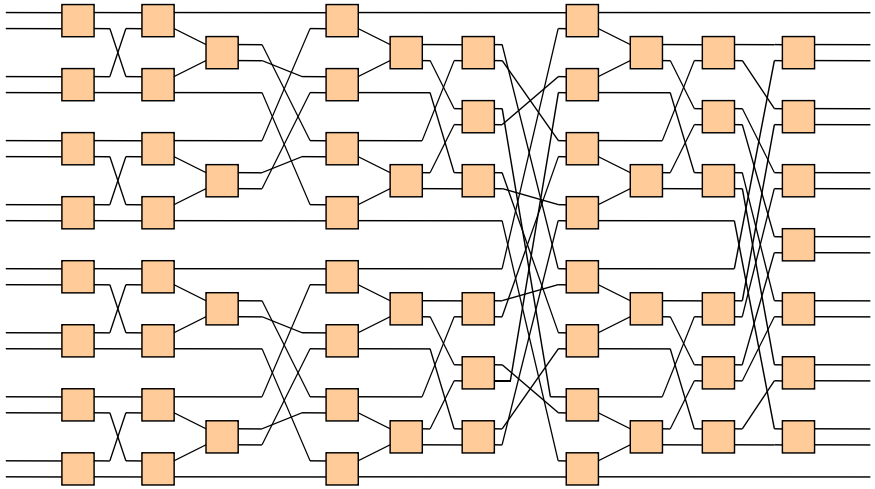


SORT(16)



Ergebnis: 63 Bausteine **sort**,
größte Tiefe: 10, Breite: 8.

SORT(16)



Korrektheit des Verfahrens: Nachdem der Aufbau des parallelen Sortieralgorithmus klar ist, müssen wir beweisen, dass die Technik "odd-even-merge" die beiden geordneten Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ tatsächlich zu der *geordneten* Folge $z_1, z_2, ..., z_{2m-1}, z_{2m}$ zusammenmischt.

Satz 10.7.9:

odd-even-merge sortiert die geordneten Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ zur geordneten Folge $z_1, z_2, ..., z_{2m-1}, z_{2m}$.

Beweis:
Wenn m=1 ist, dann werden zwei Zahlen durch den Sortierbaustein **sort** geordnet und der Satz ist richtig.

Sei daher m>1. Gegeben sind zwei sortierte Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$. Nach Definition des odd-even-merge werden die sortierten Teilfolgen mit jeweils m/2 Elementen $a_1, a_3, a_5, ..., a_{m-1}$ und $b_1, b_3, b_5, ..., b_{m-1}$ zur sortierten Folge $u_1, u_2, u_3, ..., u_m$ sowie $a_2, a_4, a_6, ..., a_m$ und $b_2, b_4, b_6, ..., b_m$ zur sortierten Folge $v_1, v_2, v_3, ..., v_m$ rekursiv gemischt.

Die Ergebnisfolge ist definiert durch $z_1 = u_1$, $z_{2m} = v_m$ und $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ für i=1,2,..,m-1. Wir zeigen, dass die z-Folge hierdurch korrekt sortiert ist.

u_1 muss das Minimum der beiden Folgen sein, da a_1 und b_1 die Minima der a- bzw. b-Folge und beide Elemente in der u-Folge sind. Analog gilt, dass v_m das Maximum der Ergebnisfolge sein muss. Also sind z_1 und z_{2m} richtig bestimmt worden.

Um zu zeigen, dass $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ richtig festgelegt wurden, genügt es zu zeigen, dass sich das Element u_{i+1} in der sortierten Ergebnisfolge an der Position 2i oder 2i+1 befinden muss (analog muss man dies für das Element v_i nachweisen). Wir beweisen dies in acht Schritten (a) bis (h).

Wir betrachten u_{i+1} für ein i zwischen 1 und m-1. O.B.d.A. nehmen wir an, dass dieses Element aus der a-Folge stammt und das j-te Element der Teilfolge mit den ungeraden Indizes ist, d.h., es gibt ein j mit $1 \le j \le m/2$ mit $u_{i+1} = a_{2j-1}$.

(a) Wie viele der Elemente der a-Folge sind kleiner als u_{i+1}?
Genau 2j-2 Elemente;
denn weil $u_{i+1} = a_{2j-1}$ ist, müssen die Elemente $a_1, a_2, ..., a_{2j-2}$ der geordneten a-Folge kleiner als u_{i+1} sein.

(b) Wie viele der Elemente der b-Folge sind kleiner als u_{i+1}?
Mindestens $2i-2j+1$ Elemente. Beweis hierfür:
Wegen $u_{i+1} = a_{2j-1}$ und wegen $a_1 \le a_3 \le a_5 \le ... \le a_{2j-3}$ müssen sich unter den ersten i Elementen $u_1, u_2, ..., u_i$ der u-Folge genau diese (j-1) Elemente aus der a-Folge befinden. Folglich müssen unter diesen ersten i Elementen der u-Folge genau i-(j-1) = i-j+1 Elemente der b-Folge sein. Da in der u-Folge aber nur die Elemente mit ungeradem Index sind, müssen dies genau die Elemente $b_1, b_3, b_5, ..., b_{2(i-j+1)-1}$ sein. Da die b-Folge sortiert ist, müssen daher mindestens die Elemente $b_1, b_2, b_3, ..., b_{2(i-j+1)-1}$ kleiner als u_{i+1} sein. Ihre Anzahl ist 2i-2j+1.

(c) Folglich stehen in der sortierten Ergebnisfolge mindestens
2j-2 + 2i-2j+1 = $2i-1$ Elemente vor u_{i+1}, d.h., in der Ergebnisfolge kann u_{i+1} frühestens das Element z_{2i} sein.

(d) Wie viele der Elemente der a-Folge sind größer als u_{i+1}?
Genau $m-2j+1$ Elemente. Grund:
wegen $u_{i+1} = a_{2j-1}$ müssen $a_{2j}, a_{2j+1}, ..., a_m$ größer als u_{i+1} sein.

(e) Wie viele der Elemente der b-Folge sind größer als u_{i+1}?
Mindestens $m-2i+2j-2$ Elemente. Beweis hierfür:
Wegen (b) muss $u_{i+1} \le b_{2(i-j+1)+1}$ sein; denn sonst wäre u_{i+1} nicht das (i+1)-te, sondern ein späteres Element der u-Folge. Also müssen mindestens die Elemente $b_{2(i-j+1)+1}, b_{2(i-j+1)+2}, ..., b_m$ größer als u_{i+1} sein. Ihre Anzahl ist m - (2i-2j+3) + 1 = m-2i+2j-2.

(f) Folglich stehen in der sortierten Ergebnisfolge mindestens
m-2j+1 + m-2i+2j-2 = $2m-2i-1$ Elemente hinter u_{i+1}, d.h., in der sortierten Ergebnisfolge muss u_{i+1} spätestens das Element $z_{2m-(2m-2i-1)} = z_{2i+1}$ sein.

(g) Nun führe man genau die gleiche Untersuchung für v_i durch. Auch hier erhält man, dass v_i frühestens das Element z_{2i} und spätestens das Element z_{2i+1} in der Ergebnisfolge sein kann.
(Führen Sie diesen Beweis selbst analog zu (a) bis (f).)

(h) Aus (f) und (g) folgt nun, dass die Elemente u_{i+1} und v_i sich an den Positionen 2i und 2i+1 der z-Folge befinden müssen. Daher muss $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ für die sortierte Ergebnisfolge $z_1, z_2, ..., z_{2m}$ gelten.

Damit ist der Satz 10.7.9 bewiesen.

Skizze zum Beweis: O.B.d.A. sei $u_{i+1} = a_{2j-1}$.

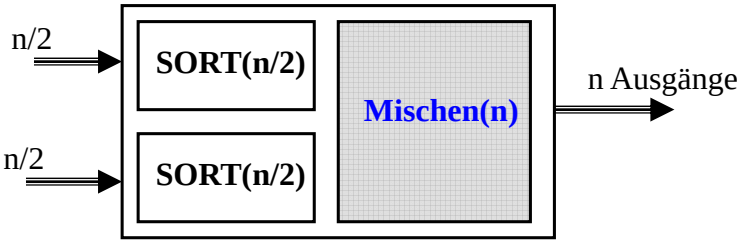
Es sind kleiner als u_{i+1}		Es sind größer als u_{i+1}
$a_1 \le a_2 \le ... \le a_{2j-2}$		$a_{2j} \le a_{2j+1} \le ... \le a_m$
$b_1 \le b_2 \le ... \le b_{2(i-j+1)-1}$	$\le u_{i+1} \le$	$b_{2(i-j+1)+1} \le b_{2(i-j+1)+2} \le ... \le b_m$

$z_1, z_2, ..., z_{2i-2}, z_{2i-1}$ z_{2i} z_{2i+1} $z_{2i+2}, z_{2i+3}, ..., z_{2m}$

[Wenn $u_{i+1} = a_{2j-1}$ ist, so ist $v_i = b_{2(i-j+1)}$. Beide müssen an den Positionen 2i oder 2i+1 in der sortierten z-Ergebnisfolge stehen. Es ist nur noch zu prüfen, ob a_{2j-1} kleiner oder größer als $b_{2(i-j+1)}$ ist. Folglich gilt $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$.]

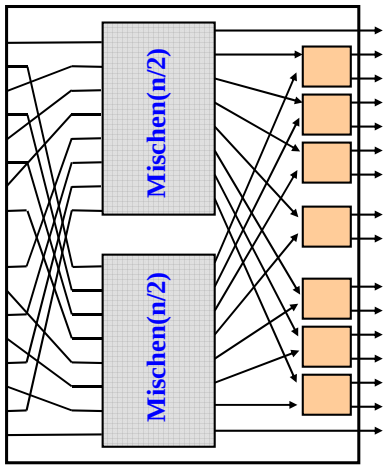
10.7.10 Vorüberlegung zur Komplexitätsabschätzung

Wie viele sort-Bausteine besitzt SORT(n)? Und wie lang ist der längste Weg in SORT(n) von einem Eingang zu einem Ausgang?
Zunächst die rekursive Struktur von SORT(n):



Sei G(n) = Zahl der Bausteine **sort** in SORT(n). Dann gilt:
G(2) = 1 und für $n = 2^k$, $k > 1$:
G(n) = 2·G(n/2) + M(n),
wobei M(n) die Anzahl **sort** in **Mischen(n)** ist.

Sodann die rekursive Struktur von Mischen(n):



Mischen(n)

Sei M(n) = Zahl der Bausteine **sort** in Mischen(n). Dann gilt:
M(2) = 1 und für n > 2: M(n) = 2·M(n/2) + n/2-1.

Es entstehen hier also sehr einfache rekursive Gleichungen für die Anzahl der sort-Bausteine.

Blättern Sie nicht weiter, sondern rechnen Sie die Funktionen M(n) und G(n) aus.

G(n) ist ein Maß für Kosten, um eine Hardware-Realisierung dieses Sortier-Netzwerkes zu bauen. Allerdings darf man die Vernetzungskosten nicht unterschätzen, da die Länge einiger Verbindungen mit n wächst und da Überkreuzungen teuer sind.

Die Geschwindigkeit hängt von der Tiefe ab, also vom längsten Weg von einem Eingang zu einem Ausgang im Netzwerk. *Definieren Sie diese Funktion "Tiefe" und berechnen Sie sie.*

Genau zu diesen Fragen finden Sie die Lösungen auf den folgenden Folien.

10.7.11 Zur Komplexität (Größe, Tiefe, Breite):

Detaillierte Analyse der Frage: "Wie groß, wie breit und wie tief ist dieser parallele Sortieralgorithmus?"

"Größe" soll ungefähr ein Maß für die technischen "Herstellungskosten" sein:
G(n) = Anzahl der Bausteine **sort** in **SORT(n)**.

"Tiefe" soll die Laufzeit des Sortierens angeben:
T(n) = Länge des längsten gerichteten Weges durch Bausteine **sort** in **SORT(n)**.

"Breite" soll die Anzahl der Mikroprozessoren angeben, die für eine Softwarelösung benötigt werden:
B(n) = Minimale Zahl der Bausteine **sort**, die parallel zueinander liegen müssen, ohne die Tiefe zu verändern.

Wir gehen nun von 10.7.10 aus:

Anzahl G(n) der Bausteine **sort**: G(2) = 1 und für $n=2^k$, k>1:
G(n) = 2·G(n/2) + M(n),
wobei M(n) die Anzahl **sort** in Mischen(n) ist:

M(2) = 1 und für $n=2^k$, k>1: M(n) = 2·M(n/2) + n/2-1.

Für die Tiefe gilt: T(n) = T(n/2) + "Tiefe von Mischen(n)".

Die Breite beträgt B(n) = n/2. Denn von jedem Eingang x_i muss ein Weg ausgehen und die Anzahl der Bausteine, die direkt hinter den Eingängen liegen, beträgt bereits n/2. Im weiteren Verlauf der Rekursion kommt man mit dieser Zahl auch aus, siehe die Formeln für SORT(n) und für Mischen(n). Dies lässt sich auch formal beweisen, worauf wir hier verzichten.

Einige Werte: M(2) = 1, M(4) = 3, M(8) = 9, M(32) = 65.
Aus M(n) = 2·M(n/2) + n/2 - 1 erhält man durch Einsetzen:

$$\begin{aligned}
M(n) &= 2\cdot M(n/2) + n/2-1\\
&= 2\cdot(2\cdot M(n/4) + n/4-1) + n/2-1\\
&= 4\cdot M(n/4) + 2\cdot n/2 - 1 - 2\\
&= 4\cdot(2\cdot M(n/8) + n/8 -1) + 2\cdot n/2 - 1 - 2\\
&= 8\cdot M(n/8) + 3\cdot n/2 - 1 - 2 - 4\\
&= \ldots\\
&= 2^{\log(n)-1}\cdot M(2) + (\log(n)-1)\cdot n/2 - (2^{\log(n)-1} - 1)\\
&= n/2 + n/2\cdot\log(n) - n/2 - n/2 + 1\\
&= n/2 \cdot (\log(n) - 1) + 1
\end{aligned}$$

Es gilt also $M(n) = n/2 \cdot (\log(n) - 1) + 1$.
(Vgl. die ähnliche Gleichung in 10.5.4 für V(n).)

Hiermit können wir nun die "Größe" G(n) ausrechnen:

$$\begin{aligned}
G(n) &= 2\cdot G(n/2) + M(n)\\
&= 2\cdot G(n/2) + n/2 \cdot (\log(n) - 1) + 1\\
&= 2\cdot(2\cdot G(n/4) + n/4 \cdot (\log(n/2) - 1) + 1) + n/2 \cdot (\log(n) - 1) + 1\\
&= 4\cdot G(n/4) + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) +1 +2\\
&= 4\cdot(2\cdot G(n/8) + n/8 \cdot (\log(n/4) - 1) + 1)\\
&\qquad + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) +1 +2\\
&= 8\cdot G(n/8) + n/2 \cdot (\log(n) - 3)\\
&\qquad + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) +1 +2 + 4\\
&= \ldots\\
&= 2^{\log(n)-1}\cdot G(2) + n/2\cdot(1+2+\ldots+(\log(n) - 1)) +1+2+\ldots+2^{\log(n)-2}\\
&= 2^{\log(n)-1} + n/4\cdot\log(n)\cdot(\log(n)-1)) + 2^{\log(n)-1} - 1\\
&= (n/4)\cdot\log(n)\cdot(\log(n)-1)) + n - 1.
\end{aligned}$$

Ergebnis: **$G(n) = (n/4)\cdot\log(n)\cdot(\log(n)-1)) + n - 1 \in O(n\cdot\log^2(n))$.**

Entscheidend für den praktischen Einsatz ist die Tiefe T(n).
Hierzu betrachten wir zunächst Mischen(n) in 10.7.10:
Die Tiefe von Mischen(n) ist die Tiefe von Mischen(n/2) + 1,
woraus sofort die Tiefe log(n) für den Teil Mischen(n) folgt.

Aus T(n) = T(n/2) + "Tiefe von Mischen(n)" folgt dann:

$$\begin{aligned}
T(n) &= T(n/2) + \log(n)\\
&= T(n/4) + \log(n/2) + \log(n) = T(n/4) + \log(n)-1 + \log(n)\\
&= T(n/8) + \log(n/4) + \log(n/2) + \log(n)\\
&= \ldots\\
&= T(2) + \log(n/2^{\log(n)-2}) + \log(n/2^{\log(n)-3}) + \ldots + \log(n)-1 + \log(n)\\
&= 1 + 2 + 3 + \ldots + \log(n)\\
&= (1/2)\cdot\log(n)\cdot(\log(n) + 1).
\end{aligned}$$

Ergebnis: **$T(n) = (1/2)\cdot\log(n)\cdot(\log(n) + 1) \in O(\log^2(n))$.**

10.7.12 Resultat:

G(n) = (n/4)·log(n)·(log(n)-1)) + n - 1

T(n) = (1/2) · log(n) · (log(n) + 1)

Faustformel:
G(n) ≈ n/2 · T(n).

n Eingänge	G(n) Größe	T(n) Tiefe	
2	1	1	
4	5	3	
8	19	6	
16	63	10	← siehe SORT(16) in 10.7.8
32	191	15	
64	543	21	
128	1471	28	
256	3839	36	
512	9727	45	
1024	24063	55	

n Eingänge	G(n) Größe	T(n) Tiefe
16	63	10
128	1471	28
1024	24063	55
16384	1490957	105

Mit 24.063 Bausteinen **sort** kann man also in 55 Schritten 1024 Zahlen sortieren.

Mit etwa 100 Millionen Bausteinen **sort** kann man in nur 210 Schritten rund 1 Million Zahlen sortieren.

Solche Algorithmen sind technisch durchaus realisierbar. Sie können in Zukunft die Leistungsfähigkeit beim Sortieren deutlich steigern.

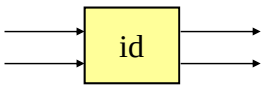
10.7.13 Durchdenken Sie eine mögliche Realisierung weiter, z.B.:

- Gibt es Probleme bei der Hardware-Realisierung? Lassen sich die vielen Leitungen problemlos verschalten / auf Platten drucken? ... Jede Leitung in unserer Skizze besitzt eine "Breite", z.B. 64 Bits zuzüglich Kontrollbits.
- Wie kann man den Sortierbaustein **sort** realisieren? Nehmen Sie an, dass die zu sortierenden Schlüssel k-stellige 0-1-Folgen sind, wobei man wegen der vielfältig möglichen Schlüssel von k = 64 ausgehen sollte. Was ist dann die minimale Tiefe (= längster Weg über Gatter von einem Eingang zu einem Ausgang) von **sort**?
- Wie kann man Millionen von Daten *gleichzeitig* an alle Eingänge legen? Welche Strukturen müssen die Speicher hierfür besitzen?

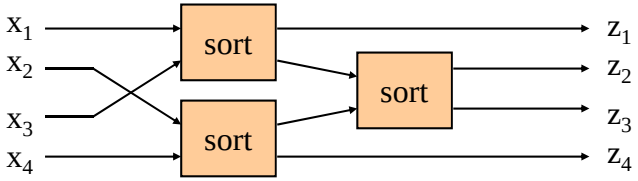
10.7.14 Wie sehen softwaremäßige Realisierungen aus?

Bei n Eingabewerten kann man n/2 Prozeduren definieren, die jeweils "den nächsten Gesamtschritt" (= alle parallel durchführbaren Sortierschritte) ausführen. Hierfür greifen sie immer wieder auf das Feld der zu sortierenden Daten x: array(1..n) of <Elementtyp> zu, aus dem in jedem Schritt gelesen und in das in jedem Schritt geschrieben wird.

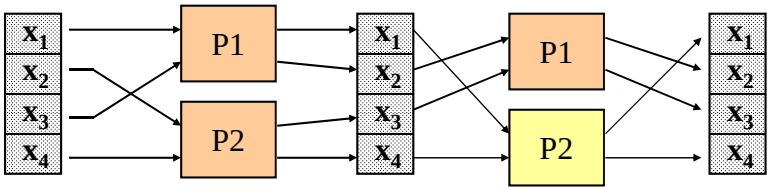
Im Algorithmus gibt es Teile, in denen ein Wert nur weitergereicht wird. Hierfür führen wir den zusätzlichen Baustein "id" (Identität) ein, der zwei Werte einliest und unverändert wieder ausgibt:

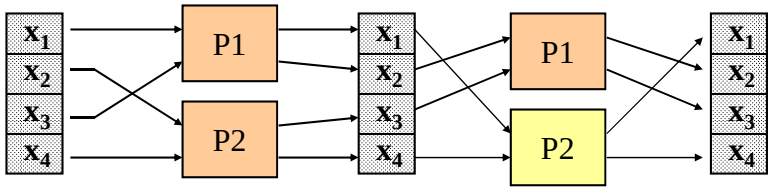


Der Mischbaustein Mischen (4) (siehe 10.7.7)



wird dann realisiert durch zwei Prozeduren P1 und P2:

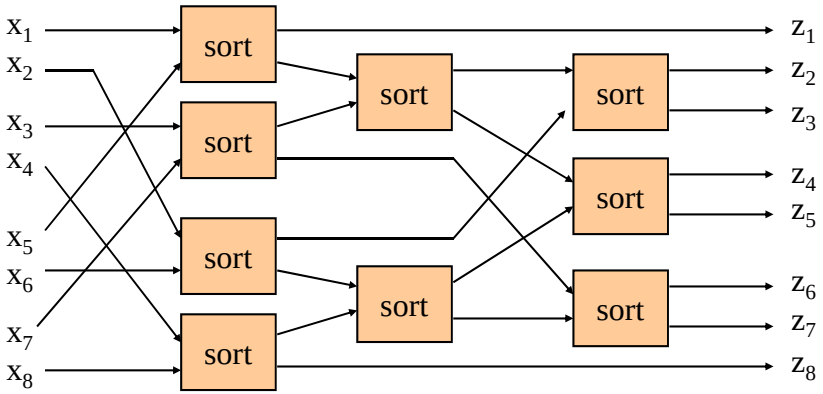




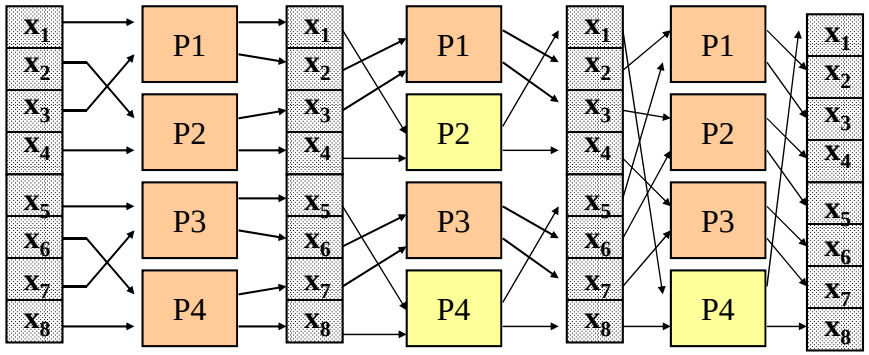

```
procedure P1 is                    -- x: array(1..n) of <Elementtyp> ist global
h1, h2: <Elementtyp>;
begin (h1, h2) := sort (x(1), x(3));  x(1) := h1; x(2) := h2; $
      (h1, h2) := sort (x(2), x(3));  x(2) := h1; x(3) := h2;
end;
procedure P2 is                    -- x: array(1..n) of <Elementtyp> ist global
h1, h2: <Elementtyp>;
begin (h1, h2) := sort (x(2), x(4));  x(3) := h1; x(4) := h2; $
      (h1, h2) := id (x(1), x(4));    x(1) := h1; x(4) := h2;
end;
```

$ steht für "Synchronisation"

Der Mischbaustein Mischen(8)



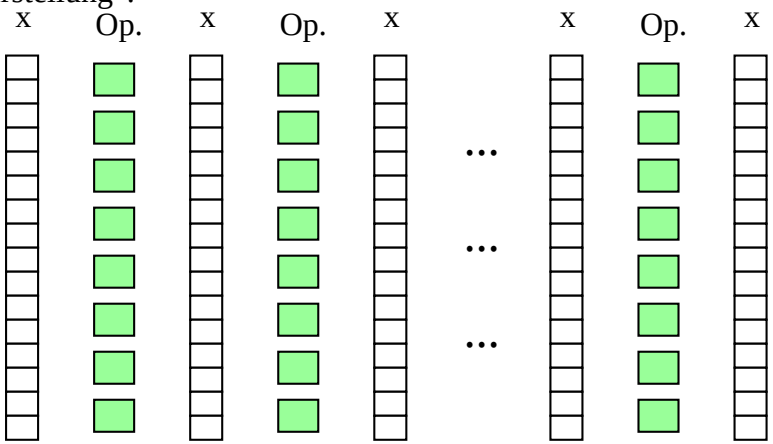
wird dann realisiert durch vier Prozeduren, die ihre Arbeitsweise synchronisieren müssen:




```
procedure Mischen8 is
    procedure P1 is begin ... end;
    procedure P2 is begin ... end;
    procedure P3 is begin ... end;
    procedure P4 is begin ... end;
begin  P1; P2; P3; P4; end;        -- die Prozeduren gleichzeitig starten
```

Die Synchronisation ist innerhalb der Prozeduren P1 bis P4 sicherzustellen. Jede Prozedur hat drei aufeinander folgende Programmstücke, durch die Synchronisation getrennt sind.

10.7.15: Man erhält auf diese Weise eine "normierte Darstellung":



x = Datenschicht, Op. = Operationsschicht. Jeder Operationsbaustein **sort** oder **id** hat zwei einlaufende und zwei ausgehende Verbindungen.

Jede Operationsschicht lässt sich durch n/2 Prozeduren softwaremäßig beschreiben; diese Prozeduren können wir ebenfalls für die nächste Operationsschicht verwenden usw., so dass wir aus Software-Sicht folgendes Ergebnis erhalten (die Operationen **id** kann man natürlich im Programm weglassen, nicht aber die Synchronisation):

Mit n/2 Prozeduren lassen sich n Elemente in
$T(n) = (1/2) \cdot \log(n) \cdot (\log(n) + 1)$
Schritten sortieren, wobei jede der n/2 Prozeduren aus einem sequentiellen Programmstück mit bis zu T(n) einzelnen Sortieroperationen **sort** besteht und T(n)-1 Synchronisationsanweisungen besitzt.

Diese Prozeduren können automatisch durch ein Programm erzeugt werden; sie hängen nur vom Parameter n ab.

Wie müssen Programmiersprachen beschaffen sein, damit der synchrone Ablauf und die Verzögerungen gesichert werden?

Diese Sprachen müssen in der Regel mehrere Anforderungen erfüllen:

- Teile eines Programms müssen unabhängig voneinander ablaufen können (Nebenläufigkeit, "tasks").
- Es muss Synchronisationsmechanismen geben.
- Man muss Zeitvorgaben (innerhalb von ... Mikrosekunden führe durch ....) machen können und es muss Verzögerungs-elemente (delay) geben.
- Es muss der parallele Zugriff auf Daten möglich sein.
- Es muss ...

Siehe weiteres Studium, Praktika, Projekte .....

11. Graphalgorithmen

Graphen

Viele zu verarbeitende Informationen besitzen eine Graph-Struktur: Sie bestehen aus Teilen (dies entspricht den Knoten), die untereinander in Beziehung stehen (dies entspricht den Kanten), wobei die Beziehungen gewichtet sind oder Kommunikation ermöglichen und die Informationen Attribute tragen oder wiederum aus Graphen bestehen (markierte Graphen, hierarchische Graphen).
Alle erforderlichen Definitionen über Graphen finden Sie in 3.7 und 3.8., sowie 8.2 und 8.8. Die Adjazenzdarstellung steht in 3.8.6; Graphdurchläufe (zunächst GD genannt, nun als Tiefen- und als Breitendurchlauf DFS bzw. BFS bezeichnet) finden Sie in 3.8.7 und 8.8.9. In 6.5.5 wird die transitive Hülle eines gerichteten Graphen in $O(n^3)$ Schritten berechnet.
Wir gehen davon aus, dass Sie die grundlegenden Begriffe bereits eingeübt haben und kennen.

Ziele des 11. Kapitels:

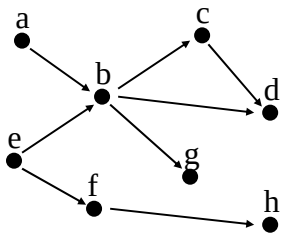
Gewisse Verfahren auf Graphen werden in den unterschiedlichsten Anwendungen gebraucht. Den Graph- und den Baumdurchlauf haben wir bereits kennen gelernt und ihn auf die Ermittlung der Zusammenhangskomponenten angewendet (DFS und BFS, inorder usw., Baumsortieren).

In diesem Kapitel lernen Sie weitere Verfahren, und zwar:
- ordne eine Halbordnung linear,
- finde den kürzesten Weg in einem kantenmarkierten Graphen,
- finde das minimale Gerüst in einem Graphen.

Sie sollen zugehörige Lösungsalgorithmen beschreiben und deren Komplexität herleiten können. Zugleich sollen Sie die Korrektheit dieser Verfahren begründen können.

11.1 Topologisches Sortieren

Topologisches Sortieren ist das Einbetten einer Halbordnung in eine totale (= lineare) Ordnung. Die Halbordnung ist durch einen azyklischen Graphen und seine transitive Hülle gegeben.



Angabe zweier Funktionen ord, vgl. Def. 8.8.2:

a	b	c	d	e	f	g	h
1	4	5	7	2	3	6	8

a	b	c	d	e	f	g	h
4	5	7	8	1	2	6	3

Es gibt noch weitere Anordnungen ord für dieses Beispiel.

Hilfssatz 11.1.1:

Es sei G ein DAG, dann gibt es mindestens einen Knoten mit Eingangsgrad 0 und mindestens einen Knoten mit Ausgangsgrad 0.

Diese Aussage ist "klar": Folge von irgendeinem Knoten den auslaufenden Kanten. Da man in einem azyklischen Graphen nicht wieder auf einen bereits besuchten Knoten stoßen darf, muss jede Kantenfolge nach spätestens n-1 Schritten in einem Knoten mit Ausgangsgrad 0 enden. Analog beim Rückwärts-Verfolgen von Kanten.

(DAG = gerichteter azyklischer Graph, siehe 3.8)

Satz 11.1.2: Ein gerichteter Graph besitzt genau dann eine topologische Sortierung, wenn er azyklisch ist.

"⇒": Ein Graph möge eine topologische Sortierung ord: $V \to \mathbb{N}$ mit $\forall\, u,v \in V$ mit $u \neq v$ gilt: $(u,v) \in E^* \Rightarrow ord(u) < ord(v)$ besitzen. Betrachte dann einen geschlossenen Weg in G: $(u_0, u_1, ..., u_{k-1}, u_0)$ mit $k \geq 0$. Wegen $(u_0, u_0) \in E^*$ muss dann $ord(u_0) < ord(u_0)$ gelten, Widerspruch.

"⇐": Setze $i = 1$. Ein gerichteter azyklischer Graph besitzt mindestens einen Knoten u mit dem Eingangsgrad 0. Setze $ord(u) = i$, erhöhe i um 1, entferne den Knoten u und die zu ihm inzidenten Kanten (es entsteht ein gerichteter azyklischer Graph G') und fahre rekursiv mit G' fort. So erhält jeder Knoten u aus G eine Nummer ord(u). Wenn nun $(u, v) \in E^*$ in G gilt, so kann der Eingangsgrad von v bei dieser Konstruktion erst dann 0 werden, wenn der Knoten u bereits entfernt ist. Dies heißt aber: ord(u) < ord(v).

11.1.3: Der Beweis liefert zugleich TopSort1 = einen *Algorithmus zur Berechnung einer topologischen Sortierung*:

```
i := 0;
for j in 1..n loop
      wähle einen Knoten u mit Eingangsgrad 0;
      i:=i+1;  setze ord(u) := i;
      entferne u aus dem Graphen;
end loop;
```

Man beachte, dass sich der Eingangsgrad der Knoten bei jedem Entfernen eines Knotens um 1 verringern kann.

Da es mehrere Knoten mit dem Eingangsgrad 0 geben kann und da jede Auswahl eines solchen Knotens zu einer topologischen Sortierung führt, sind topologische Sortierungen in der Regel nicht eindeutig.

Erinnerung: Wir übernehmen die Datenstrukturen für die Adjazenzdarstellung von Graphen aus Abschnitt 3.8.6:

```
type Knoten; type Kante;
type NextKnoten is access Knoten;
type NextKante is access Kante;
type Knoten is record
        Id: Knotenname;
        Besucht: Boolean; zahl1, zahl2: Integer;
        Inhalt: <weitere Komponenten>;
        NKn: NextKnoten;
        EIK: NextKante;
end record;
type Kante is record
        W: <Typ des Gewichts der Kanten>;
        EKn: NextKnoten;
        NKa: NextKante;
end record;
```


Leicht zu lösende Probleme bei diesem Algorithmus:
- Wie findet man am Anfang rasch einen Knoten mit dem Eingangsgrad 0? (siehe Programmstück unten)
- Wie aktualisiert man die Eingangsgrade beim Entfernen eines Knotens? (man muss dies nur für die Nachfolgerknoten tun)

```
p := Anfang;              -- Die Eingangsgrade werden in zahl1 gespeichert
while p /= null loop  p.zahl1 := 0; p := p.NKn; end loop;
p := Anfang;              -- Gehe alle Kanten durch und erhöhe den
while p /= null loop                -- Eingangsgrad des Zielknotens
    edge := p.EIK;
    while  edge /= null  loop
        edge.EKn.zahl1 := edge.EKn.zahl1 + 1;
        edge := edge.NKa;
    end loop;
    p := p.NKn;
end loop;
```

Dieses Programmstück ermittelt alle Eingangsgrade in O(n+m) Schritten.

Um schnell einen Knoten mit Eingangsgrad 0 finden zu können, speichern wir alle diese Knoten in einer Liste (oder in einem Feld) "*GradNull*". Initialisierung:

```
p := Anfang; "Setze GradNull auf die leere Liste";
while p /= null loop
      if p.zahl1 = 0 then "füge p an GradNull an" end if;
      p := p.NKn;  end loop;
```

Nun müssen wir noch die Eingangsgrade aktualisieren, sobald ein Knoten u aus dem Graphen entfernt wird.

Hierzu erniedrigen wir die Eingangsgrade aller Knoten, die über die Kantenliste von u erreichbar sind, um 1. Falls hierbei ein Eingangsgrad auf 0 absinkt, wird der entsprechende Knoten in *GradNull* aufgenommen. (u sei der Verweis auf den als nächstes zu entfernenden Knoten mit Eingangsgrad 0.) Dies liefert:


```
edge := u.EIK;
while  edge /= null  loop
    edge.EKn.zahl1 := edge.EKn.zahl1 - 1;
    if edge.EKn.zahl1 = 0
        then "füge edge.EKn an GradNull an" end if;
    edge := edge.NKa;
end loop;
"entferne den Knoten u aus dem Graphen G"
```

Die restlichen Details überlassen wir den Leser(inne)n. (Könnte z.B. zahl1 irrtümlich kleiner als 0 werden, so dass die Abfrage "if edge.EKn.zahl1 = 0 ..." übergangen wird?) Zur Datenstruktur: Entweder arbeitet man auf einer Kopie von G und trägt zusätzlich zahl1 im Original ein oder man verwendet einen Booleschen Wert, um zu simulieren, dass man den jeweiligen Knoten gelöscht hat.

11.1.4: Komplexität dieses Algorithmus TopSort1:

Zeitkomplexität: O(n+m).
Platzkomplexität O(n) (bei Verwendung eines Booleschen Wertes).

Begründung zur Zeitkomplexität:

Jeder Knoten erhält seinen Eingangsgrad: O(n+m).

Aufbau der Liste *GradNull*: O(n).

n mal: Einen Knoten mit Eingangsgrad 0 finden (= nimm stets den ersten Knoten in der Liste *GradNull*) und später entfernen, insgesamt: O(n).

m mal insgesamt: Eingangsgrade erniedrigen und Knoten mit Eingangsgrad 0 an *GradNull* anhängen: O(n+m).

Alle übrigen Operationen (i := i+1; setze ord(u) := i; usw.) erfordern insgesamt höchstens O(n) Schritte.

11.1.5: Wir betrachten folgenden Algorithmus TopSort2, der in O(n+m) Schritten arbeitet und im Wesentlichen gleich der Tiefensuche ist (n = Zahl der Knoten, m = Zahl der Kanten):

```
procedure TS2 (u: NextKnoten) is                -- nummer ist global
begin  u.Besucht := true;
      for alle Nachfolgerknoten v von u loop
            if not v.Besucht then TS2 (v); end if; end loop;
      u.zahl2 := nummer;  nummer := nummer - 1;
end;
-- Programmstück, welches TS2 aufruft:
for alle Knoten p loop  p.Besucht := false;  end loop;
nummer := n;
for alle Knoten p loop
          if not p.Besucht then TS2 (p); end if;  end loop;
```

Hinweis: Diese zahl2-Werte heißen in der Literatur auch "finish"-Werte.

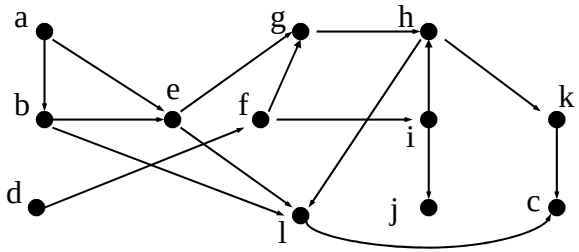
Dieser Algorithmus TopSort2 macht Folgendes:

Er beginnt in einem beliebigen Knoten und führt einen Tiefendurchlauf durch. Immer wenn er hierbei alle von einem Knoten ausgehenden Kanten abgearbeitet hat, ordnet er diesem Knoten die Zahl der Variablen "nummer" zu. nummer wird dann um 1 erniedrigt. Somit erhält jeder Knoten eine kleinere Nummer als alle seine Nachfolger. Da nummer mit n initialisiert wird, bekommt ein Knoten ohne Nachfolger den größten Wert.

Machen Sie sich klar, dass diese Nummern als ord-Werte verwendet werden können, da eine Ordnungsfunktion ord in einem DAG genau dann zu einer topologischen Sortierung gehört, wenn für jeden Knoten x gilt, dass ord(x) < ord(y) für alle Nachfolgerknoten y, d.h. für alle $(x,y) \in E$, ist.

Also berechnet der Algorithmus eine topologische Sortierung.

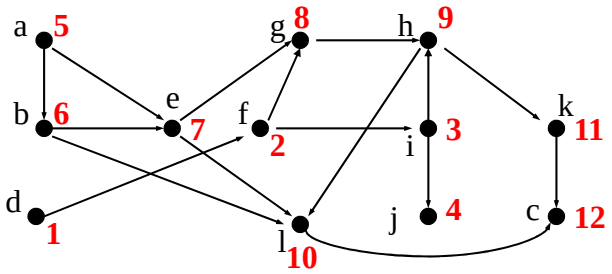
Beispiel: Azyklischer Graph mit n = 12 Knoten und m = 16 Kanten



Die Reihenfolge der Knoten in der Adjazenzliste sei k, e, h, a, f, j, i, d, c, b, g, l.

Dann liefert der obige Algorithmus TopSort2 folgende Zahlen für die Knotenkomponente *zahl2*, sofern man beim Nachfolger zuerst der Kante folgt, die bzgl. der Zahlen auf einer Uhr die kleinste ist.

Beispiel: Azyklischer Graph mit n = 12 Knoten und m = 16 Kanten



Knotenreihenfolge in der Adjazenzliste: k, e, h, a, f, j, i, d, c, b, g, l

Frage: Was liefert der Algorithmus TopSort2 für Werte bei beliebigen gerichteten Graphen?
Antwort: Man kann die Kanten dann in Kanten, die vom Algorithmus zu einem neuen Knoten durchlaufen werden (sog. Baum-Kanten), und andere Kanten unterteilen, wobei sich letztere wiederum in Rückwärts-, Vorwärts- und Quer-Kanten einteilen lassen. Siehe Vorlesungen über Graphen.

11.1.6: Vergleich der Algorithmen TopSort1 und TopSort2.

TopSort2 ist in der Praxis schneller als TopSort1.

Beide Verfahren benötigen zusätzlichen Speicherplatz: TopSort1 wegen der Liste *GradNull* und TopSort2 wegen der Rekursion.

TopSort1 legt eine *GradNull*-Liste an, die im Laufe des Verfahrens aktualisiert wird. Wenn *GradNull* leer wird, so kann man direkt ablesen, ob der Graph azyklisch war (nämlich genau dann wurde die j-Schleife bis zum Ende ausgeführt!). TopSort1 liefert also zugleich die Information, ob ein azyklischer Graph vorlag oder nicht.

TopSort2 liefert diese Information nicht unmittelbar; vielmehr arbeitet dieser Algorithmus für beliebige gerichtete Graphen G weiter und liefert eine topologische Sortierung für den Untergraphen, der aus G entsteht, indem man alle Rückwärtskanten löscht. Diese Rückwärtskanten hängen vom Durchlauf ab: Es sind Kanten, die zu einem Knoten führen, der im rekursiven Aufrufbaum oberhalb des aktuell betrachteten Knoten steht.

TopSort2 kann also in der hier vorgestellten Formulierung nur benutzt werden, wenn man schon weiß, dass der Graph azyklisch ist.

Wie muss man TopSort2 abändern, damit dieser Nachteil entfällt?

Ein Zyklus liegt genau dann vor, wenn der DFS-Durchlauf auf einen Knoten trifft, der im rekursiven Aufrufbaum ein Vorgängerknoten von sich selbst ist. Wir führen also für jeden Knoten eine weitere Boolesche Komponente "rekursion_aktiv" ein, die anfangs false ist. Wird ein Knoten v mittels DFS(v) aufgerufen, so wird v.rekursiv_aktiv := true gesetzt. Die Abfrage

```
if not v.Besucht then TS2(v); end if;
```

wird folgerichtig ersetzt durch

```
if not v.Besucht then v.rekursiv_aktiv := true; TS2(v);
elsif v.rekursiv_aktiv then "Abbruch, es liegt ein Zyklus vor"
end if;
```

Unmittelbar nach "u.zahl2 := nummer;" setzt man

```
u.rekursiv_aktiv := false;
```

Analog ergänzt man den Aufruf TSP2(p).

Aufgabe: Durchdenken Sie diese Idee. Stimmt sie wirklich oder gibt es Fälle, wo sie versagt?

11.2 Kürzeste Wege

Gegeben ist ein
gerichteter Graph $G = (V, E, \delta)$ mit $\delta: E \to \mathbb{R}^{\geq 0}$
(beachte: $\delta(e) \geq 0$ für alle Kanten e) und ein Knoten $u \in V$.

Gesucht werden alle kürzesten Entfernungen $\delta(u,v)$ für alle Knoten $v \in V$.

Wir wollen also SSSP lösen (single source shortest paths, siehe 8.8.5).

Der Standardalgorithmus zur Lösung des SSSP ist der Dijkstra-Algorithmus, der sich als Breitensuche (gewichtet bzgl. der Entfernungsfunktion δ) über den Graphen vorarbeitet. Seine worst-case-Zeitkomplexität bei Implementierung mit einem Heap liegt in $O((n+m)\cdot\log(n))$. Im Mittel läuft er sogar in $O(m + n\cdot\log(n))$.

Hinweis: Sucht man die kürzesten Abstände zwischen *allen* Knoten (all pair shortest paths, APSP), so kann man den Dijkstra-Algorithmus für jeden Knoten einmal durchführen oder man kann den Floyd-Algorithmus mit Zeitkomplexität $\Theta(n^3)$ verwenden. Der Floyd-Algorithmus ähnelt stark dem Warshall-Algorithmus in 6.5.5 und ist in Lehrbüchern gut beschrieben.

(Edgar W. Dijkstra und R. W. Floyd: holländischer bzw. amerikanischer Wissenschaftler, "Pioniere" der Informatik. Ihre Algorithmen wurden 1959 und 1962 veröffentlicht. <Aussprache dieser Namen: "Deik-stra" und "Fleut">)

11.2.1 Aufgabenstellung (SSSP)

SSSP = single source shortest paths
= alle kürzesten Wege, die von einem gegebenen Knoten zu jedem anderen Knoten führen.

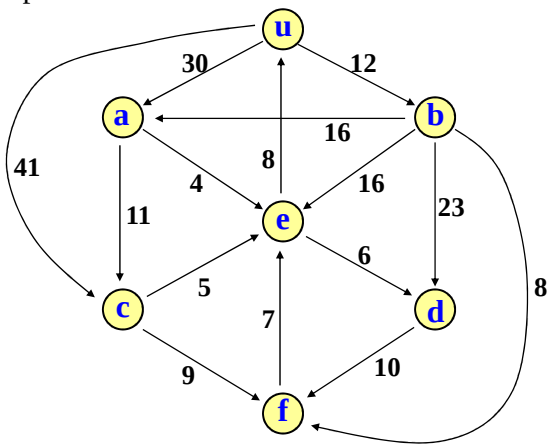
Gegeben ist ein gerichteter Kanten markierter Graph $G = (V, E, \delta)$ mit $\delta: E \to \mathbb{R}^{\geq 0}$ und ein "Startknoten" $u \in V$ (u ist "single source", also die einzige Quelle, von der ausgehend alle kürzesten Wege berechnet werden sollen; $\mathbb{R}^{\geq 0}$ ist die Menge der nichtnegativen reellen Zahlen).
Gesucht werden für jeden Knoten $v \in V$ ein kürzester Weg von u nach v und dessen Länge
$\delta(u,v)$ = Summe aller $\delta(e)$ für alle Kanten e, aus denen dieser Weg besteht (siehe 8.8.4).

Zuerst zu klären: Wie sollen diese Wege dargestellt werden? Würde man alle Wege tatsächlich hinschreiben, so benötigt man $O(n^2)$ Speicherplatz, da es n Knoten v gibt und jeder doppelpunktfreie Weg von u zu einem Knoten v bis zu n Knoten enthalten kann.

Vorschlag: Man speichert zu jedem Knoten $v \in V$ den Zeiger auf den Knoten, der auf einem kürzesten Weg von u nach v direkter Vorgänger von v ist. Hieraus kann man (rückwärts gehend) schrittweise einen kürzesten Weg von hinten nach vorne zu jedem Knoten aufbauen.
Der Graph aus der Knotenmenge V und diesen Zeigern, allerdings in umgedrehter Richtung, bilden einen Baum, den sog. "Kürzeste-Wege-Baum".

11.2.2 Beispiel

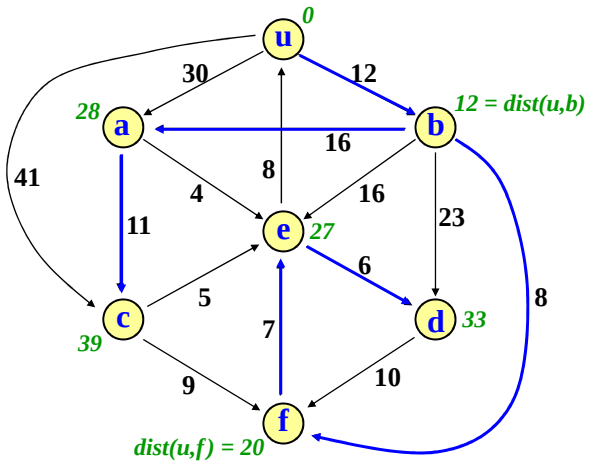
n=7, m= 15



Dieser Graph ist stark zusammenhängend. Startknoten sei u.
Wir konstruieren nun einen Teilgraphen mit kürzesten Wegen.

Beispiel:

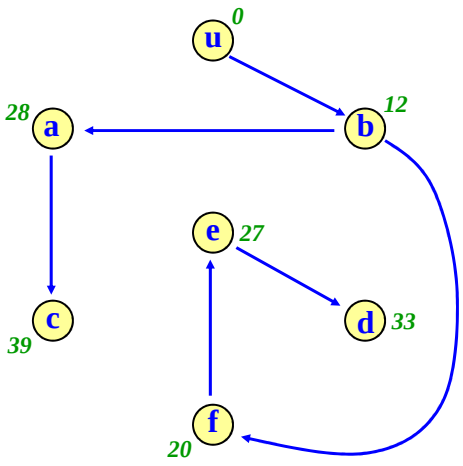
n=7, m= 15



Der Teilgraph mit den kürzesten Wegen bildet einen Baum!

Beispiel:
n=7, m= 15

Dies ist ein "Kürzeste-Wege-Baum" für den Knoten u

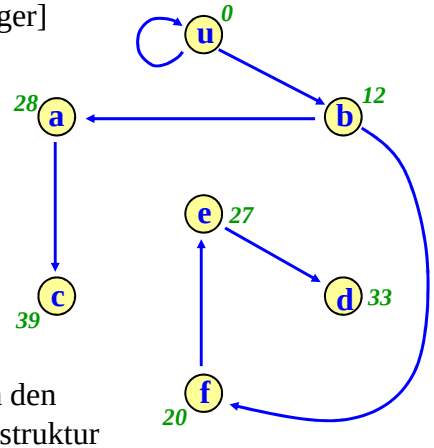


Also stellen wir kürzeste Wege bzgl. u wie folgt dar:

Schema für die Darstellung kürzester Wege:
(Knoten, Vorgängerknoten, Distanz von u)
[für u nehmen wir u als Vorgänger]

Hieraus kann man kürzeste Wege rückwärts rekonstruieren.

(u, u, 0),
(a, b, 28),
(b, u, 12),
(c, a, 39),
(d, e, 33),
(e, f, 27),
(f, b, 20).



Diese Werte schreibt man oft in den Graphen hinein, d.h., die Datenstruktur "Knoten" muss dann um entsprechende Komponenten erweitert werden.

11.2.3 Hinweis: Im Allgemeinen gibt es mehrere kürzeste Wege zwischen Knoten.

Wir interessieren uns hier nur für genau *einen* kürzesten Weg, der - wenn es mehrere gibt - beliebig ausgewählt werden darf. Unter dieser Annahme bilden die kürzesten Wege einen Baum. Grund: Wenn in diesem Teilgraphen zwei verschiedene Wege von einem Knoten x zu einem Knoten y führen würden, dann lägen mindestens zwei kürzeste Wege vor, von denen wir einen (durch Entfernen mindestens einer Kante) streichen. Entfernt man auf diese Weise alle doppelten Wege, so entsteht schließlich der genannte "Kürzeste-Wege-Baum" von u zu allen erreichbaren Knoten. (Dieser Baum ist i. A. nicht eindeutig, aber für jeden Startknoten u sind selbstverständlich die Entfernungen $\delta(u,v)$ zu allen Knoten v in jedem Kürzeste-Wege-Baum, der zu gehört, gleich.)

An obigem Beispiel sieht man die Vorgehensweise, um einen Kürzeste-Wege-Baum zu einem Knoten u zu konstruieren:

Annahme, man hat bereits zu einer Menge von Knoten B je einen kürzesten Weg mit Distanz $d(v) = \delta(u,v)$ für jedes $v \in B$ berechnet, dann wähle man als nächstes einen Knoten x ($x \notin B$), der die geringste Distanz von u besitzt, für den also gilt:
$d(v) + \delta((v,x)) \le d(v) + \delta((v,y))$ für alle Knoten $v \in B$, $y \notin B$.
Setze d(x) = "dieses Minimum $d(v) + \delta((v,x))$", füge x zu B hinzu und fahre rekursiv fort, bis alle Knoten betrachtet wurden.
Initialisierung: $B = \{u\}$ mit $d(u) = 0$.

Dieses Vorgehen wird als Dijkstra-Algorithmus bezeichnet. Problem hierbei: Finde einen solchen Knoten x möglichst schnell. Man speichert hierzu die als nächstes zu betrachtenden Knoten in einer "Prioritätswarteschlange". Dann gibt es drei Grundoperationen, die wir nun vorstellen.

11.2.4 Struktur des Dijkstra-Algorithmus

Grundlegende Mengen und Funktionen (blau gefärbt):

Gerichteter Graph G = (V, E, δ) mit δ: E→ $\mathbb{R}^{\geq 0}$ und Knoten u.
B = Menge der bereits abschließend "bearbeiteten" Knoten.
R = Menge der Knoten aus V-B, die von B aus über nur eine Kante erreichbar sind (sog. "Randknoten").
U = Menge der übrigen, bisher noch "unsichtbaren" Knoten.
Es gilt stets: V = B∪R∪U (und B, R, U sind paarweise disjunkt).
Initialisierung: B = {u}, R = {v | (u,v)∈E}, U = V-R-{u}.
D: V→ $\mathbb{R}^{\geq 0}$ mit D(v) = *vorläufiger Wert* für die kürzeste Entfernung von u zum Knoten v (stets ist D(v) ≥ δ(u,v)).
Aber: Für die Knoten v in B wird gelten: D(v) = δ(u,v).
Man stellt D als array (V) of real dar mit der Initialisierung D(v) = maximal darstellbare reelle Zahl, außer D(u) = 0.0.
Vorg: array (V) of (V∪{null}) gibt zu jedem Knoten den Vorgänger auf einem kürzesten Weg an. Initialisierung mit null.

In jedem Schritt wird ein Knoten x aus R mit seinem D-Wert in die Menge B aufgenommen, die Menge R wird um alle Knoten v aus U, die von x aus erreichbar sind, erweitert und die von x aus erreichbaren Knoten y in R werden darauf überprüft, ob über x ein kürzerer Weg von u nach y existiert.

Hierfür verwenden wir folgende drei Grundoperationen, wobei der D-Wert als "Schlüssel" (key) verwendet wird:

DELETEMIN = Finde und liefere x mit minimalem D-Wert, entferne x aus R, füge x zu B hinzu.

INSERT (y) = Berechne für y den D-Wert und füge y in R ein.

DECREASEKEY (y) = Wenn ein kürzerer Weg von u über den Knoten x nach y∈ R existiert, so berechne das kleinere D(y) und positioniere y neu in R.

Dies liefert dann folgendes Verfahren (man sieht rasch, dass man die Menge U nicht explizit braucht):

```
B := {u};  R := { v | (u,v) ∈ E };
while R ≠ ∅ do
    X := DELETEMIN;
    for all  Y ∈ S(X) and Y ∉ B  do
        if  Y ∉ R  then  INSERT (Y)
        else  DECREASEKEY (Y)  fi
    od
od;
```

Hierbei ist S(x) = {y | (x, y) ∈ E } die Menge der Nachfolgerknoten ("successors") eines Knotens x im gegebenen Graphen.
Um dies entsprechend der Bedeutungen der Grundoperationen in ein Ada-Programm umzusetzen, muss die Datenstruktur für R festgelegt werden. Alles Übrige ergibt sich hieraus.

11.2.5 Prioritätswarteschlange

Eine "abstrakte" Datenstruktur, in der diverse Elemente auf ihre Abarbeitung warten (wobei sich deren Priorität nach einem Schlüssel aus einer geordneten Menge richtet), in die jederzeit Elemente eingefügt werden dürfen und aus der jederzeit das Element mit dem kleinsten Schlüssel entfernt werden kann, bezeichnet man als Prioritätswarteschlange.

Eine Prioritätswarteschlange besitzt die drei Grundoperationen DELETEMIN, INSERT und DECREASEKEY.

Anregung: Formulieren Sie die Prioritätswarteschlange als abstrakten Datentyp (sofern Kapitel 5 bekannt ist).

Prioritätswarteschlangen realisiert man oft als aufsteigenden Heap (10.4.3).

Eine Menge von n Schlüsseln werde als Heap gespeichert.

Dann realisiert man die Operation X := DELETEMIN, indem man das oberste Element des Heaps der Variablen X zuweist, das letzte Element des Heaps an die oberste Stelle setzt und es dann absinken lässt. Zeitkomplexität: $\leq 2 \cdot \log(n)$ Vergleiche.

Die Operation INSERT (Y) wird realisiert, indem der Wert von Y als letztes Element an den Heap angehängt wird und dieses Element dann im Heap aufsteigt, d.h., man vergleicht diesen neuen Schlüssel D(Y) mit seinem Elternknoten; falls der Elternknoten ein größeres Element enthält, werden die Inhalte vertauscht und man fährt genauso an dem neuen Knoten fort. Zeitkomplexität: $\leq \log(n) + 1$ Vergleiche.

Bei DECREASEKEY (Y) lässt man das Element Y, das ja schon im Heap steht, aufsteigen, sofern D(Y) durch einen Weg über den Knoten X kleiner wird. Zeitkomplexität: $\leq \log(n) + 1$ Vergleiche.

Mit dieser Heap-Realisierung lässt sich nun die Komplexität des Dijkstra-Algorithmus abschätzen. Weil in jedem äußeren Schleifendurchlauf genau ein Element aus V-{u} bearbeitet wird, gibt es maximal n-1 Durchgänge der while-Schleife.

```
B := {u};  R := { v | (u,v) ∈ E };        -- n Schritte
while R ≠ ∅ do                           -- maximal n-1 mal
    X := DELETEMIN;                      -- ≤2·log(n) Vergleiche
    for all  Y ∈ S(X) and Y ∉ B  do      -- maximal n-1 mal
        if  Y ∉ R  then  INSERT (Y)      -- ≤log(n) Vergleiche
        else  DECREASEKEY (Y)  fi        -- ≤log(n) Vergleiche
    od
od;
```

Es gibt also höchstens
$(n-1) \cdot (2 \cdot \log(n) + (n-1) \cdot \log(n)) = O(n^2 \cdot \log(n))$ Vergleiche.

Eine genauere Betrachtung ergibt eine günstigere Schranke:

Die Abfrage "Y ∈ S(X) and Y ∉ B" wird für jede Kante genau einmal durchgeführt, insgesamt also m-mal (m = |E|). Das Einfügen (mit Aufsteigen) geschieht für jedes Element höchstens einmal und erfordert höchstens log(n) Vergleiche. Dies geschieht je Knoten höchstens einmal. DECREASEKEY kann aber bis zu m-mal erfolgen, also $O(m \cdot \log(n))$ Vergleiche.

Hinzu kommen die maximal $(n-1) \cdot 2 \cdot \log(n)$ Vergleiche, die durch DELETEMIN verursacht werden.

Satz 11.2.6: Der gesamte Aufwand erfordert höchstens
$O(\underbrace{(n-1) \cdot 3 \cdot \log(n)}_{\text{verursacht durch DELETEMIN und INSERT}} + \underbrace{m \cdot \log(n)}_{\text{verursacht durch DECREASEKEY}}) = \mathbf{O((n+m) \cdot \log(n))}$ Vergleiche.

worst-case-Laufzeit des Dijkstra-Algorithmus, wenn man ihn mit Heaps implementiert

Aufgabe: Man könnte nun folgendermaßen argumentieren:

Die Abfrage "Y ∈ S(X) and Y ∉ B" wird für jede Kante genau einmal durchgeführt, insgesamt also m-mal (m = |E|). Das Einfügen INSERT und das Aufsteigen eines Elements (mit INSERT oder später DECREASEKEY) erfordern insgesamt höchstes $n \cdot \log(n)$ Vertauschungen (spätestens dann ist das jeweilige Element an der Spitze des Heaps angekommen); alle n-1 Elemente zusammen benötigen daher maximal $(n-1) \cdot \log(n)$ Vertauschungen. Somit erfordert die innere Schleife insgesamt höchstens $O(m + n \cdot \log(n))$ Schritte.

Hinzu kommen dann nur noch die maximal $O((n-1) \cdot 2 \cdot \log(n))$ Operationen, die durch DELETEMIN verursacht werden.

Somit würden wir insgesamt folgenden Zeitaufwand erhalten:

$O((n-1) \cdot 2 \cdot \log(n) + m + (n-1) \cdot \log(n)) = O(m + n \cdot \log(n))$.

Was ist an dieser Argumentation falsch??

11.2.7 Realisierung des Dijkstra-Algorithmus in Ada

Wir geben eine "strukturierte Lösung" für die Realisierung an, also eine Lösung, die den Algorithmus aus 11.2.4 beibehält und die Prioritätswarteschlange als Modul (package) formuliert. Wer sich hier einige Stunden lang durchbeißt (und eventuelle Fehler aufdeckt), hat den Algorithmus und die Implementierung wirklich verstanden.

Eine Prioritätswarteschlange wird über einer Schlüsselmenge, die eine Ordnung besitzt, definiert. In Ada bietet sich hierfür ein generisches Paket an:
Die Typen für Knoten und Schlüssel sind generisch, zusammen mit den Operationen "Addition" und "Kleiner-Relation" auf dem Schlüsseltyp.
Das Paket muss einen privaten Typ für die Prioritätswarteschlange (pqk), die drei Grundoperationen und einige Hilfs-Funktionen (Initialisieren, Test auf Leerheit) besitzen. Die Spezifikation kann man daher unmittelbar hinschreiben:


```
generic   type Node is private; type Key is private;
          Z: constant Natural;
          with function "+"(I, J: Key) return Key;
          with function "<"(I, J: Key) return Boolean;
package Prio_Queue is
  type Item is record
     Kn: Node; Vor: Node; Sch: Key;
  end record;
  type pqk is private;
  procedure Empty;
  function Isempty return Boolean;
  function DeleteMin return Item;
  procedure Insert (K: Item);
  procedure DecreaseKey (K: Item);
  private
     type MaxHeap is 0..Z;
     type pqk is array (MaxHeap) of Item;    -- pqk ≅ priority queue w.r.t. key
end package;
```

Aktueller Knoten *Kn*, Vorgängerknoten *Vor* auf einem kürzesten Weg, *Sch* = Distanz zum Startknoten (zum Speichern des D-Werts) (*Sch* negativ => Fehlerfall)

Die Implementierung erfolgt in Ada als package body.

Die vorläufigen Werte für die kürzeste Entfernung werden in der Item-Komponente *Sch* für jeden Knoten gespeichert. Die Programmierung muss dafür sorgen, dass dieser Wert beim Löschen in R und Aufnehmen in B erhalten bleibt. Dies geschieht außerhalb des Pakets Prio_Queue. Um die kürzesten Wege später rekonstruieren zu können, benötigen wir die Komponente *Vor* (für Vorgängerknoten).
Wir stellen nun die zwölfseitige Implementierung des Pakets vor. Das Feld KK vom Typ pqk bildet den Heap der Items. In der Reihenfolge des Entfernens von Knoten aus KK wird das Feld D der Ergebnisse aufgebaut. Um auf die Knoten im Heap direkt zugreifen zu können, führen wir das Feld HeapIndex ein mit
HeapIndex(K) = "der Index i mit KK(i).Kn = K".


```
package body Prio_Queue is
type MaxHeap is 0..Z;
type pqk is array (MaxHeap) of Item;
KK: pqk;                          -- KK wird hier als Heap aufgefasst
Anzahl: MaxHeap;
HeapIndex: array (Node) of MaxHeap;
procedure Empty is
  begin Anzahl := 0; end;
function Isempty return Boolean is
  begin return (Anzahl=0); end;

-- Für das Folgende beachte man:
-- Es besteht eine Beziehung zwischen einem Knoten und seinem Index im
-- Heap. Hierfür verwenden wir HeapIndex: array (Node) of MaxHeap.
-- Im Feld HeapIndex wird für jeden Knoten K notiert, ob er sich im Heap
-- befindet; genau dann gilt: (HeapIndex(K) > 0), und in diesem Falle ist
-- HeapIndex(K) sein Index im Heap KK.
-- Delta: array (Node,Node) of Key ist die Kantenmarkierung δ im Graphen,
-- die eigentlich als Komponente W in den Kanten gespeichert ist.
```

```
-- Wir übernehmen die Prozedur sink für den Heap KK aus 10.4.4,
-- müssen allerdings die Verknüpfung zwischen einem Knoten und seinem
-- Index im Heap stets aktualisieren, d.h., HeapIndex muss beim
-- Aufsteigen oder Absteigen eines Knotens im Heap entsprechend
-- umgesetzt werden.

procedure sink (links, rechts: MaxHeap) is
i, j: Natural;  weiter: Boolean:=true;  v: Item;
begin  v := KK(links);  i := links;  j := i+i;
  while  (j <= rechts) and weiter  loop
    if  j = rechts  then
        if v.Sch < KK(j).Sch then
            KK(i):=KK(j);
            HeapIndex(KK(j).Kn) := i;
            i:=j;
        end if;
        weiter:=false;
```



```
    elsif KK(j).Sch < KK(j+1).Sch then
        if v.Sch < KK(j+1).Sch then
            KK(i) := KK(j+1);
            HeapIndex(KK(j+1).Kn) := i;
            i := j+1;
        else weiter:=false; end if;
    else if v.Sch < KK(j).Sch then
            KK(i) := KK(j);
            HeapIndex(KK(j).Kn) := i;
            i := j;
        else weiter:=false; end if;
    end if;
    j := i+i;
  end loop;
  KK(i):=v;
  HeapIndex(v.Kn) := i;
end sink;
```



```
function DeleteMin return Item is
Wurzel: Item;
begin
  if not Isempty then  Wurzel := KK(1);  Anzahl := Anzahl - 1;
  else Wurzel := KK(0);  end if;          -- dieser else-Fall tritt nie ein
  if not Isempty then  KK(1) := KK(Anzahl+1);
    sink(1, Anzahl); end if;
  HeapIndex(Wurzel.Kn) := 0;
  return Wurzel;
end DeleteMin;

-- Man beachte, dass wie üblich die zu entfernenden Elemente nicht gelöscht
-- werden, sondern dass nur der Index "Anzahl", der auf das letzte Element
-- im Heap zeigt, um 1 verringert wird. Die Manipulationen der Menge B
-- ziehen wir heraus; sie werden im Dijkstra-Algorithmus direkt ausgeführt.
-- In KK(0) speichern wir ein "null record", welches immer dann zurück
-- gegeben wird, wenn die Prioritätswarteschlange R leer ist; dieser Fall tritt
-- aber wegen  while R ≠ ∅ do  (= while not Isempty do) nicht ein.
```



```
procedure Aufsteigen(Los: MaxHeap) is      -- eine Hilfsprozedur
i, j: Natural;  v: Item;
begin  i := Los; j := i/2;      -- j zeigt auf den Elternknoten von i im Heap
  if  j = 0  then  HeapIndex(KK(1).Kn) := 1;
  else while  j > 0  and then  KK(i).Sch < KK(j).Sch  do
        v := KK(i);  KK(i) := KK(j);  KK(j) := v;
        HeapIndex(KK(i).Kn) := j;  HeapIndex(KK(j).Kn) := i;
        i := j;  j := i/2;  end loop;
  end if;
end Aufsteigen;

procedure Insert (K: Item) is
begin                              -- K am Ende einfügen und aufsteigen lassen
  Anzahl := Anzahl+1; KK(Anzahl) := K;
  HeapIndex(K.Kn) := Anzahl;
  Aufsteigen(Anzahl);
end Insert;
```

```
procedure DecreaseKey (K: Item) is
IndexK: MaxHeap;
begin  IndexK := HeapIndex(K.Kn);
    if K.Sch < KK(IndexK).Sch
       then KK(IndexK) := K;
       Aufsteigen (IndexK); end if;
end DecreaseKey;

begin Empty;                          -- Initialisierung des Pakets
       for q in Node loop HeapIndex(q) := 0; end loop;
end package Prio_Queue;
```

Mit diesem Paket lässt sich nun der Dijkstra-Algorithmus als Block (mit einem Unterblock und Prozedur) ausformulieren.


```
declare                              -- erforderliche Vorab-Deklarationen
  generic ... "generischer Teil wie oben angegeben";
  package Prio_Queue ... "Spezifikation wie oben angegeben";
  package body Prio_Queue ..."wie oben angegeben";
  type Knoten;  type NextKnoten is access Knoten;
  type Kante;  type NextKante is access Kante;
  type Knoten is record
       Id: Knotenname;
       Besucht: Boolean;  zahl1, zahl2: Integer;
       Inhalt: <weitere Komponenten>;
       NKn: NextKnoten;    -- nächster Knoten (in Knotenliste)
       EIK: NextKante;     -- erste inzidente Kante
  end record;
  type Kante is record
       W: <Typ des Gewichts der Kanten>;
       EKn: NextKnoten;    -- Endknoten dieser Kante
       NKa: NextKante;     -- nächste Kante (bzgl. Knoten)
  end record;
```

Adjazenzlistendeklaration wie in 11.1.3


```
N, M: Natural;  Anfang: NextKnoten;
Delta: array (Knoten, Knoten) of Float := (others => "∞");
procedure Graph_Aufbau (Anker: in out NextKnoten) is
"Selbst programmieren: Prozedur, die einen Graphen einliest und/
oder als Adjazenzliste aufbaut, auf deren Wurzel Anker zeigt; setze
N und M auf die Zahl der Knoten bzw. der Kanten."
end Graph_Aufbau;
function G_korrekt (Anker: NextKnoten) return Boolean is
ok: Boolean := true;  p: Nextknoten;  e: Nextkante;
begin p := Anker;
    while p /= null loop  e := p.EIK;
        while e /= null loop
        Delta(p.all, e.EKn.all) := e.W; e:=e.NKa;
        if e.W < 0.0 then ok := false; end if;
    end loop; end loop;
return ok;
end G_korrekt;
```

G_korrekt überträgt die Kantenmarkierung nach Delta und prüft, ob alle Werte nichtnegativ sind. Man sollte noch prüfen, dass keine Schlingen und Mehrfachkanten vorliegen (selbst einfügen).


```
begin                    -- Beginn des Anweisungsteils für den Graphen
  Graph_Aufbau (Anfang);      -- jetzt sind N und M bekannt
  if not G_korrekt then ... <"Fehlerfall behandeln">;
  else                        -- Die Deltatabelle ist nun gesetzt
  declare                     -- Beginn des Blockes für Dijkstra-Alg.
  package G_Prio_Queue is     -- Prio-Warteschlange für Knoten
     new Prio_Queue (Knoten, Float, N, "+", "<");
  with Ada.Integer_Text_IO; with Ada.Text_IO;
  with G_Prio_Queue;  use G_Prio_Queue;
  Startknoten: Knoten;
  Zustand: array (Knoten) of (B, R, U) := (others => U);
                   -- gibt an, ob ein Knoten in B, R oder U liegt
  D: pqk;                     -- zum Speichern der Ergebnisse
                              -- wegen "use" ist pqk hier sichtbar
  Zähler: Natural;            -- zählt die Knoten in B
```

```
procedure Dijkstra_with_Heap(Start: Knoten) is
X: Item; Y: Knoten;  Edge: NextKante;  H: Float;
begin                                  -- entsprechend Verfahren in 11.2.4
    Edge := Start.EIK;                 -- Initialisiere die Menge R (= KK).
    while Edge /= null loop
        -- Alle Nachfolger des Startknotens Start kommen nach R
        Y := Edge.EKn.all;
        Zustand(Y) := R;
        Insert((X.Kn, Y, Delta(Start, Y)));
        Edge := Edge.NKa;
    end loop;
    Zustand(Start) := B;
    D(1) := (Start, Start, 0.0);
    Zähler := 1;
```



```
    while not Isempty loop                 -- solange R nicht leer ist
        X := DeleteMin;  Zustand(X.Kn) := B;
        Zähler := Zähler+1;  D(Zähler) := X;
        Edge := X.Kn.EIK;
        while Edge /= null loop
            Y := Edge.EKn.all;
            if  Zustand(Y) /= B  then
                H := X.Sch + Delta(X.Kn, Y.all);
                if  Zustand(Y) = U  then
                    Insert((Y, X, H));  Zustand(Y) := R;
                else DecreaseKey((Y, X, H));
            end if;
            Edge := Edge.NKa;
        end loop;
    end loop;
end Dijkstra_with_Heap;
```



```
    begin
        Startknoten := <"wähle einen Knoten aus">;
        Dijkstra_with_Heap (Startknoten);
        for i in 1..Zähler loop
            Put(D(i).Kn.Id); Put(D(i).Vor.Id); Put(D(i).Sch);
        end loop;
        ...
    end;                    -- Ende des inneren Blocks für den Dijkstra-Alg.
end if;
...
end;                        -- Ende des äußeren Blocks der Vorab-Deklarationen
```


11.2.8 Hinweise

1. Obiges Programm wurde nicht getestet und kann daher (und wird vermutlich auch) noch einige Flüchtigkeitsfehler enthalten (nochmals der Hinweis: bitte selbst durcharbeiten!).

2. Da im Verfahren stets der Knoten X mit der geringsten Entfernung vom Startknoten ausgewählt wird, gehört der Dijkstra-Algorithmus zu den Greedy-Verfahren, siehe 6.7.

3. Eine andere Implementierung (mit den sog. Fibonacci-Heaps) bringt die bessere Zeitkomplexität $O(m + n \cdot \log(n))$; allerdings werden die Konstanten hierbei größer. Im Mittel (average case) ist der Dijkstra-Algorithmus ebenfalls von dieser Größenordnung, so dass sich jene effizientere Datenstruktur in der Praxis nur in speziellen Anwendungen lohnt. Details siehe Vorlesungen zur Algorithmik.

4. Dringende Empfehlung für alle: "Opfern" Sie drei Tage nur zu dem Zweck, den Dijkstra-Algorithmus in Ada zum Laufen gebracht zu haben. Anders werden Sie die Schwierigkeiten, die bereits mit dem "Programmieren im Kleinen" verbunden sind, kaum verinnerlichen. Auch erhalten Sie ein Gespür dafür, wie mühsam es ist, für die Praxis taugliche Bibliotheksprogramme (oder Pakete oder objektorientierte Klassen) zu erstellen.

5. Für die Berechnung der kürzesten Wege zwischen allen Knotenpaaren (APSP) verwendet man den sehr leicht zu implementierenden Floyd-Algorithmus. Dieser ist dem "Warshall-Algorithmus" zur Berechnung der transitiven Hülle (6.5.5) sehr ähnlich und besitzt ebenfalls die Zeitkomplexität $O(n^3)$, siehe Literatur.

6. Negative Kantengewichte: Der Dijkstra-Algorithmus nutzt aus, dass einmal gefundene kürzeste Entfernungen später nicht weiter verkürzt werden können. *Dieser Fall tritt aber bei negativen Kantengewichten ein.*

Erlaubt man negative Gewichte, also auch $\delta((x,y)) < 0$, dann verwendet man den "Bellman-Ford-Algorithmus", der bis zu (n-1)-mal für alle Kanten (x,y) ausprobiert, ob mit dieser Kante die aktuell kürzeste Entfernung dist(y) von u nach y über den Knoten x verkleinert werden kann.

Formulierung in Ada:

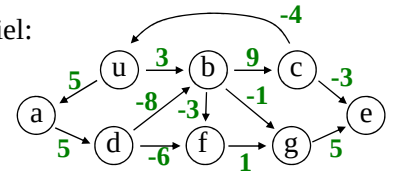

```
for zaehler in 1..n-1 loop
    for all e = (x,y) ∈ E loop
        if dist(x) + δ((x,y)) < dist(y) then
            dist(y) := dist(x) + δ((x,y));
            Vorgaengerknoten(y) := x;
        end if;
    end loop;
end loop;
```

Bellman-Ford-Algorithmus

Treten hierbei negative Zyklen (= Zyklen mit negativer Distanz) auf, so gibt es zu gewissen Knoten keine kürzesten Wege. Dies erkennt der Bellman-Ford-Algorithmus dadurch, dass man am Ende noch einmal die innere for-Schleife ausführt: Ein negativer Zyklus existiert genau dann, wenn sich herbei noch eine Verbesserung ergibt.

Aufwand: $O(n \cdot m)$. Bis heute sind keine Algorithmen mit kleinerer worst-case-Zeitkomplexität bekannt.

Beispiel:



Die Kanten werden im Beispiel in der folgenden Reihenfolge durchlaufen: (a,d), (u,a), (u,b), (c,u), (d,b), (f,g), (b,f), (d,f), (b,c), (b,g), (c,e), (g,e).

	dist	vorg	dist	vorg	dist	vorg	dist	vorg	dist	vorg	dist vorg
u	0	u	0	u	0	u	0	u	0	u	
a	∞	-	5	u	5	u	5	u	5	u	
b	∞	-	3	u	2	d	2	d	2	d	
c	∞	-	12	b	11	b	11	b	11	b	keine Verän-derun-gen mehr
d	∞	-	∞	-	10	a	10	a	10	a	
e	∞	-	7	g	6	g	5	g	5	g	
f	∞	-	0	b	-1	b	-1	b	-1	b	
g	∞	-	2	b	1	b	0	f	0	f	
	zu Beginn		nach zaehler = 1		nach zaehler = 2		nach zaehler = 3		nach zaehler = 4		nach zaehler ≥ 4

11.2.9 Korrektheit des Dijkstra-Algorithmus

Satz: Der Dijkstra-Algorithmus berechnet zu jedem Knoten $y \in V$ die kürzeste Entfernung $d(u,y) = D(y)$. Ein kürzester Weg von u nach y kann (mit dem array Vor) in O(n) Schritten berechnet werden: $y, Vor(y), Vor(Vor(y)), Vor^3(y), ..., u$.

Beweis: Es genügt, eine geeignete Schleifeninvariante für die while-Schleife anzugeben.

Invariante: Für jedes $x \in B$ ist $\delta(u,x) = D(x)$ die kürzeste Entfernung von u nach x. Ein kürzester Weg von u nach x ist mittels Vor bekannt. Für alle $x \in B$ und $z \notin B$ gilt: $D(x) = \delta(u,x) \le d(z) = \delta(u,z) \le D(z)$.

Diese Invariante gilt zweifellos zu Beginn des Algorithmus nach dem Initialisieren (wegen $\delta(u,u) = D(u) = 0$ und $\delta(e) \ge 0$ für alle Kanten e).

Wir betrachten den i-ten Schleifendurchlauf. Alle verwendeten Mengen seien die Mengen, *bevor* der i-te Schleifendurchgang beginnt. Für sie gilt die Invariante. Wir müssen zeigen, dass die Invariante auch am Ende des i-ten Schleifendurchlaufs gilt.

Sei $v \in R$ der Knoten, der zu Beginn der Schleife ausgewählt wird, d.h., $D(v) \le D(z)$ für alle $z \in R$. Wegen der Invarianten gilt $D(x) = \delta(u,x) \le \delta(u,z) \le D(z)$ für alle $x \in B$ und $z \notin B$. Wegen $B := B \cup \{v\}$ gilt diese Aussage weiterhin, außer eventuell für $x = v$. Wir müssen also nur zeigen: $\delta(u,v) = D(v)$ und für alle $z \notin B$ gilt: $D(v) \le \delta(u,z) \le D(z)$.

(1) Zu zeigen: $\delta(u,v) = D(v)$. Da D(v) die Entfernung auf einem Weg von u nach v ist, gilt stets $\delta(u,v) \le D(v)$.
Annahme, $\delta(u,v) < D(v)$, d.h., D(v) ist echt größer als $\delta(u,v)$, dann gäbe es einen Weg $u = u_0, u_1, u_2, ..., u_{r-1}, u_r = v$ von u nach v mit echt kleinerer Länge als D(v). Betrachte das kleinste j mit $u_0, u_1, u_2, ..., u_j \in B$ und $u_{j+1} \notin B$. Dann ist wegen der Invarianten $\delta(u,u_i) = D(u_i)$ für alle $i = 0, 1, ..., j$. Es folgt (beachte die Zuweisung $D(w) := D(v) + \delta(v,w)$ für alle Nachfolger w solcher Knoten v):
$d(u_0,u_1,u_2,...,u_j,u_{j+1}) = d(u_0,u_1,u_2,...,u_{j-1},u_j) + \delta(u_j,u_{j+1}) \le d(u_0,u_1,u_2,...,u_{r-1},u_r) = \delta(u,v) < D(v) = \delta(u,Vor(v)) + \delta(Vor(v),v)$.
Da Vor(v) und u_j in B liegen, gilt wegen der Invarianten $d(u_0,u_1,u_2,...,u_{j-1},u_j) = D(u_j)$ und $\delta(u,Vor(v)) = D(Vor(v))$.
Folglich ist $D(u_{j+1}) \le D(u_j) + \delta(u_j,u_{j+1}) < D(Vor(v)) + \delta(Vor(v),v) = D(v)$, d.h., in der Schleife hätte nicht v, sondern u_{j+1} ausgewählt werden müssen. Widerspruch! Also ist die Annahme "$\delta(u,v) < D(v)$" falsch. Dann muss aber $\delta(u,v) = D(v)$ sein, da D(v) als die Entfernung eines existierenden Weges von u nach v nicht kleiner als die kürzeste Entfernung sein kann.

(2) Sei nun $z \notin B$. Betrachte einen kürzesten Weg von u nach z:
$u = u_0, u_1, u_2, ..., u_{q-1}, u_q = z$.
Annahme, $D(v) > \delta(u,z)$. Dann kann man die gleiche Argumentation wie in (1) verwenden: Man betrachte den Anfangsweg
$u_0, u_1, u_2, ..., u_{j-1}, u_j$ von $u = u_0, u_1, u_2, ..., u_{q-1}, u_q = z$ mit $u_0, u_1, u_2, ..., u_j \in B$ und $u_{j+1} \notin B$. Dann folgert man ebenfalls:

$D(u_{j+1}) \le D(u_j) + \delta(u_j,u_{j+1}) < D(Vor(v)) + \delta(Vor(v),v) = D(v)$,

d.h., ein Knoten auf dem Weg von u nach z hätte anstelle von v zu Beginn der Schleife ausgewählt werden müssen. Widerspruch. Also muss $D(v) \le \delta(u,z)$ und somit auch $D(v) \le \delta(u,z) \le D(z)$ sein.

In Vor(v) steht der Vorgänger von v auf diesem kürzesten Weg. Folglich ist ein kürzester Weg auch von u nach v bekannt.

In jedem Schleifendurchlauf wächst B um einen Knoten. Knoten aus B können nicht wieder in R aufgenommen werden, d.h. nach spätestens n-1 Durchläufen terminiert die Schleife mit $R = \emptyset$. Wegen der Invarianten gibt D dann die kürzesten Entfernungen für die Knoten in B an. Für alle anderen (von u aus nicht erreichbaren) Knoten z gilt $D(z) = \infty$. Also folgt: $\forall y \in V: D(y) = \delta(u,y)$.

11.2.10 Zusammenhangskomponenten

Wiederholen Sie die Begriffe zusammenhängend, stark und schwach zusammenhängend aus Abschnitt 3.8.

Mit dem Dijkstra-Algorithmus kann man im ungerichteten Fall auch die Zusammenhangskomponente zum Startknoten bestimmen, da alle anderen Knoten am Ende die Entfernung "∞" besitzen (im Programm: Delta(v) = "∞"). Hierfür verwendet man als Kantengewicht die Zahl 1 für jede Kante.

Setzt man den Dijkstra-Algorithmus auf jeden Knoten an, so kann man alle Zusammenhangskomponenten (im gerichteten und im ungerichteten Fall) bestimmen. Dies ist allerdings zu aufwendig. Im ungerichteten Fall liefert ein modifizierter Graphdurchlauf bereits alle Zusammenhangskomponenten in Linearzeit. Überlegen Sie sich für den gerichteten Fall ein Verfahren, welches den Graphen mit DFS durchläuft und hierbei schon gefundene Teile von Zusammenhangskomponenten zu größeren Einheiten zusammenfasst, sobald man auf Knoten stößt, die bereits besucht wurden.

11.3 Minimale Spannbäume

Gegeben sei ein ungerichteter Graph $G = (V, E, \delta)$ mit $\delta: E \to \mathbb{R}$ (= Menge der reellen Zahlen).
Gesucht wird ein minimaler Spannbaum, also ein Baum $B = (V, E_B, \delta)$ mit gleicher Knotenmenge, der Teilgraph von G ist und dessen Gewicht $\delta(B)$ minimal ist bzgl. aller Spannbäume von G (siehe Def. 8.8.6).

Zur Lösung dieses Problems verwendet man den Prim-Algorithmus mit der gleichen Zeitkomplexität des Dijkstra-Algorithmus $O((n+m)\cdot\log(n))$ oder den Kruskal-Algorithmus mit der Zeitkomplexität $O(m\cdot\log(m))$.

J.B.Kruskal und R.C.Prim veröffentlichten ihre Algorithmen 1956 bzw. 1957.

11.3.1 Der Algorithmus von Prim

Der Prim-Algorithmus arbeitet genau so wie der Dijkstra-Algorithmus, jedoch mit einem modifizierten Schlüssel *Sch*, der die Reihenfolge in der Prioritätswarteschlange festlegt. Während *bei Dijkstra* für alle $y \in R$ zu jedem Zeitpunkt gilt

$$Sch(y) = Min\{ Sch(v) + \delta((v,y)) \mid v \in B \},$$

verwendet man *beim Prim*-Algorithmus den kleinsten Abstand von y zu irgendeinem Knoten aus B, d. h., für alle $y \in R$ lautet jetzt der Schlüssel *Sch* zu jedem Zeitpunkt

$$Sch(y) = Min\{ \delta((v,y)) \mid v \in B \}.$$

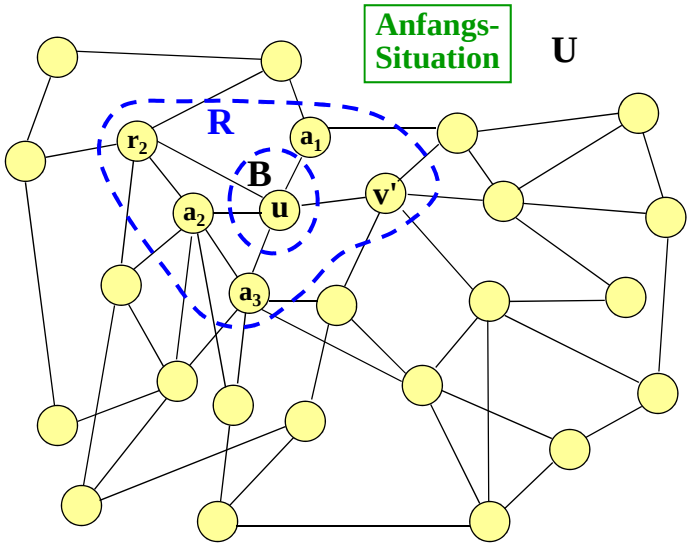
Mit dieser geringen Änderung geht das Programm 11.2.7 in den Prim-Algorithmus über. Auch hier entsteht ein Baum, da in die Menge B jeder Knoten mit genau einer Kante (zu seinem jeweiligen Vorgänger in B) aufgenommen wird.

Genauer: Nur in der Prozedur Dijkstra_with_Heap ist die Zeile
"H := X.Sch + Delta(X.Kn, Y.all);"
zu ersetzen durch
H := Delta(X.Kn, Y.all);

Die Initialisierung der Schlüsselwerte im Aufruf "Insert" zu Beginn der Prozedur Dijkstra_with_Heap erfolgt mit den Delta-Werten und ist daher bereits korrekt.

Da der Graph beim Prim-Algorithmus ungerichtet, beim Dijkstra-Algorithmus dagegen gerichtet ist, muss jede ungerichtete Kante durch zwei gerichtete Kanten (mit gleichem δ-Wert) dargestellt sein, wenn wir das Programm ohne weitere Änderungen übernehmen wollen.

Wir demonstrieren den Prim-Algorithmus an einem Beispiel.

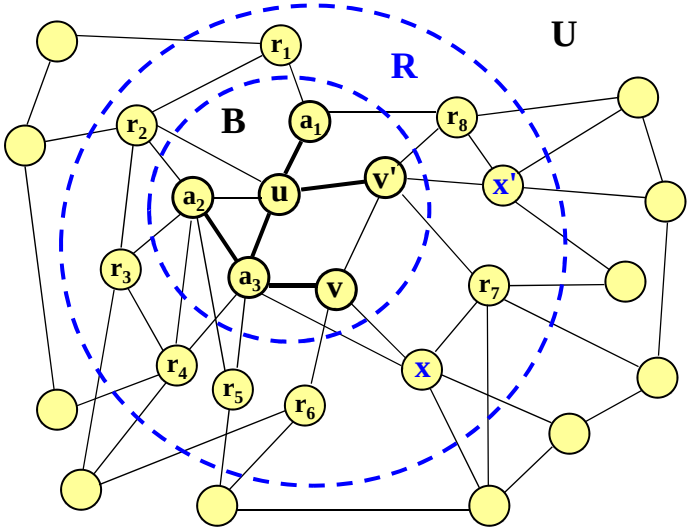


Ein Beispiel-Graph G = (V, E, δ) mit Startknoten u.
Anfangs werden $B = \{u\}$ und $R = \{r_2, a_1, a_2, a_3, v'\}$ = Menge der Nachbarknoten von u gesetzt. U = Menge der restlichen Knoten.

Jede ungerichtete Kante liegt im Programm als zwei gerichtete Kanten vor.

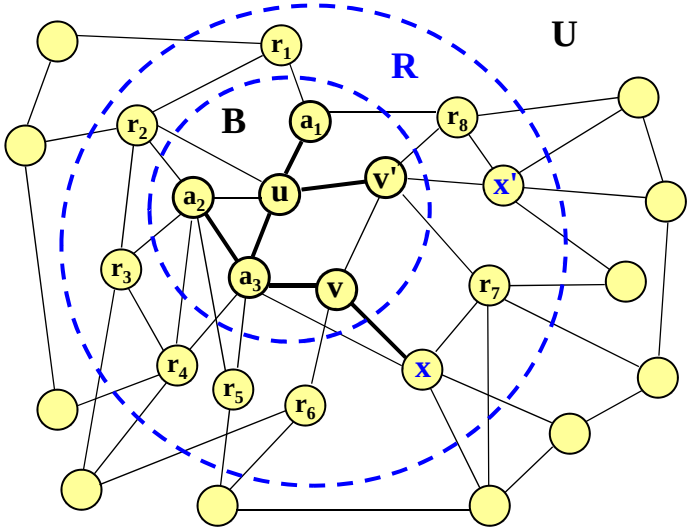
Die Werte der Kantenmarkierung δ sind hier nicht eingetragen.

Beachte: Die Menge der Kanten, die zwischen B und R verlaufen, trennt die Mengen B und V-B, d.h., jeder Weg von einem Knoten aus B zu einem Knoten aus V-B führt über mindestens eine solche Kante.



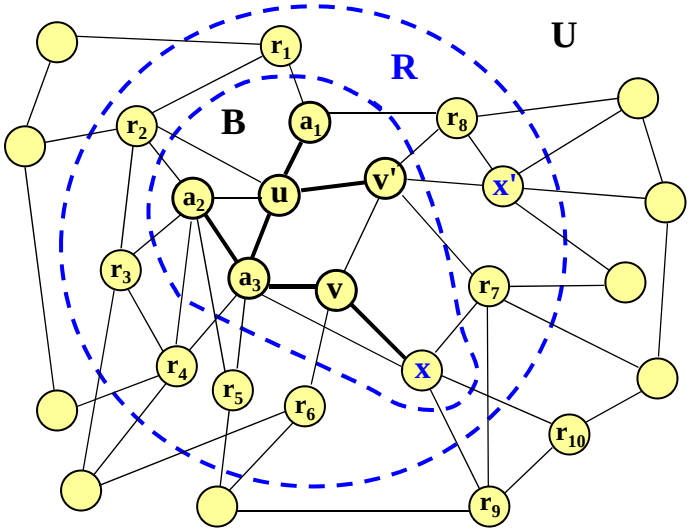
Der bereits erstellte Teil eines minimal spannenden Baums ist mit fetten Kanten und fett umrandeten Knoten (B) dargestellt. Die kleinste Kante sei nun {v, x}.

Betrachte eine aktuelle Situation beim Prim-Algorithmus:
$B = \{u, a_1, a_2, a_3, v, v'\}$, $R = \{x, x', r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8\}$,
U = Menge der restlichen Knoten (diese haben hier keine Namen).



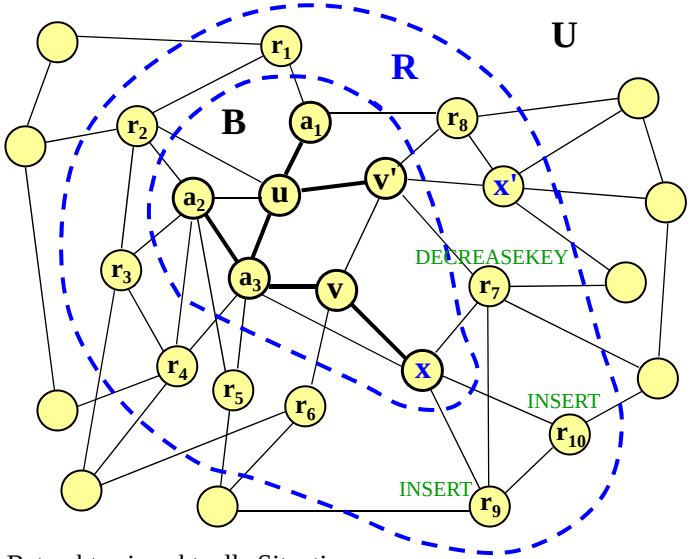
Der bereits erstellte Teil eines minimal spannenden Baums ist mit fetten Kanten und fett umrandeten Knoten (B) dargestellt. Die kleinste Kante sei nun {v, x}.
Dann wird x zusammen mit dem Verweis auf v und $\delta(\{v, x\})$ in B aufgenommen und die mit x adjazenten Knoten, die nicht in B liegen, werden in R aufgenommen oder ihr Schlüsselwert *Sch* wird ggfls. erniedrigt (falls sie schon in R liegen).

Betrachte eine aktuelle Situation beim Prim-Algorithmus:
$B = \{u, a_1, a_2, a_3, v, v'\}$, $R = \{x, x', r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8\}$,
U = Menge der restlichen Knoten (diese haben hier keine Namen).



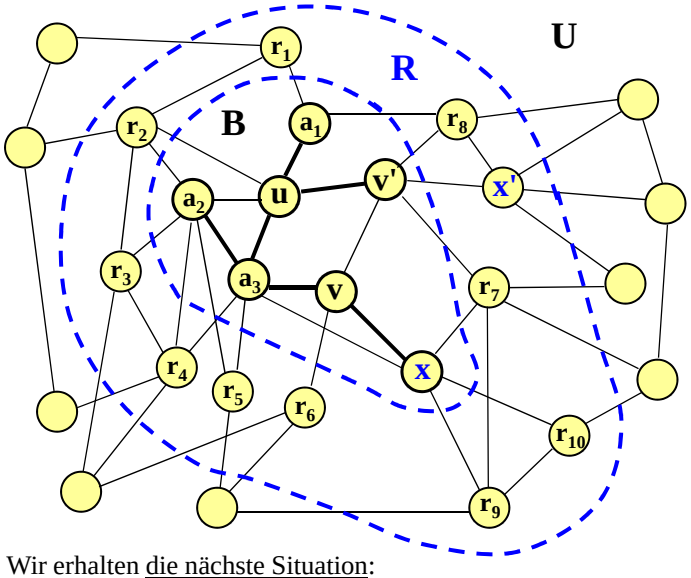
Der bereits erstellte Teil eines minimal spannenden Baums ist mit fetten Kanten und fett umrandeten Knoten (B) dargestellt. Die kleinste Kante sei nun {v, x}.
Dann wird x zusammen mit dem Verweis auf v und $\delta(\{v, x\})$ in B aufgenommen und die mit x adjazenten Knoten, die nicht in B liegen, werden in R aufgenommen oder ihr Schlüsselwert *Sch* wird ggfls. erniedrigt (falls sie schon in R liegen).

Betrachte eine aktuelle Situation beim Prim-Algorithmus:
$B = \{u, a_1, a_2, a_3, v, v'\}$, $R = \{x, x', r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8\}$,
U = Menge der restlichen Knoten (diese haben hier keine Namen).



Der bereits erstellte Teil eines minimal spannenden Baums ist mit fetten Kanten und fett umrandeten Knoten (B) dargestellt. Die kleinste Kante sei nun {v, x}.
Dann wird x zusammen mit dem Verweis auf v und $\delta(\{v, x\})$ in B aufgenommen und die mit x adjazenten Knoten, die nicht in B liegen, werden in R aufgenommen (hier: r_9 und r_{10}) oder ihr Schlüsselwert *Sch* wird ggfls. erniedrigt (falls sie schon in R liegen, hier r_7).

Betrachte eine aktuelle Situation:
$B = \{u, a_1, a_2, a_3, v, v'\}$, $R = \{x, x', r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8\}$,
U = Menge der restlichen Knoten (diese haben hier keine Namen).



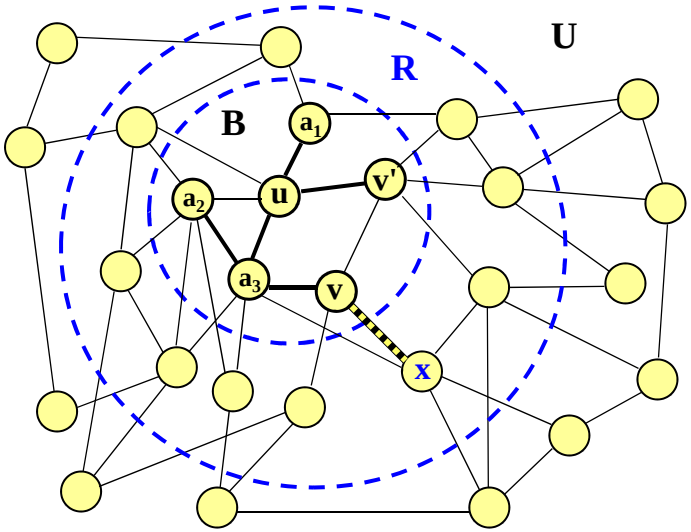
Der bereits erstellte Teil eines minimal spannenden Baums ist mit fetten Kanten und fett umrandeten Knoten (B) dargestellt. Jetzt wiederholt sich das Verfahren: Die kleinste Kante sei nun ... .

Wir erhalten die nächste Situation:
$B = \{u, a_1, a_2, a_3, v, v', x\}$, $R = \{x', r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}\}$,
U = Menge der restlichen Knoten (diese haben hier keine Namen).

11.3.2 Wie kann man beweisen, dass dieser Algorithmus tatsächlich einen minimalen Spannbaum konstruiert?

Mit Hilfe der Eigenschaft, dass die Kanten zwischen B und R die Mengen B und V-B trennen, geht dies sehr einfach (siehe Folie mit der Anfangssituation oben).

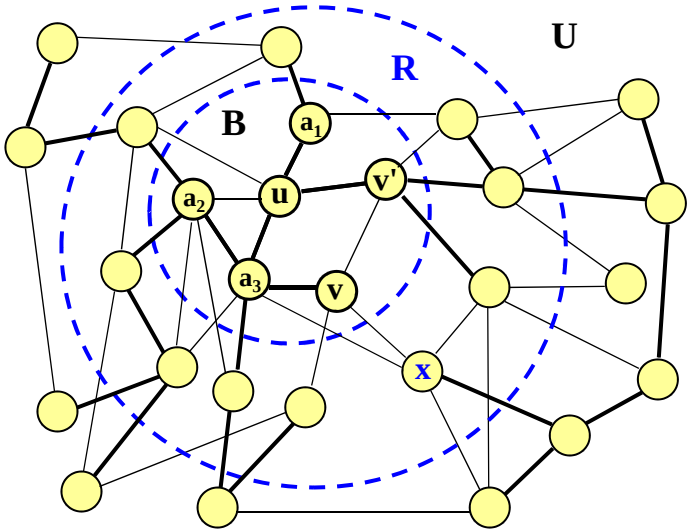
Korrektheitsbeweis: Zunächst sei der Graph zusammenhängend, denn sonst besitzt er keinen Spannbaum.
Annahme, wir hätten für den minimalen Spannbaum eine kleinste Kante nicht verwenden dürfen. D.h., in keinem minimalen Spannbaum kommt die Kante {v,x} vor, obwohl sie in einer Situation des Prim-Algorithmus kleiner gleich allen anderen Kanten, die von B nach R führten, war und folglich vom Prim-Algorithmus ausgewählt wurde. Wir betrachten eine Situation aus dem obigen Beispiel zur Illustration:



Der bereits erstellte Teil eines minimal spannenden Baums ist mit fetten Kanten und fett umrandeten Knoten (B) dargestellt. Die kleinste Kante von B nach R sei nun {v, x}.

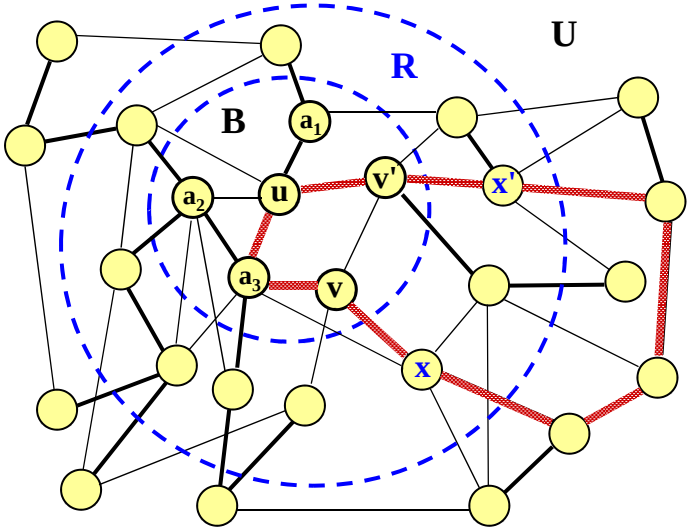
Wir nehmen nun an: Diese Kante {v,x} ist in keinem minimalen Spannbaum enthalten.

In dieser aktuellen Situation des Prim-Algorithmus sei der Schlüssel *Sch* von x minimal und die Kante {v, x} habe den kleinsten δ-Wert aller Kanten zwischen B und R.



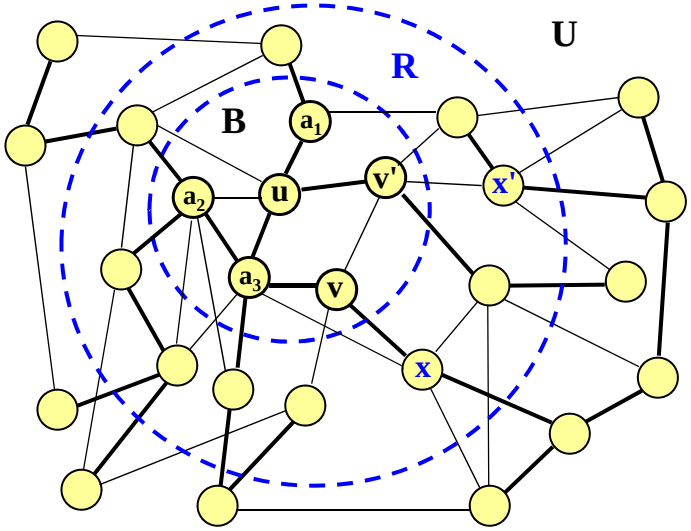
Ein minimaler Spannbaum wird mit fetten Kanten dargestellt.

Sei dies ein minimaler Spannbaum, in dem die Kante {v, x} nicht enthalten ist. Fügt man nun die Kante {v, x} zu diesem Spannbaum hinzu, so muss ein Zyklus entstehen, auf dem v und x liegen.



Wir haben {v,x} hinzugefügt. Der entstandene Zyklus ist mit dickeren Kanten hervorgehoben.

Da v in B und x in V-B lag, muss es genau eine Kante $\{v', x'\}$ auf diesem Zyklus geben, die zum Zeitpunkt, als $\{v, x\}$ ausgewählt wurde, ebenfalls vorhanden war, d.h., es gilt $\delta(\{v, x\}) \leq \delta(\{v', x'\})$.



Wir lassen nun {v',x'} weg. Der neue Spannbaum ist mit fetten Kanten dargestellt.

Ersetzt man nun $\{v', x'\}$ im minimalen Spannbaum durch $\{v, x\}$, so erhält man wieder einen Spannbaum, dessen Gewicht um den Wert $\delta(\{v', x'\}) - \delta(\{v, x\}) \geq 0$ kleiner ist.

Dieser neue Spannbaum, in dem $\{v, x\}$ vorkommt, besitzt also ein Gewicht, das kleiner oder gleich dem des zuerst betrachteten minimalen Spannbaums ist.

Folglich gibt es doch einen minimalen Spannbaum, der die Kante $\{v, x\}$ enthält, im Widerspruch zur Annahme. Daher gibt es also stets einen minimalen Spannbaum, in dem die vom Prim-Algorithmus ausgewählten Kanten vorkommen. Der Prim-Algorithmus liefert daher immer einen minimalen Spannbaum. ■

Natürlich kann es mehrere minimale Spannbäume zu einem Graphen geben, wenn es Kanten mit gleichem δ-Wert im Graphen gibt. Dann treten eventuell in der Prioritätswarteschlange mehrere Knoten mit gleichem minimalem Schlüssel *Sch* auf. Egal welchen solchen Knoten man dann wählt, man erhält am Ende stets einen minimalen Spannbaum (die nichtdeterministische Auswahl "schadet" also nicht).

11.3.3 Der Algorithmus von Kruskal

Der Kruskal-Algorithmus sortiert zuerst die Kanten bzgl. δ, beginnt dann mit der kleinsten Kante und nimmt jeweils die nächste Kante hinzu, sofern diese Kante keinen Zyklus mit den bereits ausgewählten Kanten bildet. Auf diese Weise entsteht ein minimaler Spannbaum.

Die Zeitkomplexität beträgt $O(m \cdot \log(m))$ mit $m = |E|$.

Dies gilt aber nur bei geeigneter Implementierung, insbesondere muss es eine Liste oder ein Feld der Kanten geben (hier empfiehlt sich eine Inzidenzlistendarstellung des Graphen, in der die Knoten und die Kanten in eigenen Listen hintereinander stehen und Verweise zu den Endknoten von der Kanten- in die Knotenliste existieren).

"Einziges" Problem hierbei:

Wie stellt man fest, ob eine Kante mit den bereits ausgewählten Kanten einen Zyklus bildet?

Lösung: Man bildet ein System von Mengen aus Knoten und fasst jeweils in einer Menge genau die Knoten zusammen, die untereinander durch bereits ausgewählte Kanten verbunden sind.

Für jede Kante prüft man dann, ob ihre Endknoten in der gleichen Menge liegen. Falls ja, so verwirft man die Kante, denn dann bewirkt diese Kante einen Zyklus; falls nein, so nimmt man die Kante zu den Baumkanten hinzu und vereinigt die beiden Mengen, in denen die Endknoten lagen, zu einer neuen Menge.

Kruskal-Algorithmus für einen minimalen Spannbaum:

Gegeben sei ein ungerichteter Graph $G = (V, E, \delta)$.

1. Sortiere alle Kanten nach ihrem δ-Wert.
2. Bilde für jeden Knoten $y \in V$ die Menge $\{y\}$; setze
   BK := ∅; -- dies wird die Menge der Baumkanten
3. while (die Kantenliste ist nicht leer) and (|BK| < n-1) do
   entferne die kleinste Kante e aus der Kantenliste;
   if die Endknoten x und y der Kante e liegen in verschiedenen Mengen M und N then
   bilde $M := M \cup N$; vergiss N; $BK := BK \cup \{e\}$ fi
   od;

Die Kanten von BK bilden einen minimalen Spannbaum.

11.3.4 *Korrektheit des Kruskal-Algorithmus:*

Der Beweis, dass dieses Vorgehen einen minimalen Spannbaum liefert, wird ähnlich wie der Beweis zum Prim-Algorithmus geführt. Annahme, eine Kante e, die vom Kruskal-Algorithmus ausgewählt wurde, würde in keinem minimalen Spannbaum sein, dann fügt man sie zu einem minimalen Spannbaum hinzu, wodurch ein Zyklus entsteht, auf dem mindestens eine Kante mit nicht-kleinerem δ-Wert liegen muss; gegen diese tauscht man die Kante e aus und erhält einen minimalen Spannbaum, in dem die Kante e doch vorkommt.

(Hinweis: Gäbe es auf dem Zyklus außer der Kante e nur Kanten mit kleinerem δ-Wert, so könnte die Kante e nicht vom Kruskal-Algorithmus ausgewählt worden sein, da alle anderen Kanten vorher hätten ausgewählt worden sein müssen und e dann einen Zyklus bewirkt hätte.) ■

Der Algorithmus stellt nebenbei auch fest, ob der gegebene Graph zusammenhängend war: Dies trifft genau dann zu, wenn am Ende ein Spannbaum entsteht, wenn also genau n-1 Kanten ausgewählt wurden.

Das Vereinigen der beiden Mengen wurde hier so realisiert, dass eine der Mengen (hier: M) neu auf $M \cup N$ gesetzt wird, während die andere Menge entfernt wird. Dies sollte man natürlich in dieser Weise nicht implementieren, da das Löschen zu viel Aufwand erfordern würde. Vielmehr fasst man die beiden Mengen zusammen und vergisst deren alte Bedeutung.

11.3.5 Der Kruskal-Algorithmus braucht zwei Operationen:

FIND (x) = Stelle die Menge fest, in der das Element x liegt.

UNION (M, N) = Vereinige die beiden Mengen M und N, nenne das Ergebnis wieder M und vergiss N.

Hiermit lautet der Kruskal-Algorithmus:

```
Sortiere alle Kanten nach ihrem δ-Wert in eine Liste L;
for all y ∈ V do M_y := {y} od;  BK := leere Liste;
while not isempty(L) and (|BK| < n-1) do
   e := DELETEMIN(L);    -- x und y seien die Endknoten von e;
   M := FIND(x); N := FIND(y);
   if  M ≠ N  then  UNION(M, N); füge e zu BK hinzu  fi
od;
```


Das sog. UNION-FIND-Problem fragt nach einer effizienten Implementierung der Operationen UNION und FIND unter der Nebenbedingung, dass je O(n) dieser beiden Operationen auf einer Gesamtmenge von n Elementen durchgeführt werden müssen.

Als gute Implementierung gilt die Darstellung jeder Menge als Baum, dessen Kanten von jedem Knoten zu seinem Vorgänger gerichtet sind und in dessen Wurzel zusätzlich die Anzahl der Elemente dieses Baums vermerkt ist (s.u.). Dann lässt sich

FIND(x) als Bestimmung der Wurzel des Baums, in dem sich x befindet, und

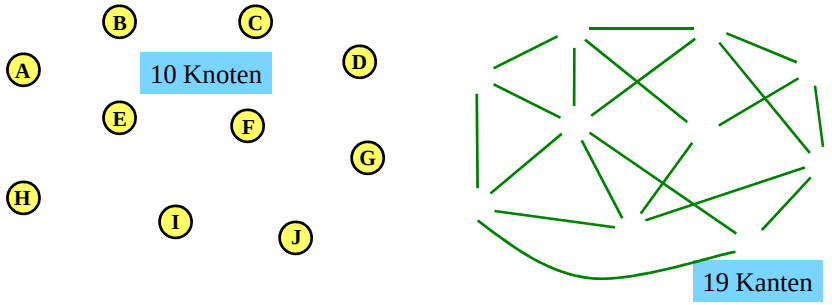
UNION(M,N) als Anhängen des kleineren Baums an die Wurzel des größeren Baums realisieren.

FIND(x) ≠ FIND(y) läuft dann auf die einfache Abfrage, ob beide Wurzeln der gleiche Knoten sind, hinaus.

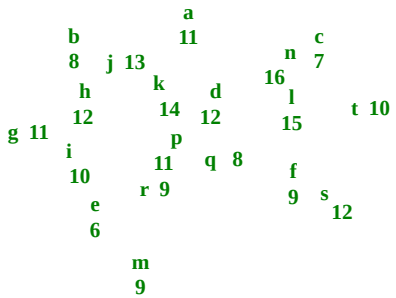
Mit dieser Darstellung benötigt jedes FIND (x) höchstens log(n) Vergleiche (warum? Beachten Sie, dass stets der kleinere an den größeren Baum gehängt wird) und UNION kann in konstant vielen Schritten ausgeführt werden. Ohne das Sortieren der Kanten benötigt der Kruskal-Algorithmus also höchstens O(m·log(n)) viele Schritte, weil die while-Schleife bis zu m Mal durchlaufen wird und jedes FIND höchstens log(n) Schritte erfordert.

Nach Kapitel 10 kostet das Sortieren der m Kanten O(m·log(m)) Schritte. In der Praxis ist n in der Regel kleiner als m, daher wird die Komplexität des Kruskal-Algorithmus durch das Sortieren der m Kanten dominiert. Insgesamt ergibt sich in dem normalen Fall n ≤ m somit eine Zeitkomplexität von

$$O(m \cdot \log(m) + m \cdot \log(n)) = O(m \cdot \log(m)).$$

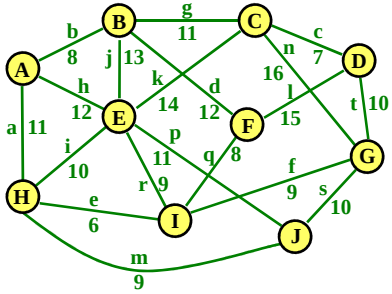


11.3.6 Erläuterung der Datenstrukturen am Beispiel.



Benennung der 19 Kanten und Gewichte $\delta : E \to \mathbb{R}$

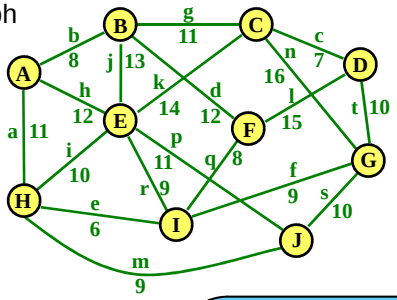
Der Graph G = (V, E, δ: E $\rightarrow$ **IR**):



Erste Aufgabe: Bilde die sortierte Liste der Kanten. Ergebnis:

(e,6), (c,7), (b,8), (q,8), (f,9), (m,9), (r,9), (i,10), (s,10), (t,10), (a,11), (g,11), (p,11), (d,12), (h,12), (j,13), (k,14), (l,15), (n,16).

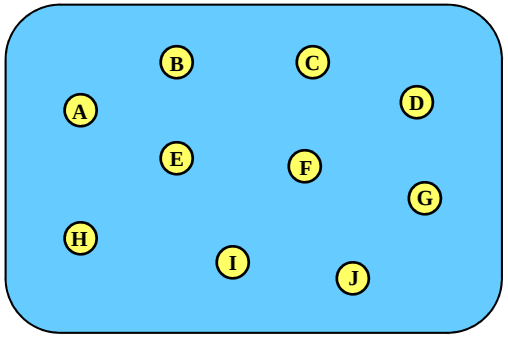
Graph



Sortierte Kantenliste

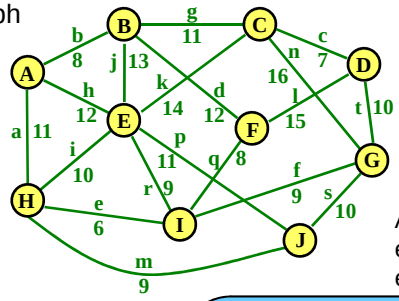
(e,6),
(c,7),
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

"Arbeitsfläche":
Ein Wald isolierter Knoten



Bilde nun die einelementigen Mengen M_y für jeden Knoten y:

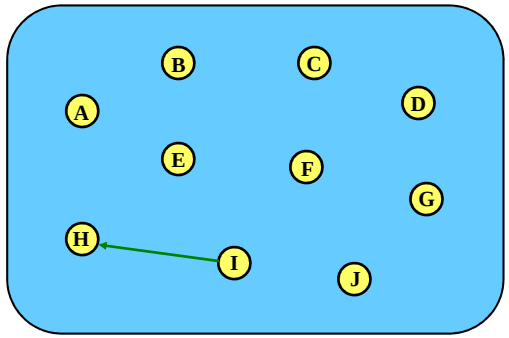
Graph



Sortierte Kantenliste

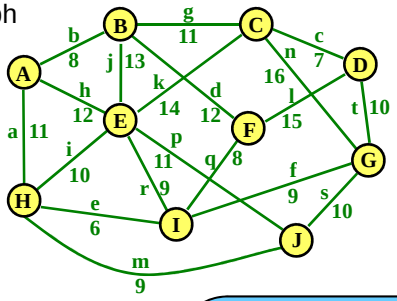
(e,6),
(c,7),
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Aus zwei Knoten (= einelementige Bäume) wird ein neuer Baum gebildet.



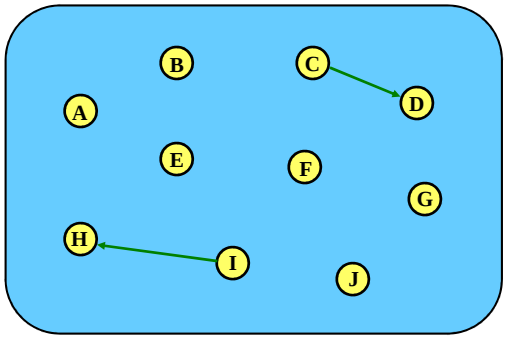
Betrachte die erste Kante der sortierten Liste, also **e**:
(Die Ausrichtung der Kanten ist bei gleich großen Mengen willkürlich, ansonsten wird stets der kleinere an den größeren Baum gehängt.)

Graph



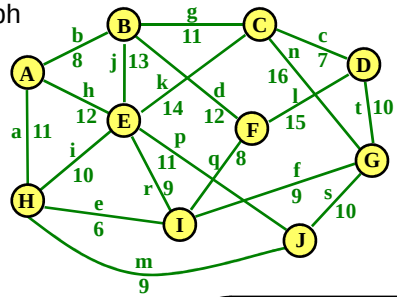
Sortierte Kantenliste

(e,6), +
(c,7),
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



Betrachte nun die zweite Kante der sortierten Liste, also **c**:
(Die Ausrichtung der Kanten ist bei gleich großen Mengen willkürlich, ansonsten wird stets der kleinere an den größeren Baum gehängt.)

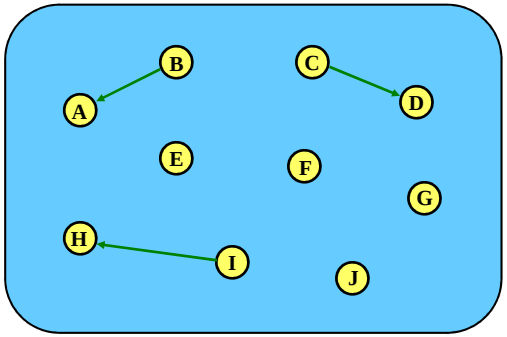
Graph



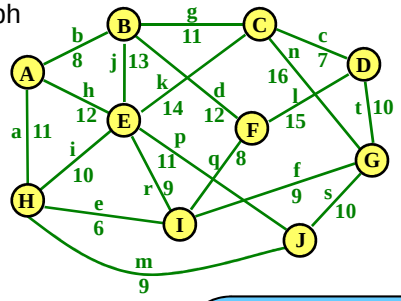
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die dritte Kante der sortierten Liste, also **b**:



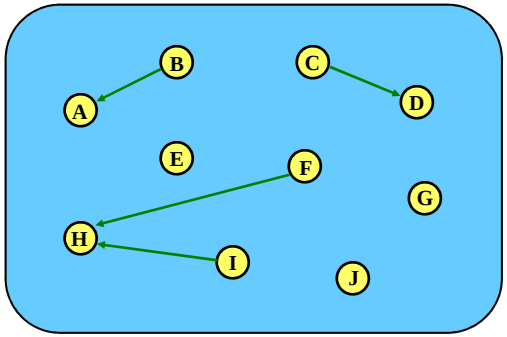
Graph



Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die vierte Kante **q**. Hänge den kleineren Baum "F" an den größeren, der zu I gehört.

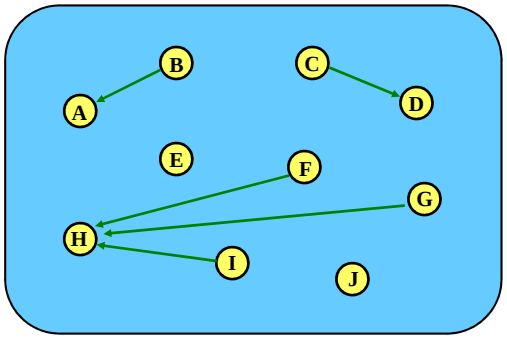


Graph

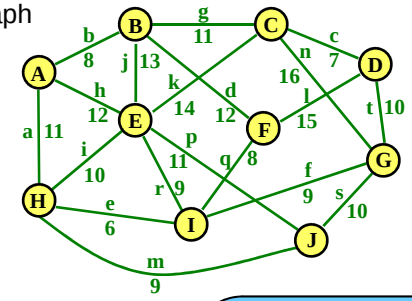
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die fünfte Kante **f**. Hänge den kleineren Baum "G" an den größeren, der zu I gehört.



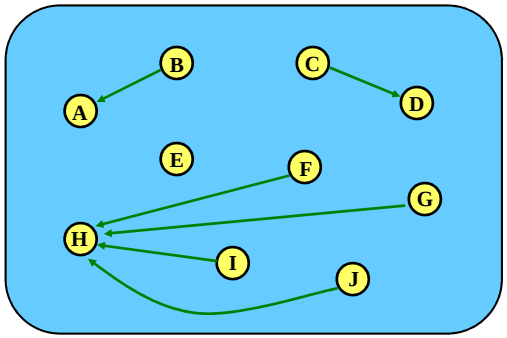
Graph



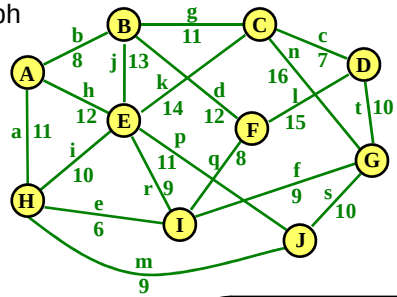
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die sechste Kante **m**. Hänge den kleineren Baum "J" an den größeren, der zu H gehört.



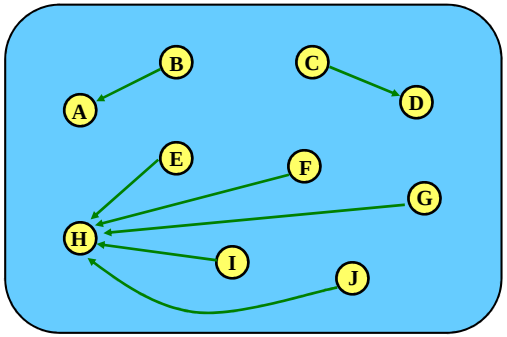
Graph



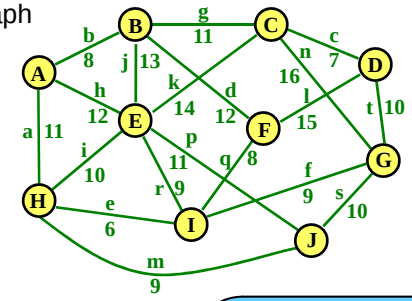
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die siebente Kante **r**. Hänge den kleineren Baum "E" an den größeren, der zu I gehört.



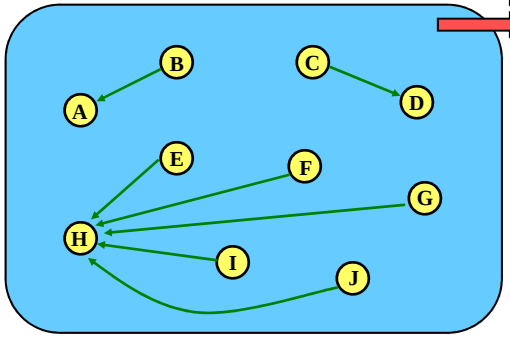
Graph



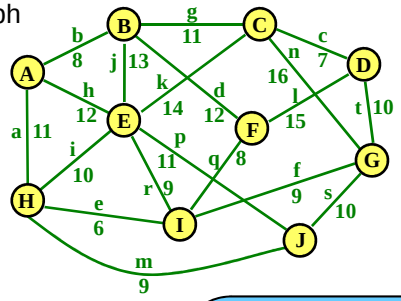
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die achte und neunte Kante. Jedes Mal trifft man auf "H", d.h., diese Kanten führen zu Zyklen.



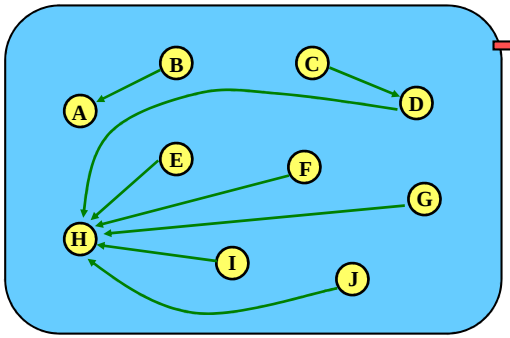
Graph



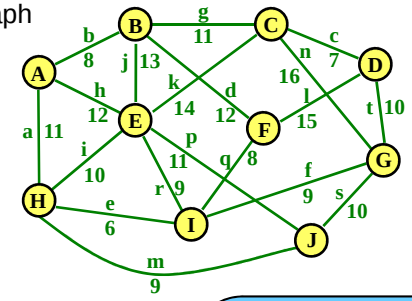
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die zehnte Kante **t**. Nun wird der Baum "D" an den größeren Baum "H" gehängt.



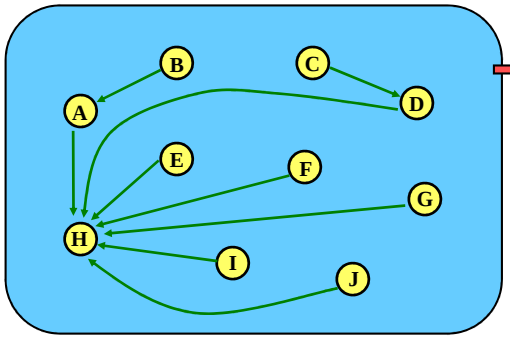
Graph



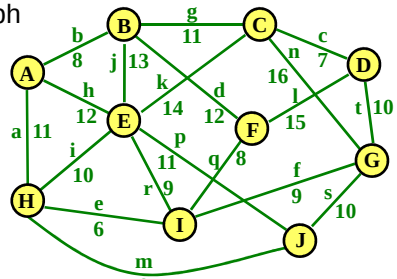
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10), +
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte jetzt die elfte Kante **a**. Nun wird der Baum "A" an den Baum "H" gehängt - und *wir sind fertig*, da n-1 Kanten gefunden wurden.



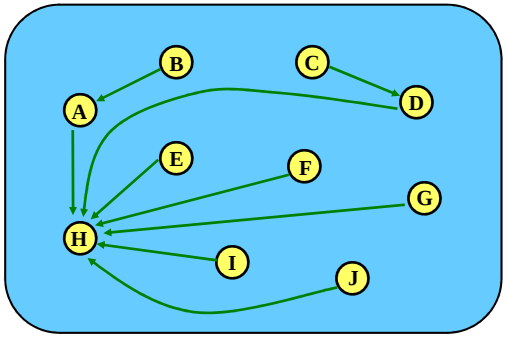
Graph



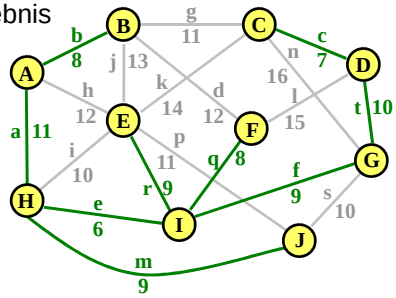
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10), +
(a,11), +
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Ergebnis: Die mit "+" markierten 9 Kanten **e, c, b, q, f, m, r, t** und **a** bilden einen minimalen Spannbaum.



Ergebnis



Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10), +
(a,11), +
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Ergebnis: Die mit "+" markierten 9 Kanten **e, c, b, q, f, m, r, t** und **a** bilden einen minimalen Spannbaum.

Zugleich wird bestätigt, dass der Graph zusammenhängend ist.

Ende des Beispiels.

Hinweis:
"Verflacht" man die Bäume bei jeder FIND-Operation (sog. "Pfadkompression": alle besuchten Knoten werden direkt an die Wurzel des Baums gehängt), dann braucht man für alle UNION- und FIND-Operationen nur $O(m \cdot \log^*(n))$ Schritte. (Siehe Vorlesungen zur Algorithmik im weiteren Studium oder siehe Literatur; log* wurde in 6.1.4 und 6.8 behandelt.)
Die Zeitkomplexität wird hierdurch allerdings nicht verringert, da das Sortieren der Kanten unverändert $O(m \cdot \log(m))$ Schritte erfordert.
[Wegen $m \leq n^2$ gilt $\log(m) \leq \log(n^2) = 2 \cdot \log(n)$. Man kann also statt $O(m \cdot \log(m))$ auch stets $O(m \cdot \log(n))$ schreiben.]

11.4 Weitere Standardalgorithmen auf Graphen

Maximales Matching: Das Problem, zu einem sog. bipartiten (s.u.) Graphen ein maximales Matching zu finden, wird in der Literatur als Heiratsproblem bezeichnet. Es wurde 1935 von dem englischen Mathematiker Philip Hall charakterisiert.

Das Heiratsproblem lautet: Gegeben sind zwei disjunkte Mengen D und H ('Damen' und 'Herren') und eine Relation $E \subseteq \{\{x,y\} \mid x \in D \text{ und } y \in H\}$. Gesucht ist eine möglichst große Teilmenge F von E, in der jedes Element von D und von H höchstens einmal vorkommt. Die Relation E wird als "wechselseitig sympathisch" und die Menge F als Menge von Hochzeiten aufgefasst.
Wir definieren dies nun exakt.

Definition 11.4.1:
$G=(V,E)$ sei ein ungerichteter Graph.
a) Eine Teilmenge der Kanten $F \subseteq E$ heißt Matching, wenn je zwei verschiedene Kanten von F disjunkt sind, d. h., für alle $\{x,y\}, \{x',y'\} \in F$ gilt:

aus $\{x,y\} \neq \{x',y'\}$ folgt $\{x,y\} \cap \{x',y'\} = \emptyset$.

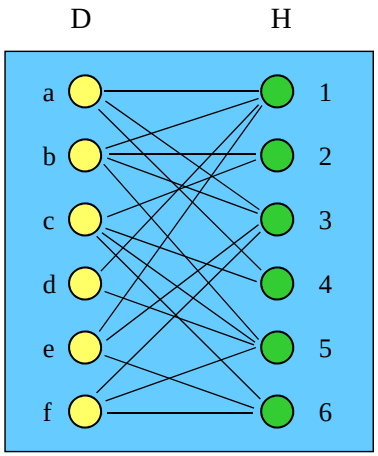
b) Ein Matching $F \subseteq E$ heißt nicht vergrößerbar, wenn es keine Kante $e \in E\text{-}F$ gibt, so dass $F \cup \{e\}$ ein Matching ist.

c) Ein Matching $F \subseteq E$ heißt maximal, wenn es kein Matching $F' \subseteq E$ mit $|F| < |F'|$ gibt.

Man könnte meinen, das Problem, ein maximales Matching zu finden, sei einfach: Ausgehend von der leeren Menge nehme man schrittweise immer wieder eine Kante hinzu, die zwei noch nicht ausgewählte Knoten miteinander verbindet. Dies führt in der Regel jedoch nicht zu einem maximalen Matching, sondern nur zu einem nicht vergrößerbaren Matching (selbst ein Beispiel konstruieren!).

Definition 11.4.2: Ein ungerichteter Graph $G=(V,E)$ heißt bipartit, wenn seine Knotenmenge V so in zwei disjunkte Teilmengen D und H zerlegt werden kann, dass Kanten nur von einer zur anderen Teilmenge führen, d.h.,
$V = D \cup H$, $D \cap H = \emptyset$ und $E \subseteq \{\{x,y\} \mid x \in D \text{ und } y \in H\}$.
Man schreibt dann auch $G=(D \cup H, E)$ oder $G=(D,H;E)$.

Beispiel: $G = (D \cup H, E)$, $D = \{a,b,c,d,e,f\}$, $H = \{1,2,3,4,5,6\}$.
Illustration der Idee, ein maximales Matching zu konstruieren.

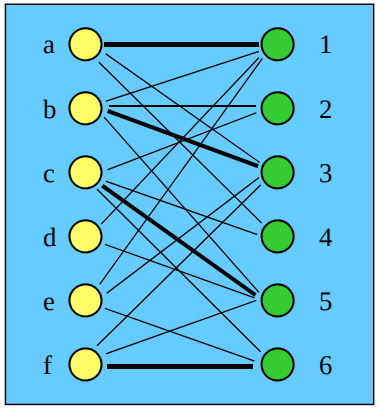


Beispiele für Matchings:

$F_1 = \{\{a,1\}, \{b,5\}, \{e,3\}\}$
$F_2 = \{\{b,1\}, \{c,4\}, \{e,1\}, \{f,6\}\}$
$F_3 = \{\{a,3\}, \{b,1\}, \{c,6\}, \{d,5\}\}$
$F_4 = \{\{a,1\}, \{b,3\}, \{c,5\}, \{f,6\}\}$

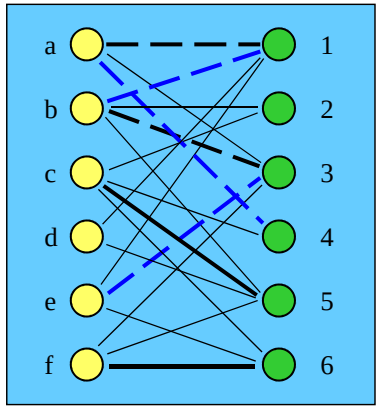
F_3 und F_4 sind nicht vergrößerbar.

Ist eines dieser Matchings maximal?
Nein. Siehe folgende Seite.



Nicht vergrößerbares Matching F_4 = {{a,1}, {b,3}, {c,5}, {f,6}}
Suche einen erweiterbaren Weg.

Erweiterbarer Weg beginnend in e und endend in 4:
{e,3}, {b,3}, {b,1}, {a,1}, {a,4}.
Ersetze schwarz gestrichelte Kanten durch die blau gestrichelten Kanten.
Dies führt zum Matching F_5 = {{c,5}, {f,6}, {e,3}, {b,1}, {a,4}}.



Ist dieses Matching maximal?

11.4.3 Maximales Matching in bipartiten Graphen

Vorgehensweise: Man baue zunächst ein nicht vergrößerbares Matching auf, indem man immer erneut eine Kante, die keine Knoten mit den bisher ausgewählten Kanten gemeinsam hat, hinzu nimmt.

Es gilt folgender Satz: Wenn ein Matching nicht maximal ist, so gibt es stets einen erweiterbaren (oder erhöhenden) Weg, englisch: "augmented path".
Ein erweiterbarer Weg $(x_0, x_1, x_2, ..., x_{k-1}, x_k)$ zu einem Matching $F \subseteq E$ in einem bipartiten Graphen $G = (D \cup H, E)$ ist ein Weg mit $x_0 \in X$, $x_k \notin X$ und abwechselnd Kanten aus E-F und aus F: $\{x_0, x_1\} \in$ E-F, $\{x_1, x_2\} \in F$, $\{x_2, x_3\} \in$ E-F, $\{x_3, x_4\} \in F$, ..., $\{x_{k-2}, x_{k-1}\} \in F$, $\{x_{k-1}, x_k\} \in$ E-F, wobei X entweder D oder H ist und kein Knoten zweimal auf dem Weg vorkommt.

In einem bipartiten Graphen kann man einen erweiterbaren Weg in O(m) Schritten finden, indem man ausgehend von einem Knoten x_0, welcher noch zu keiner Kante des Matchings F gehört, mit einem Tiefendurchlauf alle Wege durchprobiert, allerdings nur solche, deren Kanten abwechselnd zu E-F und zu F gehören. Trifft man dabei auf einen Knoten x_k, der ebenfalls zu keiner Kante des Matchings F gehört, so hat man einen erweiterbaren Weg gefunden.

Dann bilde man aus F das um eine Kante größere Matching F', indem man alle Kanten $(x_1, x_2) \in F, (x_3, x_4) \in F, (x_5, x_6) \in F, ...$ aus F entfernt und durch $(x_0, x_1), (x_2, x_3), (x_4, x_6), ..., (x_{k-1}, x_k)$ ersetzt. Findet man dagegen von x_0 ausgehend keinen solchen Weg, dann braucht man x_0 nicht weiter zu betrachten, d.h., man darf x_0 streichen und mit einem anderen Knoten fortfahren. Da man dies höchstens für alle n Knoten prüfen muss, ergibt sich eine Zeitkomplexität von $O(n \cdot m)$.

Bei diesem Vorgehen wirft man jedes Mal die Informationen, die man beim Tiefendurchlauf ermittelt hat, weg und startet mit einem neuen x_0 wieder von vorne. Merkt man sich aber gewisse Informationen aus den vorherigen Durchläufen, so lässt sich die Laufzeit auf $O(m \cdot \sqrt{n})$ verkürzen.

11.4.4 Maximales Matching in beliebigen Graphen

Die Lösungsidee für bipartite Graphen lässt sich auf beliebige ungerichtete Graphen erweitern. Hier verweisen wir auf die Literatur oder auf Vorlesungen in den höheren Semestern.

11.4.5 Hinweise zu weiteren Graphalgorithmen

Beispiele von Problemen:
Zusammenhangskomponenten in Linearzeit ermitteln
Maximale Flüsse in einem Graphen-Netzwerk berechnen
(Stichwort: MaxFlow = MinCut)
Färbbarkeit von Graphen
Größte vollständige Teilgraphen (Cliquenproblem)
zweit-, dritt-, ..., k-t kürzeste Wege berechnen
Baumisomorphie (gerichtet / ungerichtet)
Graphisomorphie und Teilgraph-Isomorphie

Eine Informatik-Anwendung: Schicke maximal viele Datenpakete durch ein Netzwerk. Wie hoch ist der Fluss? Kann man eine feste Empfangszeit garantieren? Wie hängt dies von der Zahl der Empfänger ab? ....

12. Verwaltung von Datenstrukturen

12.1 Keller und Halde

12.1.1 Überblick über die wichtigsten Speicher: Wie in 3.4.3 vorgestellt benötigen Programme mindestens drei Typen von Speichern:

1. Zur Übersetzungszeit bekannter ("statischer") Speicher.
Am Ende der Übersetzung des Programms liegen fest:
1.1: Der Platz, den das übersetzte Programm braucht.
1.2: Der Platz für alle Konstanten und für alle Variablen, die zu einem Datentyp gehören, dessen Speicherplatzbedarf von vornherein feststeht, z.B.: elementarer Datentyp, Aufzählungstyp, Unterbereiche hiervon, records, die nur aus solchen Komponenten bestehen, Felder hierüber von konstanter Länge, access-Datentypen (nur der Zeiger, nicht die hiermit aufgebaute Liste).

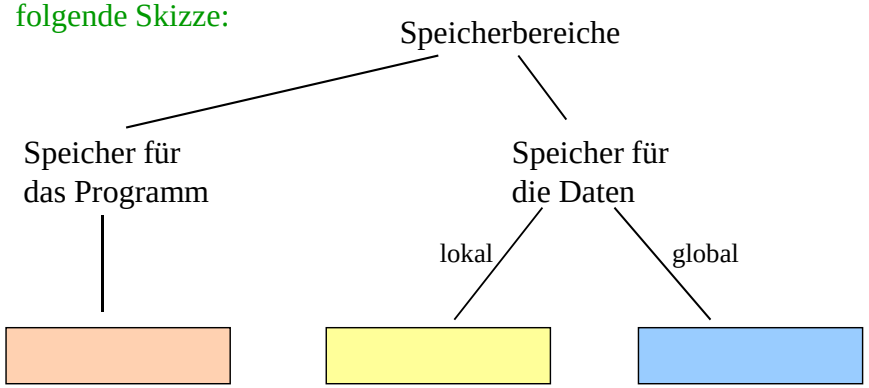
2. Zur Laufzeit beim Abarbeiten der Deklarationen weiterhin erforderlicher dynamischer Speicher (Kellerspeicher).

Viele Variablen, die in klammerartig ineinander geschachtelten Blockstrukturen oder in aufgerufenen Unterprogrammen oder anderen Einheiten stehen oder die parametrisiert sind (dynamische Felder, Records mit Diskriminanten), erhalten ihren Speicherplatz erst zur Laufzeit zugewiesen. Der Platz für solche Objekte wird beim Erreichen der zugehörigen Deklarationen hinten an den zum Programm gehörenden (lokalen) Speicher angehängt und er wird am Ende ihrer Lebensdauer wieder frei gegeben. Wegen der Klammerstruktur, in die die Deklarationen eingebunden sind, ist dieser dynamische Speicher ein Kellerspeicher (Pushdown, Keller), siehe 4.1.4.

Kellerspeicher in der Praxis:
In der Regel legt man auch die Daten, die zum statischen Speicher zählen, in diesem Kellerspeicher (ganz am Anfang) ab, wodurch man die Deklarationen zur Laufzeit einheitlich abarbeiten kann.

3. Zur Laufzeit durch Anweisungen erzeugte Daten, deren Lebensdauer sich nicht an den Programmeinheiten orientiert.
Die Speicherplätze für solche dynamisch erzeugten Daten werden mittels new angefordert und zugeordnet ("allokiert"). Dieser Speicher pulsiert nicht kellerartig, daher liegen diese Speicherplätze in einem allgemeinen Speicherbereich, der Halde. Die Halde kann von vielen Programmen gleichzeitig genutzt werden. Die Speicherplätze in der Halde werden meist explizit freigegeben, sobald die Daten nicht mehr gebraucht werden, oder sie werden mit einer Speicherbereinigung entfernt, sobald die Halde überzulaufen droht.

So erhält man folgende Skizze:



Programmspeicher (für jedes Programm gesondert): Wird beim Ablauf des Programmes nicht verändert (wie bei Turing- und Registermaschine).

Keller (für jedes Programm gesondert): Pulsiert mit dem Wechsel zwischen Blöcken, Prozeduren und weiteren Einheiten.

Halde: In einem großen Speicherbereich werden referenzierte Daten abgelegt. Im Allgemeinen für viele Programme. Wird häufig aktualisiert.

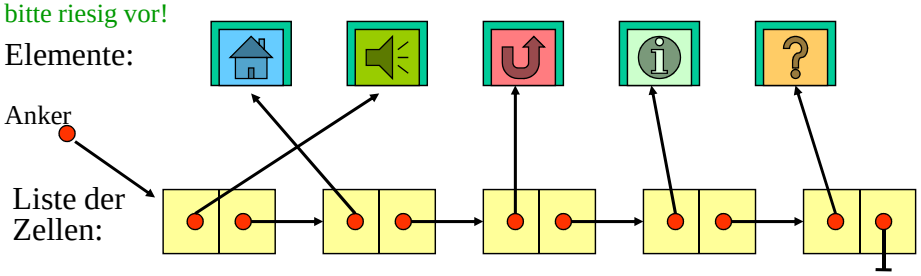
12.1.2 Erinnerung an Pointer/Zeiger/Listen:

Listen sind Folgen von Elementen des gleichen Datentyps. Sie werden durch *Zeiger (pointer, access-Datentypen)* realisiert. Die Verkettung kann einfach oder doppelt, die Anordnung sequenziell oder ringförmig sein. Das erste und/oder das letzte Element sind von außen über einen Zeiger erreichbar (auch *Anker* der Liste genannt). Der Zugriff erfolgt *sequenziell*; man durchläuft also die Liste von vorne nach hinten bzw. von hinten nach vorne, um nach einem Element zu suchen oder um ein Element einzufügen. Die mit einer Liste üblicherweise verbundenen Operationen sind in 3.5 aufgeführt.

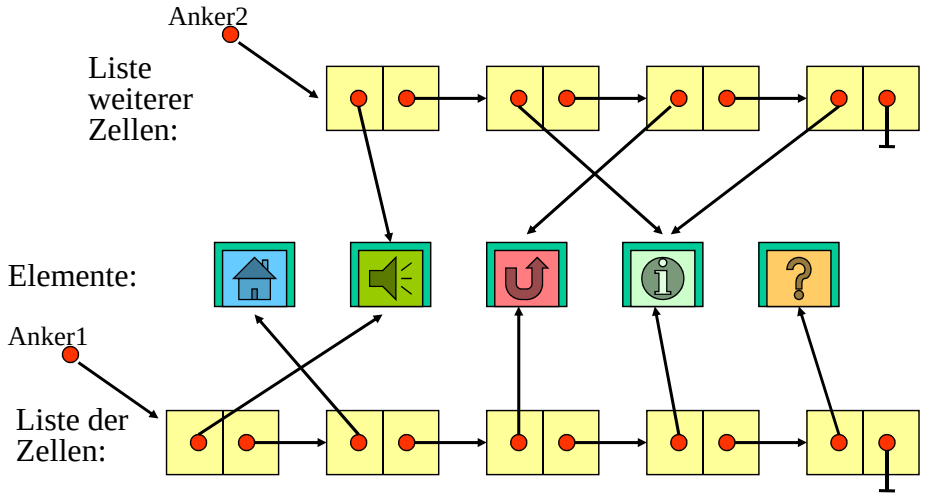
Das Schlüsselwort für Zeiger lautet in Ada access. Die Elemente einer Liste speichert man üblicherweise nur einmal und setzt in die Liste einen Zeiger auf sie. Also:


```
type Element is ....;        -- Definiere den Datentyp der Listenelemente
type Element_Zeiger is access Element;
type Zelle;                  -- Vorwärtsverweis
type Zelle_Zeiger is access Zelle;
type Zelle is   record
                    Inhalt: Element_Zeiger;
                    Next: Zelle_Zeiger;
                end record;
```

Stellen Sie sich diese Elemente bitte riesig vor!



Will man die Elemente mehrfach in einer Liste oder in mehreren Listen gleichzeitig verwenden, so braucht man keine Kopien der Elemente anzulegen; es genügen Zeiger:



Vorteile dieser Zellen-Darstellung:

- Elemente können in verschiedenen Listen sein.
- Elemente werden in allen Listen gleichzeitig geändert (da nur das Original geändert werden muss).
- Das Einfügen in andere Listen ist einfach.
- Es lassen sich weitere Zugriffsstrukturen leicht aufbauen.

Nachteile dieser Zellen-Darstellung:

- Der Zugriff auf Elemente dauert evtl. etwas länger.
- Man braucht evtl. mehr Speicherplatz (als z.B. mit arrays).
- Es muss die Halde als Speicher verwaltet werden (meist keine direkte Kontrolle über die dortigen Abläufe).

Hinweis: Listen werden in der Halde abgelegt. Nur die Anker stehen im statischen Bereich oder lokalen Keller des Programms, sofern sie deklarierte Variablen sind.

12.2 Kellerverwaltung

Hiermit ist nicht die recht einfache Verwaltung eines einzelnen Kellers (3.5.4) gemeint, sondern die Verwaltung vieler Keller in einem beschränkten linearen gemeinsamen Speicherbereich.

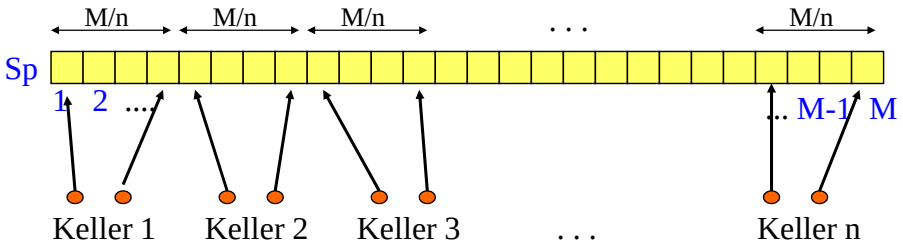
Im Computer werden viele Programme ("jobs") im gleichen Zeitraum bearbeitet. Wie kann man deren lokale Keller, die einander den Speicherplatz streitig machen, gemeinsam verwalten?.

Alle Keller liegen in einem linear angeordneten Speicherbereich Sp(1..M). Manche pulsieren stark, andere nur wenig. In der Regel wird der Gesamt-Speicherplatz nicht überschritten, aber wenn man jedem Programm einen festen Bereich zuweisen würde, so wird es oft einen Speicherüberlauf geben. Daher muss man den Gesamt-Speicherplatz geeignet auf die jeweils laufenden Programme verteilen. Allgemeine Formulierung des Problems:

12.2.1: Implementierung von mehreren Kellern

Aufgabe: Wir wollen eine ***Multikellerverwaltung*** in einem eindimensionalen Feld durchführen, d.h.:
Es sollen n Keller verwaltet werden. Insgesamt steht hierfür ein linearer Speicher Sp(1..M) zur Verfügung.

Spontane Idee: Jeder Keller erhält gleich viele Speicherplätze, nämlich M/n:



Diese Verweise werden wir durch Indizes realisieren.

Einfachster Fall: **n = 1**. Es liegt ein einzelner Keller vor. Die Ada-Formulierung hierfür ist ein generisches Paket, z.B.:

```
generic
   M: Positive := 5_000_000;          -- Initialisierung willkürlich
   type Element is private;
package Keller is
   procedure newkeller;               -- Leeren des Kellers
   function isempty return Boolean;   -- Ist der Keller leer?
   function isfull return Boolean;    -- Ist der Keller voll?
   function top return Element;       -- Oberstes Kellerelement
   procedure push (x: in Element);    -- Füge Element x oben an
   procedure pop;                     -- Lösche oberstes Element
   function length return Natural;    -- Aktuelle Kellerlänge
   unterlauf, ueberlauf: exception;   -- Ausnahmebehandlung
end Keller;
```

Hieran schließt sich der Modulrumpf an, siehe 4.3.3 und 4.3.4:

```
package body Keller is
   type Speicher is array (1..M) of Element;
   Sp: Speicher;
   index: Integer range 0..M := 0;
   procedure newkeller is begin index := 0; end;
   function isfull return Boolean is
      begin return index >= M; end;
   procedure push (x: in Element) is
      begin  if isfull then raise ueberlauf;
             else index := index + 1; Sp(index) := x; end if; end;
   ...
   < selbst schreiben: die Prozedur pop, die Funktionen isempty,
   length, top und die Ausnahmen unterlauf und ueberlauf >
end Keller;
```


Eine Instanz kann nun lauten:

```
package Ganzzahlkeller is
        new Keller (M => 800_000, Element => Integer);
```

Nächster Fall: **n = 2**. Wenn man zwei Keller auf einem linearen Speicher Sp der Größe 1 .. M unterbringen möchte, so wird man den ersten Keller von 1 an aufwärts und den zweiten Keller mit M beginnend abwärts implementieren. Eingegriffen muss werden, sobald der eine Keller auf den Speicherplatz des anderen zugreifen will, d.h., wenn der obere Index des ersten Kellers größer als der des zweiten Kellers wird. Auf diese Weise wird Sp optimal genutzt.

Aufgabe: Realisieren Sie diesen einfachen Fall selbst!

12.2.2 Allgemeiner Fall: **n ≥ 3**. Vorhandener Speicher Sp(1..M). Hier gibt es mehrere Varianten:

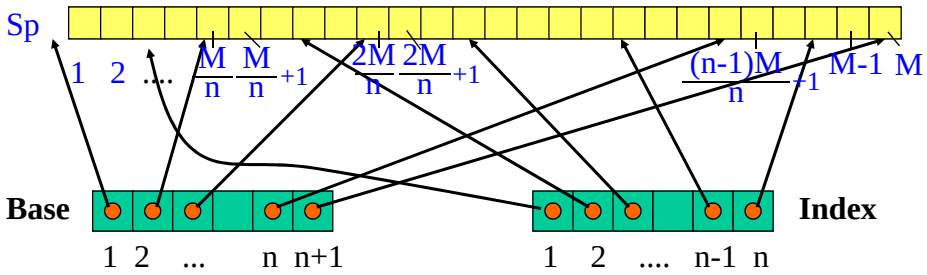
- *Variante 1:* Jeder Keller hat seine eigene maximale Größe, die in Max: array (1..n) of Natural abgelegt ist (einfachster Fall: Max(i) = ⌊M/n⌋ für alle i) und für die gilt

$$\sum_{i=1}^{n} \text{Max}(i) \le M.$$

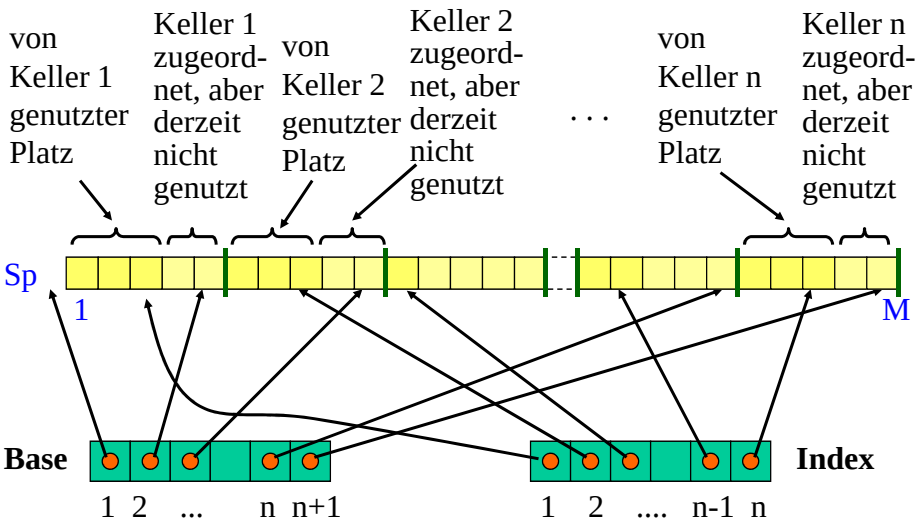
Dieser Fall wird wie "n=1" behandelt, indem überall die Nummer des Kellers hinzugefügt wird und jeder Keller unabhängig von den anderen ist. Es gibt dann zwei Felder für die Adresse vor dem Beginn des i-ten Kellers ("Base") und für seine aktuelle oberste Position ("Index") mit $0 \le \text{Index}(i) - \text{Base}(i) \le \text{Max}(i)$ für i = 1, 2, ..., n.
Wir formulieren dies nun genau aus.

Implementierung: Jeder Keller erhält den gleichen Platz der Größe M/n. Wir verwenden hierfür zwei Zeiger bzw. Indizes:

Base(i) zeigt auf den Speicherplatz, der unmittelbar vor dem Bereich für den i-ten Keller liegt;
Index(i) zeigt auf den Speicherplatz, auf dem sich das oberste Element des i-ten Kellers befindet.



Nochmals zur Illustration: Aufteilung des Speichers Sp



Die vom jeweiligen Keller nicht genutzten Speicherbereiche sind farblich etwas heller unterlegt; die dicken Striche markieren das Ende eines Kellers.

12.2.3 Ada Deklarationen hierzu: (MKV = Multikellerverwaltung)

```
generic
   M: Positive := 5_000_000;          -- willkürlicher Default-Wert
   n: Positive;                       -- Anzahl der Keller, n ≥ 2
   type Element is private;           -- Datentyp der Kellerelemente
package MKV1 is
   type ZK is Positive range (1..n+1);          -- für Zugriff auf Keller
   procedure newkeller (i: in ZK);              -- Leeren des Kellers i
   function isempty (i: ZK) return Boolean;     -- Ist der Keller i leer?
   function isfull (i: ZK) return Boolean;      -- Ist der Keller i voll?
   function top (i: ZK) return Element;         -- Oberstes Kellerelement
   procedure push (i: in ZK; x: in Element);    -- Füge x oben an
   procedure pop (i: in ZK);                    -- Lösche oberstes Element
   function length (i: in ZK) return Natural;   -- Aktuelle Kellerlänge
   unterlauf (i: ZK), ueberlauf (i: ZK): exception;
end MKV1;
```


Paketrumpf hierzu: (MKV = Multikellerverwaltung)

```
package body MKV1 is
type Adressen is Natural range 0..M;   -- Speicher“adressen“
Sp: array (Adressen) of Element;       -- Speicher
Base: array (ZK) of Adressen;          -- Beginn jedes Kellers (minus 1)
Index: array (ZK) of Adressen;         -- Aktueller Stand jedes Kellers
procedure newkeller (i: in ZK) is
   begin  Index(i) := Base(i);  end newkeller;
function isempty (i: ZK) return Boolean is
  begin  return  Base(i) = Index(i);  end isempty;
function isfull (i: ZK) return Boolean is
  begin  return  Base(i+1) >= Index(i);  end isfull;
function top (i: ZK) return Element is
  begin  return Sp(Index(i));  end top;
```


Paketrumpf MKV1 (Fortsetzung)

```
procedure push (i: in ZK; x: in Element) is
  begin  if isfull(i) then raise ueberlauf (i);
         else Index(i) := Index(i) + 1;
              Sp(Index(i)) := x; end if;
  end push;
procedure pop (i: in ZK) is
  begin  if isempty(i) then raise unterlauf(i);
         else Index(i) := Index(i) - 1; end if; end pop;
function length (i: ZK) return Natural is
  begin return Index(i)-Base(i); end length;
exception   when  ... => .....
end MKV1;
```

Eine konkrete Instanz für zehn ganzzahlige Keller könnte dann sein (hier wird der voreingestellte Wert von M genommen):

```
package Zahlenkeller is new MKV1(n => 10; Element => Integer);
use Zahlenkeller; ...
for i in 1..n loop Base(i) := (i-1)*(M/n); Index(i):=Base(i); end loop;
Base(n+1) := M; ...
```

Nachteilig ist, dass die Multikellerverwaltung zusammenbricht, falls irgendein Keller überläuft. In der Regel stehen ja noch weitere Speicherplätze in Sp zur Verfügung.

Es gibt diverse nahe liegende Veränderungen. Diese ersetzen alle „raise ueberlauf(i)“ durch den Prozeduraufruf „umordnen(i)“, um weiteren Speicherplatz bereitzustellen (siehe unten in Variante 2):

```
procedure umordnen (i: in ZK);  ...
```


12.2.4 *Variante 2:* Die Größe jedes einzelnen Kellers ist nicht vorab beschränkt und alle Keller zusammen sollen den Speicherplatz der Größe M möglichst gut nutzen. Hierfür muss es wiederum zwei Felder Base, Index: array (1..n) of Natural geben, für die zu jedem Zeitpunkt gilt

$$\sum_{i=1}^{n} \text{Index}(i)-\text{Base}(i) \le M.$$

Wenn einer der Keller überläuft (d.h. push-Operation bei Index(i) = Base(i+1)) und wenn zugleich andere Keller den ihnen zugewiesenen Bereich noch nicht voll ausnutzen, *so muss der Speicherplatz neu auf die Keller verteilt werden.*

Hier sind mehrere Untervarianten möglich.

Möglichkeit 1:
Schaue nach, ob der rechte oder linke Nachbar des Kellers i noch genügend freien Platz hat und tritt dann die Hälfte dieser Plätze an den Keller i ab.

Möglichkeit 2:
Suche einen Keller j mit maximal viel freiem Platz, das heißt, Index(j)-Base(j) ist maximal, und tritt die Hälfte dieser Plätze an den Keller i ab. Konkret muss dann der Speicherbereich zwischen den Kellern i und j um q Speicherplätze verschoben werden, wobei q die Hälfte der freien Plätze von Keller j ist.

Möglichkeit 3:
Berechne den Speicherplatz, den jeder Keller bekommen soll, neu, indem jedem Keller eine Mindestzahl an Plätzen und weitere Plätze entsprechend seines bisherigen Wachstums zugewiesen werden, und ordne den Speicher dann komplett um.

Möglichkeit 1: (nur die benachbarten Keller i-1 und i+1 betrachten)

```
procedure umordnen (i: in ZK) is              -- n muss mindestens 3 sein
   k: ZK;  q: Adressen;        -- der Typ Adressen ist Natural range 0..M
begin  k := i;     -- wir testen auch, ob im Keller k mindestens 2 Plätze frei sind
   if (i = 1) and (Index(2) + 1 < Base(3)) then k := 2;
   elsif (i = n) and (Index(n-1) + 1 < Base(n)) then k := n-1;
   elsif Base(i) - Index(i-1) + 1 < Base(i+2) - Index(i+1)
      then k := i+1;
      elsif Index(i-1) + 1 < Base(i) then k := i-1; end if;
                -- Keller k dient nun als Platz-Lieferant, sofern k≠i
   if k = i then raise ueberlauf;
   elsif k < i then                          -- Speicher nach links verschieben
      q := (Base(k+1) - Index(k))/2;       -- Hälfte des freien Platzes
      Base(i) := Base(i) - q; Index(i) := Index(i) - q;
      for j in Base(i)..Index(i) loop Sp(j) := Sp(j+q); end loop;
   else <das Gleiche, aber nach rechts verschieben; selbst einfügen> end if;
end umordnen;
```

Möglichkeit 2: (Keller k, der maximal viel Platz abgeben kann, suchen)

```
procedure umordnen (i: in ZK) is
    k: ZK;  q: Adressen;
begin  k := 1;                    -- Suche Keller k mit maximal freiem Platz
    for j in 2..n loop
        if (Base(j+1) - Index(j)) > (Base(k+1) - Index(k))
        then k := j; end if;
    end loop;
    q := (Base(k+1)-Index(k))/2;        -- Hälfte des freien Platzes
    if q <= 0 then raise ueberlauf;
    elsif k < i then        -- Speicher der Keller k+1..i nach links verschieben
        for j in k+1..i loop
            Base(j) := Base(j) - q; Index(j) := Index(j) - q; end loop;
        for j in Base(k+1)..Index(i) loop Sp(j) := Sp(j+q); end loop;
    else  < das Gleiche, nur nach rechts verschieben; selbst einfügen > end if;
end umordnen;           -- hier werden auch die zurzeit nicht genutzten Teile
                        -- mitverschoben, was man vermeiden kann ...
```


Nachteil der Möglichkeit 1:
Ein Abbruch kann geschehen, obwohl noch irgendwelche anderen Keller ihren Platz kaum benötigen. Denn man prüft ja nur die benachbarten Keller ab. Auch kann "umordnen" relativ rasch wieder aufgerufen werden.

Nachteil von Möglichkeit 2:
Eventuell wird die Prozedur "umordnen" nach q Schritten erneut aufgerufen, vor allem, wenn ein Keller schnell wächst. Komplettes Neuverteilen der Speicherbereiche vermeidet dies.

Vorteil:
Die Prozedur "umordnen" wird sehr schnell abgearbeitet.

12.2.5 Möglichkeit 3: (Garwick-Algorithmus)

- Berechne den insgesamt freien Platz aller Keller ("sum").
- Berechne den gesamten Zuwachs seit dem letzten Umordnen.
- Verteile 10% des freien Platzes gleichmäßig an alle Keller.
- Verteile 90% des freien Platzes proportional zum Zuwachs.

Um den Zuwachs zu berechnen, muss man sich in einem array ***AltIndex*** merken, welches die Indexpositionen *unmittelbar nach* dem letzten Umordnen waren. Um die Umordnung durchzuführen, muss man die neuen Basispositionen in einem array ***NewBase*** notieren. Der Zuwachs ergibt sich dann aus der Summe der Werte (Index(j)-AltIndex(j)), wobei man nur die positiven Werte addieren darf. NewBase(j) ergibt sich aus den Newbase-Werten der darunter liegenden Keller erhöht um den festen Anteil u, der jedem Keller zusteht, und dem Zuwachs-Anteil.

Die Formeln wollen wir zunächst genau angeben.

Zu berechnende Größen (u und w gerundet, weil ganzzahlig):

sum := gesamter freier Speicherplatz aller n Keller

zuwachs :=
    Summiere die nichtnegativen Werte von Index(j) - Altindex(j)
Hier werden nur die Keller berücksichtigt, die seit dem letzten Umordnen gewachsen sind.

$u := \lfloor (sum/10) / n \rfloor$
u ist 10% der Zahl freier Speicherplätze anteilig für jeden Keller; mindestens u Speicherplätze werden also jedem Keller als freie Plätze zugewiesen.
$u*n$ sind die freien Speicherplätze, die vorab an alle Keller verteilt werden.
$sum-u*n$ ist der restliche Speicherplatz, der nach Zuwachs zuzuordnen ist.

$v := (sum - u*n)/zuwachs$, $w := \lfloor v*\delta \rfloor$
v und δ sind reelle Werte. v = Zuwachs pro Speicherplatz in jedem Keller, der sich vergrößert hat. Ist ein Keller um δ Plätze gewachsen (also nur im Falle $\delta > 0$), dann erhält er $v*\delta$ zusätzliche Speicherplätze.

Damit sind die Formeln in folgender Prozedur "umordnen" erklärt (diese Prozedur braucht keine Parameter mehr):

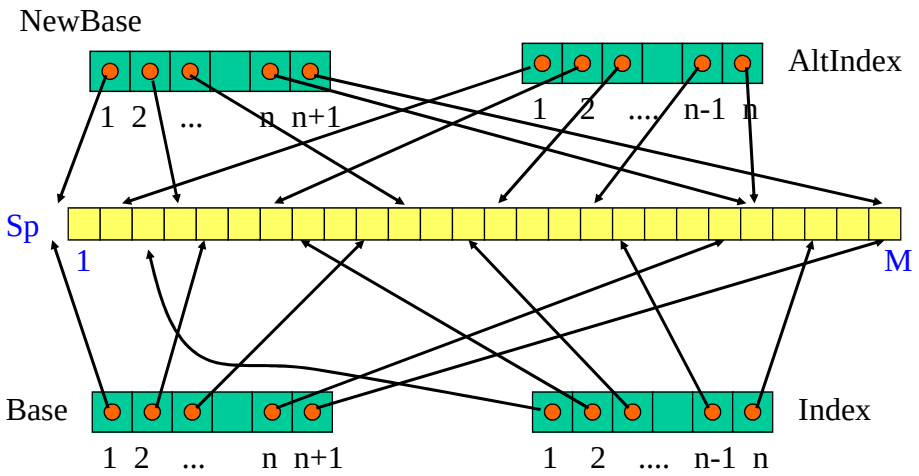
Garwick-Algorithmus: Füge zum "package body MKV1" hinzu:

```
NewBase: array (ZK) of Adressen;        -- Neuer Beginn der Kellers
AltIndex: array (ZK) of Adressen;       -- Alter Stand jedes Kellers

procedure umordnen is -- bei Möglichkeit 3 brauchen wir keinen Parameter i
   k: ZK;  sum, zuwachs, u, h: Integer; v: Float;  -- n ist global
   Delta: array (ZK) of Adressen;       -- für den Zuwachs jedes Kellers
begin  sum := 0;                        -- Addiere freien Platz in "sum" auf
   for j in 1..n loop  sum:=sum + Base(j+1) - Index(j);  end loop;
   if sum <= n then raise ueberlauf;
      -- Nicht genug Platz frei; bei sum > n vermeidet man eine unendliche
      -- Schleife durch ständig erneutes Aufrufen des Garwickalgorithmus
   else  zuwachs := 0;    -- ermittle Zuwächse seit letztem "umordnen"
      for j in ZK loop h := Index(j) - AltIndex(j);
         if h > 0 then Delta(j) := h; zuwachs := zuwachs + h;
         else Delta(j) := 0; end if;
      end loop;
```



```
      if zuwachs >= 1 then
         u := INTEGER(0.1*FLOAT(sum)/FLOAT(n));
                -- 10% Anteil, der jedem Keller zusteht
         v := (FLOAT(sum) - FLOAT(u*n))/FLOAT(zuwachs);
      else u := INTEGER(FLOAT(sum)/FLOAT(n)); v := 0; end if;
         -- Dieser else-Fall darf eigentlich nicht eintreten, da ja ein Keller
         -- überläuft; man könnte also auch eine exception erwecken.
      NewBase(1) := 0; NewBase(n+1) := M;
      for j in 2..n loop
      NewBase(j) := NewBase(j-1) + Index(j-1) - Base(j-1)
                          + u + INTEGER(Delta(j-1)*v - 0.5);  end loop;
      speicherumordnen;
      for j in ZK loop AltIndex(j) := Index(j); end loop;
   end if;
end umordnen;
-- Hier kann am Ende wegen Rundungsfehlern freier Speicherplatz übrig
-- bleiben (INTEGER(...-0.5)). Wie viel? Wie kann man dies berücksichtigen?
```


Garwick-Algorithmus: Verwaltung des Speichers Sp



Unterprozedur zu "umordnen":

```
procedure speicherumordnen is
   m, j, k: ZK;
begin  j := 2;
   while (j <= n) loop
      k := j;
      if  NewBase(k) < Base(k)  then  verschieben(k);
      else   while NewBase(k+1) > Base(k+1) loop
                                 k := k + 1;  end loop;
                        -- Diese Schleife endet spätestens für k = n
           for m in reverse j..k loop  verschieben(m);  end loop;
      end if;
      j := k + 1;
   end loop;
end speicherumordnen;
```

Unterprozedur zu "speicherumordnen":

```
procedure verschieben (i: in ZK) is
d: Integer;
begin d := NewBase(i) - Base(i);
              -- d gibt an, um wie viele Stellen Keller i verschoben werden muss
    if (d /= 0) then
        if d > 0 then
            for a in reverse Base(i)+1 .. Index(i) loop
                Sp(a+d) := Sp(a);  end loop;
        else
            for a in Base(i)+1 .. Index(i) loop
                Sp(a+d) := Sp(a);  end loop;          -- beachte: hier ist d<0
        end if;
        Index(i) := Index(i)+d;
        Base(i) := NewBase(i);
    end if;
end verschieben;
```


Hinweis: Wo kommen die Konstanten 10 und 90 im Garwickalgorithmus her? Warum nicht 30 und 70?

Dies sind Erfahrungswerte: Man hat die obige Umordnung der Kellerspeicherbereiche für eine feste Menge von Programmen inmer wieder für verschiedene Prozentzahlen durchgeführt und hierbei festgestellt, dass der Wert 10% für die Festzuweisung (und somit 90% für den Zuwachs) den besten Durchsatz, d.h., im Mittel die geringste Zahl an Umordnungen ergab.

12.2.6 Verwaltung durch ein Betriebssystem

- Der gesamte Kellerspeicher wird vom Betriebssystem verwaltet.
- Jedes Programm, das neuen Speicherplatz benötigt oder alten zurückgibt oder auf seine Daten zugreifen möchte, schickt eine Anfrage an das Betriebssystem mit "Nummer des Programms i" und "Relative Speicherplatzadresse a".
- Die zugehörige absolute Speicherplatzadresse im Rechner ist dann Base(i) + a. Ist dieser Wert kleiner oder gleich Base (i+1), stellt das Betriebssystem den Wert zur Verfügung bzw. speichert ihn ab bzw. legt ihn neu an oder gibt ihn frei; anderen-falls wird die Prozedur umordnen durchgeführt und erst danach entweder die Anfrage bedient oder, falls neuer Speicherplatz nicht mehr zugewiesen werden kann, die weitere Bearbeitung des Programms i unterbrochen und der Konfliktfall irgendwie gelöst (notfalls mit dem Abbruch mindestens eines Programms; in der Regel lagert man aber einen Prozess auf die Festplatte aus und bearbeitet ihn später, wenn wieder Platz vorhanden ist).

12.3 Freispeicher- und Haldenverwaltung

Um Datenobjekte zu verwalten, organisiert man diese mittels Zeigern in einem Graphen. In jedem Datenobjekt muss dann mindestens ein Zeiger existieren. Gibt es nur einen Zeiger, so kann man nur Listen aufbauen. Ab zwei Zeigern lassen sich stark vernetzte Strukturen realisieren.

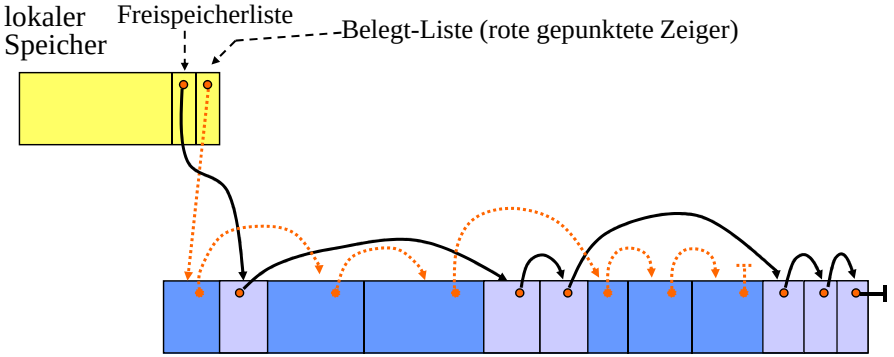
In der Implementierung werden "Zeiger" stets durch die "Adresse eines Speicherplatzes", ab der die referenzierte Struktur beginnt, dargestellt. Grund: Heutige Rechner besitzen in der Regel einen ein-dimensionalen Speicher, auf dessen Speicherplätze über einen Index von 0 bis 2^s-1 (für eine natürliche Zahl s) zugegriffen wird. Einen Zeiger implementiert man daher als die Adresse derjenigen Speicherzelle, ab der das Objekt, auf das verwiesen wird, steht.

12.3.1 Halde *(engl.: Heap)*: Dies ist ein Speicherbereich, dessen Größe hinreichend groß ist, um die mittels new erzeugten Datenobjekte (Zugriff über Zeiger!) abzulegen. Wenn diese Datenobjekte im Laufe der Rechnungen nicht mehr gebraucht werden, sollten sie explizit wieder frei gegeben werden (in Ada mit Hilfe des pragmas "Controlled" und der Prozedur FREE). Die Verwaltung erfolgt oft über eine *Freispeicherliste*.

Werden die nicht mehr benötigten Speicherplätze nicht wieder frei gegeben, so liegen nach einiger Zeit in der Halde viele unnötige Datenobjekte herum (= Daten, auf die nicht mehr von irgendeinem Programm über dessen lokalen Speicher zugegriffen werden kann). So kann die Halde rasch voll werden. Um dann noch weiterarbeiten zu können, müssen die nicht mehr benötigten Datenobjekte erkannt, ihre Speicherplätze frei gegeben und die Halde in geeigneter Weise umorganisiert werden (*Speicherbereinigung*).

12.3.2 Freispeicherliste

Um die dynamischen Daten der Halde zu verwalten, tragen wir die freien Speicherplätze in eine "Freispeicherliste" ein, aber nicht jede Speicherzelle einzeln, sondern immer ganze "Datenblöcke".



Halde (blau = belegte Bereiche)

Benutzen mehrere Programme die Halde, so wird die "Freispeicherliste" (und eventuell auch eine "Belegt-Liste") vom Betriebssystem verwaltet. Folgende Aufgaben sind unter anderen Fragestellungen zu lösen:

1. Ein Programm fordert einen Speicherplatzbereich der Größe "G" an. Weise dem Programm einen geeigneten Bereich in der Halde zu und modifiziere die Freispeicherliste.

2. Ein Programm gibt einen Speicherplatzbereich wieder frei. Füge diesen Bereich "geschickt" in die Freispeicherliste ein.

3. Verschmelze aneinander grenzende freie Datenblöcke der Halde zu größeren Einheiten.

4. Falls keine Zuweisung erfolgen kann, ordne die Halde so um, dass alle freien Bereiche nebeneinander liegen (das ist nicht trivial, weil fast alle Zeiger abzuändern sind). Füge hierbei alle Datenblöcke, die nicht mehr benutzt werden, in die Freispeicherliste ein.
5. Falls auch dies nicht erfolgreich ist, führe einen Austausch der Speicherinhalte mit dem Hintergrundspeicher durch (Stichwort: Seitenaustauschstrategien, Paging; siehe Vorlesungen über Betriebssysteme).

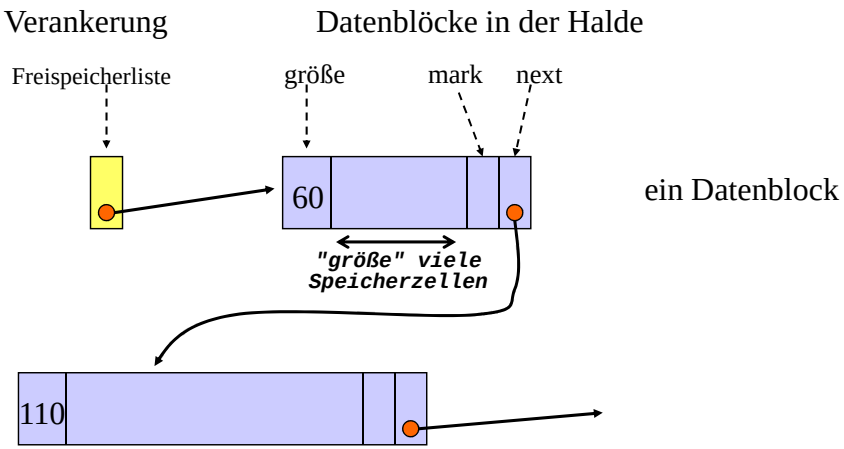
Um diese Aufgaben durchzuführen, muss die Freispeicherliste oft durchlaufen werden. Hierfür muss jeder Datenblock drei Speicherplätze enthalten: einen Zeiger für den Verweis auf den nächsten freien Datenblock, eine natürliche Zahl für die Größe des Datenblocks und einen oder mehrere Boolesche Werte, damit wir bei späteren Verfahren markieren können, welchen Datenblock wir bereits besucht haben, ob der Datenblock zur Belegt-Liste gehört bzw. welche andere Eigenschaft er erfüllt.

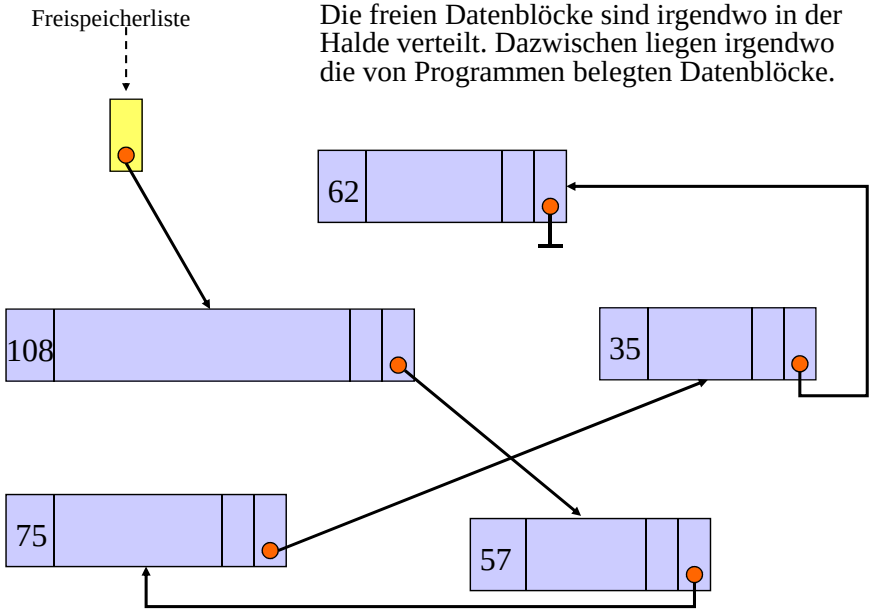
Wir legen hier folgenden Datentyp "DBlock" für Datenblöcke fest. Es sei eine natürliche Zahl "maxgröße" (= die maximale Größe an Speicherplätzen je Datenblock) vorgegeben (vgl. auch 1.12.3 "Diskriminanten für Größenangaben"):

```
type DBlock;
type DBlockzeiger is access DBlock;
subtype AnzahlZellen is Positive range 1..maxgröße;
type DBlock (größe: AnzahlZellen := maxgröße) is record
        inhalt: array (1..größe) of Speicherzelle;
        -- Komponenten, die genau "größe" viele Speicherzellen belegen
        mark: Boolean;
        next: DBlockzeiger;
end record;
```

Jeder DBlock belegt also größe + x viele Speicherzellen in der Halde; hierbei ist x die Zahl der Speicherzellen, die für "größe", "mark" und "next" benötigt werden.

Skizze:





Die freien Datenblöcke sind irgendwo in der Halde verteilt. Dazwischen liegen irgendwo die von Programmen belegten Datenblöcke.

12.3.3 Bearbeitungsstrategien: Ein Programm fordert einen Datenblock mit m Speicherplätzen an.

Algorithmus 1: **First Fit**
Gehe die Freispeicherliste durch, bis ein Datenblock D mit größe ≥ m + x gefunden ist.
Mache hieraus zwei Datenblöcke: Einen mit m + x und einen mit größe-m-x Speicherplätzen (x = Zahl der Speicherplätze für größe, mark und next, siehe Datentyp DBlock in 12.3.2).
Füge diese beiden Datenblöcke in die Freispeicherliste anstelle des DBlocks D ein.
Klinke den ersten dieser beiden Datenblöcke aus der Freispeicherliste aus und ordne ihn dem Programm zu.
Hinweise: Falls der zweite Block "zu klein" ist, vermeide die Aufspaltung in zwei Datenblöcke und weise ganz D dem Programm zu. Falls kein geeigneter Block D existiert, rufe die Speicherbereinigung auf, siehe später.

Algorithmus 2: **Best Fit**
Gehe die gesamte Freispeicherliste durch und ermittle den kleinsten Datenblock D mit größe ≥ m + x.
Fahre anschließend fort wie bei "First Fit".

Welche Strategie ist besser?

Bei beiden Methoden entstehen im Lauf der Zeit viele kleine Datenblöcke, die verstreut in der Halde liegen. Diese sog. "Fragmentierung" des Speichers erfordert häufige Aufrufe der Speicherbereinigung. In der Praxis erweist sich die Best-Fit-Strategie gegenüber der "First-Fit-Strategie" nach einiger Zeit als schlechter (!), da hierbei besonders kleine Datenblöcke entstehen; außerdem muss bei Best-Fit stets die gesamte Freispeicherliste durchlaufen werden.

Aus der Praxis weiß man: Solange der freie Speicher etwa ein Drittel der Halde ausmacht, ist die First-Fit-Strategie gut anwendbar. Wird aber der freie Platz geringer, so muss oft eine zeitaufwendige Speicherbereinigung durchgeführt werden, die zu Wartezeiten bei den "Kunden" führt.

Recht nachteilig ist die Zeit, die beim Durchlaufen der Liste verstreicht. Zwei Ideen zur Verbesserung:

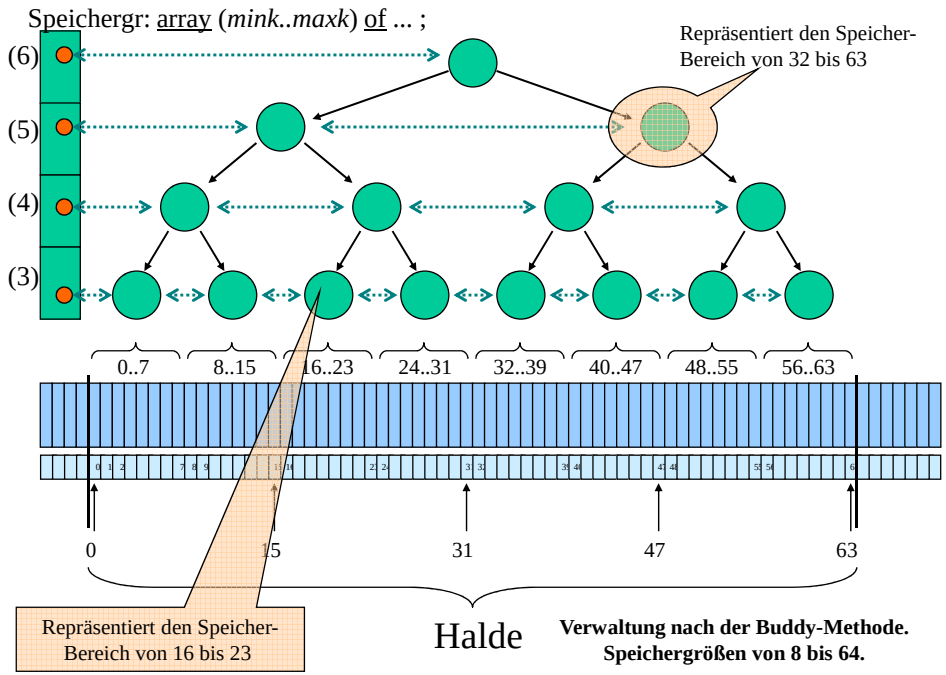
- Halte die Freispeicherliste stets nach der Größe der Datenblöcke sortiert. Nachteil: Das Einfügen freigegebener Speicherbereiche ist dann aufwendiger, dafür findet man aber einen passenden DBlock im Mittel rascher.
- Lege einen binären Suchbaum über die Freispeicherliste. Die Suche erfolgt dann schneller. Nachteile: Weiterer Speicherplatz; diese Suchbäume müssen ebenfalls in der Halde untergebracht werden, da sie dynamische Datenstrukturen sind; sie müssen eventuell ständig geändert werden.

12.3.4 Buddy-Verfahren

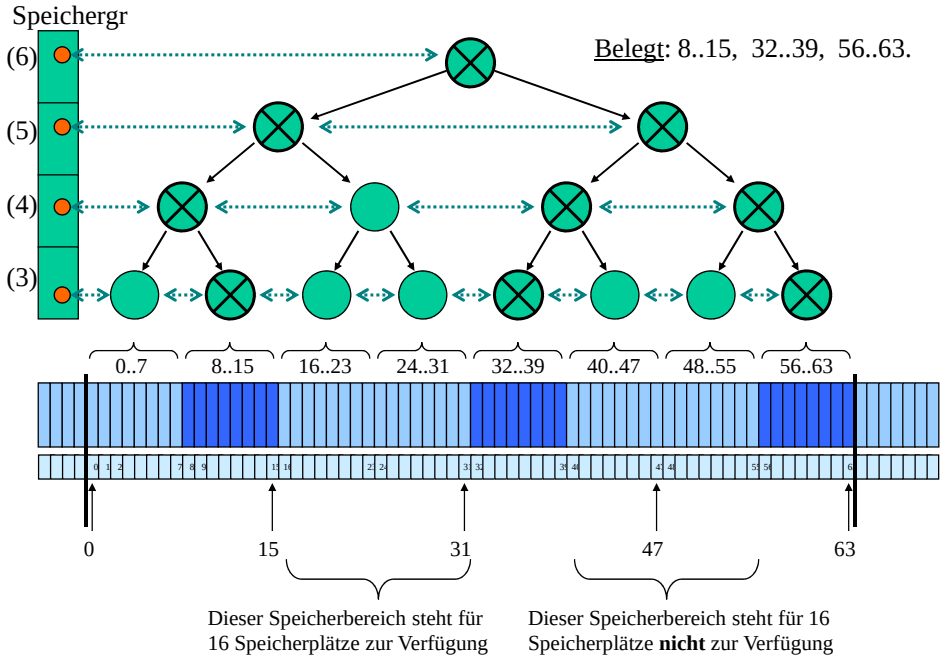
Wir stellen kurz einen alten Vorschlag zur Verwaltung des freien Speicherplatzes *anstelle* einer Freispeicherliste vor, der leicht implementiert werden kann, aber gewisse Nachteile besitzt, die sog. Buddy-Methode. (Buddy = (engl.) Kamerad, Kumpel.)

Idee: Das Verfahren legt einen gleichverzweigten binären Baum über die Halde. Das Aufspalten und das Verschmelzen von Datenblöcken sind aber nicht beliebig möglich.

Vorgehen: Die Halde wird in Datenblöcke unterteilt, deren Länge jeweils eine Zweierpotenz ist. Jeder Datenblock muss genau 2^k Speicherplätze belegen für eine natürliche Zahl k mit $mink \le k \le maxk$. (Man schränkt in der Praxis k ein z.B. zwischen *mink* = 10 und *maxk* = 25.) Jedem Datenblock wird genau einer seiner beiden Nachbarn als "Buddy" zugeordnet.



Verwaltung nach der Buddy-Methode. Speichergrößen von 8 bis 64.



Wir nummerieren die Halde also von 0 bis $2^{maxk} - 1$ durch. Die kleinste Blockgröße sei 2^{mink}. Alle Speicherblöcke mit festem $mink \le k \le maxk$ stehen in einer Liste, erreichbar über den Zeiger des Feldelements Speichergr(k).

Zu jedem Speicherblock, der an der Adresse x beginnt und die Größe 2^k besitzt, sei **$buddy_k(x)$** die Anfangsadresse seines Buddy (dieser liegt entweder links oder rechts von ihm und besitzt die gleiche Größe).

Es gilt:

$$\textbf{buddy}_k(x) = \begin{cases} x + 2^k \text{ , falls } x = 0 \mod 2^{k+1} \\ x - 2^k \text{ , falls } x = 2^k \mod 2^{k+1} \end{cases}$$

Wir programmieren diese Form der Speicherverwaltung nicht aus, sondern geben nur die Vorgehensweisen an.

Beginn: Anfangs werden alle Speicherbereiche als frei markiert.

Speicheranforderung: Ein Programm fordert einen Datenblock der Größe m an. Berechne k so, dass $2^{k-1} < m \le 2^k$ gilt.

Suche: Die Speicherverwaltung durchläuft dann die Liste, die über Speichergr(k) erreichbar ist. Diese Liste hat 2^{maxk-k} Knoten.

Zuordnung und Aktualisierung: Wird hier ein freier Speicherbereich gefunden, so wird er dem Programm zugewiesen; zugleich werden dieser Bereich, alle Knoten in seinem Unterbaum und seine Vorgänger im Baum bis zur Wurzel als belegt markiert.

Ablehnung und "Wiedervorlage": Wird kein freier Speicherbereich gefunden, so lege die Speicheranforderung in einer Warteschlange des Systems ab, sende dem Programm einen "Wartehinweis" und prüfe später erneut.

ok, aber ...

Beispiel: Wird in der Situation der obigen Folie ein Bereich der Größe 14 angefordert, so ist k = 4 und es wird ausgehend von Speichergr(4) die Liste der Speicherblöcke der Größe $2^4 = 16$ durchsucht. Bereits der zweite Block ist frei, so dass dem anfordernden Programm der Bereich 16..31 zugewiesen wird.

Speicherfreigabe: Ein Programm gibt einen Datenblock der Größe 2^k wieder zurück. Dieser Block wird in der Liste zu Speichergr(k) als frei markiert. Ist sein Buddy frei, so wiederhole diesen Vorgang mit seinem Vorgängerknoten.

Beispiel: Wird in der Situation der obigen Folie der Bereich 8..15 der Größe 8 freigegeben, so kann dieser Block, aber auch sein Vorgänger und dessen Vorgänger frei gegeben werden, so dass anschließend die linken drei als belegt markierten Knoten im Baum wieder als frei markiert sind.

Vorteile der Buddy-Methode:

- Einfach zu handhaben und leicht zu programmieren.
- Das Verfahren bewährt sich in der Praxis hinreichend gut.

Nachteile:

- Benachbarte Bereiche, die nicht Buddys sind, können nicht verschmolzen werden, und es gibt nicht genutzte Speicherbereiche (Fragmentierung), da immer nur Blöcke von der Länge einer Zweierpotenz zugewiesen werden.
- Es können Verklemmungen entstehen, wenn zwei Programme Speicher nachfordern, die nicht verfügbar sind. (Man kann dies durch Überwachung durch das Betriebssystem abfangen, oder man kann verbieten, dass ein Programm eine zweite Speicheranforderung stellt, was aber bei rekursiven oder nebenläufigen Programmen nicht sinnvoll ist.)

12.3.5 Speicherbereinigung (garbage collection)

Wir nehmen nun an, die Programme legen immer mehr dynamische Datenstrukturen (also: verzeigerte Strukturen) in der Halde an. Es ist absehbar, dass in Kürze kein Speicherplatz mehr zur Verfügung steht.

Nun muss geprüft werden, ob die Datenobjekte, die in der Halde stehen, wirklich alle benötigt werden, oder ob man sie löschen und auf diese Weise neuen Speicherplatz bereitstellen kann. Dies ist die Aufgabe der "Speicherbereinigung", die in der Regel automatisch erfolgt.

Standardtechniken zur Lösung dieser Aufgabe sind: Verweiszählermethode, Graphdurchlauf und Kopieren.

(Hinweis: In Ada 95 gibt es keine Speicherbereinigung, vielmehr muss alles explizit mit FREE freigegeben werden.)

12.3.6 Verweiszählermethode ("Reference Counting")

Wenn die Zeigerstrukturen keine Kreise bilden (also "azyklisch" sind), dann kann man in jeden Knoten einen "Verweiszähler" aufnehmen, der angibt, wie oft auf dieses Objekt verwiesen wird. Wird ein Knoten mit k Zeigern hinzugefügt, so müssen die Verweiszähler der k Knoten, auf die diese Zeiger zeigen, jeweils um 1 erhöht werden. Wird ein Knoten gelöscht, so muss man die Verweiszähler in den k Objekten, auf die die Zeiger zeigten, um jeweils 1 erniedrigen. Wird ein Verweiszähler hierbei 0, dann muss man auch in allen ihren nachfolgenden Knoten den Verweiszähler um 1 erniedrigen. Entsprechende Operationen kann der Compiler in den übersetzten Code einfügen, ohne dass der Programmierer hiervon etwas bemerkt. Benötigt man irgendwann Speicherplatz, so kann man zu einem gegebenen Zeitpunkt genau alle die Knoten löschen, deren Verweiszähler 0 ist.

Bei zyklischen Strukturen funktioniert dieses einfache Verfahren aber nicht mehr. (Klar!?)

12.3.7 Graphdurchlauf

Hier verfolgt man alle Zeiger, die von den lokalen Speichern der Programme ausgehen, und markiert alle Datenobjekte, die auf diese Weise auf irgendeinem Weg erreichbar sind. Die nicht-markierten Datenobjekte kann man löschen.

Die Halde ist nichts anderes als ein großer Graph mit mehreren Einstiegspunkten (= den Zeigern der lokalen Speicher in den Programmen). Ausgehend von diesen Einstiegspunkten wird der Graph mit den bekannten Techniken (3.8.7 und 8.8.9) wie DFS und BFS oder gewissen Varianten durchlaufen, wobei die erreichbaren Knoten mit "true" markiert werden.

Das Verfahren besteht aus zwei Teilen:
- Markiere alle erreichbaren Objekte,
- Entferne alle nicht-markierten Objekte.

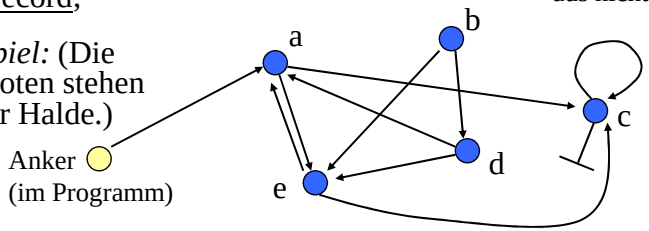
Wir behandeln nur den ersten Teil, da der zweite ein einfacher Halden-Durchlauf mit Umhängen in die Freispeicherliste ist.

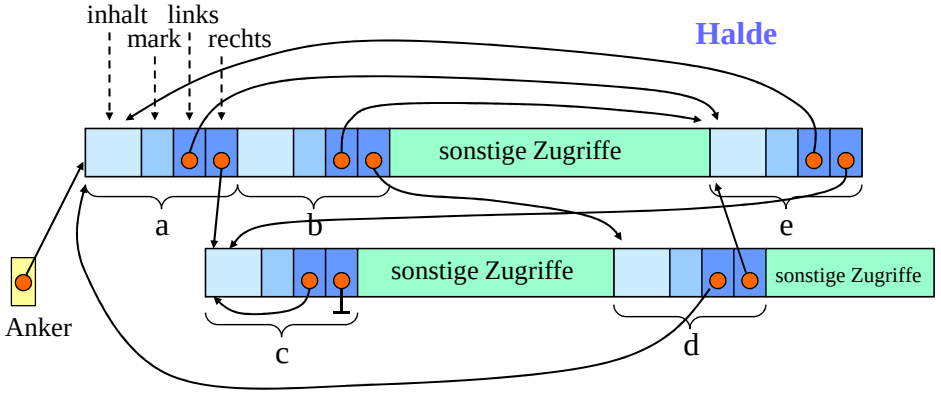
Vereinfachung: Um das Vorgehen zu erläutern, betrachten wir hier Datenobjekte mit zwei Zeigern (statt nur mit einem wie bei Listen), also Objekte des folgenden Typs "Knoten":

```
type Knoten;
type Kante is access Knoten;
type Knoten is record
        inhalt:  ...
        mark: Boolean;
        links, rechts: Kante;
end record;
```

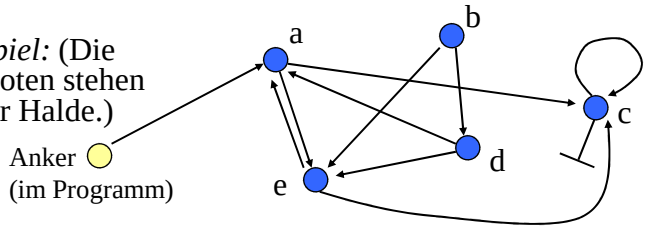
Die Knoten b und d sind vom Programm aus nicht erreichbar.

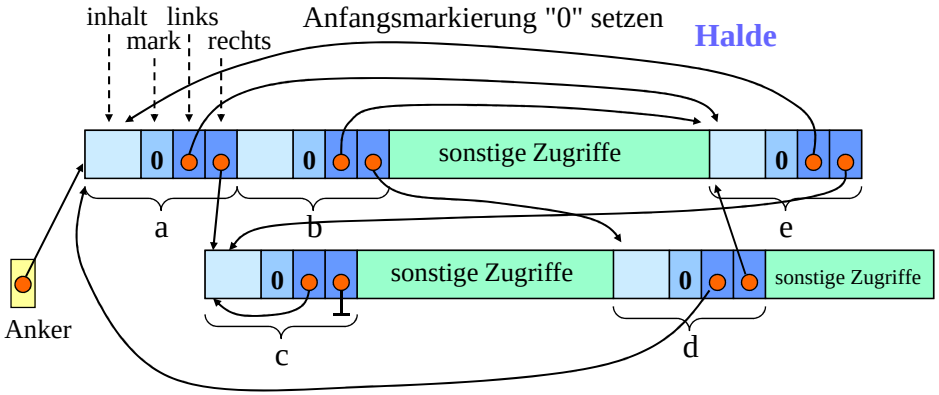
Beispiel: (Die 5 Knoten stehen in der Halde.)



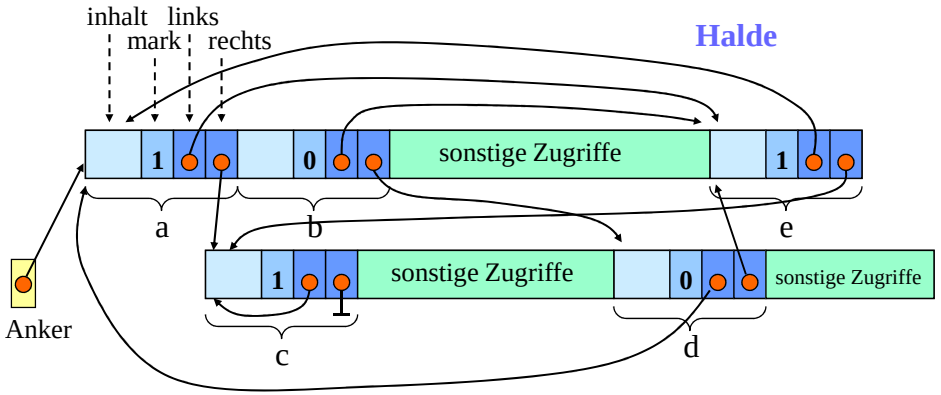


Beispiel: (Die 5 Knoten stehen in der Halde.)





Ziel muss es nun sein, die Knoten b und d als löschbare Knoten zu erkennen. Hierzu durchläuft man ausgehend von "Anker" alle Zeiger und markiert die erreichten Knoten mit true (oder einer "1"). Dies geschieht für alle "Anker", die aus den lokalen Speichern in die Halde verweisen. Danach löscht man alle mit false (oder "0") markierten Objekte und schiebt ggf. den Speicher zusammen.



Ergebnis des Durchlaufs: Ausgehend von "Anker" wurden alle Zeiger nachverfolgt und die hierbei erreichten Knoten mit einer "1" markiert; die nicht erreichbaren bleiben mit "0" markiert.

Anschließend kann man den Speicherbereich neu organisieren, sofern man möglichst große Freispeicherbereiche benötigt.

12.3.8 Wir kommen nun zum Algorithmus zur Markierung der erreichbaren und der unerreichbaren Knoten unter der Annahme, dass kein freier Speicherplatz für den Algorithmus zur Verfügung steht:

Schritt 1:
Markiere alle Knoten in der Halde mit "false" (bzw. mit 0).

Schritt 2:
Markiere alle Knoten in der Halde, die von einem der lokalen Speicher direkt erreicht werden können, mit "true" (bzw. mit 1).

Schritt 3 (eigentlicher Algorithmus): Die Halde möge von Adresse 0 bis Adresse M im Speicher nummeriert sein. Jeder Knoten möge genau r Speicherplätze (Adressen) belegen.
u, v sind vom Typ Knoten, i und j sind Adressen in der Halde.

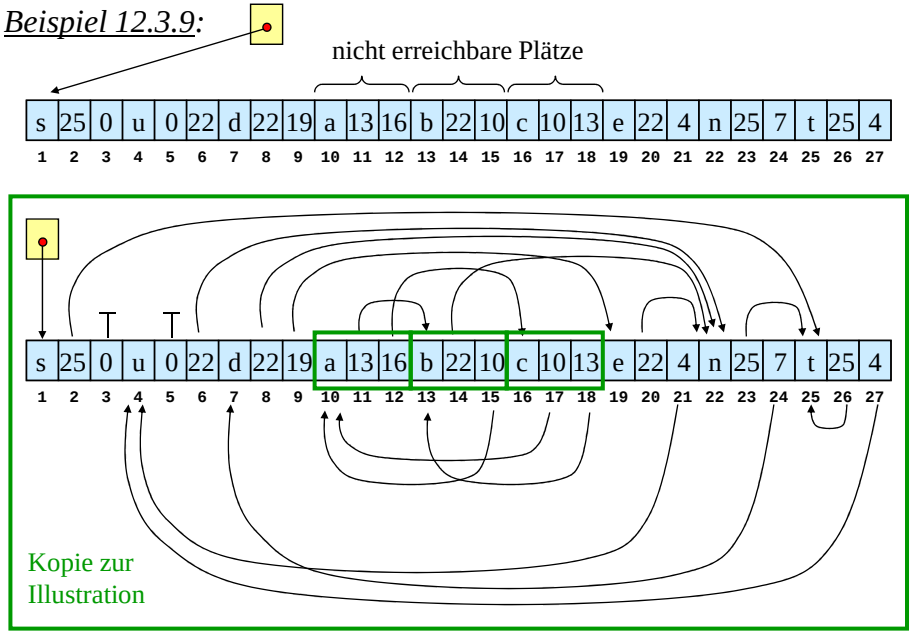

```
i := 0;                          -- i ist die Adresse des betrachteten Knotens
while i <= M loop
    j := i + r;                  -- j wird die Adresse des nächsten Knotens
    if der Knoten mit Adresse i ist mit "true" markiert
       and then der Knoten mit Adresse i besitzt mindestens einen
            Nachfolger (d.h.: (links /= null) or (rechts /= null))
    then  if (links /= null) and then (der Knoten u, auf den links
                verweist, ist mit "false" markiert)
          then  markiere den Knoten u mit "true";
                j := Minimum (j, Adresse von u);  end if;
          if (rechts /= null) and then (der Knoten v, auf den rechts
                verweist, ist mit "false" markiert)
          then  markiere den Knoten v mit "true";
                j := Minimum (j, Adresse von v);  end if;
    end if;
    i := j;                      -- zum nächsten Knoten gehen
end loop;
```


Idee dieses Vorgehens: Durchlaufe die Knoten von vorne nach hinten in der Halde. Es interessieren nur die mit true markierten Knoten (nur sie sind bisher vom Programm aus erreichbar). Betrachte deren beide Nachfolgeknoten. Markiere sie mit true und setze das Verfahren an der minimalen Adresse der drei Knoten
- nächster Knoten in der Halde
- linker Nachfolgeknoten
- rechter Nachfolgeknoten

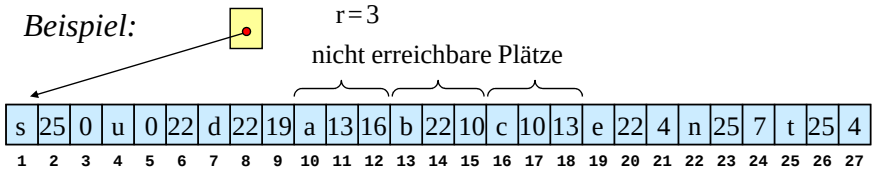
fort.

Auf diese Weise gelangt man schließlich an alle erreichbaren Knoten. Beachte: Dieser Algorithmus ist eine rekursionsfreie Variante des Graphdurchlaufs, der in 3.8.7 beschrieben wurde. Um ihn zu verstehen, müssen Sie ihn präzise nachvollziehen.

Beispiel 12.3.9:



Beispiel:



Ablaufdiagramm für i, j und "Markierung auf true setzen":

i	j	Mark.
		1
1	4	
	4	25
4	7	
7	10	
10	13	
13	16	
16	19	
19	22	
22	25	
25	28	

i	j	Mark.
	4	4
4	7	
7	10	22
10	13	
13	16	
16	19	
19	22	
22	25	
25	28	
	7	7
7	10	

i	j	Mark.
	10	19
10	13	
13	16	
16	19	
19	22	
	22	
22	25	
25	28	
28		
Ende		

Dies ist bereits der worst case: Es gibt viele Verweise vom Ende der Halde nach vorne (vgl. Komplexitätsabschätzung unten).

Aufwand dieses Verfahrens? (Im worst case quadratisch in M.)

Zeitkomplexität: Im ungünstigsten Fall beim Durchlauf durch die Halde ist die if-Bedingung erst beim letzten Knoten erfüllt und dessen Verweis führt auf den ersten Knoten zurück, siehe obiges Beispiel.

Nach dem zweiten Durchlauf geschieht das Gleiche mit dem vorletzten Knoten usw.

Wenn n eine (ungerade) Zahl der Knoten in der Halde ist, so muss man $n + (n-2) + (n-4) + ... + 3 + 1 = ((n+1)/2)^2 \in O(n^2)$ Schritte ausführen. Wegen $n \approx M/r$ erhält man ein $O(M^2)$-Verfahren. (M = Anzahl der Speicherplätze in der Halde).

In der Tat erweist sich dieser Algorithmus in der Praxis auch im Mittel als ein quadratisch mit M wachsendes Verfahren.

Die Speicherplatzkomplexität ist dagegen konstant.

Hinweis:
Es ist klar, wie man dieses Verfahren auf Knoten, die mehr als zwei Nachfolger haben können oder deren Größe im Datenobjekt selbst gespeichert ist, erweitern kann:
- for all Nachfolgeknoten (im äußersten then-Teil),
- ersetze j := i+r durch j:=i+größe_des_aktuellen_Knotens.

Das oben genannte Verfahren eignet sich besonders dann, wenn man (fast) keinen freien Speicherplatz mehr zur Verfügung hat. Gibt es dagegen noch Speicherplatz, den man für einen Keller S nutzen kann, dann empfiehlt sich folgender deutlich schnellere Algorithmus, der die weiter zu verfolgenden Zeiger im Keller S ablegt (machen Sie sich klar: dies ist der Algorithmus GD aus 3.8.7 beschränkt auf Ausgangsgrad 2):

12.3.10 Kellerverfahren

Schritt 1: Markiere alle Knoten in der Halde mit "false".
Schritt 2: Markiere alle Knoten in der Halde, die von einem lokalen Speicher direkt erreicht werden können, mit "true" und lege sie (in der Praxis: ihre Adressen) im Keller S ab.
Schritt 3: K sei vom Typ Kante (="Zeiger auf Knoten").

```
while not isempty(S) loop
    while not isempty(S) and (top(S) hat keinen Nachfolger)
        loop pop(S); end loop;
    if not isempty(S) then
        K := top(S);  pop(S);
        if (K.links /= null) and then (not K.links.mark) then
            K.links.mark := true;  push(S, K.links);  end if;
        if (K.rechts /= null) and then (not K.rechts.mark) then
            K.rechts.mark := true;  push(S, K.rechts);  end if;
    end if;
end loop;
```

Aufwand dieses Keller-Verfahrens? (Im worst case linear in M.)

Das Verfahren durchläuft jeden Zeiger, der in einem Knoten auftritt, höchstens einmal. Da es höchstens doppelt so viele Zeiger wie Knoten gibt, handelt es sich bei Schritt 3 also um ein O(n') -Verfahren (n' = Zahl der erreichbaren Knoten in der Halde, $n' \leq n$ = Zahl der Knoten in der Halde).

Allerdings bezahlt man diese Schnelligkeit mit dem benötigten Speicherplatz für den Keller S. Dieser kann bis zu n/2 Knoten groß werden. Im Mittel wird man aber deutlich weniger Platz brauchen.

Die uniforme Zeitkomplexität dieses Keller-Verfahrens wird also vor allem durch Schritt 1 bestimmt, welcher $n \approx M/r$ Zeiteinheiten benötigt. Insgesamt ergibt sich damit ein O(M)-Verfahren sowohl bzgl. der Zeit als auch bzgl. des Platzes.

Man kann nun die beiden Algorithmen kombinieren:

Variante 1: Solange noch genügend Platz für den Keller S vorhanden ist, arbeite nach dem Kellerverfahren. Sobald der Keller überläuft, suche man die kleinste Adresse im Keller und schalte mit ihr beginnend auf das andere Verfahren um.

Variante 2: Man verwende statt des Kellers eine sortierte Liste, worin die Zeiger geordnet nach ihrer Adresse eingetragen werden und verwende stets den Zeiger mit der kleinsten Adresse als nächsten zu untersuchenden Knoten. Da die Liste bei jedem Eintrag durchlaufen werden muss, wird dieses Vorgehen zu einem $O(n'^2)$-Verfahren und damit letztlich zu einem $O(M^2)$-Verfahren.

Überlegen Sie sich weitere Kombinationen, Varianten oder Verbesserungen. Ist es z.B. sinnvoll, zuerst den Nachfolgeknoten mit der größeren Adresse in den Keller S zu legen?

Man erkennt nun auch die Abhängigkeit der Speicherbereinigung von der jeweiligen Programmiersprache: Man muss wissen, wie die Datenobjekte / Blöcke / Knoten usw. aufgebaut sind, um Zeiger auch als Zeiger erkennen zu können. Andererseits kann natürlich das Betriebssystem ein universelles Datenformat vorgeben, in dem zum Beispiel die Informationen über Zeiger und die Markierungen an vorgegebenen Stellen notiert werden müssen.

Siehe auch 12.4.

Zum Durchlaufalgorithmus in Linearzeit ohne einen Keller vgl. den Durchlauf durch binäre Bäume (Stichwort: Schorr-Waite-Algorithmus, siehe Literatur).

12.3.11 Nachdem wir nun die erreichbaren Knoten bzw. Datenblöcke mit "true" markiert haben, kann man alle anderen in die Freispeicherliste (oder in die Buddy-Verwaltung) eintragen und normal weitermachen.

Oft möchte man zusätzlich den Speicher "zusammenschieben" ("kompaktifizieren") und dabei die im Laufe der Rechnungen entstandenen kleinen Fragmente (= nicht nutzbaren freien Speicherbereiche) beseitigen. Dieser Kompaktifizierungs-Algorithmus ist bei beliebiger Verzeigerung aufwendig. Man kann Verschiebe-Zeiger mitführen, die die Lage nach der Kompaktifizierung angeben. Dies ist insbesondere bei der Kopiermethode (12.3.12) vorteilhaft.

Diese und weitere Fragen zur Verwaltung von Programmen und Daten lernen Sie in Vorlesungen über Betriebssysteme oder auch in speziellen Praktika kennen.

12.3.12 Kopiertechniken

Man verwendet aktuell immer nur den halben Speicher. Läuft dieser über, so kopiert man (ausgehend von den Zeigern in den lokalen Speichern der Programme) die erreichbaren Objekte in die andere Hälfte des Speichers, wobei man die relative Anordnung unverändert lässt. Auf diese Weise werden die nicht-markierten Objekte zu Lücken im neuen Speicher und können in die Freispeicherliste eingetragen werden.

Mit etwas erhöhtem Aufwand können die Objekte beim Kopieren von vorne nach hinten nacheinander abgelegt werden, wodurch zugleich der Speicherplatz optimal genutzt wird. In mehreren Durchläufen können hierbei die Verweise entsprechend der neuen Anordnung umgesetzt werden. *(Überlegen Sie, wie so etwas geschehen könnte! Z.B. mit zusätzlichen Verweiszeigern, die den neuen Speicherplatz im alten Objekt speichern.)*

Hinweis: Um sich in dieses Problem hineinzudenken, sollten Sie die Frage lösen, wie man einen Graphen kopiert.

Dies sieht einfach aus, hat aber seine Tücken, wenn man keine Zusatzzeiger mitführen oder (wegen fehlendem Speicherplatz) keine Rekursion verwenden darf.

12.4 Historische Hinweise

Mit der Entwicklung der ersten Computer in den 1940er Jahren entstanden auch sogleich eindimensionale Felder, da diese genau die Speicherstruktur wiedergaben. Allgemeine Felder finden sich bereits in den Programmiersprachen der 1950er Jahre (Fortran 58, Algol 60, APL). Verbunde und Folgen von Buchstaben treten in Cobol auf (ab 1961). Verschiedene Konzepte der Datenstrukturen wurden in Algol 68 (Standard: 1975) zusammengeführt. Dessen "gut verständlicher" und "gut implementierbarer" Anteil wurde von Nikolaus Wirth in die Sprache PASCAL (1972) eingebracht, die bis heute als "didaktisches Vorbild" für Programmiersprachen gilt. (Deren Sprachelemente gehören zum Kern des "Programmieren im Kleinen" und sind in Ada enthalten.)

Die Datenstruktur "Keller" entstand ca. 1954 mit ersten Arbeiten über die korrekte Auswertung von arithmetischen Ausdrücken. Sie wurde zugleich ab 1960 verwendet, um den Aufruf von (auch rekursiven) Prozeduren und generell alle klammerartigen Strukturen in Programmen und Programmier-sprachen korrekt zu implementieren.

Listen bilden die Grundlage der Programmiersprache LISP (McCarthy, ab 1959). Statt der expliziten Zeiger wurden Operationen wie "head" und "tail" für den Zugriff auf das erste Element einer Liste bzw. auf die "Restliste ohne das erste Element" benutzt. In SIMULA und PL/I (beide ab 1965) konnten Zeiger verwendet werden. Die Unterscheidung zwischen einem statischen und einem dynamischen Speicher geschah bereits in den ersten Programmiersprachen; die Halde wurde in SIMULA (Koroutinenkonzept) und PL/I erforderlich.

Dass sehr allgemeine Datenstrukturen korrekt übersetzbar sind, demonstrierten die Compiler von SIMULA 67 und etwas später von PL/1. Probleme bereiteten aber die ganz allgemeinen Datenstrukturen von Algol 68, bei denen kartesische Produkte, Vereinigungen, Referenzen, Potenzmengen, Funktionenbildung usw. beliebig miteinander verknüpft werden können: Die Laufzeitsysteme wurden derart kompliziert, dass jeder Algol-68-Compiler gewisse Einschränkungen machen musste.

Für die automatische Speicherbereinigung des Betriebssystems ist es unverzichtbar, *einen Zeiger auch als Zeiger zu erkennen!* Hierzu legt der Compiler (unsichtbar für den Benutzer) für jeden Zeiger einen "Deskriptor" an, aus dem die Struktur des referenzierten Objektes (sein Typ einschl. der Weiterverweise und das Markierungsbit) zu ersehen ist.

Mitte der 1960er Jahre entstanden die ersten Betriebssysteme, die mehrere Programme gleichzeitig verwalten konnten. Ab dieser Zeit entwickelte man diverse Verfahren für die Speicherverwaltung (wie Multikeller, Freispeicher, Bereinigung usw.). Die in diesem Kapitel 12 behandelten Vorgehensweisen sind also bereits als "klassische Verfahren" einzustufen.

Für *Echtzeitsysteme* und für *Verteilte Systeme* wurden in den 80er und 90er Jahren neue Verfahren entwickelt. Echtzeitsysteme z. B. können nicht einfach unterbrochen werden, so dass die Standardverfahren zu "inkrementellen Techniken" erweitert wurden: Während des Programmablaufs läuft ein Prozess mit, der nicht erreichbare Objekte aufspürt und in die Freispeicherliste einträgt. Die beiden Prozesse dürfen allerdings nicht gleichzeitig auf gewisse Teile zugreifen ("wechselseitiger Ausschluss", siehe 13.4), was der Compiler automatisch in das übersetzte Programm einfügt.

13. Netze und Prozesse

13.1 Stellen-Transitions-Netze

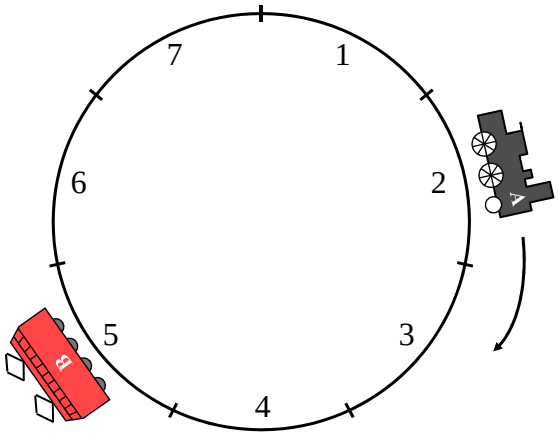
Bisher: Sequentielle Programmierung, d.h., höchstens eine Stelle im Programm wird in jedem Augenblick bearbeitet. Jeder Ablauf wird hierbei in eine Folge nacheinander auszuführender Aktivitäten zerlegt.

Im Folgenden wollen wir unabhängig voneinander ablaufende Programme (Prozesse, Objekte) und deren Kommunikation beschreiben. Man spricht von Nebenläufigkeit (engl.: concurrency) und von nebenläufigen, von verteilten und von parallelen Systemen. Wir beginnen mit einem Kalkül.

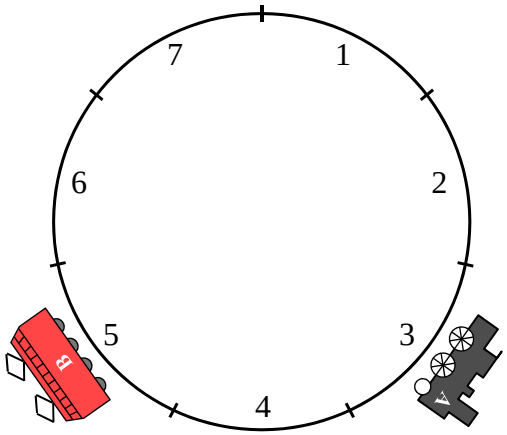
Ansatz: Verallgemeinere endliche Automaten, in denen mehrere Zustände gleichzeitig oder nacheinander aktiv sind. Standardbeispiel: Züge auf einer Kreisstrecke.

13.1.1 Beispiel: Züge auf einer kreisförmigen Strecke.

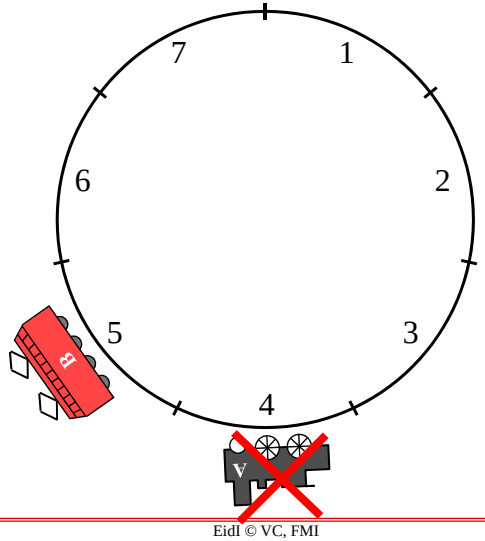
Damit die Züge nicht zusammenstoßen, verlangen wir, dass zwischen ihnen mindestens ein Streckenabschnitt frei bleibt.



Damit die Züge nicht zusammenstoßen, verlangen wir, dass zwischen ihnen mindestens ein Streckenabschnitt frei bleibt.

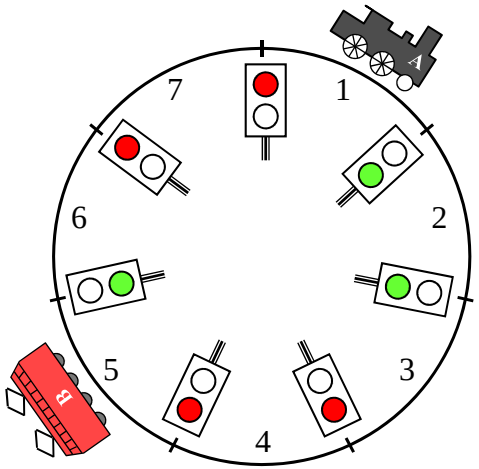


Diese Situation darf also nicht eintreten. Wie kann man dies sicherstellen?



Übliche Lösung: Wenn eine Lok von Abschnitt i nach (i mod 7 +1) einfährt, so muss die Einfahrt in den Abschnitt i durch ein rotes Signal gesperrt und die Einfahrt nach i-1 durch ein grünes Signal geöffnet werden.

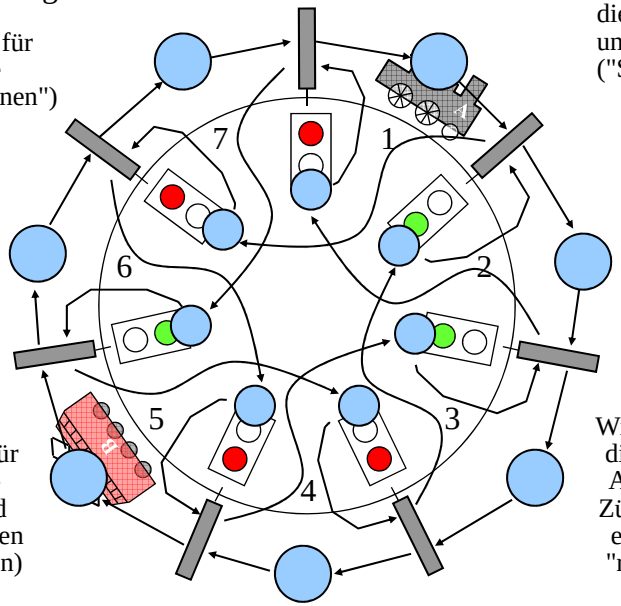
Steuerung durch Ampeln: Grünes Signal bedeutet, dass in den Streckenabschnitt hineingefahren werden darf.



Umwandlung in ein Netz:

Rechtecke für Übergänge ("Transitionen")

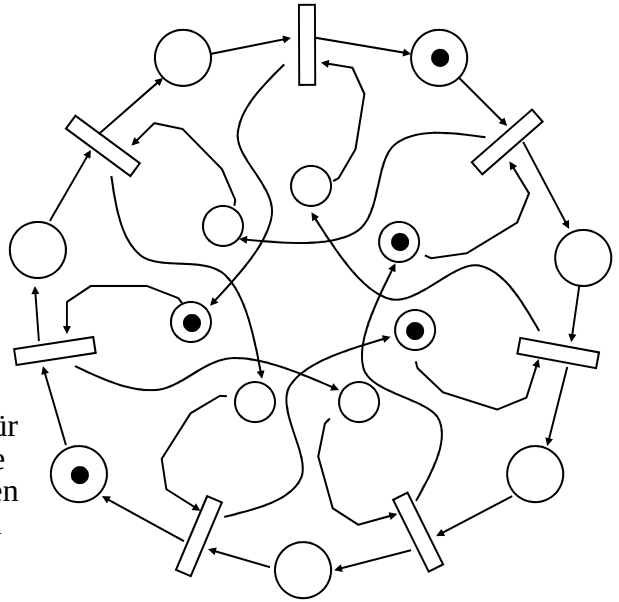
Kreise für die Strecken und Ampeln ("Stellen")



Pfeile (=Kanten) für Vorausset-zungen und Auswirkungen (Flussrelation)

Wir lassen nun die Strecken, Ampeln und Züge weg und erhalten das "reine Netz".

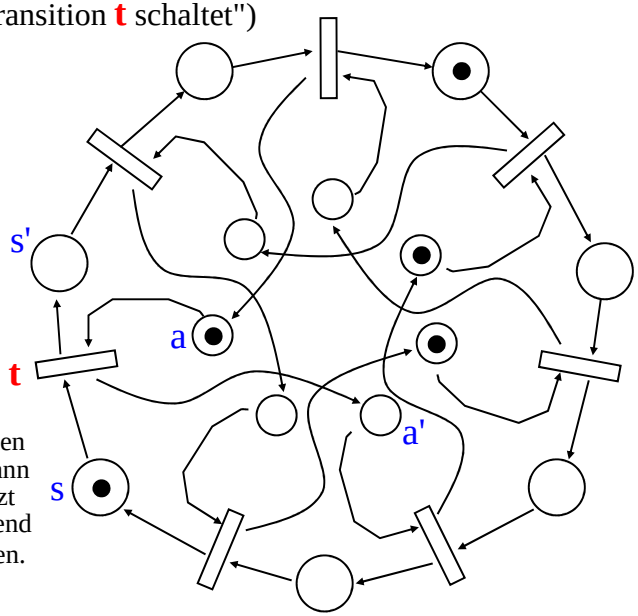
Das resultierende Netz:



Marken für mögliche Situationen einfügen

Eine Aktivität bei **t** durchführen ("die Transition **t** schaltet")

Die Transition t "schaltet":

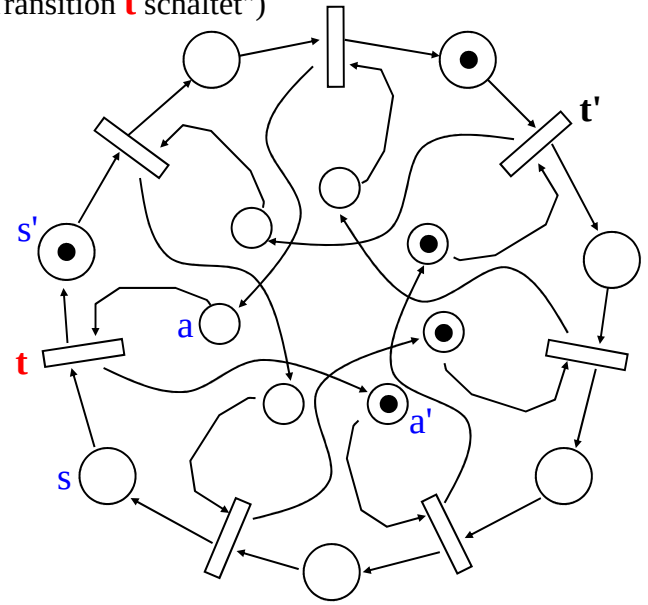


Die Marken werden dann umgesetzt entsprechend den Kanten.

Ein Zug kann von Strecke s nach Strecke s' fahren (= in Stelle s liegt eine Marke). Die Ampel steht auf grün (= in der Stelle a liegt eine Marke). Der Zug fährt nun von s nach s' (= die Transition t schaltet).

Eine Aktivität bei **t** durchführen ("die Transition **t** schaltet")

Ergebnis des Schaltens:



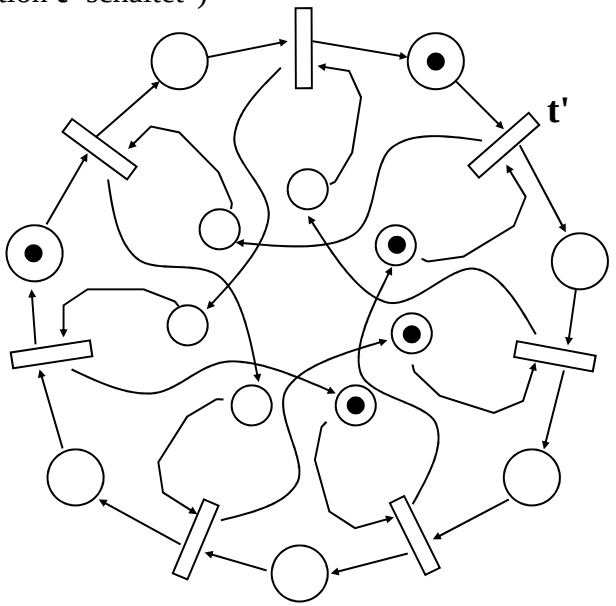
Der Zug ist nun auf der Strecke s' (= in Stelle s' liegt eine Marke). Die Ampel a steht auf rot (= in der Stelle a liegt keine Marke). Dafür wird aber a' auf grün gestellt (= in Stelle a' liegt eine Marke).

Eine Transition t schaltet bedeutet also:
Voraussetzung: Auf allen Stellen, von denen eine Kante nach t führt, muss mindestens eine Marke liegen.
Aktion: Von jeder dieser Stellen wird eine Marke abgezogen. Danach wird zu jeder Stelle, zu der eine Kante von t führt, eine Marke hinzugefügt.

Man nennt dies die "Schaltregel". Wir werden sie im Folgenden exakt definieren.

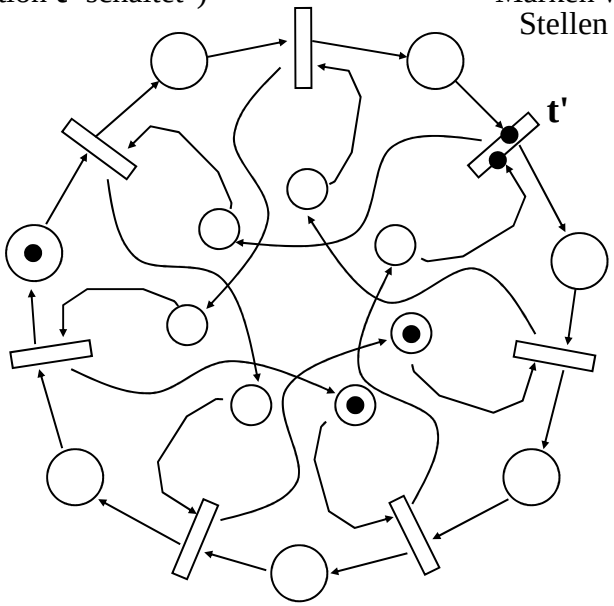
In unserem Beispiel: Der linke Zug kann nicht weiterfahren, weil die zugehörige Ampel keine Marke enthält. Aber der rechte Zug kann weiterfahren, d. h., die Transition t' kann jetzt schalten. Das Ergebnis (= die neue Verteilung der Marken) finden Sie auf den nächsten Folien.

Nächste Aktivität durchführen
("die Transition **t'** schaltet")



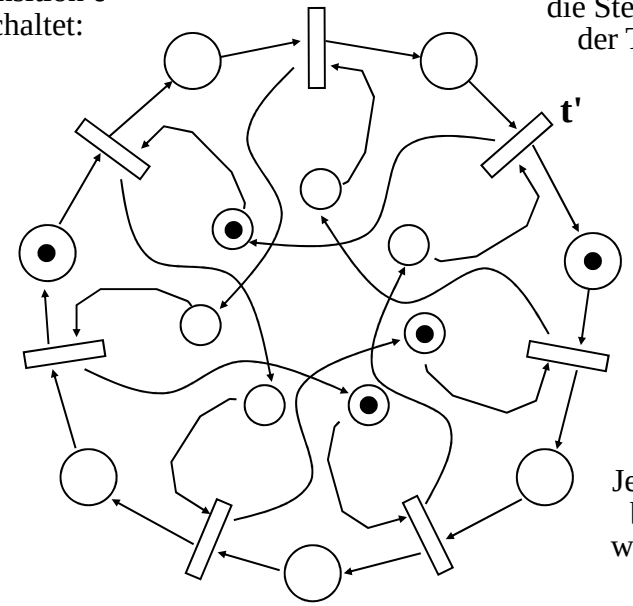
Nächste Aktivität durchführen
("die Transition **t'** schaltet")

Zwischensituation:
Marken werden von Stellen abgezogen



Die Transition **t'** hat geschaltet:

Marken werden auf die Stellen hinter der Transtition gelegt



Jetzt können beide Züge weiterfahren usw.

Definition 13.1.2: Stellen-Transitionsnetz (S/T-Netz)

$N = (S, T, F, K, W, M_0)$ heißt Stellen-Transitions-Netz $\Leftrightarrow$

(1) S ist eine endliche Menge (Menge der "Stellen"),

(2) T ist eine endliche Menge (Menge der "Transitionen"),

(3) $F \subseteq (S \times T) \cup (T \times S)$ ist die "Flussrelation" (Kantenmenge),

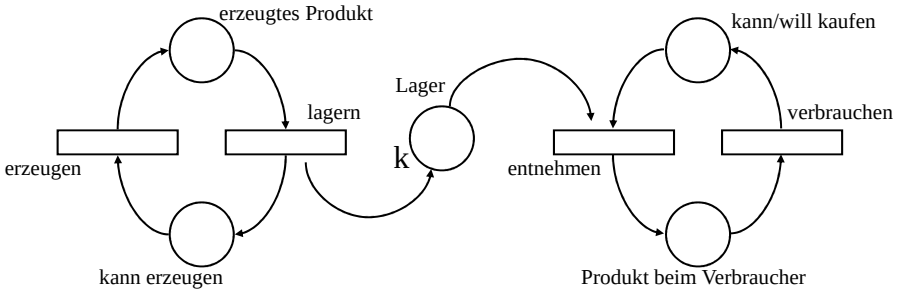
(4) $K: S \to \mathbb{N} \cup \{\infty\}$ ist die Kapazität für jede Stelle,

(5) $W: F \to \mathbb{N}$ ist die Gewichtsfunktion ("weight") der Kanten,

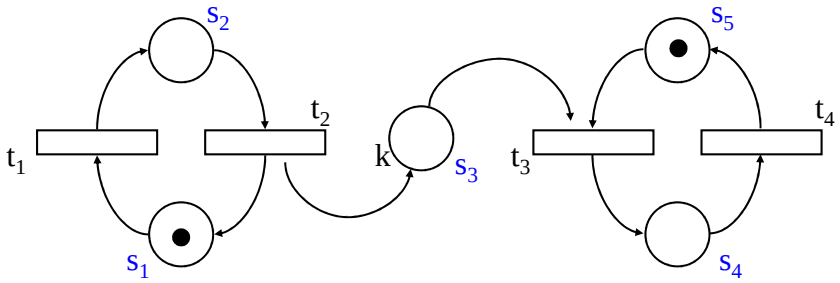
(6) $M_0: S \to \mathbb{N}_0 \cup \{\infty\}$ ist die Anfangsmarkierung, für die gelten muss: $\forall s \in S: M_0(s) \le K(s)$, d.h., in keiner Stelle dürfen mehr Marken liegen, als die Kapazität zulässt (die Markierungen schreibt man in der Regel als Vektoren).

Beispiel 13.1.3: Erzeuger-Verbraucher-Kreislauf (Producer-Consumer-Cycle)

Ein Erzeuger erzeugt ein Produkt, legt dieses in einem Lager, das maximal $k \ge 1$ Stellplätze besitzt, ab und wiederholt diesen Prozess.
Ein Verbraucher entnimmt ein Produkt aus dem Lager, konsumiert dieses und wiederholt diesen Prozess.

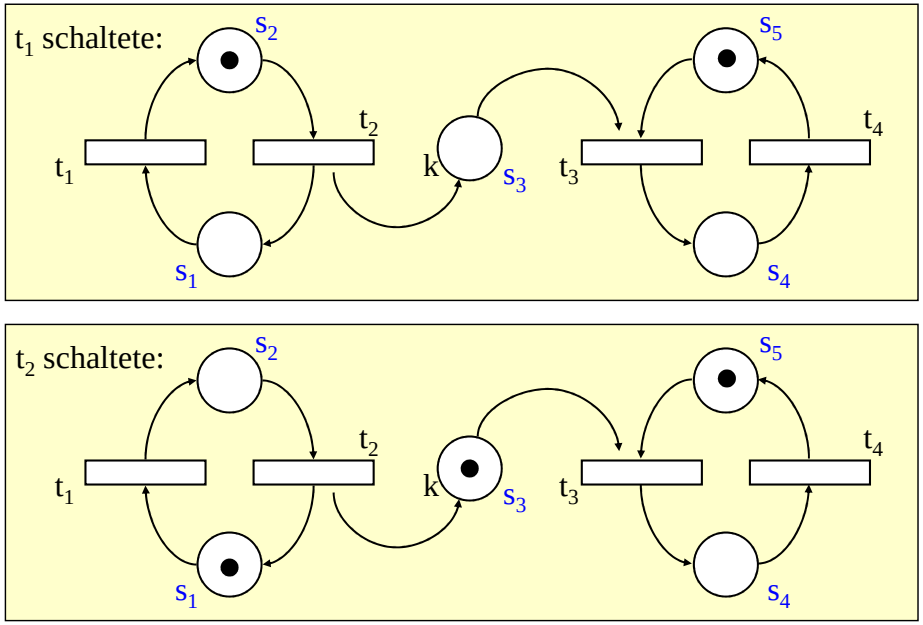


Hinweis: Das Lager hat die Kapazität k, alle anderen Stellen haben die Kapazität ∞.



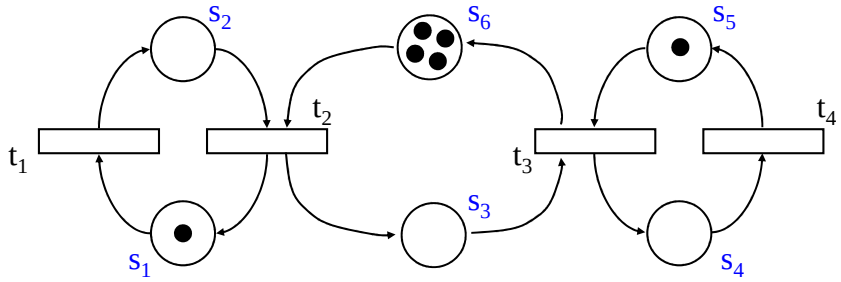
Verbraucher-Erzeuger-System
mit Anfangsmarkierung $M_0 = (1,0,0,0,1)$.

Formal: $S = \{s_1, s_2, s_3, s_4, s_5\}$, $T = \{t_1, t_2, t_3, t_4\}$, $M_0 = (1,0,0,0,1)$,
$F=\{(s_1,t_1),(s_2,t_2),(t_1,s_2),(t_2,s_1),(t_2,s_3),(s_3,t_3),(s_5,t_3),(t_3,s_4),(s_4,t_4),(t_4,s_5)\}$,
$W((x,y))=1$ für alle Kanten (x,y), $K(s_1)=K(s_2)=K(s_4)=K(s_5)=\infty$, $K(s_3)=k$.
In dieser Anfangssituation kann nur die Transition t_1 schalten.
Aus der Anfangsmarkierung (1,0,0,0,1) ensteht dann die Folge-Markierung (0,1,0,0,1). Nun kann t_2 schalten und es entsteht die Markierung (1,0,1,0,1). Jetzt können t_1 oder t_3 schalten, wobei die Markierungen (0,1,1,0,1) bzw. (1,0,0,1,0) entstehen usw.



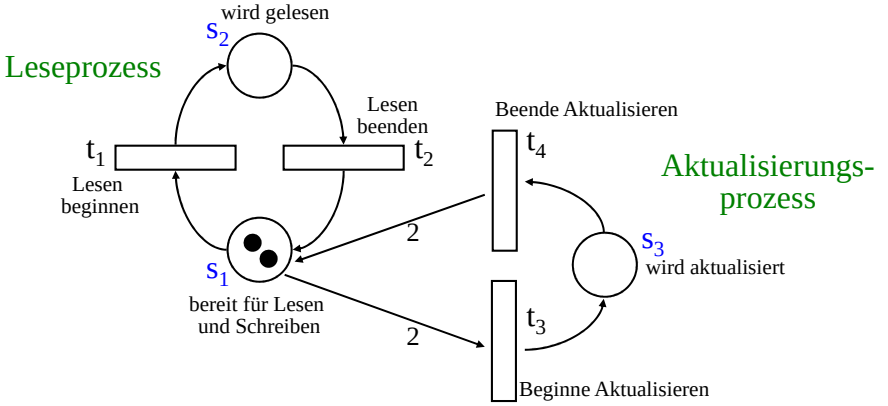
Hinweis: "Komplementärstellen"

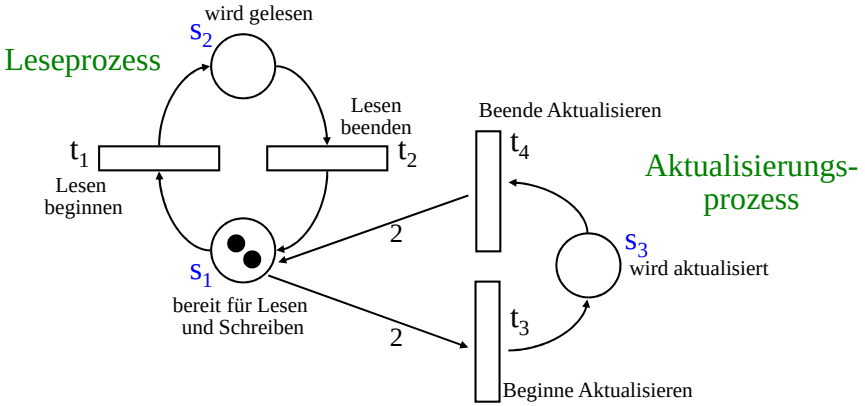
Man kann die Kapazitätsfunktion stets auf den Wert ∞ setzen, indem man für eine Stelle, deren Kapazität nicht ∞ ist, eine neue Stelle "freie Plätze" einführt, diese Stelle "geeignet" verbindet und sie anfangs mit k Marken füllt. Wir demonstrieren dies nur am Beispiel. Sei k = 4 in unserem Beispiel. Wir führen für das Lager s_3 eine weitere ("Komplementär-") Stelle s_6 (= "freie Plätze" in s_3) ein und betrachten nun anstelle des bisherigen Netzes das folgende Netz:



Beispiel 13.1.4: Lese-Schreib-Konflikt

Eine Datenbank steht zwei Benutzern zum Lesen zur Verfügung. Ab und zu sollen die Inhalte von einem Autor aktualisiert werden; zu diesem Zeitpunkt darf nur der Autor auf die Datenbank zugreifen können. Modell hierzu?



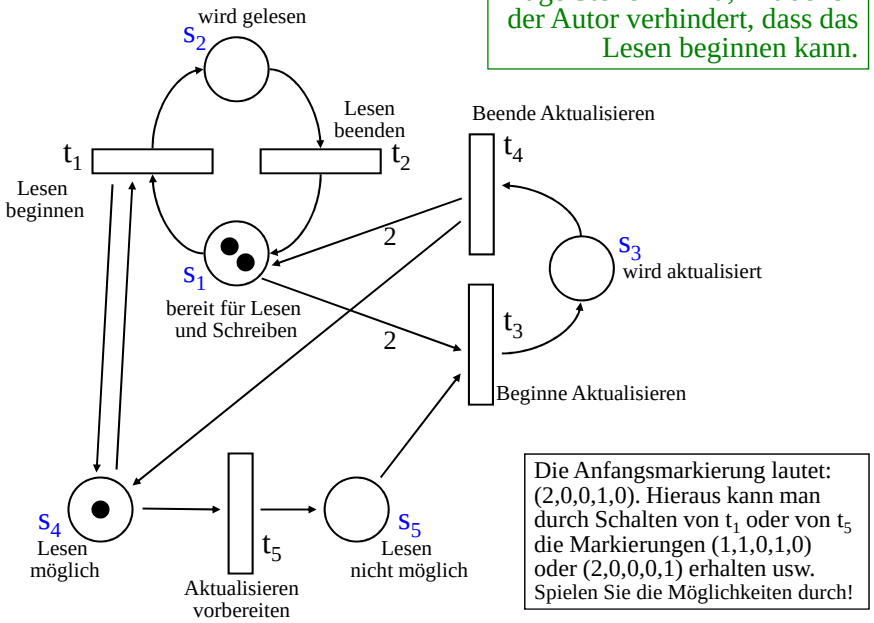


Nun kann eine unerwünschte Folge von Transitionen auftreten:

$t_1\, t_1\, t_2\, t_1\, t_2\, t_1\, t_2\, t_1\, t_2\, t_1\, t_2\, \ldots$

Hierbei ist stets mindestens eine Marke in der Stelle s_2, so dass niemals aktualisiert werden kann. Wie kann man erzwingen, dass der Autor das Lesen unterbrechen kann?

Füge Stellen hinzu, mit denen der Autor verhindert, dass das Lesen beginnen kann.



Die Anfangsmarkierung lautet: (2,0,0,1,0). Hieraus kann man durch Schalten von t_1 oder von t_5 die Markierungen (1,1,0,1,0) oder (2,0,0,0,1) erhalten usw. Spielen Sie die Möglichkeiten durch!

13.1.5 Fragen :

1. Erreichbarkeitsproblem: Wie kann man feststellen, ob eine angestrebte Situation (= Markierung) von einer gegebenen Situation (= Markierung) aus erreicht werden kann?
2. Beschränktheit: Wie kann man nachweisen, dass die Zahl der Marken in allen Stellen beschränkt bleibt?
3. Wie kann man beweisen, dass das Netz nicht in "Verklemmungen" gerät, also in eine Markierung, von der aus es nicht mehr weitergeht?
4. Wie stellt man sicher, dass das Netz nicht in "sinnlose Schleifen" (z. B: Iteration von schaltenden Transitionen, die letztlich nichts verändern) gerät?

Um diese Fragen beantworten zu können, müssen wir zuerst die Arbeitsweise und die erforderlichen Begriffe exakt definieren.

Definition 13.1.6: Begriffe und Schreibweisen

N = (S, T, F, K, W, M_0) sei ein Stellen-Transitions-Netz.

(1) Jede Abbildung M: $S \to \mathbb{N}_0 \cup \{\infty\}$ heißt Markierung von N. M heißt zulässig, wenn für alle $s \in S$ gilt: $M(s) \le K(s)$.

(2) Eine Markierung schreibt man in der Regel als Spaltenvektor (oder in Texten auch als Zeilenvektor). Hierbei wird vorausgesetzt, dass die Menge der Stellen S geordnet ist: $S = \{s_1, s_2, s_3, \dots, s_n\}$ mit $s_1 < s_2 < s_3 < \dots < s_n$

(3) Für $x \in S \cup T$ heißen ${}^\bullet x = \{(y,x) \mid (y,x) \in F\}$ der Vorbereich von x und $x^\bullet = \{(x,y) \mid (x,y) \in F\}$ der Nachbereich von x.

(4) Die Flussrelation $F \subseteq (S \times T) \cup (T \times S)$ zusammen mit der Gewichtsfunktion W wird meist auf die gesamte Menge $(S \times T) \cup (T \times S)$ wie folgt zu W': $S \to \mathbb{N}_0$ fortgesetzt:
W'((x,y)) := if $(x,y) \in F$ then W((x,y)) else 0 fi.
(W' beschreibt F und W bereits eindeutig.)

Definition 13.1.7: Arbeitsweise von S/T-Netzen

N = (S, T, F, K, W, M_0) sei ein Stellen-Transitions-Netz.

(1) Eine Transition $t \in T$ heißt unter der Markierung M aktiviert
$\Leftrightarrow \forall s \in {}^\bullet t: W((s,t)) \le M(s)$ und $\forall s \in t^\bullet: W((t,s)) + M(s) \le K(s)$.
Man schreibt hierfür: $M[t>$.

(2) *Schaltregel:* Es sei t eine Transition, die unter der zulässigen Markierung M aktiviert ist. Dann kann t schalten und es entsteht aus M die (zulässige) Folge-Markierung M' mit:

$$M'(s) = \begin{cases} M(s) & s \notin {}^\bullet t \text{ und } s \notin t^\bullet, \\ M(s) - W((s,t)) & s \in {}^\bullet t \text{ und } s \notin t^\bullet, \\ M(s) + W((t,s)) & s \notin {}^\bullet t \text{ und } s \in t^\bullet, \\ M(s) - W((s,t)) + W((t,s)) & s \in {}^\bullet t \text{ und } s \in t^\bullet. \end{cases}$$

[Wir hätten auch $M'(s) = M(s) - W'((s,t)) + W'((t,s))$, $\forall s \in S$ schreiben können, siehe Definition 13.1.6 (4).]

noch Definition 13.1.7:

(3) Schreibweise: Wenn t unter M aktiviert ist und nach dem Schalten von t die Markierung M' entsteht, so schreibt man $M[t>M'$ oder $M \xrightarrow{t} M'$.

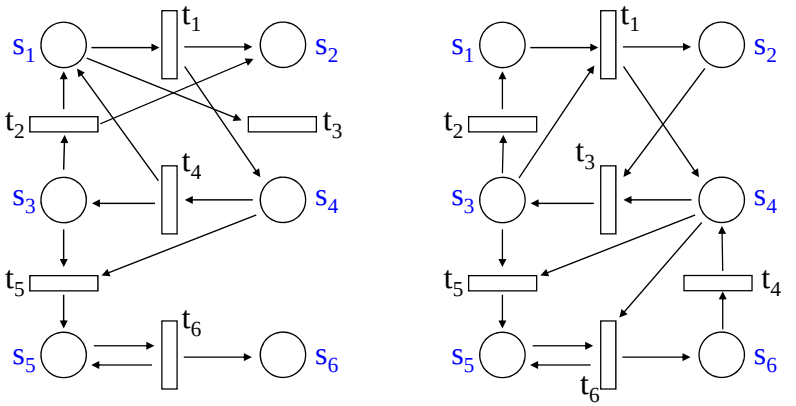
(4) Fortsetzung der einstelligen Relation $[>$ auf Folgen von Transitionen (also auf T^*):
$M[t_1 t_2 t_3 \dots t_r> \Leftrightarrow$
es gibt Markierungen $M_1, M_2, \dots, M_{r-1}$ mit
$M[t_1>M_1$, $M_1[t_2>M_2$, $M_2[t_3>M_3$, ..., $M_{r-2}[t_{r-1}>M_{r-1}$
und $M_{r-1}[t_r>$.

noch Definition 13.1.7:

(5) Fortsetzung der Relation [> auf Folgen von Transitionen):
$M[t_1 t_2 t_3 ... t_r > M' \Leftrightarrow$ es gibt Markierungen $M_1, M_2, ..., M_{r-1}$ mit $M[t_1 > M_1$, $M_1[t_2 > M_2$, $M_2[t_3 > M_3$, ..., $M_{r-1}[t_r > M'$.
(Im Falle $r = 0$ muss $M = M'$ sein.)

(6) ERR(M) = {M' | es existiert $t_1 t_2 t_3 ... t_r$ mit $M[t_1 t_2 t_3 ... t_r > M'$} heißt Erreichbarkeitsmenge bzgl. M.
(Beachte: $M \in$ ERR(M), insbesondere ist ERR(M) nie leer.)
Wenn M' in ERR(M) liegt, so sagt man auch, M' ist von M aus erreichbar.
ERR(N) = {M' | es existiert $t_1 t_2 t_3 ... t_r$ mit $M_0[t_1 t_2 t_3 ... t_r > M'$} heißt Erreichbarkeitsmenge des Netzes N.

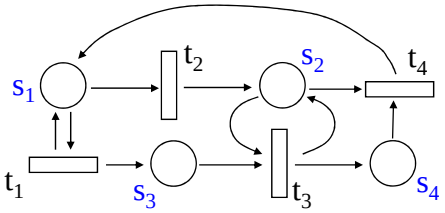
1.13.8 Beispiele zum "Spielen": Starten Sie bei irgendeiner Anfangsmarkierung, z.B. (1,0,0,0,0,0,), und prüfen Sie, ob Sie (1,1,1,1,1,1) erreichen können. (K = ∞, W = 1)



Untersuchen Sie das folgende Netz.

Zeigen Sie: Von der Anfangsmarkierung (1,0,0,0) sind alle Markierungen (1,0,0,a) für jedes $a \geq 0$ erreichbar.

Kann man auch (1,1,1,1) erreichen?



Oft schreibt man S/T-Netze nur in der Form N = (S, T, F, M_0). In diesem Fall ist K(s) = ∞ für alle Stellen s und W((x,y)) = 1 für alle Kanten (x,y) einzusetzen.
Dies gilt auch für Zeichnungen: Fehlen die Angaben für K oder W an einer Stelle bzw. Kante, so ist unbeschränkte Kapazität bzw. Kantengewicht 1 gemeint.
In diesem Kapitel sprechen wir oftmals nur von einem "Netz" und meinen damit stets ein S/T-Netz.

Die Erreichbarkeit stellt die Bedeutung der S/T-Netze dar.

Um die Arbeitsweise eines Netzes im Ganzen zu verstehen, konstruiert man schrittweise alle Markierungen, die man von der Anfangsmarkierung M_0 aus erreichen kann. Das Ergebnis ist der "Erreichbarkeitsgraph" des Netzes.

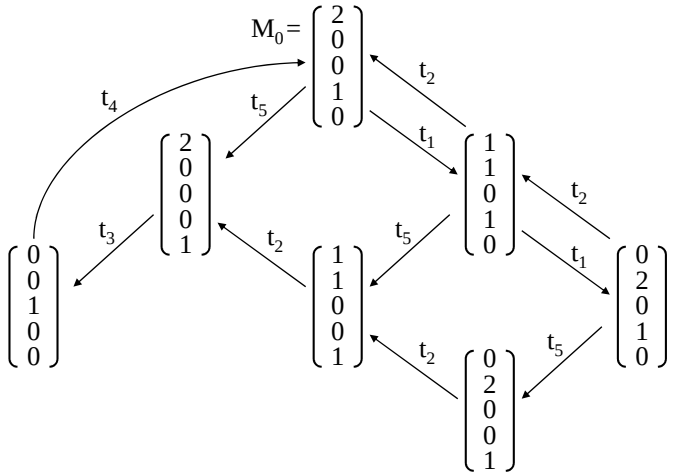
13.1.9 Erreichbarkeitsgraph

Es sei N = (S, T, F, K, W, M_0) ein Netz mit der (eventuell unendlich großen) Erreichbarkeitsmenge ERR(M_0) = ERR(N).
Der gerichtete, Kanten markierte Graph (ERR(N), E, γ) heißt
Erreichbarkeitsgraph von N ⇔

$$E = \{(M, t, M') \mid M, M' \in ERR(N) \text{ und } M[t>M'\}.$$

Anmerkungen:
Diese Darstellung ist formal nicht ganz korrekt: Eigentlich ist (M, M') eine Kante und $t = \gamma(M, M')$ für jedes (M, t, M') ∈ E. Man beachte hier, dass Mehrfachkanten möglich sind, die aber jeweils eine andere Markierung $\gamma(M, M') \in T$ besitzen. Daher wählen wir die Tripel-schreibweise (M, t, M'), die die Kanten einschl. ihrer Markierung $\gamma(M, M') = t$ bezeichnet.
Wie man einen Erreichbarkeitsgraphen konstruiert, ist offensichtlich: Man starte mit M_0 und füge jeweils die nächsten direkt erreichbaren Markierungen mit den zugehörigen Transitionen hinzu. Man breche ab, wenn keine neuen Markierungen mehr hinzukommen oder wenn eine Zeitschranke überschritten wurde.

13.1.10: *Konstruktion des Erreichbarkeitsgraphen* des letzten S/T-Netzes aus Beispiel 13.1.4: Ausgehend von M_0 werden nach-einander alle im nächsten Schritt erreichbaren Markierungen notiert:

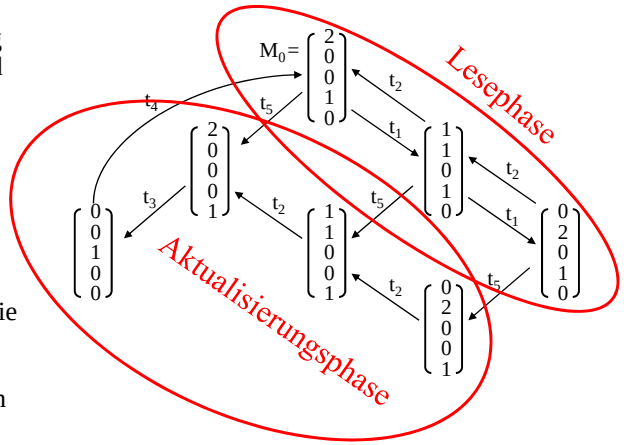


Dieser Graph besitzt einige Eigenschaften, die uns Hinweise geben, ob das angegebene Netz unser gestelltes Lese-Schreib-Problem tatsächlich löst:

1. Man erkennt die (korrekte) Trennung zwischen Lesen und Schreiben

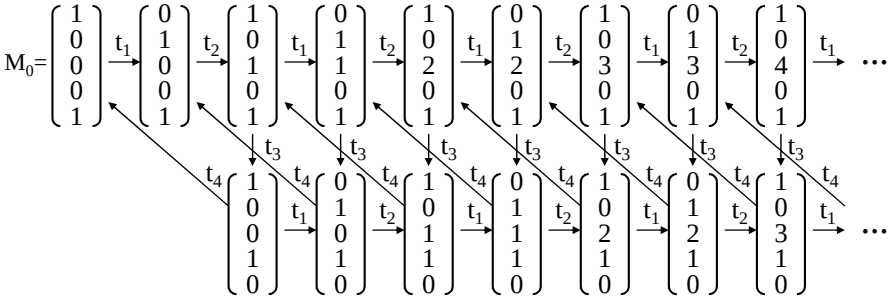
2. Man entdeckt eine "Invariante" der erreichbaren Markierungen M: (1,1,3,1,1)·M = 3.

3. Man sieht, dass die Transitionenfolgen $(t_1t_2)^k$ und $(t_5t_3t_4)^k$ korrekt von M_0 nach M_0 führen ($\forall k \in \mathbb{N}$).



4. Man erkennt, dass jede Transition irgendwann noch einmal schalten kann, dass also keine Transition irgendwann überflüssig wird.

13.1.11: Konstruktion des Erreichbarkeitsgraphen aus 13.1.3



Falls die in 13.1.3 angegebene Größe k eine natürliche Zahl ist, so besitzt der Erreichbarkeitsgraph genau 4k+2 Markierungen.
Ist die Kapazität des Lagers k = ∞, so ist der Ereichbarkeitsgraph unendlich groß.

13.1.12: Beschränktheit eines Netzes

Ein Netz soll beschränkt heißen, wenn es eine natürliche Zahl k gibt, so dass keine Markierung im Netz erreicht werden kann, in der eine Stelle mehr als k Marken besitzt.

Definition: Sei $N = (S, T, F, K, W, M_0)$ ein S/T-Netz.

(1) Es sei k eine natürliche Zahl. Eine Stelle s des Netzes N heißt k-beschränkt, wenn für jede von M_0 aus erreichbare Markierung M gilt, M(s) ist nicht größer als k, d.h., $\forall M \in ERR(N): M(s) \leq k$.

(2) N heißt k-beschränkt, wenn jede Stelle k-beschränkt ist, d.h., $\forall s \in S \; \forall M \in ERR(N): M(s) \leq k$.

(3) s heißt beschränkt, wenn es eine natürliche Zahl k gibt, so dass s k-beschränkt ist, d.h., $\exists k \in \mathbb{N} \; \forall M \in ERR(N): M(s) \leq k$.

(4) N heißt beschränkt, wenn jede Stelle von N beschränkt ist, d.h., $\exists k \in \mathbb{N} \; \forall s \in S \; \forall M \in ERR(N): M(s) \leq k$.

13.1.13: Folgerung

Ein Netz ist genau dann beschränkt, wenn sein Erreichbarkeitsgraph endlich ist.

Beweis: Sei S die Menge der Stellen des Netzes N.

"⇒" Wenn N beschränkt ist, dann gibt es ein k, so dass alle Markierungen des Erreichbarkeitsgraphen nur Komponenten besitzen, die kleiner oder gleich k sind. Dann kann es aber höchstens $(k+1)^{|S|}$ Markierungen in ERR(N) geben, d.h., der Erreichbarkeitsgraph ist endlich.

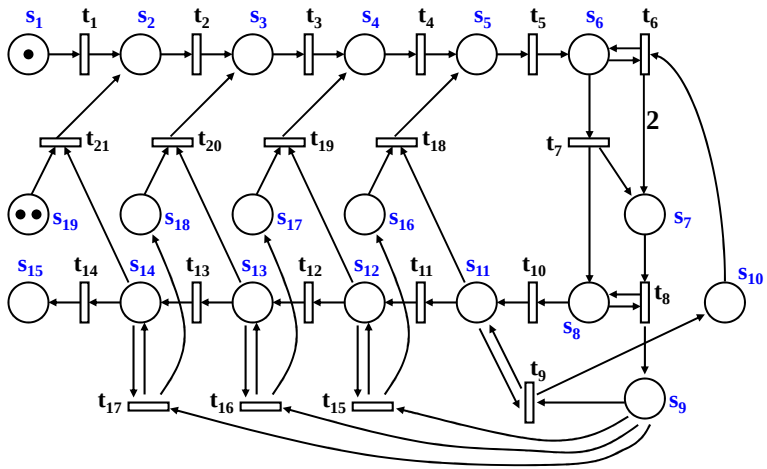
"⇐" Wenn der Erreichbarkeitsgraph endlich ist, dann existiert das Maximum m für alle Komponenten von Markierungen in ERR(N). Jede Stelle ist dann m-beschränkt, d.h., das Netz ist beschränkt.

Beispiele: Der Erreichbarkeitsgraph in 13.1.10 ist endlich und das Netz ist 2-beschränkt; das Netz in 13.1.11 ist k-beschränkt bzw. (falls die Kapazität von s_3 unendlich ist) unbeschränkt

Warnung: Die Erreichbarkeitsgraphen von S/T-Netzen können gewaltig wachsen.

Das S/T-Netz auf der folgenden Folie mit 19 Stellen und 21 Transitionen vollzieht die Berechnung der so genannten Ackermann-Funktion A (in der fünften Stufe) nach. Der Erreichbarkeitsgraph dieses S/T-Netzes ist endlich, besitzt aber mehr als

$2^{2^{2^{\cdots^{2}}}}$ (65535 mal) Knoten! Vollziehen Sie etwa 80 Schritte nach, um die Wirkungsweise des Netzes zu erahnen.



13.1.14 Lebendigkeit

Ein Netz soll lebendig heißen, wenn jede Transition irgendwann noch einmal schalten könnte. Genauer: Für jede Transition t muss gelten: Wenn man sich, ausgehend von M_0, in irgendeiner Markierung M befindet, dann muss von M aus eine Markierung M' erreichbar sein, unter der t aktiviert ist (also schalten kann).

Definition: Sei N = (S, T, F, K, W, M_0) ein S/T-Netz.

(1) Eine Transition t des Netzes N heißt lebendig, wenn es zu jeder von M_0 aus erreichbaren Markierung M eine von M aus erreichbare Markierung M' mit M'[t> gibt. Formal: $\forall M \in ERR(N)\ \exists M' \in ERR(M): M'[t>$.

(2) N heißt (stark) lebendig, wenn jede Transition von N lebendig ist.

(3) Eine Transition heißt lebendig bzgl. einer Markierung M $\Leftrightarrow$ $\forall M' \in ERR(M)\ \exists M'' \in ERR(M'): M''[t>$. Ist t nicht lebendig bzgl. M, so sagt man auch, t ist tot bzgl. M

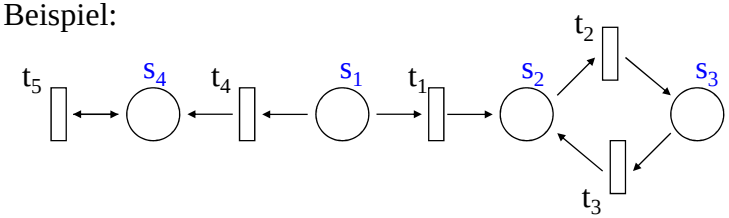
Man könnte ein Netz auch lebendig nennen, wenn es immer weiterschalten kann. Im Netz darf es dann keine "Verklemmung" geben, d.h., es darf keine von M_0 aus erreichbare Markierung ohne Folge-Markierung geben.

Fortsetzung der Definition:

(4) Eine Markierung M heißt Verklemmung ("Deadlock") von N, wenn unter M keine Transition aktiviert ist, d.h. $\neg \exists t \in T: M[t>$.

(5) Das Netz N heißt schwach lebendig, wenn es keine von M_0 aus erreichbare Verklemmung besitzt, d.h. $\forall M \in ERR(N)\ \exists t \in T: M[t>$.

Beispiel:



Keine Transition ist bzgl. (1,0,0,0) lebendig. Also ist das Netz nicht (stark) lebendig.

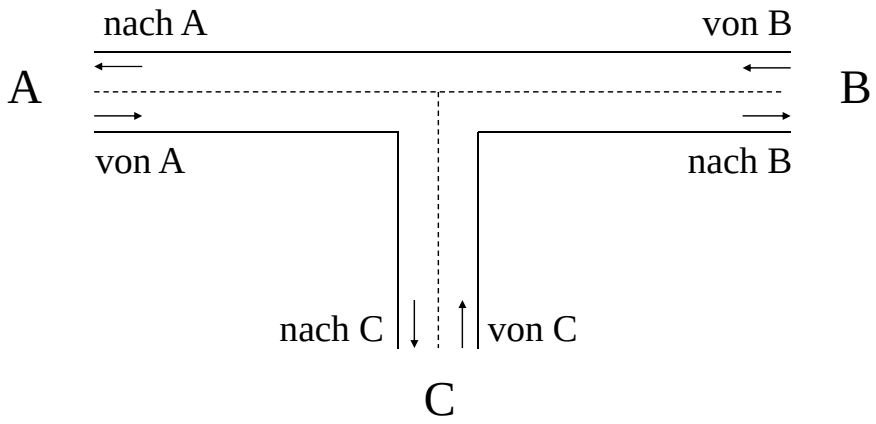
t_1, t_4, t_5 sind bzgl. (0,1,0,0) tot.

t_2, t_3 sind bzgl. (0,1,0,0) lebendig.

Für jede Markierung mit mindestens einer Marke kann das Netz ständig weiterschalten, d.h., es ist schwach lebendig.

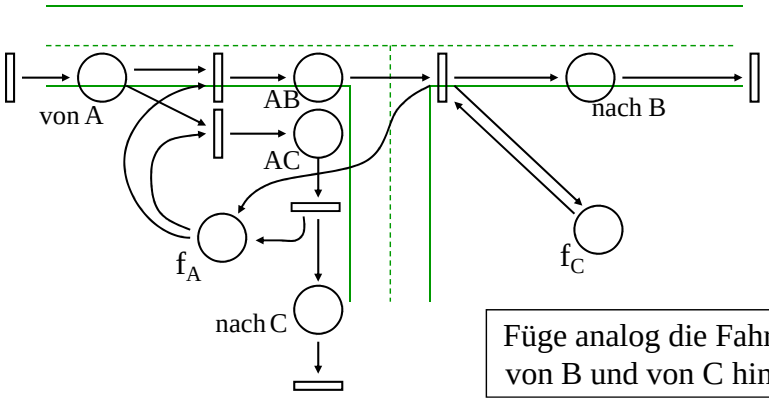
13.1.15: Beispiel

Straßenkreuzung mit Vorfahrtsregel "rechts vor links".



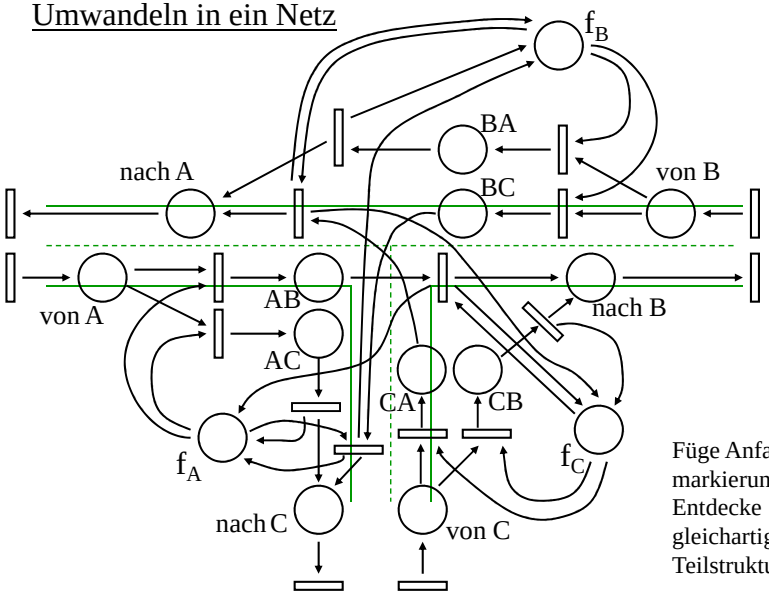
Stellen sind Straßenabschnitte, auf denen maximal ein Fahrzeug stehen kann. Unsere Stellen s sollen daher alle die Kapazität K(s) = 1 besitzen!

Umwandeln in ein Netz, wobei explizit "von rechts kommt niemand" durch die Stellen f modelliert wird. Wir betrachten zunächst die von A kommenden Fahrzeuge, für die nur wichtig ist, ob die Strecke von C zur Kreuzung frei ist (f_C). *(Beachte: Jede Stelle s besitzt hier die Kapazität K(s)=1.)*



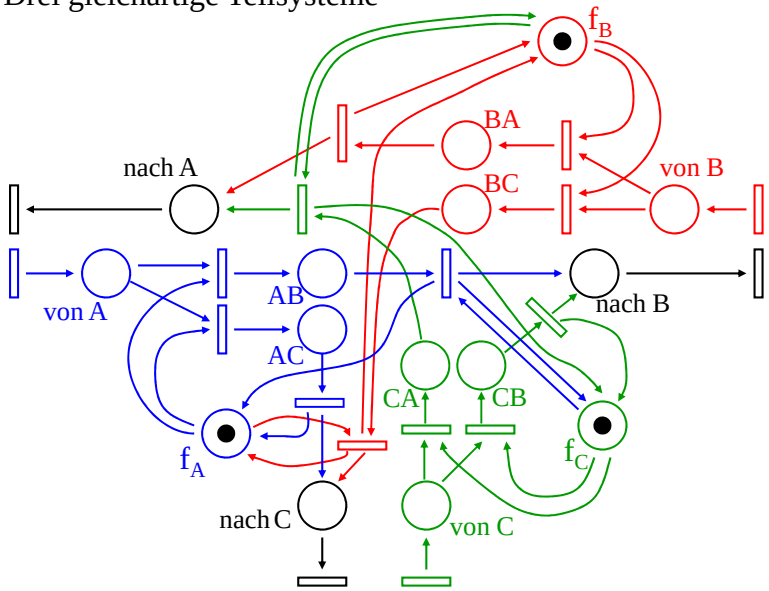
Füge analog die Fahrten von B und von C hinzu.

Umwandeln in ein Netz

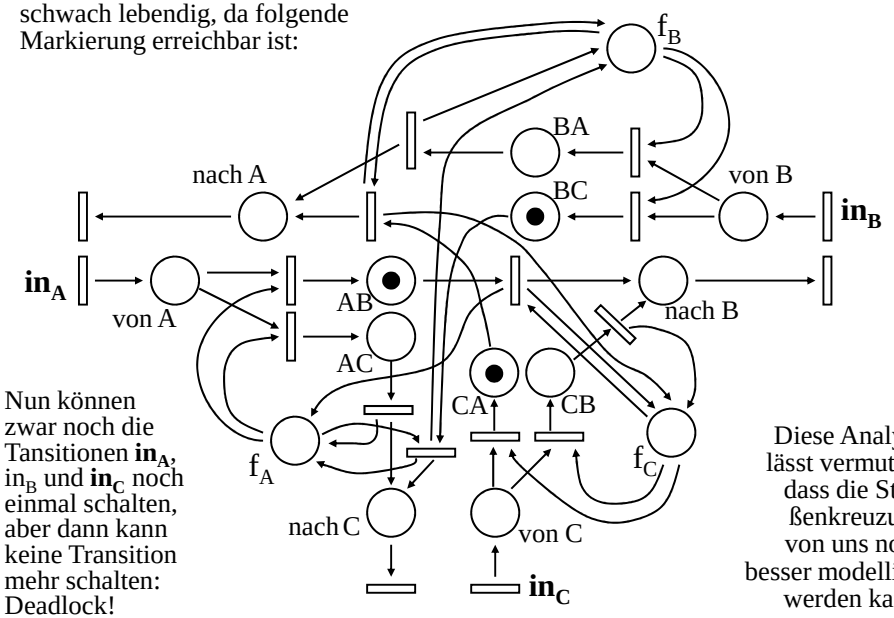


Füge Anfangsmarkierung hinzu. Entdecke gleichartige Teilstrukturen:

Drei gleichartige Teilsysteme



Dieses Netz ist leider nicht schwach lebendig, da folgende Markierung erreichbar ist:

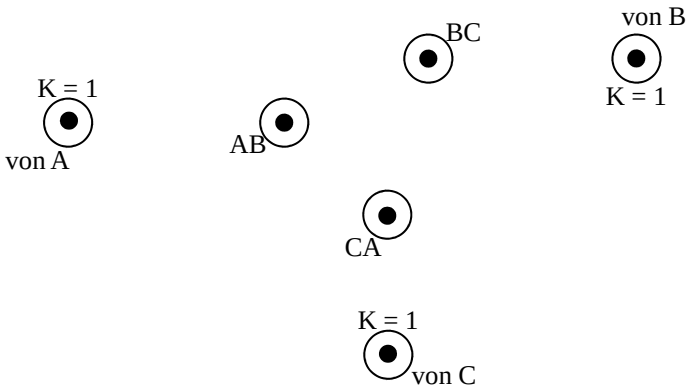


Nun können zwar noch die Tansitionen $\mathbf{in_A}$, in_B und $\mathbf{in_C}$ noch einmal schalten, aber dann kann keine Transition mehr schalten: Deadlock!

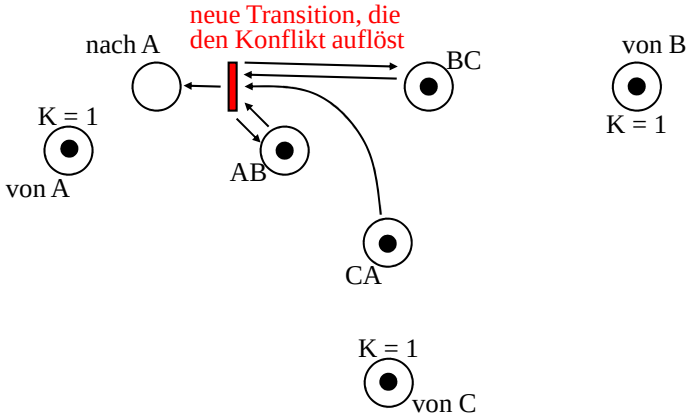
Diese Analyse lässt vermuten, dass die Straßenkreuzung von uns noch besser modelliert werden kann.

Wir hatten alle Stellen, also auch die Eingangs-Stellen "von A", "von B" und "von C" mit der Kapazität 1 belegt.

In diesem Fall gibt es folgende Verklemmung (wir zeigen nur den relevan-ten Ausschnitt aus dem Netz, also die Stellen der Verklemmung, die noch Marken tragen):

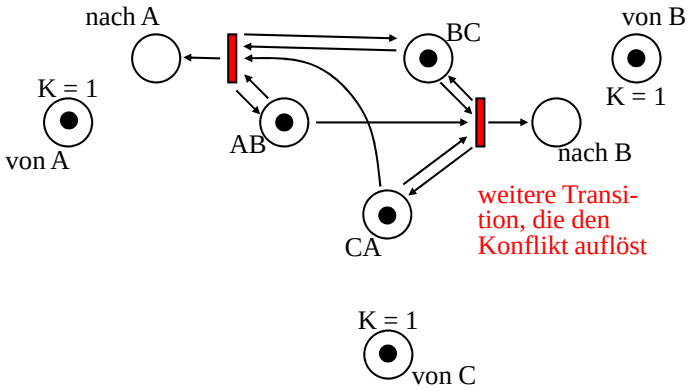


Diese Verklemmung lässt sich beseitigen, indem eine Transition eingefügt wird, die genau die Stellen, die die Verklemmung bewirken, als Vorbereich hat und ihnen eine Folgemarkierung erlaubt. In der folgenden Skizze darf das von C nach A fahrende Fahrzeug nun die Kreuzung überqueren:



Wir fügen auf der nächsten Folie zusätzlich eine Transition hinzu, die das von A nach B fahrende Fahrzeug weiterfahren lässt.

Hier darf nun auch das von A nach B fahrende Fahrzeug über die Kreuzung fahren. Beachten Sie, dass die anderen beiden Fahrzeuge warten, d.h., ihre Marken werden wieder auf BC und CA zurückgelegt.



Als drittes können wir eine Transition hinzufügen, die das von B nach C fahrende Fahrzeug im Konfliktfall über die Kreuzung lässt.

Dies führt zu einem Netz, das relativ realistisch die Situationen an einer Kreuzung widerspiegelt. Es enthält alle Möglichkeiten, eine Verklemmung aufzulösen und unsinnige Überlastungen in gewissen Teilen des Netzes zu vermeiden.

Die Leser(innen) mögen diese Einzelheiten in das bereits vorhandene Netz eintragen und das neue Netz untersuchen.

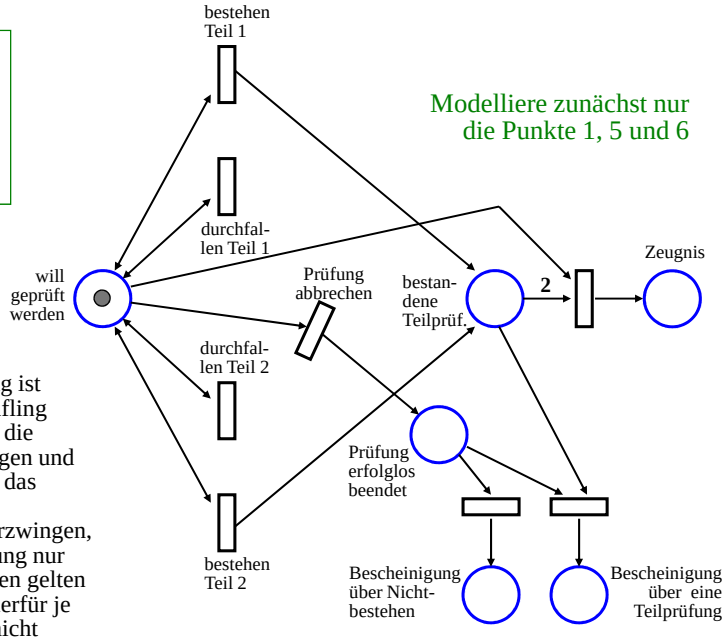
Es ist klar, dass man solche Modelle nun "simulieren" kann, d.h., man kann das reale Verhalten an Kreuzungen im Rechner nachvollziehen, indem man Marken in das System hineingibt und wieder aus ihm herausnimmt und hierbei den Durchsatz, die Konflikte, das Stauverhalten usw. untersucht.

Ende Beispiel 13.1.15

13.1.16 Aufgabe:
Modelliere eine Prüfung mit folgenden Regeln:

1. Die Prüfung besteht aus zwei Teilprüfungen. Die Prüfung ist bestanden, wenn beide Teilprüfungen bestanden sind.
2. Jede Teilprüfung kann einmal wiederholt werden.
3. Eine der Teilprüfungen darf zweimal wiederholt werden.
4. Eine bestandene Teilprüfung darf nicht wiederholt werden.
5. Ist die Prüfung bestanden, wird ein Zeugnis ausgestellt.
6. Sind alle Wiederholungsmöglichkeiten erfolglos ausgeschöpft oder bricht der Prüfling von sich aus die Prüfung ab, so wird eine Bescheinigung über das Nichtbestehen oder über eine bereits bestandene Teilprüfung ausgestellt.

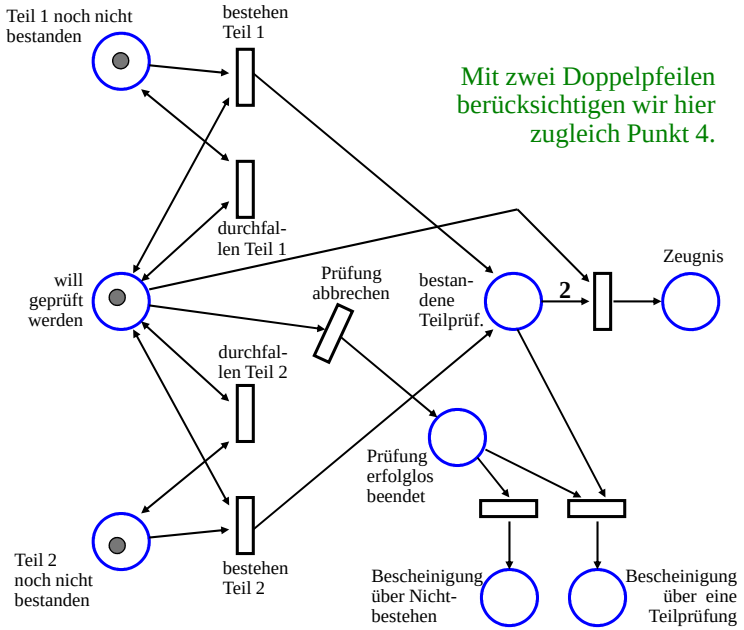
Hinweis:
Statt
⟸⟹
schreiben wir hier
⟺



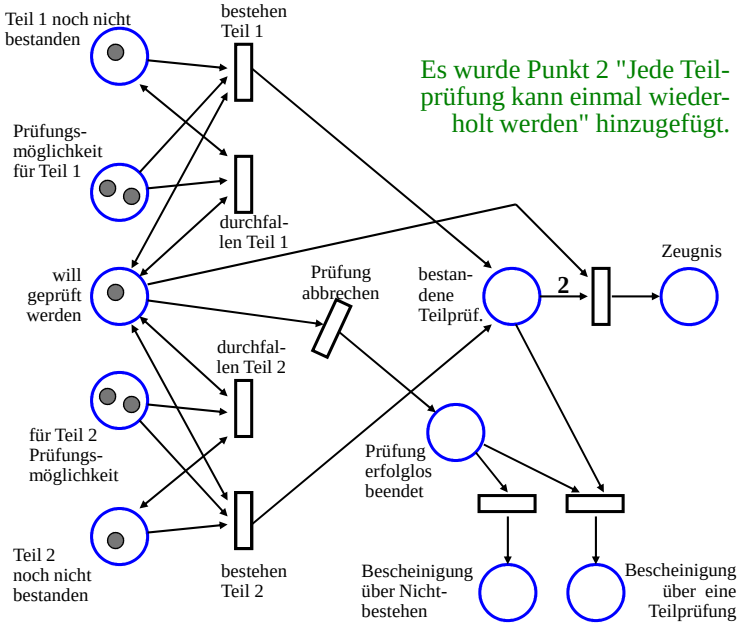
Modelliere zunächst nur die Punkte 1, 5 und 6

Diese Modellierung ist fehlerhaft. Der Prüfling kann hier zweimal die Teilprüfung 1 ablegen und erhält dann bereits das Zeugnis.
Wir müssen also erzwingen, dass jede Teilprüfung nur einmal als bestanden gelten kann. Wir fügen hierfür je eine Stelle "noch nicht bestanden" hinzu.

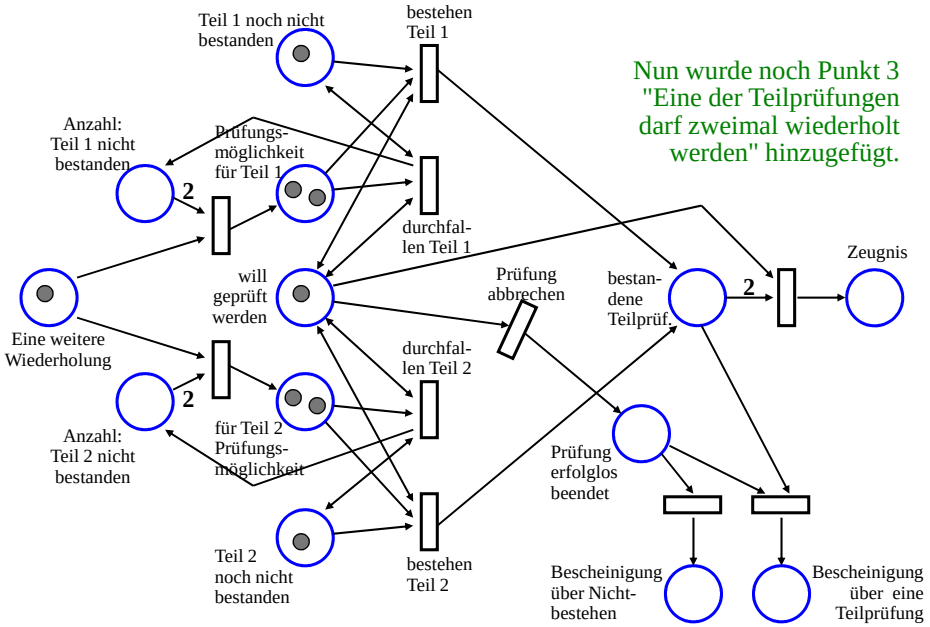
Kritik: Ist es sinnvoll, dass man nach zwei bestandenen Teilprüfungen die Prüfung noch abbrechen kann und dann nur eine Bescheinigung über höchstens eine Teilprüfung erhält? Beachten Sie, dass die Punkte 5 und 6 diesen Fall zulassen. Wir ignorieren dieses Problem hier aber.
(Aufgabe: Verändern Sie die Anforderungen und das Modell geeignet.)



Mit zwei Doppelpfeilen berücksichtigen wir hier zugleich Punkt 4.



Es wurde Punkt 2 "Jede Teilprüfung kann einmal wiederholt werden" hinzugefügt.



Nun wurde noch Punkt 3 "Eine der Teilprüfungen darf zweimal wiederholt werden" hinzugefügt.

Meist versucht man, diese Endsituation zu erreichen. (Dies geht tatsächlich.)

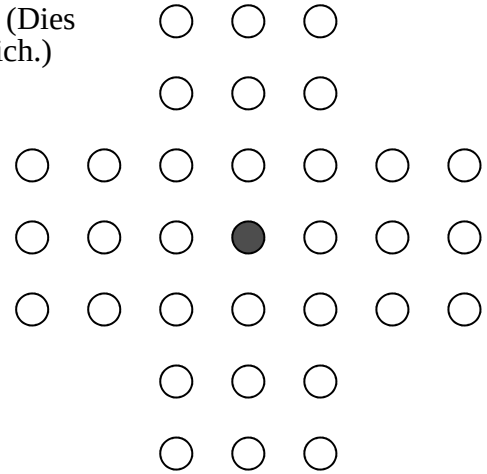
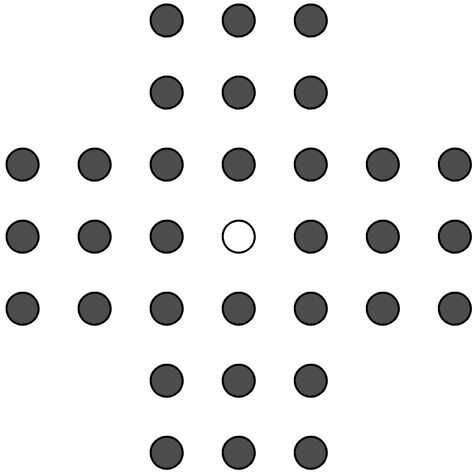
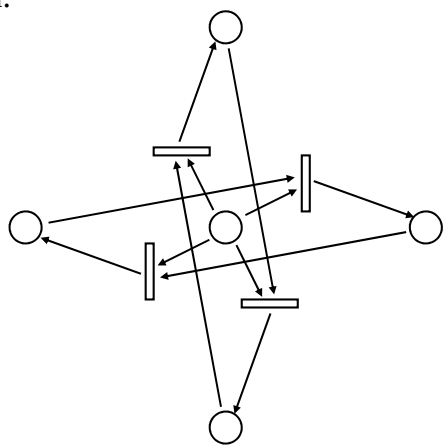


Illustration 13.1.17: Beispiel Solitaire-Spiel

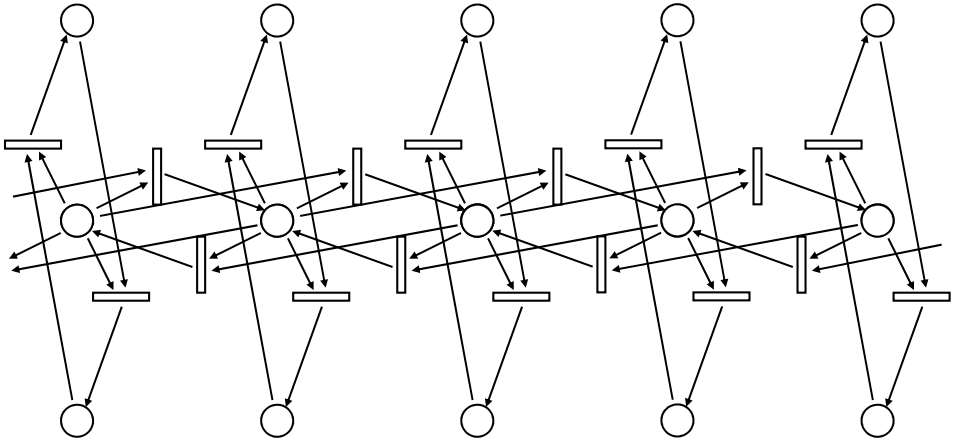


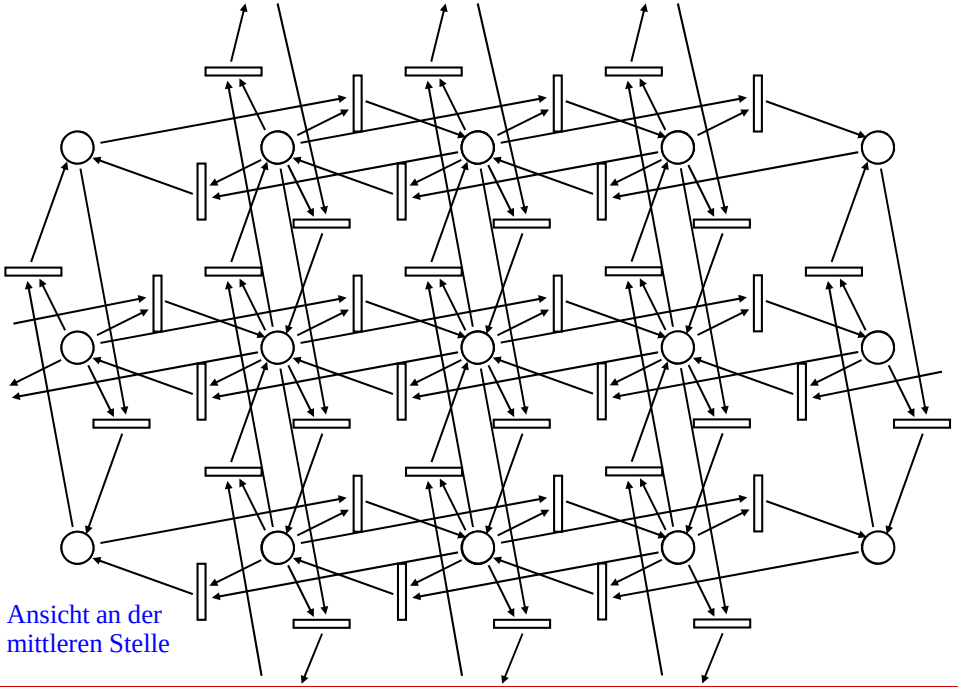
33 Stellen, 32 Steine, d.h., eine Stelle bleibt frei. Ein Zug = Springe in gerader Richtung (nicht diagonal) über einen Nachbarstein auf eine freie Stelle und entferne den übersprungenen Stein. Ziel: Am Ende soll nur noch ein Stein (möglichst auf einer vorgegebenen Stelle) übrig sein.

Solitaire als S/T-Netz formulieren: Betrachte eine Stelle mit ihren Nachbar-Stellen und formuliere das Überspringen durch Transitionen:

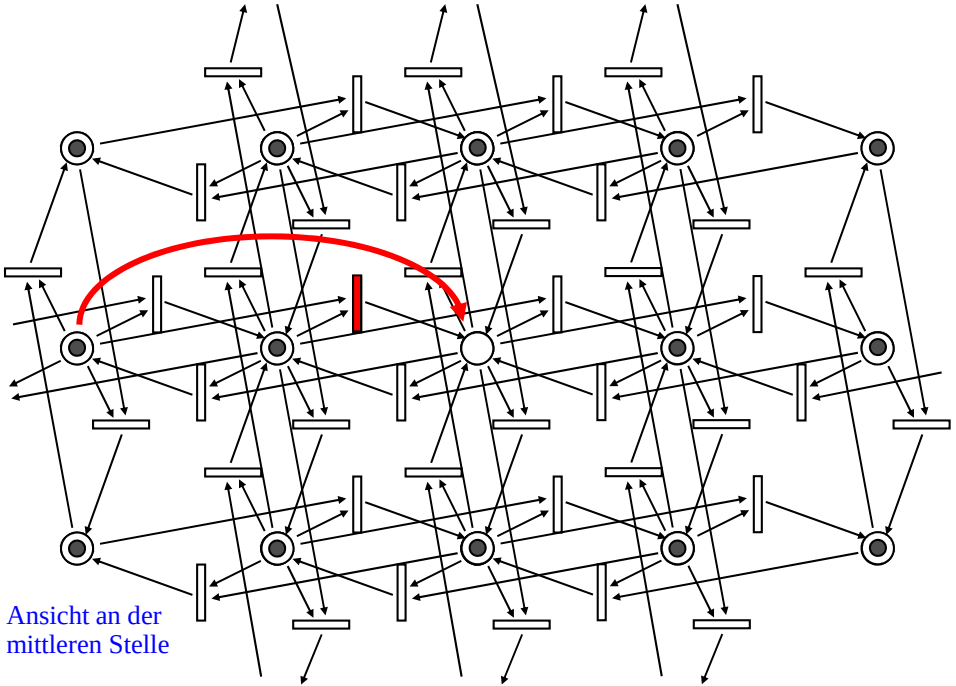


Dieses Muster muss nun an jeder Stelle eingefügt werden.

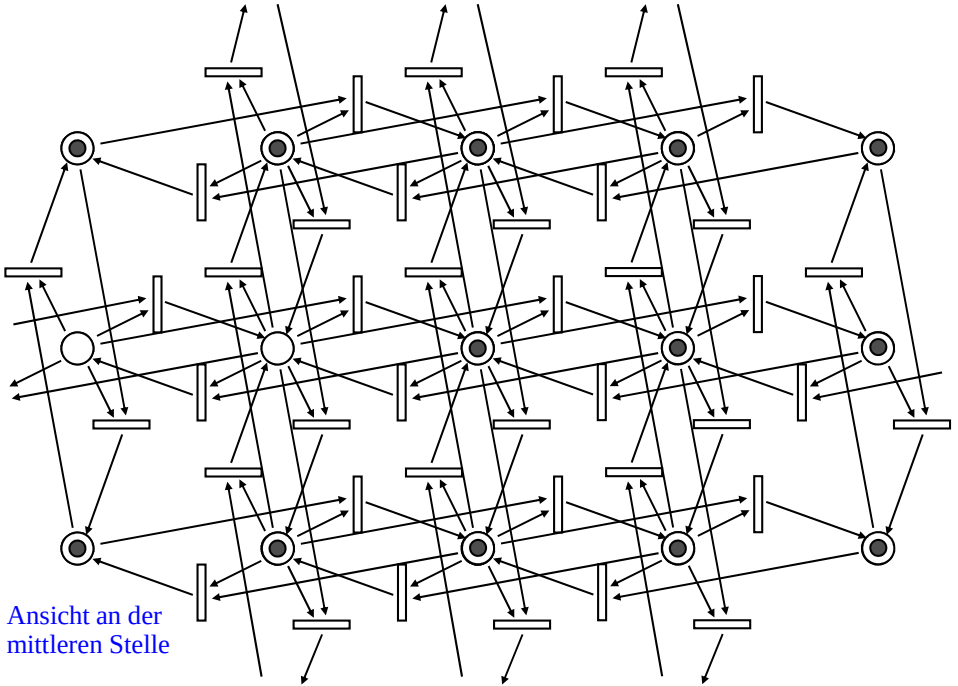




Ansicht an der mittleren Stelle



Ansicht an der mittleren Stelle

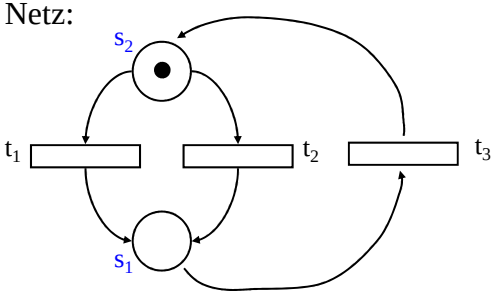


Ansicht an der mittleren Stelle

Das Ziel des Spiels besteht nun darin nachzuweisen, dass von der Anfangsmarkierung
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
eine vorgegebene Markierung, zum Beispiel
(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),
erreicht werden kann.
Zugleich ist eine Transitionenfolge zu finden, die diese Erreichbarkeit bewirkt.

13.1.18 Fairness

Fairness ist eine Eigenschaft, die erst "im Unendlichen" zum Tragen kommt. Sie bedeutet, dass eine Transition, die in einer unendlichen Schaltfolge (= Folge von Markierungen und schaltenden Transitionen) immer wieder geschaltet werden könnte, irgendwann geschaltet werden muss. Betrachte das Netz:



Hier ist die unendliche Folge von Transitionen $t_1t_3t_1t_3t_1t_3 \ldots = (t_1t_3)^\infty$ *möglich*, die verhindert, dass t_2 jemals schaltet. Wenn es solche Möglichkeiten gibt, wird ein Netz als "unfair" bezeichnet.

Die formale Definition muss sich auf unendliche Schaltfolgen beziehen. Endlich lange Schaltfolgen sind per Definition fair. Wir unterscheiden zwei Fälle (schwach und stark fair).

Definition: Wir betrachten eine unendliche Schaltfolge

$$\sigma = M_1[t_1> M_2[t_2> M_3[t_3> \ldots > M_i[t_i> M_{i+1}[t_{i+1}> \ldots$$

(1) Die Schaltfolge σ heißt schwach fair bzgl. einer Transition $t \in T$, wenn gilt: Falls t ab einem Index i ständig aktiviert ist, dann kommt t unter den t_j für $j \geq i$ auch vor. Formal:

$$\exists i \forall j \geq i\ M_j[t> \Rightarrow \exists k \geq i\ (t_k = t).$$

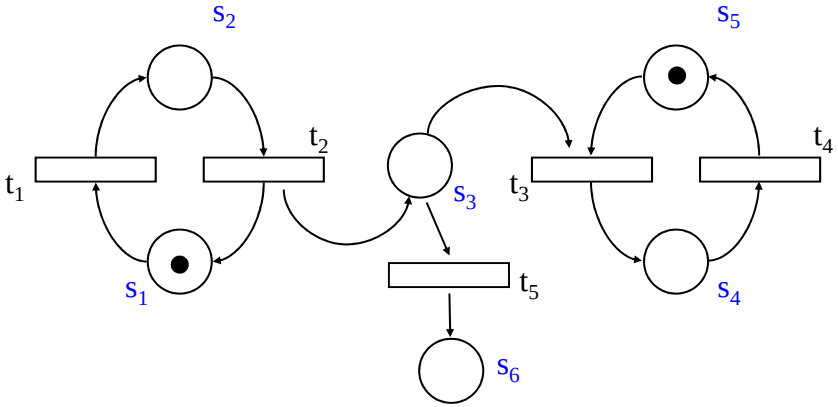
(2) Die Schaltfolge σ heißt stark fair bzgl. $t \in T$, wenn gilt: Ist t ab einem Zeitpunkt i unendlich oft bzgl. der Markierungen von σ aktiviert, so kommt t auch mindestens einmal (und somit unendlich oft) unter den Transitionen von σ vor.

Formal: $\exists i \forall^\infty j \geq i\ M_j[t> \Rightarrow \exists k \geq i\ (t_k = t).$

(3) Ein Netz heißt (stark bzw. schwach) fair, wenn jede seiner zulässigen unendlichen Schaltfolgen bzgl. jeder Transition (stark bzw. schwach) fair ist.

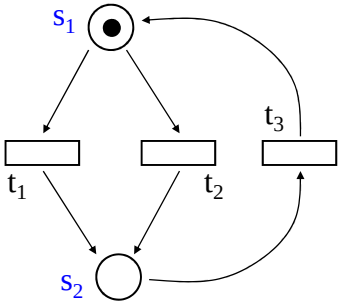
(Bezeichnungen: $\forall^\infty$ = "für unendlich viele"; $\exists^\infty$ = "es gibt unendlich viele".)

Beispiel: Unbeschränktes Lager mit Diebstahl

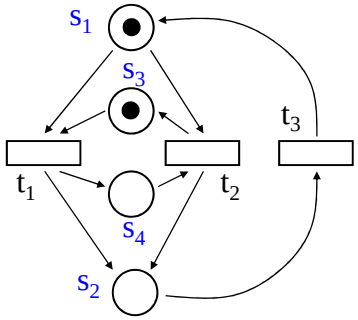


Dieses Netz ist nicht beschränkt; es ist lebendig; es ist weder stark noch schwach fair, wegen $t_1t_2(t_1t_2t_3t_4)^\infty$, denn t_5 kommt in dieser Schaltfolge nicht vor, obwohl t_5 nach dem ersten Schalten von t_2 ständig aktiviert ist.

Beispiel: Schalter einfügen



Linkes Netz (siehe Beginn von 13.1.18): beschränkt und lebendig; schwach fair, weil t_2 nicht ständig aktiviert sein kann; aber nicht stark fair.



Rechtes Netz: wie linkes Netz, aber mit einem Umschalter (= Stellen s_3 und s_4) versehen; das Netz ist beschränkt und lebendig, schwach und stark fair.

13.1.19 Abarbeitung von Jobs im Rechner

Ein Betriebssystem verwaltet die Reihenfolge, in der Aufträge ("Jobs") abgearbeitet werden. In der Regel erhält jeder Job eine Zeiteinheit zum Rechnen (z.B. 100 Millisekunden), dann wird der nächste Job bedient usw. Die Jobs befinden sich in einer zyklischen Liste; auf einen Job zeigt ein Zeiger "aktuell". Die Strategie lautet: Nimm den Job, auf den "aktuell" verweist. Falls dieser aktiviert ist (d.h., die Ressourcen, die er benötigt, stehen tatsächlich zur Verfügung), dann führe ihn eine Zeiteinheit lang durch; anderenfalls bzw. anschließend setze "aktuell" auf das nächste Element der Liste.

Diese Strategie heißt Umlaufverfahren (engl.: *Round-Robin*-Verfahren). Im Folgenden betrachten wir das Ein- und Ausfügen von Jobs der Liste zunächst noch nicht.

Programmstück. Es sei L die zyklische Liste von rechenwilligen Jobs (Transitionen); "aktuell" sei der Zeiger auf ein Element.

```
begin
   L := empty;                          -- starte mit <leere Liste>
   for i in 1..n loop append (L,ti); end loop; -- <Liste mit n Jobs füllen>
   aktuell := ...;                      -- <irgendein Element der Liste L>
   while true loop
      t := aktuell.Job;                 -- zu aktuell gehörender Job
      if  (t ist aktiviert)  then  schalte t  end if;
      aktuell := aktuell.next;   -- "schalte t" = rechne t eine Zeiteinheit lang
   end loop;
end;
```

Es könnte nun Jobs in der Liste geben, die durch Blockieren von Ressourcen ständig dafür sorgen, dass ein spezieller Job t nicht aktiviert ist, wenn er dran ist, aber ansonsten rechnen könnte. t wird dann nie rechnen, obwohl t unendlich oft aktiviert ist. Dieses Umlaufverfahren ist also schwach, aber nicht stark fair.

Bessere Strategie: L sei eine Liste von rechenwilligen Jobs (Transitionen) und aktuell der Zeiger auf ein Element und first der Zeiger auf das erste Element.

```
begin
   L := empty; i := 1; aktuell und first zeigen auf das erste Element von L;
   while true loop
      if (neuer Job kommt in das System) then
         nenne den Job ti; i := i+1; append (L,ti); end if;
      if (L ist nicht leer) then
         aktuell := first; t := aktuell.Job;
         while (t ist nicht aktiviert) and (aktuell /= null) then
            aktuell := aktuell.next; t := aktuell.Job; end loop;
         if (aktuell /= null) then              -- das heißt: t ist aktiviert
         then  schalte t; entferne t aus L;
               if (t noch nicht fertig gerechnet) then append (L,t); end if
         end if;              -- t hat geschaltet und kommt dann ans Ende der Liste
      end if;
   end loop;
end;
```

Nicht aktivierte Jobs rutschen also schrittweise in der Liste nach vorne und kommen schließlich tatsächlich an die Reihe, sofern sie irgendwann einmal aktiviert werden. Dieses Verfahren ist daher stark fair.

13.1.20 Hinweise zur Entscheidbarkeit

Die Erreichbarkeit ist entscheidbar, allerdings i. A. nur mit einem gewaltigen Zeitaufwand.

Das Gleiche gilt für die Beschränktheit. Dies werden wir im Folgenden genauer untersuchen. Letztlich erfolgt der Nachweis der Beschränktheit mit sog. Überdeckungsgraphen. Diese kann man auch für die Untersuchung der Lebendigkeit heran ziehen. Lebendigkeit ist jedoch nicht entscheidbar.

Das Problem, ob ein beliebiges S/T-Netz fair ist, also für alle Transitionen nur faire Schaltfolgen besitzt, ist weder für schwache noch für starke Fairness entscheidbar.

13.2 Invarianten

13.2.1: Wirkung des Schaltens einer Transition t_j.
Zur modifizierten Gewichtsfunktion W' siehe 13.1.6 (4).

$$\Delta t_j = \begin{pmatrix} W'((t_j,s_1)) - W'((s_1,t_j)) \\ W'((t_j,s_2)) - W'((s_2,t_j)) \\ W'((t_j,s_3)) - W'((s_3,t_j)) \\ \dots \\ W'((t_j,s_n)) - W'((s_n,t_j)) \end{pmatrix} \in \mathbf{Z}^n \quad \text{mit} \quad n = |S|$$

Wenn t_j schaltet, wird die Markierung genau um diesen Vektor verändert, d.h.: Aus $M[t_j>M'$ folgt $M' = M + \Delta t_j$.

Wir übertragen diese Aussage nun auf eine Folge von Transitionen. Hierzu sei $T = \{t_1, t_2, ..., t_m\}$.

Wir definieren für $w \in T^*$ den Anzahlvektor #w mit den Komponenten: $\#_j w$ = Anzahl der t_j, die in w vorkommen.

$$\#w = \begin{pmatrix} \#_1 w \\ \#_2 w \\ \#_3 w \\ \dots \\ \#_m w \end{pmatrix}$$

Formal: $\#_j \varepsilon = 0$ und $\#_j vt = \underline{\text{if}}\ t_j = t\ \underline{\text{then}}\ \#_j v + 1\ \underline{\text{else}}\ \#_j v\ \underline{\text{fi}}$ $(\forall v \in T^*, \forall t \in T)$.

Dann gilt für jede Folge w von Transitionen mit M[w>M':

$$M' = M + \#_1 w \cdot \Delta t_1 + \#_2 w \cdot \Delta t_2 + \#_3 w \cdot \Delta t_3 \dots + \#_m w \cdot \Delta t_m .$$

Dies formulieren wir in Matrizenschreibweise.

13.2.2 Definition:
Gegeben sei ein Netz $N = (S, T, F, K, W, M_0)$. Es seien $S = \{s_1, s_2, ..., s_n\}$ und $T = \{t_1, t_2, ..., t_m\}$. Die (n,m)-Matrix

$$C = (\Delta t_1, \Delta t_2, \Delta t_3, ..., \Delta t_m)$$

heißt Inzidenzmatrix des Netzes N.

13.2.3 Folgerung:

Für alle Markierungen M und M' und für alle Schaltfolgen $w \in T^*$ gilt:

Aus $M[w>M'$ folgt: $M' = M + C \cdot \#w$.

13.2.4 Definition:

(1) Jeder (Zeilen-) Vektor $y \in \mathbf{Z}^n$ mit $y \cdot C = 0$ heißt S-Invariante.

(2) y heißt echte S-Invariante $\Leftrightarrow$ y ist eine S-Invariante mit nichtnegativen Komponenten (d.h. $y \geq 0$) und $y \neq 0$.

(3) Jeder (Spalten-) Vektor $x \in \mathbf{Z}^m$ mit $C \cdot x = 0$ heißt T-Invariante.

Beachtet man 13.2.3, so erkennt man:
Eine T-Invariante gibt an, wie oft die einzelnen Transitionen schalten müssen, damit die ursprüngliche Markierung wieder hergestellt wird.
Eine S-Invariante y besagt, dass $y \cdot M = y \cdot M'$ für alle von M aus erreichbaren Markierungen M' gilt.

13.2.5 Satz:

Sei $N = (S, T, F, K, W, M_0)$ ein Netz.

(1) Sei y eine S-Invariante, dann gilt für alle $M' \in ERR(M)$: $y \cdot M = y \cdot M'$.

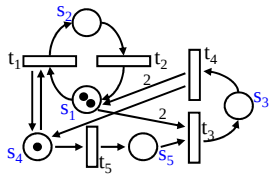
(2) Sei y eine echte S-Invariante, dann gilt für jede Stelle s von N, deren zugehörige Komponente in y positiv ist: s ist k-beschränkt für $k = y \cdot M_0$.

(3) Wenn es eine Markierung M und eine Schaltfolge $w \in T^*$ mit $M[w>M$ gibt, dann ist #w eine T-Invariante von N.

Dieser Satz folgt unmittelbar aus den bisherigen Ausführungen.
Bei (2) beachte man für die Komponente M(s):
$M(s) \leq y(s) \cdot M(s) \leq y_1 \cdot M_1 + y_2 \cdot M_2 + ... + y_n \cdot M_n = y \cdot M = y \cdot M_0 = k$
Aus der linearen Algebra wissen wir, dass die Menge der S- bzw. T-Invarianten einen Untervektorraum bildet.

Anwenden auf das Beispiel 13.1.4:

$$C = \begin{pmatrix} -1 & 1 & -2 & 2 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 0 & 1 \end{pmatrix} \quad M_0 = \begin{pmatrix} 2 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$



Eine echte S-Invariante lautet: y = (1, 1, 3, 1, 1). Nach Satz 13.2.5 (2) sind daher alle Stellen 3-beschränkt und somit ist das Netz beschränkt.
(1,1,0,0,0) und (0,0,1,1,1) sind T-Invarianten, daher verändern die Schaltfolgen t_1t_2 oder t_2t_1 bzw. $t_3t_4t_5$ oder $t_3t_5t_4$ oder $t_4t_3t_5$ oder $t_4t_5t_3$ oder $t_5t_3t_4$ oder $t_5t_4t_3$ (sofern sie schalten können) die Markierung nicht.

Um festzustellen, ob ein Netz beschränkt ist, berechnet man zunächst eine Basis $\{y_1, y_2, ..., y_r\}$ für die S-Invarianten, also linear unabhängige Vektoren $y_i \in \mathbf{Z}^n$ mit $y_i \cdot C = 0$. Dies erfolgt in $O(n^3)$ Schritten mit dem üblichen Eliminationsverfahren für lineare Gleichungssysteme.

Man prüfe, ob es Linearkombinationen dieser y_i gibt, die echte S-Invarianten sind. (Im Zweifel durch Ausprobieren lösen, ansonsten sei auf den Farkas-Algorithmus verwiesen, der später behandelt wird.)

Jede Stelle, deren Komponente in einer echten S-Invariante positiv ist, ist beschränkt.

Über alle anderen Stellen weiß man zunächst nichts. Sie können beschränkt sein, müssen es aber nicht. Der Test mit den echten S-Invarianten ist also nur hinreichend, aber nicht notwendig.

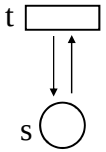
Ein Netz N heißt von echten S-Invarianten überdeckt, wenn es für jede Stelle s eine echte S-Invariante gibt, deren zu s gehörende Komponente positiv ist.

Aus Satz 13.2.5 folgt: Wenn ein Netz von echten S-Invarianten überdeckt wird, so ist es beschränkt. (Die Umkehrung gilt leider nicht.)

Die S-Invarianten bilden einen Unterraum des Raumes $\mathbf{Z}^n$. Insbesondere ist mit zwei echten S-Invarianten auch deren Summe eine echte S-Invariante. Wenn ein Netz von echten S-Invarianten überdeckt wird, so kann man diese S-Invarianten alle addieren, wodurch eine echte S-Invariante mit nur positiven Komponenten entsteht. Folglich wird ein Netz genau dann von echten S-Invarianten überdeckt, wenn es eine S-Invariante gibt, deren sämtliche Komponenten positiv sind.
Um die Beschränktheit eines Netzes nachzuweisen, suche man daher zunächst nach einer solchen positiven S-Invariante.

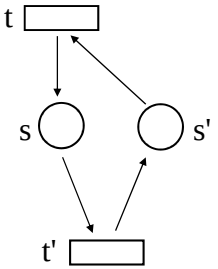
13.2.6 Schlingen

Es sei N = (S, T, F, K, W, M_0) ein Netz. Enthält F zwei Kanten (s,t) und (t,s), so nennt man diese eine "Schlinge".



Solche Schlingen sind in der Inzidenzmatrix nicht "sichtbar", da in (13.2.1) W'((t,s)) - W'((s,t)) = 0 ist.

Schlingen wollen wir daher ausschließen. Dies ist keine wesentliche Annahme, da man sie stets durch Hinzunahme einer weiteren Transition und einer weiteren Stelle durch ein schlingenfreies Netz ersetzen kann, siehe Skizze rechts.



Satz 13.2.7:
Es sei N = (S, T, F, K, W, M_0) ein lebendiges Netz.
Ein Vektor $y \in \mathbf{Z}^n$ ist genau dann eine S-Invariante, wenn für alle von M_0 erreichbaren Markierungen M gilt $y \cdot M = y \cdot M_0$.

Beweis:

"$\Rightarrow$" folgt aus Satz 13.2.5 (1).

"$\Leftarrow$": Da N lebendig ist, gibt es zu jedem $t \in T$ mindestens eine Markierung $M \in ERR(M_0)$, so dass t unter M schalten kann: $M_0\ [u > M\ [t > M + \Delta t$, für ein geeignetes u (weil $M \in ERR(M_0)$). Nach Voraussetzung gilt $y \cdot M_0 = y \cdot M = y \cdot M + y \cdot \Delta t$, denn M und $M+\Delta t$ sind von M_0 aus erreichbar. Es folgt: $y \cdot \Delta t = 0$. Dies gilt für jedes t, also ist $y \cdot C = 0$, da C aus allen Vektoren Δt zusammengesetzt ist. Folglich ist y eine S-Invariante.

13.3 Minimale Invarianten

13.3.1: Definition

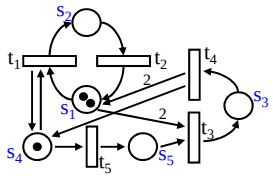
(1) Es sei $z \in \mathbf{Z}^n$ ein n-Tupel ganzer Zahlen. Die Menge

supp(z) = { i | $z_i > 0$ } heißt Träger von z ("support").

(2) Eine S-Invariante $y \in \mathbf{Z}^n$ heißt minimal $\Leftrightarrow$ y ist eine echte S-Invariante und es gibt keine echte S-Invariante y' mit supp(y') $\subset$ supp(y) und supp(y') $\neq$ supp(y).

Beachten Sie: Minimale S-Invarianten sind nach Definition immer auch echte S-Invarianten.

Betrachte erneut Beispiel 13.1.4:



$$C = \begin{pmatrix} -1 & 1 & -2 & 2 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

Minimale S-Invarianten sind
(1,1,2,0,0) mit dem Träger {1, 2, 3} und
(0,0,1,1,1) mit dem Träger {3, 4, 5}.

(1, 1, 3, 1, 1) ist zwar eine echte S-Invariante, aber ihr Träger {1, 2, 3, 4, 5} umfasst die Träger der obigen beiden echten S-Invarianten; sie ist also nicht minimal.

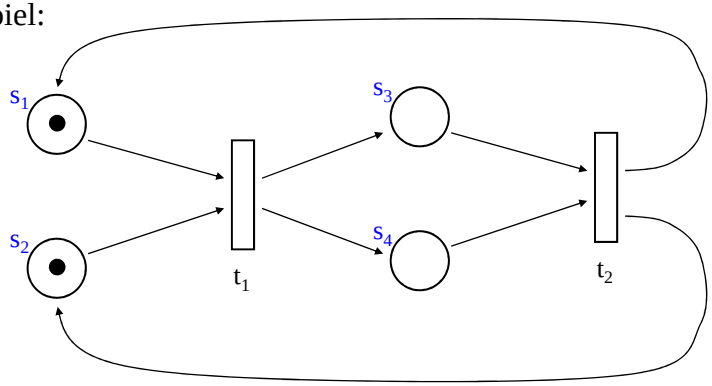
13.3.2 Anzahl kleinster minimaler S-Invarianten

Wie viele verschiedene minimale S-Invarianten kann ein Netz haben? Hierbei wollen wir nur "kleinste" Vektoren zählen.

Genauer: Wenn $(y_1,y_2,...,y_n)$ eine minimale S-Invariante ist, dann ist $k\cdot(y_1,y_2,...,y_n) = (k\cdot y_1,k\cdot y_2,...,k\cdot y_n)$ für jede natürliche Zahl k ebenfalls eine minimale S-Invariante. Uns interessieren daher nur die minimalen S-Invarianten $(y_1,y_2,...,y_n)$, für die $ggT(y_1,y_2,...,y_n) = 1$ ist.

Aber bereits deren Anzahl kann exponentiell mit der Zahl der Stellen wachsen, wie folgendes Beispiel zeigt.

Beispiel:

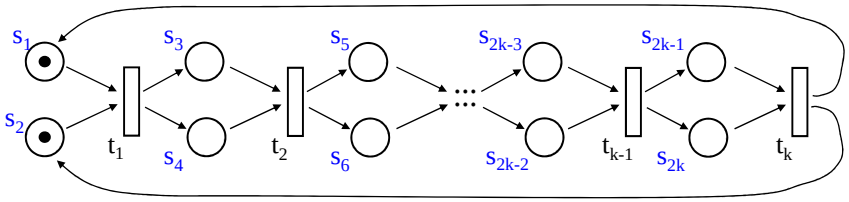


$$C = \begin{pmatrix} -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \end{pmatrix}$$

Minimale S-Invarianten sind:

$y^{(1)} = (1,0,1,0)$, $y^{(2)} = (1,0,0,1)$,
$y^{(3)} = (0,1,1,0)$, $y^{(4)} = (0,1,0,1)$.

Wir verallgemeinern dieses Beispiel (n = |S|, k = n/2):



$$C = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & & \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 1 & -1 \end{pmatrix}$$

Es gibt 2^k minimale S-Invarianten:

y = (. , . , . , . , . , .) ... (. , . , . , . , . , .)

genau an einer der beiden Stellen steht eine 1 | genau an einer der beiden Stellen steht eine 1 | genau an einer der beiden Stellen steht eine 1 | genau an einer der beiden Stellen steht eine 1

13.3.3 Farkas-Algorithmus (1902)

(Julius Farkas, 1847-1930, ungarischer Mathematiker)

Mit diesem Algorithmus kann man alle minimalen S-Invarianten zu einer Inzidenzmatrix C berechnen. Es seien n = |S| die Anzahl der Stellen und m = |T| die Anzahl der Transitionen des Netzes.

Idee: Beginne mit der Matrix $D_0 = (C\ E_n)$, d. h., man schreibe die Matrix C hin und ergänze sie nach rechts um die n-dimensionale Einheitsmatrix E_n. C ist eine (n,m)- und E_n eine (n,n)-Matrix. Wenn $D_{i-1} = (L_{i-1}\ R_{i-1})$ für i > 0 die aktuelle Matrix ist, wobei L_{i-1} eine (k_{i-1},m)- und R_{i-1} eine (k_{i-1},n)-Matrix ist (für Zahlen k_{i-1}, die sich im Laufe der Konstruktion ergeben), dann konstruiere hieraus die Matrix $D_i = (L_i\ R_i)$ mit der (k_i,m)-Matrix L_i und der (k_i,n)-Matrix R_i. Die ersten i Spalten von L_i enthalten nur Nullen. Für die Matrix $D_m = (L_m\ R_m)$ gilt dann: L_m ist eine Null-Matrix und R_m enthält alle minimalen S-Invarianten von C. (In der Regel enthält R_m aber noch weitere echte S-Invarianten von C.)

Wie konstruiert man D_i aus D_{i-1}?

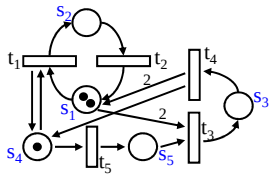
Gegeben ist die Matrix D_{i-1}. Für je zwei Zeilen d_1 und d_2 in D_{i-1}, deren i-te Komponenten $d_{1,i}$ und $d_{2,i}$ nicht Null sind und die verschiedenes Vorzeichen haben (d.h., für die $d_{1,i} \cdot d_{2,i} < 0$ gilt), bilde den (n+m)-dimensionalen Vektor $d = |d_{2,i}| \cdot d_1 + |d_{1,i}| \cdot d_2$; wegen der verschiedenen Vorzeichen ist die i-te Komponente von d Null. Teile dann d durch den größten gemeinsamen Teiler aller seiner Komponenten und füge den so erhaltenen Vektor unten an die Matrix D_{i-1} an.

Hat man dies für alle möglichen Zeilen d_1 und d_2 getan, so streiche man alle Zeilen aus D_{i-1}, deren i-te Komponente nicht Null ist. Die so erhaltene Matrix ist D_i.

Betrachte wiederum das Beispiel aus 13.3.1:

$$C = \begin{pmatrix} -1 & 1 & -2 & 2 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

n = 5, m= 5



$$D_0 = \begin{pmatrix} -1 & 1 & -2 & 2 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Hier kann man nur die erste und die zweite Zeile kombinieren. Diese werden anschließend gestrichen. Ergebnis ist:

$$D_1 = \begin{pmatrix} 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -2 & 2 & 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Da die zweite Spalte nur Nullen enthält, muss im nächsten Schritt nichts getan werden, d.h., es ist $D_1 = D_2$.

$D_2 = D_1$

Dritte Spalte: Nun können die Zeilen 1 und 3 und die Zeilen 1 und 4 kombiniert werden; es entstehen zwei neue Zeilen; danach werden die bisherigen Zeilen 1, 3 und 4 gestrichen:

$$D_3 = \begin{pmatrix} 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 2 & 0 & 0 \end{pmatrix}$$

Vierte Spalte: Es können nur die Zeilen 1 und 2 kombiniert werden:

$$D_4 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} = D_5$$

Die fünfte Spalte enthält nur Nullen, also ist $D_5 = D_4$. Wegen m = 5 endet der Algorithmus jetzt und wir erhalten

$$R_5 = \begin{pmatrix} 1 & 1 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Die Zeilen von R_5 enthalten alle minimalen S-Invarianten des gegebenen Netzes.

Beachten Sie: Ausgehend von $D_0 = (C\ E_n)$ berechnet der Algorithmus die Matrizen $D_1, D_2, ..., D_m$, wobei die neuen Zeilenvektoren wegen $|d_{1,i}| > 0$ und $|d_{2,i}| > 0$ positive Linearkombinationen bereits vorhandener Zeilen sind. Die rechten n Spalten (also die Teilmatrix R_{i-1}) sind von Anfang an nicht-negativ und bleiben bei positiven Linearkombinationen auch nicht-negativ. Folglich enthält R_i nur nicht-negative Zahlen und in jeder Zeile mindestens eine von 0 verschiedene Zahl (wegen $R_0 = E_n$).

Später beweisen wir (Hilfssatz 13.3.5):
Für jedes $D_i = (L_i\ R_i)$ gilt die Gleichung: $R_i \cdot C = L_i$.

Nach Konstruktion ist L_m die Nullmatrix. Für jede Zeile y von R_m gilt also $y \cdot C = 0$. Daher ist jede Zeile von R_m eine echte S-Invariante des Netzes. Folglich müssen wir "nur noch" die obige Gleichung $R_i \cdot C = L_i$ beweisen und zeigen, dass alle minimalen S-Invarianten als Zeilen in R_m vorkommen.

Satz 13.3.4:

Es sei $N = (S, T, F, K, W, M_0)$ ein Netz. Dann gilt:

(i) Jede echte S-Invariante von N ist eine positive Linearkombination von minimalen S-Invarianten.
Das heißt: Zu jeder echten S-Invariante y von N gibt es minimale S-Invarianten $z_1, z_2, ..., z_r$ von N und positive ganze Zahlen $a_1, a_2, ..., a_r$ mit $y = a_1 \cdot z_1 + a_2 \cdot z_2 + ... + a_r \cdot z_r$.

(ii) Minimale S-Invarianten mit gleichem Träger sind linear abhängig. Speziell gilt: Zu allen minimalen S-Invarianten $u_1, u_2, ..., u_r$ mit gleichem Träger $supp(u_1) = ... = supp(u_r)$ gibt es eine eindeutig bestimmte minimale S-Invariante u mit $supp(u) = supp(u_1) = ... = supp(u_r)$, so dass jedes u_i Vielfaches von u ist.

Beweis zu (i):

Sei y eine echte S-Invariante von N. Wenn y minimal ist, so ist nichts zu beweisen. Sei also y nicht minimal. Dann muss es nach Definition 13.3.1 (2) eine echte S-Invariante z mit $supp(z) \subset supp(y)$ und $supp(z) \neq supp(y)$ geben.

Einschub: Beispiel. Seien y = (7,1,0,3,3) und z = (4,0,0,2,0), dann kombiniere man z und y so, dass eine echte S-Invariante mit kleinerem Träger entsteht. Die ersten Komponenten lauten 7 und 4, man könnte also $4 \cdot y - 7 \cdot z = (0,4,0,-2,12)$ bilden; dies ist aber ungeeignet, da eine negative Komponente entsteht. Wir betrachten daher die vierten Komponenten 3 und 2 und bilden $2 \cdot y - 3 \cdot z = (2,2,0,0,6) = a$. Wählt man die Komponenten y(s) und z(s) so, dass der Quotient q = y(s)/z(s) minimal ist, dann erhält man eine echte S-Invariante a mit kleinerem Träger als y. Dies tun wir nun.

Zu y und z wähle $s \in supp(z)$ so, dass

$$\frac{y(s)}{z(s)} = \text{Min}\ \{\ \frac{y(s')}{z(s')}\ |\ s' \in supp(z)\ \} =: q$$

Da $s \in supp(z)$ und somit auch $s \in supp(y)$ ist, gelten $y(s) \neq 0$ und $z(s) \neq 0$ und daher $q > 0$. Bilde nun den Vektor

$$a = z(s) \cdot y - y(s) \cdot z.$$

Für a gilt an der Stelle s:

$a(s) = z(s) \cdot y(s) - y(s) \cdot z(s) = 0$ und hieraus folgt:

$supp(a) \subset supp(y) - \{s\}$, also eine echte Teilmenge von supp(y).

Weiterhin ist a eine echte S-Invariante; denn als Linearkombination der beiden S-Invarianten y und z ist a ebenfalls eine S-Invariante und alle Komponenten von a sind nicht negativ.
[Weil: Gäbe es ein s' mit $a(s') = z(s) \cdot y(s') - y(s) \cdot z(s') < 0$, so folgt $y(s')/z(s') < y(s)/z(s)$ im Widerspruch zur Wahl von s.]

Aus $a = z(s) \cdot y - y(s) \cdot z$ folgt (mit $q = y(s)/z(s)$):

$$y = \frac{1}{z(s)} \cdot a - q \cdot z$$

Damit ist der Beweis zu (i) auch schon beendet; denn y lässt sich als Linearkombination zweier echter S-Invarianten mit echt kleineren Trägern darstellen, wobei z bereits minimal ist. Falls a nicht minimal ist, so wende man das gleiche Argument auf a an: Auch a lässt sich als Linearkombination zweier echter S-Invarianten mit echt kleineren Trägern darstellen usw. Da der Träger mindestens eine Komponente enthält, muss dieser Prozess nach spätestens |S|-1 Schritten mit minimalen S-Invarianten enden und dann liegt (durch rückwärts einsetzen) eine Linearkombination von y durch minimale S-Invarianten vor.

Beweis zu (ii):

Betrachte zwei minimale S-Invarianten y und y' mit gleichem Träger supp(y) = supp(y').

Konstruiere nun zu y mit z = y' wie im Beweis zu (i) den Vektor $a = z(s) \cdot y - y(s) \cdot z$. Da y minimal ist und a mindestens eine Komponente, die ungleich Null ist, weniger als y hat, muss der Träger von a leer sein, d.h., a muss der Nullvektor sein. Daraus folgt $y = q \cdot z$, und somit ist y ein Vielfaches von z. Weil y und z ganzzahlig sind, muss z(s) ein Vielfaches des ggT der Komponenten von z sein.

Bilde daher den größten gemeinsamen Teiler aller Komponenten von y: $g = ggT\{y(s) \mid s \in S\}$.

Der Vektor $u = \frac{1}{g} \cdot y$ ist dann die gesuchte minimale S-Invariante zum Träger supp(y).

Hilfssatz 13.3.5:

Es sei $D_0, D_1, D_2, ..., D_m$ die Folge der Matrizen, die der Farkas-Algorithmus konstruiert. Jedes D_i besitzt genau m+n Spalten mit m = |T| und n = |S|. Mit L_i bezeichnen wir die Matrix der ersten m Spalten von D_i und mit R_i die Matrix der restlichen n Spalten.

Für jedes $D_i = (L_i\ R_i)$ gilt: $R_i \cdot C = L_i \quad (i = 0, 1, 2, ..., m)$.

Beweis durch Induktion.

Verankerung: Für i = 0 ist der Hilfssatz richtig, denn aus $D_0 = (C\ E_n)$ folgt $L_0 = C$, $R_0 = E_n$ und somit $R_0 \cdot C = L_0$.

Induktionsannahme: Wir nehmen nun an, der Hilfssatz ist bis i-1 bewiesen. Wir untersuchen nun die Matrix $D_i = (L_i\ R_i)$.

Betrachte die j-te Zeile d_j aus $D_i = (L_i\ R_i)$. d_j besteht aus zwei Teilvektoren $d_j = (f_j, r_j)$, wobei f_j die j-te Zeile von L_i und r_j die j-te Zeile von R_i ist. (Anmerkung: Wir verwenden hier f statt eines kleinen "L", da sich "l" drucktechnisch nicht von der Ziffer "1" unterscheidet.) Die Gleichung $R_i \cdot C = L_i$ ist gleichbedeutend damit, dass $r_j \cdot C = f_j$ für alle j gilt.

Fall 1: Die Zeile d_j ist eine Zeile, die bereits in der Matrix D_{i-1} vorkommt. Dann gilt $r_j \cdot C = f_j$ nach Induktionsannahme.

Fall 2: Die Zeile d_j ist eine positive Linearkombination der zwei Zeilen $d = (f, r)$ und $d' = (f', r')$ aus der Matrix D_{i-1}, also $d_j = b_1 \cdot d + b_2 \cdot d'$ mit ganzen Zahlen b_1 und b_2. Dies gilt auch für die Teilvektoren: $f_j = b_1 \cdot f + b_2 \cdot f'$ und $r_j = b_1 \cdot r + b_2 \cdot r'$. Nach Induktionsannahme ist: $r \cdot C = f$ und $r' \cdot C = f'$, woraus folgt: $r_j \cdot C = b_1 \cdot r \cdot C + b_2 \cdot r' \cdot C = b_1 \cdot f + b_2 \cdot f' = f_j$.

In beiden Fällen gilt also $R_i \cdot C = L_i$, womit Hilfssatz 13.3.5 durch Induktion bewiesen ist.

Satz 13.3.6:

Das Netz möge keine Schlingen enthalten (vgl. 13.2.6).
Der Farkas-Algorithmus liefert nur echte S-Invarianten, darunter alle minimalen.

Beweis: Zunächst führen wir Bezeichnungen ein.

$N = (S, T, F, K, W, M_0)$ sei das gegebene Netz mit $K(s) = \infty$ für alle $s \in S$, $n = |S| > 0$, $m = |T| > 0$. S und T seien fest durchnummeriert: $S = \{s_1, s_2, ..., s_n\}$ und $T = \{t_1, t_2, ..., t_m\}$. $C \in \mathbf{Z}^{n \times m}$ sei die Inzidenzmatrix von N.

Setze für $j = 0, 1, ..., m$: $T_j = \{t_1, t_2, ..., t_j\}$ mit $T_0 = \emptyset$. Sei $F_j = \{(v, w) \in F \mid v, w \in S \cup T_j\}$. Bilde hiermit die Netze $N_j = (S, T_j, F_j, K, W_j, M_0)$, wobei W_j die Abbildung W eingeschränkt auf F_j ist. N_j ist das Unter-Netz von N mit den Transitionen t_1 bis t_j. Speziell ist N_0 das Netz, welches zwar alle Stellen, aber keine Transitionen besitzt, und es gilt $N_m = N$.

Behauptung: Für jedes $j = 0, 1, ..., m$ gilt: Die Matrix R_j enthält nur echte S-Invarianten von N_j, darunter alle minimalen.

Wir beweisen dies durch Induktion über j.

j = 0: N_0 enthält keine Transition. Es ist C der Nullvektor und für jeden Vektor $y \in \{(1,0,0,...,0), (0,1,0,...,0), ..., (0,0,...,0,1)\}$ gilt $y \cdot C = 0$. Folglich sind alle Einheitsvektoren genau die minimalen S-Invarianten von N_0. Wegen $R_0 = E_n$ trifft die Behauptung für j = 0 daher zu.

$j \Rightarrow j+1$: Für N_0 bis N_j sei die Behauptung bereits bewiesen. Betrachte nun N_{j+1}.
(1) Zunächst gilt, dass in R_{j+1} nur nicht-negative Zahlen stehen. [Begründung: Die Zeilen von R_{j+1} werden entweder von R_j übernommen oder entstehen als positive Linearkombinationen der Zeilen von R_j. In $R_0 = E_n$ stehen nur nicht-negative Zahlen, daher sind alle Zahlen in $R_1, R_2, ...$ ebenfalls nicht-negativ.]

(2) Jede echte S-Invariante von N_{j+1} (und somit auch jede minimale S-Invariante von N_{j+1}) ist eine positive Linearkombination der minimalen S-Invarianten von N_j.
[Denn es gilt: y ist echte S-Invariante von N_{j+1}
$\Rightarrow$ $y \cdot (\Delta t_1, \Delta t_2, ..., \Delta t_{j+1}) = 0$
$\Rightarrow$ $y \cdot (\Delta t_1, \Delta t_2, ..., \Delta t_j) = 0$
$\Rightarrow$ y ist echte S-Invariante von N_j
$\Rightarrow$ y ist nach Satz 13.3.4 (i) positive Linearkombination der minimalen S-Invarianten von N_j.]

(3) Weiterhin gilt: Jede minimale S-Invariante von N_{j+1} ist eine positive Linearkombination von *höchstens zwei* Zeilen aus R_j.
[Begründung: Betrachte eine minimale S-Invariante y von N_{j+1}. y ist insbesondere eine echte S-Invariante von N_{j+1}. Nach (2) muss sich y als positive Linearkombination der Zeilen von R_j darstellen lassen; denn nach Induktionsannahme kommen alle minimalen S-Invarianten von N_j in R_j vor.

Wir nehmen nun an, y sei die positive Linearkombination von drei oder mehr Zeilen aus R_j: $y = b_1 \cdot r_1 + b_2 \cdot r_2 + b_3 \cdot r_3 + ... + b_k \cdot r_k$ mit $k \geq 3$ und alle $b_i > 0$. Betrachte dann $y' = b_1 \cdot r_1 + b_2 \cdot r_2$.

Es gilt $supp(y') \subseteq supp(y)$, weil durch Addition von $b_3 \cdot r_3 + ... + b_k \cdot r_k$ (dies sind positive Vektoren!) der Träger höchstens größer werden kann. Da y minimal ist, muss aber $supp(y') = supp(y)$ gelten. Nach Satz 13.3.4 (ii) ist dann y ein Vielfaches von y', d.h., man erhält y (bis auf einen konstanten Faktor) bereits durch Linearkombination von höchstens zwei Zeilen von R_j.]

(4) Es sei nun y eine minimale S-Invariante von N_{j+1}. Insbesondere muss $y \cdot \Delta t_{j+1} = 0$ gelten. Wegen (3) ist y eine positive Linearkombination von höchstens zwei Zeilen aus R_j.
(4.1) Wenn y bereits in R_j vorkommt, dann gibt es eine Zeile (f,y) in D_j und in D_{j+1}. Wegen $y \cdot \Delta t_{j+1} = 0$ folgt aus Hilfssatz 13.3.5, dass die (j+1)-te Komponente von f gleich 0 sein muss.

(4.2) Sei y eine positive Linearkombination aus zwei Zeilen von R_j, d.h.: $y = b_1 \cdot r_1 + b_2 \cdot r_2$ mit $b_1 > 0$ und $b_2 > 0$.
Es sei (f,y) die zugehörige Zeile in D_{j+1} (mit f in L_{j+1}).
Es seien f_1 und f_2 die zu r_1 und r_2 gehörenden Zeilen aus L_j.
Dann folgt aus der Konstruktionsvorschrift des Farkas-Algorithmus: $f = b_1 \cdot f_1 + b_2 \cdot f_2$. Also gilt
$y \cdot \Delta t_{j+1} = (b_1 \cdot r_1 + b_2 \cdot r_2) \cdot \Delta t_{j+1} = b_1 \cdot r_1 \cdot \Delta t_{j+1} + b_2 \cdot r_2 \cdot \Delta t_{j+1} = 0$
= die (j+1)-te Komponente des Vektors f in L_{j+1}.
(Letzteres folgt aus Hilfssatz 13.3.5.)

Folglich muss die (j+1)-te Komponente von $f = b_1 \cdot f_1 + b_2 \cdot f_2$ Null sein. Wegen $b_1 > 0$ und $b_2 > 0$ müssen f_1 und f_2 an der (j+1)-ten Komponente verschiedenes Vorzeichen haben, und diese Bedingung reicht bereits aus, um eine positive Linearkombination mit der Eigenschaft $y \cdot \Delta t_{j+1} = 0$ zu erzeugen.

(4.3) Wir haben also gezeigt: Jede minimale S-Invariante y von N_{j+1} muss in R_{j+1} vorkommen, wenn
entweder y bereits in R_j auftritt und der zu y gehörige Vektor f in L_j an der Stelle j+1 eine Null besitzt
oder wenn y positive Linearkombination zweier Vektoren r_1 und r_2 aus R_j ist, wobei die zu r_1 und r_2 gehörenden Vektoren f_1 und f_2 in L_j an der (j+1)-ten Stelle verschiedenes Vorzeichen besitzen.

Dies ist aber genau die Konstruktionsvorschrift des Farkas-Algorithmus, das heißt: Jede minimale S-Invariante von N_{j+1} tritt in R_{j+1} auf (es können aber noch weitere echte S-Invarianten in R_{j+1} stehen). Folglich gilt die Behauptung auch für j+1, womit die Behauptung gezeigt ist.

Für j = m folgt aus der Behauptung der Satz 13.3.6.

13.3.7 Größe der Matrizen D_i

Aus 13.3.2 wissen wir bereits, dass die Matrix D_m $2^{n/2}$ Zeilen besitzen kann. Die Anzahl der minimalen S-Invarianten ist wegen Satz 13.3.4 (ii) aber durch die Anzahl der Trägermengen, d.h. durch die Anzahl der Elemente der Potenzmenge von S, also durch 2^n beschränkt.

Dies ist aber noch keine obere Schranke für die Zahl der Zeilen in der Matrix D_m, da im Allgemeinen auch nicht-minimale echte S-Invarianten bei der Konstruktion entstehen. Der theoretisch denkbar schlimmste Fall tritt ein, wenn eine Matrix D_i mit k_i Zeilen in der Spalte i+1 genau $k_i/2$ positive und $k_i/2$ negative Zahlen besitzt; denn dann kann die Matrix D_{i+1} bis zu $(k_i/2)^2$ Zeilen haben. Aus $k_0 = n$ Zeilen können also Matrizen mit $(n/2)^2 = n^2/2^2$, $((n^2/4)/2)^2 = n^4/2^6$, $(n^4/64)/2)^2 = n^8/2^{14}$ und allgemein mit $4 \cdot (n/4)^{2^i}$ Zeilen entstehen.

Ein dramatisches Wachstum kann tatsächlich auftreten. Deshalb muss man bei der Anwendung des Farkas-Algorithmus die Zeilenzahl von D_i ständig überwachen.

Die Zeilenzahl kann in Grenzen gehalten werden, falls man beim Hinzufügen neuer Zeilen stets prüft, ob es in D_i bereits eine Zeile mit kleinerem oder gleichem Träger gibt. Durch Streichen solcher Zeilen erreicht man, dass D_i immer nur minimale S-Invarianten bzgl. des Netzes N_i enthält.

In der Praxis sind die Netze meist recht groß, so dass der Speicherplatz zum Problem wird. Schon kleine Beispiele zeigen die Notwendigkeit, in jedem Schritt die D-Matrizen zu verkleinern.

Beispiele: n = 5, m = 4: Mit der Hand kann man noch die links stehende Matrix mit dem Farkas-Algorithmus durchrechnen. Dagegen benötigt man für die in der Mitte stehende Matrix einen Rechner, weil die Zahlen in den 91 entstehenden Zeilen zu groß (genauer: über 2 Millionen) werden. Man erhält hier eine minimale S-Invariante: (32, 24, 91, 92, 21).

$$\begin{pmatrix} 1 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & 0 & 1 & -1 \\ -1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} -2 & -3 & 0 & 0 \\ 1 & 1 & -3 & 2 \\ 2 & -1 & -1 & -2 \\ -2 & 2 & 2 & 1 \\ 2 & -1 & -1 & 2 \end{pmatrix}$$

Aus der "kleinen" rechts stehenden Matrix mit n = 7 und m = 5 erzeugt der Farkas-Algorithmus *ohne* Reduktionen eine Matrix mit **4502138** (!) Zeilen. Verkleinert man die Matrix dagegen in jedem Schritt, so kommt man mit 12 Zeilen aus. Ergebnis sind zwei minimale S-Invarianten. (Prüfen Sie es einfach einmal nach.)

$$\begin{pmatrix} -1 & 1 & 1 & -1 & 0 \\ 1 & 0 & -1 & -1 & 0 \\ -1 & -1 & 0 & 1 & 1 \\ -1 & 1 & -1 & 1 & 0 \\ 1 & 0 & -1 & 1 & -1 \\ 1 & 0 & 1 & -1 & 0 \\ -1 & -1 & 1 & 1 & 0 \end{pmatrix}$$

13.4 Nachrichtenaustausch

Wie können verschiedene Prozesse Daten austauschen? Bei den S/T-Netzen müssen sich die Daten im Vorbereich der Transition befinden; sie werden "lokal" über den Nachbereich anderen Prozessen zur Verfügung gestellt.

Wir betrachten nun zwei andere Mechanismen:
- gemeinsamer Speicherbereich (shared variables),
- Aufbau von Kanälen.

Der Datenaustausch kann synchron und asynchron erfolgen.

Hierbei gehen wir sofort von einer konkreten Programmiersprache aus. Diese ist eine Erweiterung der Sprachelemente, die zu Beginn dieser Vorlesung eingeführt wurden. Wir beginnen mit dem gemeinsamen Speicherbereich.

13.4.1 Elementare Anweisungen (diese übernehmen wir aus Abschnitt 2.1.5 und fügen await hinzu):

skip	Nichtstun.
X := α	Wertzuweisung. α ist ein Ausdruck. (Rechne den Ausdruck α aus und lege den erhaltenen Wert in der Variablen X ab.)
read (X)	Leseanweisung. (Lies den nächsten Wert ein und lege ihn in der Variablen X ab.)
write (α)	Schreibanweisung. (Drucke den Wert, den der Ausdruck α besitzt, aus.)
await β	Warten. *Bedeutung:* Warte, bis β wahr ist. (β ist ein Boolescher Ausdruck.)
$F(X_1, ..., X_n)$	Aufruf. (Führe den Algorithmus F mit den Werten der Variablen $X_1, ..., X_n$ aus.)

13.4.2 Zusammengesetzte Anweisungen (siehe wiederum Abschnitt 2.1.5):

Hintereinanderausführung oder Sequenz: Wenn γ_1 und γ_2 Anweisungen sind, dann ist auch $\gamma_1;\gamma_2$ eine Anweisung.

Alternative: Wenn γ_1 und γ_2 Anweisungen und β ein Boolescher Ausdruck sind, dann ist auch

if β then γ_1 else γ_2 fi .

eine Anweisung (else skip darf man weglassen.)

Schleife: Wenn γ eine Anweisung und β ein Boolescher Ausdruck sind, dann ist auch while β do γ od eine Anweisung.

Hinzu kommen folgende zusammengesetzte Anweisungen (d. h., wenn γ, γ_1, γ_2, .., γ_n Anweisungen sind, dann sind auch die folgenden Konstrukte ... Anweisungen):

Nichtdeterministische Auswahl für n≥2:

(γ_1 or γ_2 or γ_3 or ... or γ_n)

Nebenläufige Abarbeitung für n≥2:

(γ_1 | γ_2 | γ_3 | ... | γ_n)

(Die nebenläufigen Anweisungen γ_i nennt man auch "Prozesse").

Blöcke mit gemeinsamen Variablen:

[local <lokale Deklarationen>; γ]

Wie in Ada lassen wir in den lokalen Deklarationen Konstanten und Initialisierungen zu. Weiterhin darf man jeder Anweisung eine Marke mittels <Name>:: voranstellen.

Hinweis: Unsere Beispiele haben meist die Form:

P:: [local <lokale Deklarationen>; (γ_1 | γ_2 | γ_3 | ... | γ_n)]

Beispiel 13.4.3:

```
[ local  L: Integer := 0;
         max: constant Integer := 2;
  ( P1:: while true do
         await L < max;
         (L := L+1  or  skip)
         od;
  |
    P2:: while true do
         await L > 0;
         (L := L-1 or skip)
         od;
  )
]
```

Dieses Programm beschreibt den Erzeuger-Verbraucher-Kreislauf, siehe 13.1.3, mit der Anzahl L der Elemente im Lager, die zwischen 0 und max (=Kapazität) liegen darf.

Dieses Programms besitzt zwei Prozesse. Was ist seine Bedeutung? Ist es genau die gleiche wie die des S/T-Netzes?

Wir führen den Begriff des Zustands für Programme mit mehreren Prozessen ein. Ein Zustand gibt zu jedem Zeitpunkt an, welche Werte die Variablen besitzen und an welchen Stellen im Programm sich die Prozesse befinden.

Hierfür müssen wir die "Stelle im Programm" definieren. Grob gesprochen ist dies eine Stelle zwischen zwei Aktionen; wie in Kapitel 7 sind dies elementare Anweisungen oder Bedingungen. Wir nummerieren diese Stellen einfach durch, z.B.:

```
( P1::(1) while (2) true do
    (3) await  L < max;
        ((4) L := L+1  or  (5) skip)
    (6) od;
|
  P2:: (1) while (2) true do
    (3) await L > 0;
        ((4) L := L-1 or (5) skip)
    (6) od;
)
```


Hier besitzt man eine gewisse Willkür. Man kann beispielsweise auch die Stelle vor der inneren nebenläufigen Anweisung markieren, also

(4)((5)L := L+1 or (5')skip)

Man kann auch die Berechnungen der Ausdrücke feiner unterteilen, also

(4)((5) L (7) := (6)L+1 or (5')skip)

Dies hängt davon ab, ob die Programme in Zeittakten bearbeitet werden und ob es Berechnungsteile gibt, die von außen nicht unterbrochen werden können.

Wir formulieren nun die Menge der möglichen Zustände zu der Unterteilung, die auf der vorigen Folie angegeben wurde.

P1:: (1) while (2) true do
(3) await L < max;
((4) L := L+1 or (5) skip)
(6) od;

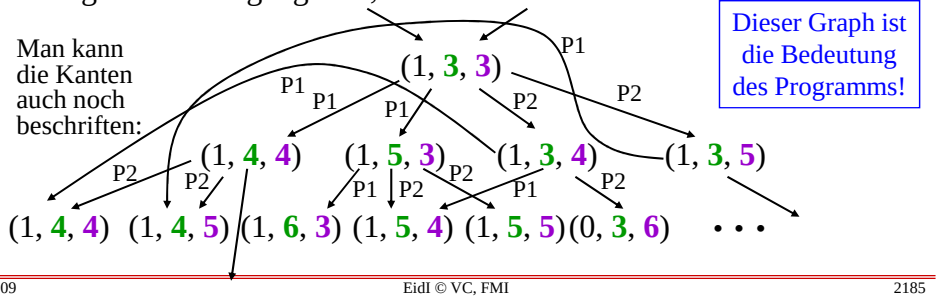
P2:: (1) while (2) true do
(3) await L > 0;
((4) L := L-1 or (5) skip)
(6) od;

Zustandsmenge

Z = { (a, **i**, **j**) | a ist der Wert von L, **i** ist die Stelle im Programm P1 und **j** ist die Stelle im Programm P2 }

Nun tragen wir wie beim Erreichbarkeitsgraphen 13.1.9 die möglichen Übergänge ein, z.B.:

Man kann die Kanten auch noch beschriften:

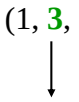


Dieser Graph ist die Bedeutung des Programms!

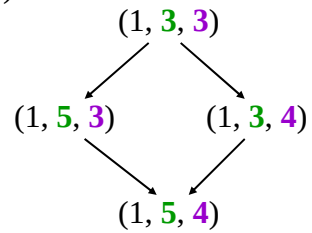
P1:: (1) while (2) true do
(3) await L < max;
((4) L := L+1 or (5) skip)
(6) od;

P2:: (1) while (2) true do
(3) await L > 0;
((4) L := L-1 or (5) skip)
(6) od;

In der Regel betrachtet man hierbei keine "Gleichzeitigkeit". Zum Beispiel ist der Übergang



hier nicht zulässig, man muss vielmehr zwei Schritte in irgendeiner Reihenfolge durchführen:



13.4.4: Legt man die Bedeutung (Semantik) nebenläufiger Programme so fest, dass zwei Prozesse niemals gleichzeitig einen Schritt (= Wechsel des Zustands) ausführen können, sondern stets eine Reihenfolge erzwungen wird, so spricht von einer "Interleaving"-Semantik.

Die Interleaving-Semantik lässt sich aus theoretischer Sicht leichter behandeln als eine Semantik, in der auch Gleichzeitigkeit zugelassen ist. Weiterhin beschreibt die Interleaving-Semantik genau die Verhältnisse, die bei einem *Monoprozessor* vorliegen, der die verschiedenen nebenläufigen Programme alleine ausführen muss (der also die Nebenläufigkeit nur vortäuscht und in Wahrheit die Prozesse verzahnt sequenziell ausführt).

13.4.5: Zur "Granularität" (feinkörnig / grobkörnig): Um die Zustände exakt definieren zu können, muss man festlegen, welche Anweisungsteile "nicht unterbrechbar" sind. Solche Teile werden als in sich geschlossene Einheiten angesehen, deren Ausführung von anderen Ereignissen nicht gestört wird.

Sind in unserem obigen Beispiel "L:=L+1" und "L:=L-1" nicht unterbrechbar, so hat nach der nebenläufigen Abarbeitung von (Q1:: L:=L+1 | Q2:: L:=L-1) die Variable L ihren Wert nicht verändert. Sind aber nur die arithmetischen Operationen und die Wertzuweisungen jede für sich nicht unterbrechbar, so kann folgende Möglichkeit bei dieser Abarbeitung auftreten:

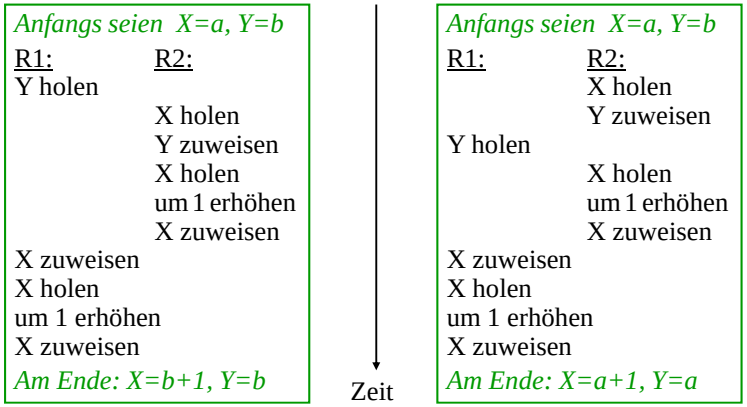
Hole den Wert a von L bzgl. Q1; hole den Wert a von L bzgl. Q2; bilde a+1 bzgl. Q1; bilde a-1 bzgl. Q2; weise a+1 der Variablen L zu bzgl. Q1; weise a-1 der Variablen L zu bzgl. Q2.

Am Ende hat L also den Wert a-1 (anstelle des erwarteten Wertes a).

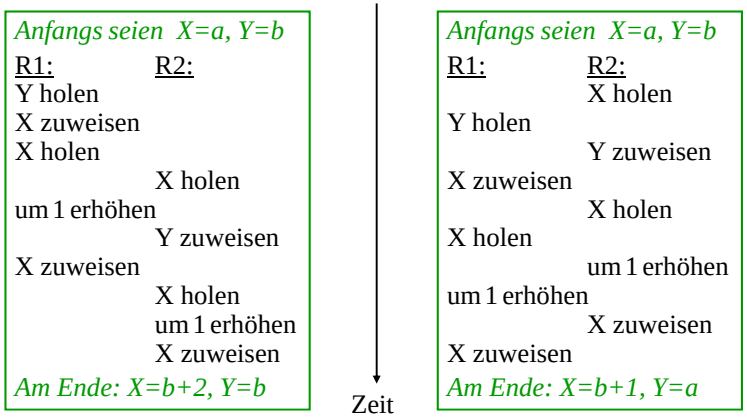
Sind überhaupt keine Anweisungsteile "nicht unterbrechbar", dann sind alle zeitlich verschachtelten Reihenfolgen möglich.

Beispiel: (R1:: X:=Y; X := X+1 | R2:: Y:=X; X:=X+1)

Man kann diese Anweisungen nun beliebig in der zeitlichen Reihenfolge ineinander stecken. Vier Beispiele sind:



(R1:: X:=Y; X := X+1 | R2:: Y:=X; X:=X+1)



Prüfen Sie: Lassen sich auch *Am Ende: X=b+2, Y=b+1* oder *Am Ende: X=a+2, Y=b+1* erreichen? Wie viele verschiedene Möglichkeiten gibt es bei diesem Beispiel?

Hierbei können Konflikte auftreten. Beispielsweise muss man vermeiden, dass ein Prozess Werte in eine Variable (= in einen Speicherbereich) schreibt, während ein anderer Prozess diese Variable ebenfalls verändert. Das Gleiche gilt, wenn irgendein anderes Betriebsmittel (Eingabegerät, Drucker, Übertragungmedium, Beamer usw.) exklusiv genutzt werden muss.

Die Anweisungsfolge, die ungestört von einem nebenläufigen Prozess ausgeführt werden muss, nennt man einen *kritischen Abschnitt*. In der Regel muss hierbei ein Betriebsmittel (z. B. der Prozessor) exklusiv genutzt werden.

Gewisse exklusive Zugriffe lassen sich hardwaremäßig sicherstellen, etwa der Zugriff auf eine Speicherzelle. Für Anweisungsfolgen muss man in der Praxis softwaremäßige Lösungen finden.

13.4.6 Definition:

Ein sequentiell abzuarbeitender Teil eines Programms heißt kritischer Abschnitt, wenn es ein Betriebsmittel gibt, das während der Ausführung dieses Programmteils von keinem anderen (hierzu nebenläufigen) Prozess genutzt werden darf. Prozesse, die auf das gleiche Betriebsmittel zugreifen wollen, *konkurrieren* um dieses Betriebsmittel und *bilden einen Konflikt*.

In einem kritischen Abschnitt für ein Betriebsmittel darf sich zu jedem Zeitpunkt höchstens einer der konkurrierenden Prozesse befinden. Kann man sicherstellen, dass sich bei einem Konflikt zu jedem Zeitpunkt höchstens einer der konkurrierenden Prozesse in seinem kritischen Abschnitt befindet, so spricht man vom wechselseitigen Ausschluss (mutual exclusion).

13.4.7 Beispiel

Ein kritischer Abschnitt ist in einem Programm das Ausdrucken von Daten, wenn nur ein Drucker vorhanden ist. Im einfachsten Fall konkurrieren nur zwei Prozesse um den Drucker.

Wir beschreiben im Folgenden den wechselseitigen Ausschluss zuerst mit einem S/T-Netz: "Stellen" sind die elementaren Anweisungen der Prozesse, "Transitionen" sind die Übergänge zu den jeweils nächsten Berechnungen.

Anschließend realisieren wir den wechselseitigen Ausschluss softwaremäßig durch geeignete Kontrollvariable in den beiden Prozessen.

Beispielprogramm: Zwei Prozesse wollen X bzw. Y drucken.

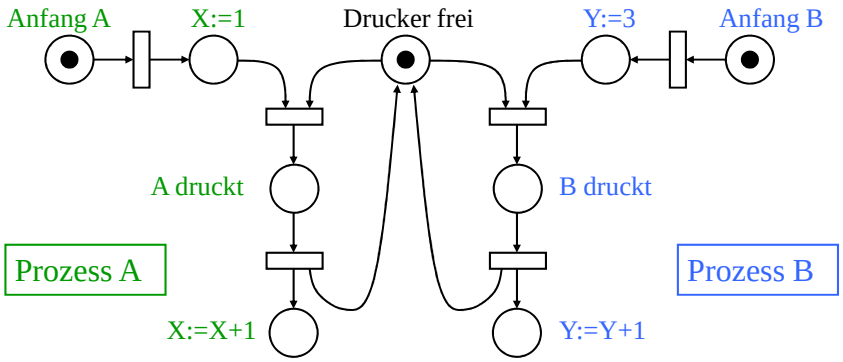
[local X, Y : Integer;

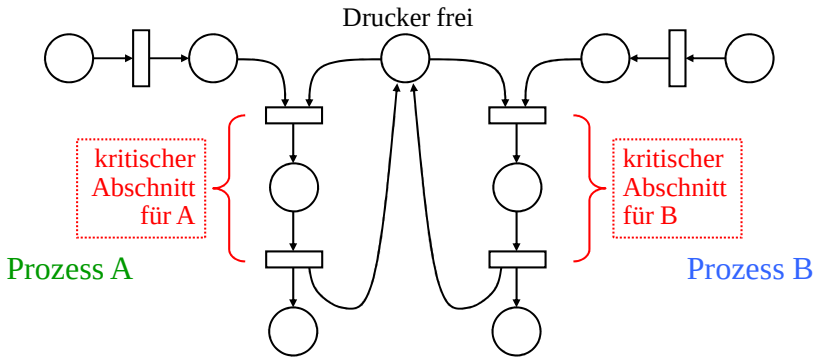
(A:: X:=1; write(X); X:=X+1 | B:: Y:=3; write(Y); Y:=Y+1)]

[local X, Y : Integer;

(A:: X:=1; write(X); X:=X+1 | B:: Y:=3; write(Y); Y:=Y+1)]

Die Programmteile write(X) und write(Y) bilden für jeden der beiden Programmteile kritische Abschnitte. Für dieses Beispiel lässt sich der wechselseitige Ausschluss mit S/T-Netzen leicht modellieren, indem man dem Betriebsmittel "Drucker" eine Stelle "frei" zuordnet :





Beschreibung des wechselseitigen Ausschlusses mit einem S/T-Netz.

Für die Realisierung in der Programmierung wird die Stelle "Drucker frei" durch eine Boolesche Variable dargestellt:

Obiges Beispiel als Programm mit wechselseitigem Ausschluss:

```
[ local X, Y: Integer; frei: Boolean := true;
   (A:: X:=1;                  |   B::  Y:=3;
       await frei;             |       await frei;
       frei:=false;            |       frei:=false;
       write(X);               |       write(Y);
       frei := true;           |       frei := true;
       X:=X+1                  |       Y:=Y+1 )      ]
```

Diese Realisierung ist nur dann korrekt, wenn die Anweisungsfolge " await frei; frei := false " nicht unterbrechbar ist! Anderenfalls könnten beide Prozesse (fast) gleichzeitig auf die Variable "frei" zugreifen und sie beide als true erkennen. Beide Prozesse würden dann fälschlicherweise gleichzeitig ihre kritischen Abschnitte betreten können.

Hinweis: Wenn k Drucker für mehrere Prozesse vorliegen, so würde man im S/T-Netz die Stelle "Drucker frei" anfangs mit k Marken belegen. Bei der Übertragung in ein Programm muss man dann eine Variable

```
Drucker_frei: Natural := k;
```

deklarieren. Will einer der Prozesse in seinen kritischen Abschnitt eintreten, so würde man schreiben:

```
await Drucker_frei > 0;
Drucker_frei := Drucker_frei - 1;
< kritischer Abschnitt >;
Drucker_frei := Drucker_frei + 1;
```

Im allgemeinsten Fall würde man bei der Programmierung noch prüfen, ob die Variable Drucker_frei einen maximalen Wert MAX nicht überschreiten kann, d.h., man würde vor dem Erhöhen von Drucker_frei noch await Drucker_frei < MAX einfügen.

13.4.8 a Definition (das Semaphor, eingeschränkte Form)

Eine Variable vom Typ Natural, die eine Menge von maximal MAX Ressourcen "bewacht", zusammen mit den nicht unterbrechbaren Operationen "Warten und Erniedrigen" und "Warten und Erhöhen" bezeichnet man als Semaphor.
(Im Falle MAX=1 kann man auch eine Variable vom Typ Boolean verwenden, siehe oben.)

Erläuterung des Namens: Unter einem Semaphor versteht man einen Signalmast, auch "Flügeltelegraph" genant. Solche Masten wurden ab 1790 für die optische Übermittlung von Nachrichten benutzt. Ab 1840 (in Europa ab 1850) wurden sie rasch durch die elektrische Nachrichtenübertragung ("Telegraph") verdrängt. Es gibt sie noch als "Windtelegraphen" in der Schifffahrt. Vorgänger waren bereits bei den alten Griechen um 500 v. Chr. in Gebrauch, vor allem als Feuerzeichen-Übertragung nachts.

Hinweis: Das allgemeine Semaphorkonzept wurde 1968 von dem niederländischen Informatiker E. W. Dijkstra eingeführt. Neben der Kontrollvariablen S besitzt jedes solche Semaphor eine Warteschlange, in die alle Prozesse nacheinander eingetragen werden, die zur Zeit noch nicht auf das Betriebsmittel zugreifen können. Das Semaphor aktiviert die Prozesse in der Warteschlange, sobald ein angefordertes Betriebmittel frei ist.

In der Literatur bezeichnet man die Operation "Warten und Erniedrigen", also await S > 0; S := S - 1 auch als P-Operation der Semaphorvariablen S (nach dem niederländischen Wort Passeer = Betreten) oder als Warteoperation. Die andere Operation heißt V-Operation (vom niederländischen Verlaat = Verlassen) oder Signaloperation.

13.4.8 b Definition (das Semaphor, ausführliche Form)

Ein Semaphor besteht aus einer Variablen S des Typs Natural (oder 0..MAX), einer Warteschlange W(S) für Prozesse und folgenden beiden nicht-unterbrechbaren Operationen P und V:

```
procedure P (S: in out Natural);
begin  if S > 0 then S := S-1;
       else  <Stoppe diesen Prozess>;
             <trage ihn in die Warteschlange W(S) ein>;  end if;
end P;

procedure V (S: in out Natural);
begin  S := S+1;
if not isempty(W(S)) then <Wähle einen Prozess A aus W(S) aus>;
       < aktiviere dessen Ausführung ab der P-Operation in A,
         durch die A gestoppt wurde >;  end if;
end V;
```

Obiges Beispiel als Programm mit allgemeinem Semaphor:

```
[ local X, Y : Integer; S: semaphor;
   (A:: X:=1;                    |   B:: Y:=3;
        P(S);                    |        P(S);
        write(X);                |        write(Y);
        V(S);                    |        V(S);
        X:=X+1                   |        Y:=Y+1 )      ]
```

Semaphore sind eine Standardtechnik, um den wechselseitigen Ausschluss zu realisieren, bzw. allgemein, um eine gegebene Menge von Betriebsmitteln mehreren auf sie zugreifenden Prozessen zur Verfügung zu stellen, ohne dass eine Verklemmung eintreten kann.
(Vgl. "Scheduler" in Vorlesungen über Betriebssysteme.)

Problem: Kann man durch ein Programm, das nur unsere Anweisungen benutzt (also keine allgemeinen Semaphore kennt), den wechselseitigen Ausschluss sicherstellen, falls keine unterbrechbaren Operationen vorliegen (nur das Schreiben in eine Variable sei nicht-unterbrechbar)?

Wie muss man also das bisherige Programm

```
[ local X, Y : Integer; frei: Boolean := true;
   (A:: X:=1;                    |   B:: Y:=3;
        await frei;              |        await frei;
        frei:=false;             |        frei:=false;
        write(X);                |        write(Y);
        frei := true;            |        frei := true;
        X:=X+1                   |        Y:=Y+1 )      ]
```

abändern? Oder kann man dies gar nicht garantieren??

Vorschlag: Führe eine Variable ein, die nur die Werte PA und PB annehmen kann.

```
[ local  type prozess is (PA, PB);
         Nr: prozess := PA;  X, Y: Integer;

  (A:: X:=1;                   |   B:: Y:=3;
       Nr := PB;               |        Nr := PA;
       await Nr = PA;          |        await Nr = PB;
       write(X);               |        write(Y);
       Nr := PB;               |        Nr := PA;
       X:=X+1                  |        Y:=Y+1    )    ]
```

Jeder Prozess gibt dem anderen Prozess die Berechtigung, als erster in den kritischen Abschnitt einsteigen zu können.
Ist dies ein Konzept, das Fehler stets vermeidet?

Dieses Programm verhindert zwar, dass sich beide Prozesse gleichzeitig in ihrem kritischen Abschnitt befinden können, aber wenn die Anweisungsfolgen (wie oft üblich) in einer unendlichen Schleife stehen, dann können die beiden Prozesse nur abwechselnd ihren kritischen Abschnitt betreten, und beide Prozesse müssen dauernd aktiv bleiben, sonst wartet der andere Prozess ewig:

⇒ Unbrauchbare Lösung

```
[ local  type prozess is (PA, PB);
         Nr: prozess := PA;  X, Y: Integer;

  (A:: while true do           |   B:: while true do
         X:=1; Nr := PB;       |          Y:=3; Nr := PA;
         await Nr = PA;        |          await Nr = PB;
         write(X);             |          write(Y);
         Nr := PB; X:=X+1      |          Nr := PA; Y:=Y+1
       od                      |        od    )          ]
```

13.4.9 Problemformulierung: Gesucht wird also eine Softwarelösung, bei der jeder Prozess unabhängig vom anderen ist, außer in dem Fall, dass beide zur gleichen Zeit in ihren kritischen Abschnitt eintreten wollen. Hierzu gibt es diverse Lösungen in der Literatur (z.B.: T. Dekker 1965, G. L. Peterson 1981, S. Owicki und L. Lamport 1982).

Versuchen Sie zunächst selbst, für "Vorbereitung A" bzw. B und "Nachbereitung A" bzw. B eine Lösung des allgemeinen Problems zu finden:

```
[ local <Deklarationen>;
   (A:: while true do                 |  B::  while true do
          Vorbereitung A;             |          Vorbereitung B;
          kritischer Abschnitt A;     |          kritischer Abschnitt B;
          Nachbereitung A             |          Nachbereitung B
        od                            |        od    )              ]
```


Wir geben hier nur die Lösung von Peterson an. Jeder Prozess hat hierbei eine eigene Boolesche Variable PrA bzw. PrB, die den Wunsch, in den kritischen Abschnitt einzutreten, signalisiert. Zusätzlich gibt es eine Variable "dran", die dem anderen Prozess den Vortritt lässt, sofern dieser auch gerade in den kritischen Abschnitt will.

Diese Lösung erfüllt die geforderten Eigenschaften:

- Es kann sich zu jedem Zeitpunkt nur ein Prozess in seinem kritischen Abschnitt befinden.
- Jeder Prozess kann in seinen kritischen Abschnitt gelangen unabhängig davon, wo sich der andere Prozess befindet oder ob er noch aktiv ist.
- Es tritt keine Verklemmung auf.

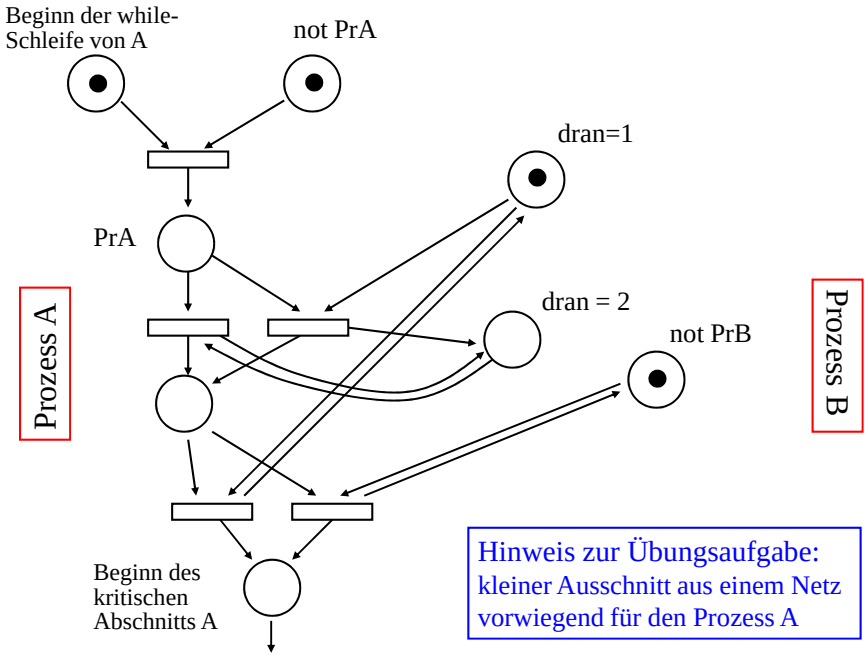
13.4.10 Die Lösung von Peterson (1981):
(OR ist hier das Oder in einem Booleschen Ausdruck)

```
[ local  dran: Integer:=1;  PrA, PrB: Boolean:= false;

(A:: while true do                          B:: while true do
       PrA := true;                                PrB := true;
       dran := 2;                                  dran := 1;
       await (not PrB) OR (dran=1);                await (not PrA) OR (dran=2);
       < kritischer Abschnitt A >;                 < kritischer Abschnitt B >;
       PrA := false;                               PrB := false;
       ...                                         ...
     od                                          od                )          ]
```


In der Vorlesung wird an der Tafel die Arbeitsweise dieses Vorgehens genauer erläutert. Machen Sie sich diese Arbeitsweise an einem Ablaufdiagramm klar!

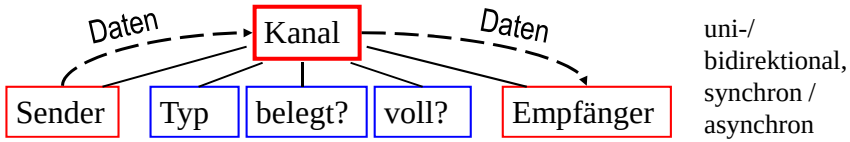
Zu diesem Programm kann man ein Stellen-Transitions-Netz zeichnen, das die Arbeitsweise genau widerspiegelt. Einen Beweis für die Korrektheit der Peterson-Lösung kann man dann über dieses S/T-Netz führen.

Übungsaufgabe: Konstruieren Sie dieses S/T-Netz zu der oben angegebenen Peterson-Lösung.



13.4.11 Kanäle: Wir haben einige Aspekte des Nachrichtenaustausches vorgestellt, der über einen gemeinsamen Speicherbereich erfolgt. Anders funktioniert das Telefonieren: Dort wird jedem solchen Nachrichtenaustausch ein eigener Kanal zur Verfügung gestellt, der nach dessen Beendigung einer anderen Kommunikation zugeordnet werden kann.

Anstelle des Ablegens von Informationen in einem gemeinsamen Speicherbereich betrachten wir nun also die Nutzung von Kanälen, über die eine Verbindung zwischen zwei Partnern hergestellt werden kann.



uni-/
bidirektional,
synchron /
asynchron

Für Kanäle erlauben wir übergreifend den Datentyp
"channel [1..max] of T"
in den Daten d vom Typ T mittels CH ← d vom Sender hineingelegt und aus dem Daten dieses Typs vom Empfänger mittels CH → X seiner Variablen X zugewiesen werden können. (CH ist eine Variable des eingeführten Datentyps.). Benutzen zwei Prozesse den gleichen Kanal, so muss einer Sender und einer Empfänger sein und es können Daten nur vom Sender an den Empfänger geschickt werden.

Ein Kanal ist wie eine Warteschlange organisiert und er besitzt in der Regel eine Kapazität. Die Daten, die zuerst hineingesteckt werden, kommen auch als erste wieder heraus (FIFO-Prinzip) und die Warteschlange kann meist nur die begrenzte Zahl "max" von Daten aufnehmen.

Mit einem Kanal muss weiterhin eine Boolesche Variable "belegt" verbunden sein, die einen Kanal nicht frei gibt, sofern er derzeit benutzt wird.

Die Anweisung CH ← X in einem Prozess P besagt also:

Wenn der Kanal CH nicht belegt ist, so wird er als belegt gekennzeichnet und P ist der Sender für CH;
wenn CH belegt ist und bisher nur der Empfänger dem Kanal zugeordnet ist, so wird P der Sender von CH;
wenn CH belegt ist und schon zwei Partner besitzt, so muss P unter diesen Partnern der Sender sein.

Trifft eine dieser drei Bedingungen zu, so wird geprüft, ob die Daten (hier: der Wert von X) vom Typ T sind und ob CH noch Platz für die Aufnahme eines weiteren Datums besitzt.

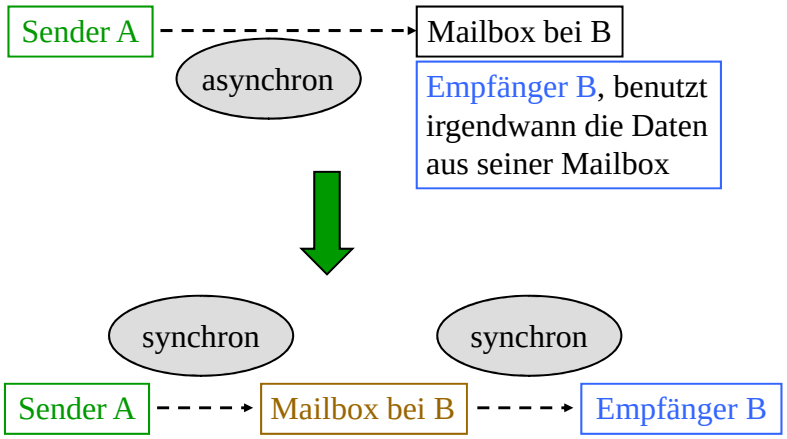
Trifft auch dies zu, so wird der Wert von X in den Kanal gelegt; anderenfalls erfolgt eine geeignete Fehlermeldung.

Analog ist die Anweisung CH → X in einem Prozess P zu interpretieren.

Will man einen Datenaustausch zwischen zwei Prozessen installieren, so muss man zwei Kanäle verwenden (wie beim Telefonieren). Will man den Zustand der Kanäle noch überwachen oder unabhängig von den Daten Kontrollinformationen übertragen, so muss man einen oder zwei weitere Kanäle hinzunehmen (wie beim Telefon).

Man muss noch festlegen, ob eine synchrone Verbindung besteht (wenn A sendet, so muss B zeitgleich empfangen; A kann erst weiterarbeiten, wenn B alle Daten empfangen hat) oder ob die Daten aynchron überliefert werden (z.B. in eine Mailbox gelegt werden; allerdings muss dann die Mailbox mit A oder mit dem Kanal synchron zusammenarbeiten).

13.4.12: Hieraus folgt: Eine asynchrone Kommunikation zwischen zwei Prozessen kann man durch zwei synchrone Kommunikationen zwischen drei Prozessen simulieren:



Ein solcher Fall liegt beim Erzeuger-Verbraucher-Kreislauf vor (siehe 13.1.3): Der Erzeuger schickt asynchron seine Produkte an den Verbraucher. Fasst man das Lager als zusätzlichen Prozess auf, so lässt sich dieser Vorgang synchron darstellen (siehe nächste Folie).

Wir wollen diese Ausführungen nun mit einem Beispiel beenden, an dem die prinzipielle Arbeitsweise von Kanälen abgelesen werden kann. Das Thema des Nachrichtenaustausches wird in Vorlesungen über Betriebssysteme, Verteilte Systeme und Sichere Systeme weiter vertieft.

Als Beispiel wählen wir, wie oben gesagt, den Erzeuger-Verbraucher-Kreislauf mit der Kapazität 10 des Lagers (hier bedeutet "or" die nicht-deterministische Auswahl aus 13.4.2; die erzeugten und verbrauchten Elemente sind vom Datentyp T).

13.4.13 Beispiel EL, LV: channel [1..1] of T;

```
local  X: T;                 local  Z: T;  k: 0..10 :=0;        local  Y: T;
                             puffer: array [1..10] of T;
while true do                while true do                      while true do
"erzeuge X";                   ( if k < 10 then                 LV → Y;
EL ← X                             EL → Z; k := k+1;            "verbrauche Y"
od                                 puffer[k] := Z  fi           od
                               or
                               if k > 0 then
                                   LV ← puffer[k];
                                   k := k-1  fi )
                             od
```

Erzeuger | Lager | Verbraucher

13.4.14 Zum Abschluss seien noch einige Begriffe aufgelistet:

Geblockte Übertragung: Daten werden meist nicht einzelnen, sondern in größeren Einheiten (Blöcken) übertragen. Dies erhöht vor allem die Effizienz der Übertragung.

Gepackte Daten: Daten werden oft noch komprimiert, damit sie weniger Platz benötigen. Beim Empfänger müssen sie dann wieder "entpackt" werden (Beispiel: zip-Files).

Synchron: Der Empfänger übernimmt die Daten, während der Sender sendet.

Asynchron: Die Daten werden irgendwo gepuffert, bis der Empfänger sie abholt. Das Puffern kann auch im Kanal integriert sein.

Blockierendes Senden: Der Sender kann erst weiterarbeiten, wenn alle gesendeten Daten entweder beim Empfänger angekommen sind oder vom Puffer des Kanals aufgenommen wurden.

Blockierendes Empfangen: Der Empfänger kann erst weiterarbeiten, wenn alle gesendeten Daten bei ihm abgespeichert sind.

Den Datenaustausch mittels synchronem blockierendem Senden und blockierendem Empfangen bezeichnet man als Rendezvous. Dies ist z.B. in Ada realisiert. Vertiefungen zu diesem Themenbereich siehe: Verteilte Systeme, Sichere Systeme, Eingebettete Systeme, Architekturen usw.

14. Pseudo-Zufallszahlen

14.1 Modulo-Rechnen

14.2 "Zufallszahlen"

Dieses kurze Kapitel dient vor allem dazu, die Sensibilität gegenüber den Zufallsgeneratoren in Informatiksystemen zu erhöhen, da die Verwendung von randomisierten Verfahren zunimmt.

14.1 Modulo-Rechnen

14.1.1 Wir wiederholen aus der Mathematik einige Begriffe.

Es sei Z die Menge der ganzen Zahlen ("integers"). Mit den beiden Operationen "+" und "·" und den zugehörigen Einheiten "0" und "1" wird Z zu einem kommutativen Ring $(Z,+,\cdot,0,1)$, d.h., $(Z,+,0,-)$ ist eine kommutative Gruppe, $(Z,\cdot,1)$ ist ein kommutatives Monoid und "+" und "·" erfüllen das Distributivgesetz.

Wenn $(Y,\oplus,\bullet,o,e)$ ebenfalls ein (kommutativer) Ring ist, dann heißt eine Abbildung $h: Z \to Y$ ein (Ring-) Homomorphismus, wenn gilt:
(1) $h(0) = o$ und $h(1) = e$ (Einheiten werden auf Einheiten abgebildet)
(2) $h(a+b) = h(a) \oplus h(b)$ für alle $a,b \in Z$,
(3) $h(a \cdot b) = h(a) \bullet h(b)$ für alle $a,b \in Z$.

Homomorphismen sind strukturerhaltende Abbildungen. (Diese Definition gilt entsprechend für alle algebraischen Strukturen.)

Auf der Menge der ganzen Zahlen kann man zwei Operationen "div" und "mod" definieren: die ganzzahlige Division und die Restbildung bzgl. dieser Division. Sie erfüllen stets die Gleichung $a = (a \text{ div } b) \cdot b + (a \text{ mod } b)$. Grundlage ist der

Satz:
Für alle $a, b \in Z$ mit $b \geq 1$ existieren genau zwei Zahlen $p, q \in Z$ mit
(1) $0 \leq q < b$,
(2) $a = p \cdot b + q$.

Beweis, zugleich mit der Definition von div und mod:
Fall 1: $0 \leq a$: Dann wähle p als die Zahl, für die gilt $p \cdot b \leq a < (p+1) \cdot b$. Diese Zahl p existiert und man erhält sie, indem man ausgehend von 0 die Zahl b solange aufaddiert, bis diese Ungleichung erfüllt ist. p ist eindeutig, denn wenn p und p' diese Ungleichung erfüllen würden, so wären $0 \leq a - p \cdot b < b$ und $0 \leq a - p' \cdot b < b$, also $-b < (p - p') \cdot b < b$, was nur für $p = p'$ zutrifft. Setze $p =: a \text{ div } b$ und $q = a - p \cdot b =: a \text{ mod } b$.
Fall 2: $a < 0$: Sei $a'' = -a$, dann erfüllen $p = -1 - ((a''-1) \text{ div } b) =: a \text{ div } b$ und $q = a - p \cdot b =: a \text{ mod } b$ die Beziehungen (1) und (2).

14.1.2 Restklassenring von Z modulo n

Es sei n eine natürliche Zahl, $n \geq 2$. Dann sei $Z_n = \{0, 1, 2, ..., n-1\}$ die Menge der ersten n natürlichen Zahlen, beginnend mit 0. Auf Z_n werden zwei Operationen "+" und "·" definiert. Für alle $a,b \in Z_n$ sei

$a + b = (a+b) \bmod n$, wobei a+b in Z zu bilden ist,
$a \cdot b = (a \cdot b) \bmod n$, wobei $a \cdot b$ in Z zu bilden ist.

Dann ist $(Z_n,+,\cdot,0,1)$ ein kommutativer Ring.

Für jedes n existiert ein Homomorphismus $h_n: Z \to Z_n$, definiert durch $h_n(a) = a \bmod n$ für alle $a \in Z$. Weisen Sie elementar nach, dass h_n für jedes $n \geq 2$ ein (surjektiver) Ring-Homomorphismus ist.

Z_n bzw. $(Z_n,+,\cdot,0,1)$ heißt "Restklassenring von Z modulo n".

Genau dann, wenn n eine Primzahl ist, ist Z_n ein Körper (d.h., dann ist $(Z_n - \{0\},\cdot,1)$ sogar eine Gruppe).

Statt + und · schreiben wir nun wieder + und ·.

14.1.3 Der chinesische Restklassensatz (chinese remainder theorem)

Bezeichnungen: Im Folgenden sei $m \geq 2$ eine natürliche Zahl. Weiterhin seien $q_1, q_2, ..., q_m$ m paarweise teilerfremde natürliche Zahlen. Es sei Q das Produkt dieser Zahlen, also $Q = q_1 \cdot q_2 \cdot ... \cdot q_m$.
Für jedes $i = 1, 2, ..., m$ sei Q_i das Produkt der m Zahlen, aber ohne die Zahl q_i, also $Q_i = Q/q_i$.

Für jede natürliche Zahl $a \in \mathbb{N}_0$ sei (der m-dimensionale Vektor)

$a^* = (a \bmod q_1, a \bmod q_2, ..., a \bmod q_m)$

die Restedarstellung von a bzgl. $q_1, q_2, ..., q_m$.

Es sei $V_Q = \{a^* \mid a \in \mathbb{N}_0\}$ die Menge aller möglichen Restedarstellungen. Dann ist $|V_Q| \leq Q$, da in der i-ten Komponente höchstens q_i verschiedene Werte auftreten können. Auf V_Q sind die Operationen "+", "-" und "·" komponentenweise modulo q_i definiert. Z.B. ist
$a^* \cdot b^* = ((a \bmod q_1 \cdot b \bmod q_1) \bmod q_1, ..., (a \bmod q_m \cdot b \bmod q_m) \bmod q_m)$.

Es ist $Z_Q = \{0, 1, 2, ..., Q-1\} \subset \mathbb{N}_0$ der Restklassenring modulo Q. Wir zeigen, dass Z_Q und V_Q isomorph sind.

Satz *(chinesischer Restklassensatz)*

(1) Zu je m natürlichen Zahlen $c_1, c_2, ..., c_m \in \mathbb{N}_0$ existiert genau eine Zahl $x \in Z_Q$ mit $x = c_i \bmod q_i$ für $i = 1, 2, ..., m$.

(2) Für alle natürlichen Zahlen y mit $y = c_i \bmod q_i$ für $i = 1, 2, ..., m$ gilt $x = y \bmod Q$.

(3) Die Abbildung $h: Z_Q \to V_Q$ mit $h(z) = z^*$ ist ein Isomorphismus.

Zu (3): Betrachte x und $x' \in Z_Q$ mit $h(x) = h(x')$, d.h., $0 \leq x < Q$, $0 \leq x' < Q$, $x = c_i \bmod q_i$ und $x' = c_i \bmod q_i$ für $i = 1, 2, ..., m$. Dann gilt für jedes i: $(x - x') = 0 \bmod q_i$ für $i = 1, 2, ..., m$. Die Differenz von x und x' ist also ein Vielfaches von q_1 und ein Vielfaches von q_2 und ... und ein Vielfaches von q_m. Da die q_i jedoch paarweise teilerfremd sind, muss x-x' ein Vielfaches der Zahl $q_1 \cdot q_2 \cdot ... \cdot q_m = Q$ sein. Wegen x und $x' \in Z_Q$ muss gelten: $-Q < x - x' < Q$, also $(x-x') = 0$. Aus $h(x) = h(x')$ folgt daher $x = x'$, d.h., h ist injektiv und (weil $|V_Q| \leq Q = |Z_Q|$) somit auch bijektiv. Da Abbildung $h_n: Z \to Z_n$ in 14.1.2 ein Homomorphismus ist, folgt sofort, dass auch h ein Homomorphismus sein muss. $\Rightarrow$ (3).

Zu (1): Da Z_Q und V_Q isomorph sind, gibt es zur Restedarstellung $(c_1 \bmod q_1, c_2 \bmod q_2, ..., c_m \bmod q_m) \in V_Q$ also genau ein zugehöriges $x \in Z_Q$, woraus Teil (1) folgt.

Zu (2): Wegen $Q = 0 \bmod q_i$ für $i = 1, 2, ..., m$ gilt für alle Zahlen y: $y = (y + Q) \bmod q_i$ für $i = 1, 2, ..., m$. Zu jeder Zahl y mit $y = c_i \bmod q_i$ für $i = 1, 2, ..., m$ kann man durch wiederholtes Addieren oder Subtrahieren genau eine Zahl $x' = y + k \cdot Q$ mit $k \in Z$, $0 \leq x' < Q$ und $x = c_i \bmod q_i$ für $i = 1, 2, ..., m$ gewinnen. Da die Zahl x aus Teil (1) und diese Zahl x' die gleiche Restedarstellung besitzen, muss wegen (3) also $x = x'$ sein, woraus $x = y + k \cdot Q$ und daher $x = y \bmod Q$ folgt.

Damit ist der Satz bewiesen.

Folgerung: Statt in Z können wir also auch in Z_Q rechnen, sofern wir diesen Zahlenbereich nicht verlassen. Beachte: Statt direkt in Z_Q kann man auch parallel in den m Restklassenringen Z_{q_i} rechnen. Dadurch können spezielle Rechnungen stark beschleunigt werden.

Umwandlungsverfahren: Um diesen Vorteil nutzen zu können, müssen wir die Zahlen und die Restdarstellungen wechselseitig ineinander überführen können.

Zahl a $\rightarrow$ Restedarstellung a^*: Bilde m-mal a $\underline{\text{mod}}$ q_i.

Restedarstellung $a^* = (a_1, a_2, ..., a_m) \rightarrow$ Zahl a:
Bilde $a = a_1 \cdot d_1 + a_2 \cdot d_2 + ... \; a_m \cdot d_m$.

Hierbei ist d_i die Zahl, deren Restedarstellung der "Einheitsvektor" e_i ist mit:

$e_i = (0, 0, ..., 0, 1, 0, ..., 0)$ mit $d_i^* = e_i$.

genau die i-te Komponente ist 1,
alle anderen Komponenten sind 0.

Offenbar ist e_i die Restedarstellung einer Zahl $k \cdot Q_i$, weil stets $Q_i \underline{\text{mod}} q_j = k \cdot Q_i \underline{\text{mod}} q_j = 0$ für alle $j \neq i$ gilt. k muss nur die Gleichung $k \cdot Q_i \underline{\text{mod}} q_i = 1$ erfüllen. Dieses k kann man durch Ausprobieren oder mit dem ggT-Verfahren von Abschnitt 1.7.7 ermitteln. (Selbst nachdenken.)

In der Praxis wählt man die q_i als Primzahlpotenzen, z.B. m = 6 und

$q_1 = 2048 = 2^{11}$, $q_2 = 2187 = 3^7$, $q_3 = 3125 = 5^5$,
$q_4 = 2401 = 7^4$, $q_5 = 2197 = 13^3$, $q_6 = 2209 = 47^2$.

Dann kann man die Zahlen von 0 bis Q = 163 097 269 323 206 400 000 $\approx 1{,}63 \cdot 10^{20}$ erfassen, wofür man 6 parallele arithmetische Einheiten bauen muss, die möglichst schnell modulo $q_1, q_2, ..., q_6$ rechnen können. (Solche Fragen werden in der Technischen Informatik beantwortet.)

Bei den Anwendungen dürfen - wie schon erwähnt - keine Größer- oder Kleiner-Abfragen vorkommen, da diese sich nur mit größerem Aufwand aus den Restedarstellungen ermitteln lassen. Die Zahlen d_i, die zu den Einheitsvektoren e_i gehören, lassen sich leicht berechnen.

In obigem Beispiel ist:
$d_1 = k \cdot Q_1 = k \cdot q_2 \cdot q_3 \cdot q_4 \cdot q_5 \cdot q_6 = k \cdot 79637338536721875$
mit $k \cdot Q_1 \underline{\text{mod}} 2048 = 1$. Man berechnet zunächst $Q_1 \underline{\text{mod}} 2048 = 1491$ und muss nun noch $(k \cdot 1491) \underline{\text{mod}} 2048 = 1$ untersuchen. Man erhält: k = 603 und somit $d_1 = 48\,021\,315\,137\,643\,290\,625 \approx 0{,}48 \cdot 10^{20}$. (Mit Rechnerhilfe findet man, nach Eingabe der q_i, alle d_i in weniger als einer Sekunde.)

14.2 "Zufallszahlen"

14.2.1 Einleitung: Oft braucht man Zufallszahlen, also eine Folge von Zahlen $z_1, z_2, z_3, ...$, die sich wie Zahlen verhalten, die bei einem zufälligen Prozess entstanden sind. Meist sollen diese Zahlen "gleichverteilt" sein, also: Jede Zahl soll zufällig mit gleicher Wahrscheinlichkeit zwischen 0 und m-1 liegen oder gleichverteilt eine reelle Zahl mit $0.0 \leq z_k < 1.0$ sein.

Eine Zahlenfolge, die von einem Algorithmus erzeugt wird, kann jedoch niemals "zufällig" sein. Man kann aber eine Zahlenfolge berechnen lassen, die viele Eigenschaften hat, die auch eine Zufallsfolge besitzt und die daher anstelle einer Zufallsfolge verwendet werden kann ("Pseudozufallszahlen").

Die Zahlen der Folge mögen aus einer endlichen Menge stammen. Beim Würfeln ist dies z.B. die Menge {1, 2, 3, 4, 5, 6}.

"Ohne Beschränkung der Allgemeinheit"

O.B.d.A. betrachten wir die m-elementige Menge $\{0, 1, ..., m\text{-}1\}$. Der Algorithmus realisiert dann eine Funktion

$f: \{0, 1, ..., m\text{-}1\} \rightarrow \{0, 1, ..., m\text{-}1\}$.

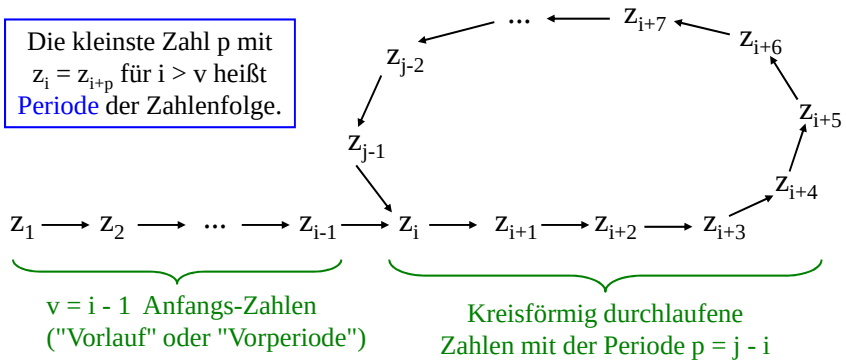
Wir gehen von einer Zahl $z_0 \in \{0, 1, ..., m\text{-}1\}$ aus und berechnen aus der Zahl z_{k-1} mit dem gegebenen Algorithmus die jeweils nächste Zahl z_k:

$z_1 = f(z_0)$, $z_2 = f(z_1)$, $z_3 = f(z_2)$, $z_4 = f(z_3)$, ...

Es gilt also $z_k = f^k(z_0) = \underbrace{f(f(f(...f}_{k \text{ mal}}(z_0))...)))$.

Wegen $|\{0, 1, ..., m\text{-}1\}| = m$ muss in der Folge $z_1, z_2, ..., z_m, z_{m+1}$ mindestens eine Zahl zweimal vorkommen. Seien $1 \leq i < j \leq m+1$ die kleinsten Indizes mit $z_i = z_j$. Dann muss folgende Situation vorliegen:

Die kleinste Zahl p mit $z_i = z_{i+p}$ für $i > v$ heißt Periode der Zahlenfolge.



Zu jeder Funktion $f: \{0,1,...,m-1\} \to \{0,1,...,m-1\}$ und jedem Anfangswert z_0 gibt es also zwei Zahlen v und p mit $0 \le v \le m-1$ und $1 \le p \le m$ und $v+p \le m$, so dass die Folge der Zahlen $z_k = f^k(z_0)$ identisch ist mit der Folge ($z_i \ne z_j$ für alle $1 \le i < j \le v+p$)

$z_1, z_2, ..., z_v, z_{v+1}, z_{v+2}, ..., z_{v+p}, z_{v+1}, z_{v+2}, ..., z_{v+p}, z_{v+1}, z_{v+2}, ..., z_{v+p}, z_{v+1}, ...$

Der gesuchte Algorithmus sollte ein möglichst kleines v und ein möglichst großes p haben, im Idealfall v = 0 und p = m.

Wir halten folgende Aussage fest:

14.2.2 Satz

Es seien m und $z_0 < m$ natürliche Zahlen ("Modul" und "Startwert") und f eine Funktion $f: \{0,1,...,m-1\} \to \{0,1,...,m-1\}$. Dann kann man die Folge der iterierten Anwendung dieser Funktion $z_i = f^i(z_0)$ für i = 1, 2, 3, ... eindeutig in einen (eventuell leeren) Anfangsteil und einen sich hieran anschließenden periodischen Teil aufspalten, d.h.:

Es gibt es zwei (eindeutig bestimmte) Zahlen v und p mit

$0 \le v \le m-1$ und $1 \le p \le m$ und $v+p \le m$

(wobei man p minimal mit der folgenden Eigenschaft wählt)

so dass die unendliche Folge der z_i die Form

$z_1\ z_2\ ...\ z_v\ (z_{v+1}\ z_{v+2}\ ...\ z_{v+p})^\infty$

besitzt.

p heißt die Periode und v der Vorlauf der Folge $\{z_i\}_{i>0}$. Den Startwert z_0 bezeichnet man im Englischen als "seed".

14.2.3 Beispiele für $f: \{0,1,...,m-1\} \to \{0,1,...,m-1\}$:

(1) $f(z) = (z+1) \bmod m$ und $z_0 = 0$.

Dann lautet die Folge $z_1, z_2, ...$:

1, 2, ..., m-1, 0, 1, 2, ..., m-1, 0, 1, ...

Die Parameter sind: v = 0 und p = m.

Diese Folge ist alles andere als zufällig.

(2) $f(z) = (z+m-1) \bmod m$ und $z_0 = 0$.

Dann lautet die Folge $z_1, z_2, ...$:

m-1, m-2, m-3, ..., 2, 1, 0, m-1, m-2, ..., 2, 1, 0, m-1, ...

Die Parameter sind: v = 0 und p = m.

Diese Folge ist als Zufallszahlenfolge ebenfalls völlig unbrauchbar. Sie entspricht Beispiel (1), durchläuft die Zahlen aber in umgekehrter Reihenfolge.

(3) $f(z) = (z+b) \bmod m$ (für eine Zahl $0 < b < m$) und $z_0 = 0$.

Dann lautet die Folge $z_1, z_2, ...$:

(b mod m), (2b mod m), (3b mod m), (4b mod m), ...

Analyse dieser Folge:

Es sei k = ggT(b,m), dann gibt es teilerfremde Zahlen x und y mit $b = k \cdot x$ und $m = k \cdot y$. Wir betrachten die Bedingung für eine Periode: $(i+q) \cdot b \bmod m = i \cdot b \bmod m$.

Dies gilt für $q > 0$ nur, wenn $q \cdot b$ ein Vielfaches von m ist, also $q \cdot b = r \cdot m$ für ein $r > 0$, also $qk \cdot x = rk \cdot y$ und $q \cdot x = r \cdot y$. Die kleinste natürliche Zahl q, für die dies zutrifft, ist $q = (r \cdot y)/x$, woraus r = x folgt, da x und y teilerfremd sind. Folglich ist y das kleinste positive q, für das $(i+q) \cdot b \bmod m = i \cdot b \bmod m$ gilt, und daher ist y die Periode dieser Zahlenfolge.

Die Parameter sind also: v = 0 und p = m/ggT(b,m) = y. Die Beispiele (1) und (2) sind Spezialfälle hiervon.

Diese Folge ist nicht "zufällig", da zwei aufeinander folgende Zahlen (modulo m) eine konstante Differenz aufweisen.

(4) $f(z) = (a \cdot z) \bmod m$ (für eine Zahl $1 < a < m-1$) und $z_0 = 1$ (sog. "multiplikativer Kongruenzgenerator").

Dann lautet die Folge $z_1, z_2, \ldots$:

$(a \bmod m), (a^2 \bmod m), (a^3 \bmod m), (a^4 \bmod m), \ldots$

Die Analyse dieser Folge ist bereits schwierig; denn es stellt sich heraus, dass bei geschickter Wahl von a und m Zahlenfolgen entstehen, die sich wie Zufallszahlen verhalten. Meist entstehen aber keine solchen Folgen.
Betrachte z. B. die Zahlenfolgen $\{z_i\}_{i>0}$ für m = 15, $z_0 = 1$ und alle a; die Periode beträgt hier für jede Wahl von a höchstens 4, der Vorlauf ist 0:

a = 2: 2 4 8 1 2 4 8 1 2 4 8 1 2 4 8 1 ...
a = 3: 3 9 12 6 3 9 12 6 3 9 12 6 3 9 12 6 ...
a = 4: 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 ...
a = 5: 5 10 5 10 5 10 5 10 5 10 5 10 5 10 5 10 ...
a = 6: 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 ...
a = 7: 7 4 13 1 7 4 13 1 7 4 13 1 7 4 13 1 ...
a = 8: 8 4 2 1 8 4 2 1 8 4 2 1 8 4 2 1 ...
a = 9: 9 6 9 6 9 6 9 6 9 6 9 6 9 6 9 6 ...
a = 10: 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 ...
a = 11: 11 1 11 1 11 1 11 1 11 1 11 1 11 1 11 1 ...
a = 12: 12 9 3 6 12 9 3 6 12 9 3 6 12 9 3 6 ...
a = 13: 13 4 7 1 13 4 7 1 13 4 7 1 13 4 7 1 ...

Als zweites Beispiel betrachten wir die Zahlenfolge für $m = 243 = 3^5$ und a = 29 mit dem Startwert $z_0 = 1$; die Periode beträgt 162, der Vorlauf ist 0:

29 112 89 151 5 145 74 202 26 25 239 127 38 130 125 223 149
190 164 139 143 16 221 91 209 229 80 133 212 73 173 157 179
88 122 136 56 166 197 124 194 37 101 13 134 241 185 19 65
184 233 196 95 82 191 193 8 232 167 226 236 40 188 106 158
208 200 211 44 61 68 28 83 220 62 97 140 172 128 67 242 214
131 154 92 238 98 169 41 217 218 4 116 205 113 118 20 94 53
79 104 100 227 22 152 34 14 163 110 31 170 70 86 64 155 121
107 187 77 46 119 49 206 142 230 109 2 58 224 178 59 10 47
148 161 52 50 235 11 76 17 7 203 55 137 85 35 43 32 199 182
175 215 160 23 181 146 103 71 115 176 1 29 112 89 151 5 145
74 202 26 25 239 127 38 130 125 223 149 190 164 139 143 16
221 91 209 229 80 133 212 73 173 157 179 88 122 136 56 166
197 124 194 37 101 13 134 241 185 19 65 184 233 196 95 82
191 193 8 232 167 226 236 40 188 106 158 208 200 211 44 61 68
28 83 220 62 97 140 172 128 67 242 214 131 ...

Dies sieht schon recht gut aus; die Zahlen scheinen über den Bereich von 1 bis 242 recht gleichmäßig verteilt zu sein?

Eine "Zufälligkeit" liegt jedoch nicht vor; denn wenn man diese Zahlen für das Würfeln verwenden will (bilde die Folge $(z_i \bmod 6 + 1)$), so erhält man:

Aus obiger Zahlenfolge abgeleitete Zahlenfolge "modulo 6 plus 1":

$6 = 29 \bmod 6 + 1$, $5 = 112 \bmod 6 + 1$, $6 = 89 \bmod 6 + 1$, ...

(für m = 243, a = 29 und $z_0 = 1$):

6 5 6 2 6 2 3 5 3 2 6 2 3 5 6 2 6 5 3 2 6 5 6 2 6 2 3 2 3 2 6 2 6 5 3 5 3 5 6 5 3 2
6 2 3 2 6 2 6 5 6 5 6 5 6 2 3 5 6 5 3 5 3 5 3 5 3 2 3 2 3 5 6 5 3 2 3 5 3 2 3 5 6 5
3 5 3 2 6 2 3 5 3 2 6 5 3 5 6 2 3 5 6 5 3 5 3 2 3 2 3 5 3 5 6 2 6 2 6 5 6 2 3 5 3 2
3 5 3 5 6 5 6 5 6 5 3 2 6 5 6 2 6 2 6 2 6 2 3 2 3 2 6 5 6 2 3 2 6 2 3 2 6 5 6 2 6 2
3 5 3 2 6 2 3 5 6 2 6 5 3 2 6 5 6 2 6 2 3 2 3 2 6 2 6 5 3 5 3 5 6 5 3 2 6 2 3 2 6 2
6 5 6 5 6 5 6 2 3 5 6 5 3 5 3 5 3 5 3 2 3 2 3 5 6 5 3 2 3 5 3 2 3 5 6 5 3 5 3

Die Häufigkeit der Augenzahl des Würfels ist hier keineswegs zufällig verteilt, sondern: Die 1 und die 4 kommen gar nicht vor, kein einziges Mal wird eine Zahl zweimal hintereinander gewürfelt und niemals folgt eine 2 auf eine 5. Die Folge mit m = 243 und a = 29 ist daher als Zufallszahlenfolge völlig ungeeignet (prüfen Sie nach, dass dies sogar unabhängig vom Startwert z_0 gilt).

Man sieht: Es wird nicht leicht sein, a und m "geschickt" zu wählen.

Die zu $f(z) = (a \cdot z) \bmod m$ (für eine Zahl $1 < a < m-1$, $z_0 = 1$) gehörende Folge $(a \bmod m), (a^2 \bmod m), (a^3 \bmod m), \ldots$ kann höchstens die Periode (m-1) besitzen; denn wenn einmal die Zahl 0 auftritt, so wäre ab dann jede weitere Zahl Null.

Unter welchen Bedingungen tritt die maximal mögliche Periode (m-1) auf? Es sei ggT(a,m) = k, also $a = k \cdot x$ und $m = k \cdot y$ für geeignete Zahlen x und y. Jede Zahl a^i ist ebenfalls durch die Zahl k teilbar und daher enthält die Zahlenfolge ausschließlich durch k teilbare Zahlen; dies sind aber nur y verschiedene Zahlen, d.h., die Periode kann dann höchstens y sein. Für eine Periode m-1 ist also notwendig, dass ggT(a,m) = 1 gilt.

Das obige Beispiel mit m=243 und a=29 besitzt die Periode 162, d.h., die Bedingung ggT(a,m) = 1 ist nicht hinreichend.

(Untersuchen Sie dieses Problem selbst weiter.)

(5) $f(z) = (a \cdot z + b) \bmod m$ ("linearer Kongruenzgenerator")
für Zahlen $1 \le a < m$ und $0 \le b < m$ und $z_0 = 1$.
Dann lautet die Folge $z_1, z_2, ...$:

$(a+b) \bmod m$, $(a^2+ab+b) \bmod m$, $(a^3+a^2b+ab+b) \bmod m$, ...

Nun ist auch m als Periode möglich, z.B.: für $m = 3^5 = 243$,
$a = 28$, $b = 2$, $z_0 = 1$ ist die Periode 243 = m:

30 113 7 198 200 13 123 44 19 48 131 25 216 218 31 141 62 37 66
149 43 234 236 49 159 80 55 84 167 61 9 11 67 177 98 73 102 185
79 27 29 85 195 116 91 120 203 97 45 47 103 213 134 109 138 221
115 63 65 121 231 152 127 156 239 133 81 83 139 6 170 145 174
14 151 99 101 157 24 188 163 192 32 169 117 119 175 42 206 181
210 50 187 135 137 193 60 224 199 228 68 205 153 155 211 78 242
217 3 86 223 171 173 229 96 17 235 21 104 241 189 191 4 114 35
10 39 122 16 207 209 22 132 53 28 57 140 34 225 227 40 150 71
46 75 158 52 0 2 58 168 89 64 93 176 70 18 20 76 186 107 82 111
194 88 36 38 94 204 125 100 129 212 106 54 56 112 222 143 118
147 230 124 72 74 130 240 161 136 165 5 142 90 92 148 15 179
154 183 23 160 108 110 166 33 197 172 201 41 178 126 128 184 51
215 190 219 59 196 144 146 202 69 233 208 237 77 214 162 164
220 87 8 226 12 95 232 180 182 238 105 26 1 30 113 ...

Die Analyse der Funktion $f(z) = (a \cdot z + b) \bmod m$ hat D. E. Knuth im Band 2 seines Buches "The Art of Computer Programming" zusammengefasst. Dort beweist er folgende notwendige und hinreichende Bedingung für die Periode m (der Vorlauf ist dann immer 0):

14.2.4 Satz
Gegeben sei eine lineare Kongruenz $f(z) = (a \cdot z + b) \bmod m$.
Die zugehörige Zahlenfolge besitzt die Periode m dann und nur dann, wenn gilt:
(1) Jeder Primteiler von m teilt a-1,
(2) ggT(b,m) = 1,
(3) ist m durch 4 teilbar, dann ist auch a-1 durch 4 teilbar.

Hinweis:
Eine Zahl p heißt Primteiler einer Zahl n, wenn p eine Primzahl ist und p zugleich ein Teiler von n ist. Zum Beispiel sind 3 und 5 die Primteiler von 75 und 2, 3 und 5 die Primteiler von 360.

Aus dem Satz folgt, dass die Periode m nicht vorliegen kann, wenn m eine Primzahl ist; denn dann wäre m ein Primteiler von m und müsste die Zahl a-1 < m teilen. Auch der Fall b = 0 kann nicht zur Periode m führen, da ggT(0,m) = m ist. Auf z_0 kommt es bei der Periode m nicht mehr an, da dann alle Zahlen von 0 bis m-1 einen großen Zyklus bzgl. f bilden und es gleichgültig ist, mit welcher Zahl man beginnt.

Beispiele für Zahlenkombinationen, die den Satz erfüllen:
m = 9, a = 7, b = 4 (Folge: 2 0 4 5 3 7 8 6 1 2 0 4 5 3 7 ...)
m = 243, a = 28, b = 2 (Folge: siehe oben)
m = 1024, a = 33, b = 5 (Folge: 38 235 592 85 762 575 548 681
974 403 1016 765 674 743 972 337 886 571 416 421 586 911
372 1017 798 739 840 77 498 55 796 673 710 907 240 ...)
$m = 2 \cdot 3^{10} = 39366$, $a = 6^4 + 1 = 1297$, $b = 7^3 = 343$ (Folge: 1640
1659 26302 23081 18240 38023 29942 20241 35164 22223
7662 17725 39290 19869 25072 2411 17496 17839 29684 ...)

Doch eine lange Periode garantiert noch keine Ähnlichkeit mit Zufallszahlen. Man betrachte nochmals m = 243, a = 28, b = 2 und gehe zur Würfel-Folge ($z_i \bmod 6 + 1$) über:

1 6 2 1 3 2 4 3 2 1 6 2 1 3 2 4 3 2 1 6 2 1 3 2 4 3 2 1 6 2 4 6 2 4 3 2 1 6 2 4 6
2 4 3 2 1 6 2 4 6 2 4 3 2 1 6 2 4 6 2 1 3 2 1 3 2 4 6 2 1 3 2 1
3 2 4 6 2 1 3 2 1 3 2 4 6 2 1 3 2 1 3 2 4 6 2 1 3 2 4 3 2 4 6 2 1 6 2 4 3 2 4 6 5
1 6 5 4 3 5 4 6 5 1 6 5 4 3 5 4 6 5 1 6 5 4 3 5 1 3 5 1 6 5 4 3 5 1 3 5 1 6 5 4 3
5 1 3 5 1 6 5 4 3 5 1 3 5 1 6 5 4 3 5 1 3 5 1 6 5 4 6 5 1 3 5 4 6 5 4 6 5 1 3 5 4
6 5 4 6 5 1 3 5 4 6 5 4 6 5 1 3 5 4 6 5 4 6 5 1 3 5 4 3 5 1 6 5 1 3 5 4 3 2 1 6 ...

Diese Folge ist keineswegs zufällig, denn in ihr folgt auf eine 1 immer nur eine 3 oder 6, auf eine 2 immer nur eine 1 oder 4 usw., in der ersten Hälfte kommt die 5 nicht vor, in der zweiten Hälfte kommt die 2 nicht vor, die Teilfolge 4 3 2 ist anfangs viel zu häufig und später zu selten usw.

Noch augenfälliger ist die Würfel-Folge, die aus der Folge mit m = 39366, a = 1297, b = 343 entsteht: Prüfen Sie dies nach!
Also: Die Periode m garantiert zwar, dass jede Zahl unter den ersten m Zahlen genau einmal vorkommt, aber man braucht diverse Zusatzkriterien für "möglichst zufällige" Folgen.

(6) Es sei die Anfangs-Folge der z_i für i = 1, 2, ..., k gegeben. Wähle zwei feste Zahlen i und j mit $1 \le i < j \le k$ (oft verlangt man i = 1). Setze dann für alle n > k:

$$z_n = (z_{n-i} + z_{n-j}) \bmod m$$ ("Fibonacci-Generator").

(7) Kombination von Zahlenfolgen: Es seien $z_1, z_2, z_3, ...$ und $u_1, u_2, u_3, ...$ zwei Zahlenfolgen (z.B. jede durch eine lineare Kongruenz erzeugt), dann bilde die kombinierte Folge

$$v_{i+1} = (z_i + u_i) \bmod m$$

Hierbei sind viele Variationen möglich, z.B.

$$v_{i+1} = (a_1 \cdot z_i + a_2 \cdot u_{i-1}) \bmod m$$

für zwei Konstanten a_1 und a_2. Das sog. "Wichmann-Hill"-Verfahren benutzt vier lineare Kongruenzgeneratoren, um hieraus eine Zahlenfolge mit einer sehr großen Periode zu erzeugen.

14.2.5 Selbstdefinierte Zahlenfolgen ?

Manche definieren sich ihren Zufallszahlengenerator selbst, indem sie undurchschaubare Operationen aneinander reihen. Zum Beispiel: Wähle m, a, y. Führe dann fünfmal die Operationen

$$a := (a^2+y+43) \bmod m \quad \text{und} \quad y := a \cdot y \bmod (m/3)$$

durch. Die Folge der Werte a sei unsere Zahlenfolge. Solch eine Folge der a oder der y wird doch hoffentlich ziemlich zufällig sein?!

Wir testen dies. Als Startwerte wählen wir willkürlich m = 30007, a = 13123 und y = 78. Ein Programm in Ada, welches die zugehörige Folge der (a,y) erzeugt, ist rasch geschrieben. Ausgehend von (13123,78) erhält man:

(16164, 1068)	(12224, 5172)	(15690, 4086)	(25668, 5766)
(23905, 2076)	(21971, 2514)	(27318, 3294)	(23677, 7320)
(24736, 6144)	(4603, 4734)	(6795, 6654)	(11114, 7698)
(13861, 1320)	(477, 2694)	(7012, 8592)	(7012, 8592) ...

Aber: Nach 203 Iterationen erhält y den Wert 0 und nach weiteren 3 Iterationen beginnt dann eine kurze Periode der Länge 6. Obige Vorschrift liefert also eine Folge mit Vorlauf 206 und Periode 6.

Wir geben einige Ergebnisse für veränderliches y bei festem Modul m = 30007 und festem Startwert a = 13123 an:

Startwert	y =	78,	Vorlauf:	206,	Periode:	6
Startwert	y =	79,	Vorlauf:	102,	Periode:	1016
Startwert	y =	80,	Vorlauf:	576,	Periode:	34
Startwert	y =	81,	Vorlauf:	121,	Periode:	6
Startwert	y =	82,	Vorlauf:	205,	Periode:	34
Startwert	y =	83,	Vorlauf:	96,	Periode:	34
Startwert	y =	84,	Vorlauf:	475,	Periode:	1016
Startwert	y =	85,	Vorlauf:	9,	Periode:	353
Startwert	y =	86,	Vorlauf:	623,	Periode:	6
Startwert	y =	87,	Vorlauf:	224,	Periode:	34
Startwert	y =	88,	Vorlauf:	243,	Periode:	6
Startwert	y =	89,	Vorlauf:	24,	Periode:	6
Startwert	y =	1218,	Vorlauf:	0,	Periode:	1016
Startwert	y =	2358,	Vorlauf:	795,	Periode:	34
Startwert	y =	5912,	Vorlauf:	798,	Periode:	34

Variiere den Anfangswert von y von 1 bis m. Dabei findet man nur die hier auftretenden 4 verschiedenen Perioden, aber einige hundert unterschiedlich lange Vorläufe von 0 bis 798.

Man erkennt: Diese willkürliche Wahl eines selbstdefinierten Zufallszahlengenerators war sehr schlecht.

14.2.6 Merke:

Eine Folge, die auf Operationen beruht, die man selbst nicht versteht, ist fast niemals eine zufällige Folge!

Begründung: Um dies plausibel zu machen, untersuchen wir die Frage: Wie groß ist die Wahrscheinlichkeit, dass eine zufällig gewählte Funktion $f: \{0, 1, ..., m\text{-}1\} \to \{0, 1, ..., m\text{-}1\}$ mindestens einen Fixpunkt besitzt?

Ein Fixpunkt von f ist eine Zahl s mit f(s) = s. Eine Funktion f für Zufallsfolgen darf keine Fixpunkte besitzen, denn wenn man in einen solchen Fixpunkt gerät, so besteht die Zahlenfolge ab dann nur noch aus diesem einen Wert s.

Eine Funktion f ist bekanntlich eine Tabelle (ein "array"), z.B.:

x	0	1	2	3	4	...
f(x)	5	2	0	3	7	...

Für solche Tabellen gilt:

Mit Wahrscheinlichkeit (m-1)/m ist f(x) ≠ x für jedes x.

Also ist die Wahrscheinlichkeit, dass eine zufällig ausgewählte Funktion f *keinen* Fixpunkt besitzt:

$((m-1)/m)^m = (1-1/m)^m \approx e^{-1} \approx (2{,}718281828459...)^{-1} \approx 0{,}36788.$

Folglich ist die Wahrscheinlichkeit, dass f mindestens einen Fixpunkt besitzt: $1 - (1-1/m)^m \approx 0{,}63212$, also rund 63 %.

Mit hoher Wahrscheinlichkeit besitzt eine Funktion auch "Zyklen der Länge der 2", d.h., es gibt Zahlen s und t mit f(s) = t und f(t) = s, was ebenfalls für Zufallszahlen nicht akzeptabel. Ebenso: Zyklen der Länge 3, der Länge 4 usw.

Fazit: Die Wahrscheinlichkeit, dass eine beliebig ausgewählte Funktion als Basis für eine Folge von Zufallszahlen dienen kann, ist extrem klein.

Genau dieser Fall ist bei unserem willkürlich gewählten Beispiel oben eingetreten. Bereits die Periode (sie entspricht der "Zyklenlänge") ist für alle y viel zu klein; dies ändert sich auch nicht wesentlich, wenn man den Startwert für a verändert.

Untersuchen Sie einige selbst erfundene Beispiele und machen Sie sich klar, dass es einer genauen aufwendigen Analyse bedarf, bevor man eine Zahlenfolge als zufällig einstufen kann.

Wählt man einen linearen Kongruenzgenerator zur Erzeugung einer Folge von (Pseudo-) Zufallszahlen, so ist es sicher sinnvoll, einen Generator mit maximaler Periode m zu verwenden. Schon hiervon gibt es nur noch wenige, doch auch die meisten dieser Generatoren erzeugen Zahlenfolgen, die keine Ähnlichkeit mit zufälligen Folgen haben. Wie kann man die besten Generatoren auswählen? *"Ganz einfach":* Wir werden verlangen, dass die erzeugten Zahlenfolgen möglichst viele Gesetzmäßigkeiten der Wahrscheinlichkeitsrechnung erfüllen. Hierfür müssen sie gewissen Kriterien genügen.

14.2.7 Einige Kriterien (K1) bis (K8) können wir hier nur auflisten, da die Kenntnisse aus noch zu besuchenden Vorlesungen über Wahrscheinlichkeitsrechnung bzw. Statistik nicht vorausgesetzt werden können.

(K1) Chi-Quadrat-Test (χ^2-Test, Abweichung der Häufigkeiten)

(K2) Grenzwertsätze (Normalverteilung usw.)

(K3) Lückentest für Intervalle (die wievielte nächste Zahl liegt wieder im untersuchten Intervall?)

(K4) Gleichverteilungstest nach Kolmogorov und Smirnov

(K5) Mehrdimensionaler Spektraltest

(K6) Autokorrelationstest (die Korrelation zweier Teilfolgen muss soll nahe bei 0 liegen)

(K7) Permutations- und Teilfolgentests, Pokertest

(K8) Coupon Collector Test (Wartezeit, bis alle Teilfolgen einer gewissen Länge vollständig aufgetreten sind)

Es gibt Testprogramme (sog. "Testbatterien"), die diese und weitere Kriterien bei einer Zahlenfolge prüfen. Besteht eine Zahlenfolge alle Tests und ist ihre Periode hinreichend groß (mindestens 2 Milliarden), so kann sie als Pseudo-Zufallszahlenfolge verwendet werden.

In Tests ergab sich: Für normale Anwendungen reichen lineare Kongruenzgeneratoren in der Regel aus. Wo eine gute Zufälligkeit aber von hoher Wichtigkeit ist, sollte man auf den Mersenne Twister oder auf kombinierte Verfahren zurückgreifen.

Die Generatoren für Zufallszahlen werden nicht immer veröffentlicht. Dann gilt der Grundsatz: Misstrauen Sie dem vordefinierten Generator, testen Sie ihn gründlich, bevor Sie ihn für wichtige Experimente verwenden, oder ersetzen Sie ihn durch einen eigenen gut getesteten Generator.

14.2.8 Welche linearen Kongruenzgeneratoren verwendet man in Programmiersprachen? Beliebt ist eine Zweierpotenz für m, da man hierfür schnelle Hardware-Operationen verwenden kann. Für 32-Bit-Rechner findet man in Wikipedia (Stand 17.11.2008) folgende Angaben:

Programmiersprache	m	a	b
Numerische Anwendungen	2^{32}	1664525	1013904223
Borland C/C++	2^{32}	22695477	1
ANSI C	2^{32}	1103515245	12345
Borland Delphi	2^{32}	134775813	1
Microsoft C/C++	2^{32}	214013	2531011
Apple (im Numerikbereich)	$2^{31} - 1$	$16807=7^5$	0
Frühere IBM-Rechner	$2^{31} - 1$	65539	0

Benutzt man zwei 32-Bit-Register für die Darstellung von Zahlen, so gilt

$m = 2^{59}$ $a = 5^{11}$ $b = 0$

als brauchbarer Generator. Ist die Periode nicht m oder m-1, dann spielt auch noch der Anfangswert z_0 eine wichtige Rolle. Im Computer-Algebra-System Maple verwendet man die als sehr gut getesteten Zahlen (die Periode ist m-1) m = 999 999 999 999 999 999 996 467, a = 671 354 420 908 421 773 035 669, b = 0.

14.2.9 Literaturhinweise zu Zufallszahlen:

Knuth, Donald E. (2002), The Art of Computer Programming, ("TAoCP") Band 2, 3. Auflage, Addison-Wesley, Reading, Mass.

Marsaglia Programme (Multicarry) und "Testbatterie Diehard" von 1996: http://www.stat.fsu.edu/pub/diehard/

TEST-RAND (seit 1981) von E.J.Dudewicz und T.G.Ralley

Weitere Stichwörter für Zufallszahlengeneratoren: Super-Duper, Knuth-TAoCP (Fibonacci-Folgen, Periode bei 2^{129}), Mersenne-Twister (Bitfolge, Periode rund 2^{19937}), Tausworthe-Generatoren (Bitfolgen durch Polynomberechnungen), Wichmann-Hill (Periode etwa 2^{80}, besteht aus vier linearen Kongruenzgeneratoren).

Zu Zufallszahlen und den Beweisen siehe auch Kapitel XI in: Kiyek, Schwarz, "Mathematik für Informatiker 2", Teubner-Verlag, 1999

Abschließende Bemerkungen zur Vorlesung

In weniger als 30 Jahren hat sich der Wirtschaftszweig "IT", der sich mit der Datenverarbeitung, den Prozessen, der Kommunikation und den zugehörigen Geräten befasst, von einem kleinen Teilgebiet der Bürosysteme zur größten Branche in der Europäischen Union (1995) entwickelt. Zugleich hat er hierbei unsere Arbeitswelt und unsere Lebensbedingungen grundlegend umgestaltet. Dieser bisher wohl einmalige Vorgang in der Wirtschaftsgeschichte ist der Öffentlichkeit kaum bewusst, auch wenn dies die Basis für den Übergang in die kommende Informationsgesellschaft ist. Grundlegend hierfür sind die Informatik und ihre (intelligenten) Anwendungen.

Die nun zu Ende gehende Vorlesung vermittelte langlebige Inhalte, Methoden und Denkweisen des Faches Informatik. Doch die Informatik befindet sich noch in einer stürmischen Entwicklung, was sich an den vielen Diskussionen über "den besten Weg" und "den richtigen Einstieg" in der Ausbildung zeigt. Immer aufs Neue entdecken die einzelnen Arbeitsbereiche der Informatik neue Ansätze, die sie als fundamental einstufen.

Wissenschaften haben oft ein sie kennzeichnendes Attribut. Man sagt scherzhaft, die Mathematik sei "rein", die Philosophie "transzendent", die Chemie "stinkt und knallt" usw. Für die Informatik kann man heute ironisch behaupten, sie sei "orientiert", denn in den Diskussionen geben sich die diversen Informatikverfechter problem-, sprach-, hardware-, algorithmen-, logik-, funktions-, anwendungs-, objekt-, system-, KI-, technik-, netz-, dienst-, windows-, kommunikations-, agenten-, architektur- usw. -orientiert.

In der Vorlesung wurden diverse Orientierungen angesprochen, es dominierte letztlich aber die Algorithmen-Orientierung. Nach meiner Auffassung finden sich hier Ideen und Vorgehensweisen, die typisch für die Informatik sind und die von anderen Wissenschaften kaum behandelt werden. Dazu gehören auch die "fundamentalen Ideen", wie sie in der Didaktik herausgearbeitet werden und die sich durch "breite Anwendbarkeit", "Vermittlung auf unterschiedlichen Niveaustufen", "Dauerhaftigkeit", "Alltagstauglichkeit" und "Nützlichkeit bei der Lösung schwerer wissenschaftlicher Probleme" auszeichnen.

Mein Wunsch war ursprünglich, eine Vorlesung ausschließlich für Informatikstudierende zu konzipieren. Eine solche Vorlesung hätte dann kompakter und vertiefter ausgesehen. Die Universitätsgremien haben jedoch beschlossen, dass auch Studierende der Computerlinguistik, der Mechatronik, der Wirtschaftsinformatik sowie Nebenfachstudierende diverser Fachrichtungen die Vorlesung besuchen müssen, weshalb die Inhalte in diesem Jahr oft breit dargestellt und zu selten vertieft wurden. Die Zusammensetzung zu Beginn des Studienjahrs 2008/2009 war: Gesamtzahl: 359, davon Informatik 136 (38%), Softwaretechnik 89 (25%), Mechatronik 43 (12%), Wirtschaftsinformatik 34 (9%), Computerlinguistik 23 (6,5%), Mathematik 20 (5,5%), sowie 14 "sonstige" (Lehramt Informatik, Technikpädagogik, B.A.-Nebenfach). Die Zustimmung, die in den Evaluationen der Vorlesung ausgelotet wurde, fiel überwiegend positiv aus und ungefähr in der obigen Reihenfolge, d.h., die meiste Zustimmung kam von den Informatikstudierenden, was vermutlich an der "Orientierung" lag. Die Erfahrung der letzten 7 Jahre zeigt aber eindeutig, dass die Informatik für ihre eigenen Studierenden umgehend eine eigene Vorlesung anbieten muss und getrennt hiervon Dienstleistungsveranstaltungen für andere Studiengänge, möglichst im Rahmen der Umstellung zum Bachelor.

Im Bachelorstudium könnte ab 1.10.09 alles anders werden, aber nicht im ersten Studienjahr, in dem die Moduln "Informatik I" und "Inf. II" durch "Programmierung und Softwaretechnik" und "Algorithmen und Datenstrukturen" ersetzt werden, wobei zugleich der Programmierkurs halbiert wird. Eine wichtige Veränderung wird der konsequente Wechsel des "Einstiegs" sein: Startend mit einem funktionalen Ansatz soll danach die Objektorientierung frühzeitig gestärkt werden. Dies entspricht dem bundesdeutschen Trend und fördert die Ingenieurkomponente der Informatik. Dieser Ansatz bedeutet, dass die Kapitel 1 bis 4 der vorliegenden Vorlesung neu gestaltet werden müssen, während die anderen Kapitel (angepasst an die Programmiersprache und das gewünschte Niveau) wiederverwendet werden können.

Universitäres Studium macht nur Sinn mit Theorie-Bildung, da hierdurch lange gültige Erkenntnisse gefunden und vermittelt werden. Dieses Herausarbeiten von langlebigem Wissen und beweisbar guten Strukturen muss und wird im Zentrum der Grundvorlesungen der Informatik stehen. Spätestens im Jahre 2015 sollte der Fachbereich seine bis dahin neu orientierte Lehre evaluieren und aktualisieren.

Anhang: Ein paar Literaturhinweise

Stand: 13.10.2008

Literaturangaben im Grundstudium sind immer subjektiv verfärbt. Es gibt weitere gute Lehrbücher, auf die ich aus welchen Gründen auch immer nicht direkt zurückgegriffen habe. Die folgende Auflistung besitzt daher keinen Anspruch auf Vollständigkeit und enthält keine Wertungen.

Manuskripte: V.Claus (WS 03/04 bis SS 2006), K.Lagally (WS 01/02 und 04/05), E.Plödereder (SS 01).
Appelrath, Hans-Jürgen und Ludewig, Jochen, "Skriptum Informatik - eine konventionelle Einführung", Verlag der Fachvereine Zürich und B.G. Teubner Stuttgart, 4. Auflage 1999
Balzert, Helmut, "Lehrbuch Grundlagen der Informatik", Elsevier, 2. Auflage, 2004
Broy, Manfred, „Informatik. Eine grundlegende Einführung“. Band 1: Programmierung und Rechnerstrukturen, Springer, 1998. Band 2: Systemstrukturen und Theoret. Informatik, Springer-Verlag, 1998
Cormen, Leiserson, Rivest, "Introduction to Algorithms", MIT Press, 1996 (zweite Auflage 2001), auch auf Deutsch erhältlich.
Goos, Gerhard, "Vorlesungen über Informatik", Band 1 und 2, dritte Auflage, Springer-Verlag, Berlin 2000 und 2001 (vierte Auflage, zusammen mit Wolf Zimmermann, 2005/2006)
Gumm, H.-P., Sommer, M., „Einführung in die Informatik“, 7. Auflage, Oldenbourg-Verlag, München 2006
Güting, R.H., "Datenstrukturen und Algorithmen", B.G.Teubner, 3. Auflage, Wiesbaden 2004
Hotz, Günter, "Einführung in die Informatik", Teubner-Verlag, Stuttgart, 1992
Klaeren, H., Sperber, M., "Die Macht der Abstraktion: Einführung in die Programmierung", B.G. Teubner, Wiesbaden, 2007
Knuth, D.E., "The Art of Computer Programming", 4 Bände, Addison-Wesley (Standardwerk seit 1968)
Levi, P., Rembold, U., "Einführung in die Informatik", Hanser-Verlag, 2003
Loeckx, J., Mehlhorn, K., Wilhelm, R., "Grundlagen der Programmiersprachen", Teubner, Stuttgart 1986
Ottmann, T., Widmayer, P., "Algorithmen und Datenstrukturen", Spektrum Verlag, Heidelberg, 4. Auflage 2002.
Saake, G., Sattler, K.-U., "Algorithmen und Datenstrukturen", d.punkt-Verlag, Heidelberg, 2004
Schöning, Uwe, "Algorithmik", Spektrum Akademischer Verlag, Heidelberg 2001
Sedgewick, Robert, "Algorithms in C", 3rd Edition, Addison-Wesley, 1998, sowie "Algorithms in Java", Addison-Wesley, 1998 und weitere Varianten ("Parts 1 to 5") ab 1998

Bücher zu Ada 95 bzw. Ada 2005:
Ein recht kompakt geschriebenes Buch stammt von Manfred Nagl; hier sind zugleich Programmierprinzipien und tiefer gehende Probleme angesprochen: Nagl., M., „Softwaretechnik mit Ada 95. Entwicklung großer Systeme.“, Vieweg-Verlag, Wiesbaden 1999
Andere Bücher (alle englisch-sprachig):
Barnes, J.G.P., „Programming in Ada 95“, 2. Auflage, Addison-Wesley 1998
Barnes, John, „Ada 95 Rationale. The Language - The Standard Libraries”, Springer-Verlag, Lecture Notes in Computer Science (Vol. 1247), 1997
Barnes, John, „Ada 2005 Rationale. The Language - The Standard Libraries”, Springer-Verlag, Lecture Notes in Computer Science (Vol. 5020), 2007 (auch im Netz verfügbar)
English, J., "Ada 95: The Craft of Object-Oriented Programming", http://www.it.bton.ac.uk/staff/je/adacraft/
Feldman, M.B. und Koffman, E.B., „Ada 95 - Problem Solving and Program Design“, 2. Auflage, Addison-Wesley 1997 (3. Auflage Textbook Binding, 1999)
Nagl, M., "Softwaretechnik mit Ada 95. Entwicklung großer Systeme.", Vieweg-Verlag, Wiesbaden, 1999
Riehle, R., „Ada Distilled – ...“, 2003, http://www.adapower.com/ (- Books & Tutorials)

Weitere Literatur:
Als Einstieg:
Appelrath, Boles, Claus, Wegener, "Starthilfe Informatik", Teubner-Verlag, Stuttgart-Leipzig, 2001
Für diverse Definitionen und Erläuterungen:
"Duden Informatik", dritte Auflage, Bibliografisches Institut, Mannheim, 2001; vierte Auflage 2007
Für Mathematik und Ideen:
Kiyek, K. und Schwarz, F., "Mathematik für Informatiker 1 und 2", Teubner-Verlag, 3. bzw. 2. Auflage, 2000 und 1999
Meinel, Christoph und Mundhenk, Martin, "Mathematische Grundlagen der Informatik", Teubner-Verlag, Wiesbaden, 2. Auflage, 2002
Schöning, Uwe, "Ideen der Informatik", Oldenbourg-Verlag, München, 2002