Einführung in die Informatik II

Sommersemester 2002, Volker Claus

0. Vorbemerkungen

0.1. Planung im Studium

Es gibt einen Plan der Vorlesung (siehe verteilte Unterlage).

Die Übungen sind durchorganisiert (ähnlich zum WS 01/02).

Die Prüfungen richten sich nach den jeweiligen Prüfungsordnungen. Leider darf ich Ihnen daher Leistungen aus den Übungsgruppen nicht anrechnen.

Wir werden zwei Tests (auch zur Selbstkontrolle) anbieten, die auf die Übungen angerechnet werden.

Planen Sie nun genau, wie Sie an der Vorlesung und an den Übungen teilnehmen wollen/können. Und planen Sie genau Ihre Arbeitszeit!

0.2. Ihre Arbeitsleitung

Wenn Sie die Veranstaltung erfolgreich bestehen wollen, dann müssen Sie mitarbeiten sowohl in der Vorlesung als auch in den Übungen. Zur Unterstützung bieten wir Ihnen zusätzliche Vortragsübungen an.

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

Ihr Aufwand für die "Einführung in die Informatik II" alles inklusive: Während der Vorlesungszeit (also vom 15.4. bis zum 20.7.02) sind pro Woche 12 bis 14 Zeitstunden aufzuwenden. Wichtig ist: Halten Sie durch!

0.3. Ziele des Studiums, Ziele der Vorlesung

Details siehe verteilte Unterlagen.

Allgemeine Hinweise für diese spezielle Veranstaltung:

- Wissen und Fähigkeiten mit Theorieunterbau (ca. 65%)
- Fertigkeiten, Handwerk, Routinewissen (ca. 30%)
- Fachübergreifendes, Persönlichkeitsentwicklung (ca. 3%)
- Entwicklung eines Beurteilungsvermögens (ca. 2%)

0.4. Voraussetzung für diese Vorlesung

Einführung in die Informatik I

Programmierkenntnisse (in Ada 95)

Mathematikkenntnisse aus dem WS 01/02

Konkretisierung von Ideen/Formalismen an Beispielen

Siehe auch: verteilte Unterlagen.

0.5. Ablauf und Ihre Mitwirkung

Vorlesung, Übungen, Vortragsübungen: siehe Unterlagen.

Wie können Sie sich stärker äußern?

- In den Übungen (leider *nicht* in der Vorlesung)
- Kontakte zu Tutoren und Mitarbeitern
- Elektronische Medien (Email, Forum)
- Wahl von Vorlesungssprecher(inne)n und Rückmeldungen über diesen Personenkreis

0.6. Zusammensetzung der Vorlesung

Fragebogen

0.7. Bücher, Skript, Folienkopien, Mitschrift

Jede(r) kaufe sich mindestens ein Buch! (Bücher sind ausgereift, enthalten kaum Fehler, geben Zusatzhinweise in die Geschichte, die Literatur, weitere Übungsaufgaben usw.)

Das Skript von Prof. Plödereder (SS 01) ist prinzipiell für den größten Teil des Stoffes hinreichend. Die Vorlesung greift auf diese Folien auch teilweise direkt zurück.

Zusätzlich hierzu können die Beamerpräsentationen und Folien 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 umgehend auf.

0.8. Unerwünschtes

An einer vergleichbaren Universität in den USA müssten Sie rund 2000 Euro allein für diese 6-SWS-Veranstaltung bezahlen und dieser Gegenwert soll allen Interessierten zugute kommen. Das bedeutet vor allem:

Keine Störungen während der Veranstaltung.

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

Wir wollen, dass Sie etwas lernen. Das erreicht man nicht durch Abschreiben. Wir wünschen daher **keine identischen oder sehr ähnlichen Abgaben** in den Übungen (hier: außer im Rahmen kleiner vorab festgelegter Gruppen), bei den Tests und schon gar nicht bei den Klausuren.

0.9. Fragen?

Einführung in die Informatik II

Sommersemester 2002

1. Algorithmen

Gliederung dieses Kapitels:

1.1. Charakteristika von Algorithmen

Vorbemerkung: Begriffe für Mengen M
"endlich", "unendlich", "abzählbar", "überabzählbar", "aufzählbar", "beschränkt".

M ist endlich ⇔ M ist leer oder es gibt eine natürliche Zahl n und eine Bijektion f: M → {1, ..., n}.

M ist unendlich ⇔ M ist nicht endlich.

M abzählbar (unendlich) ⇔ Es gibt eine Bijektion zu den natürlichen Zahlen.

M ist höchstens abzählbar ⇔
M ist endlich oder M ist abzählbar unendlich.

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 zuordnet, und zu jedem Element von M gibt es mindestens eine hierdurch zugeordnete natürliche Zahl.

Der Begriff "beschränkt" ist ein relativer Begriff; er bezieht sich auf die jeweils betrachtete Umgebung. Hierzu muss man *vorher* eine Schranke k festlegen. Bei Zahlenmengen lautet eine solche Definition dann beispielsweise:
M ist beschränkt ⇔
Es gibt eine vorab festgelegte natürliche Zahl k, so dass M nicht mehr als k Elemente besitzt.

Beispiel 1.1:
Die leere Menge ∅ ist endlich. Sie besitzt 0 Elemente.

Für jede natürliche Zahl n ist die Menge {1, 2, ..., n} endlich.

Die Menge der natürlichen Zahlen ∠ = {1, 2, 3, ...} bzw. $\angle_0$= {0, 1, 2, 3, ...} und die Menge der ganzen Zahlen 9 = { ..., -2, -1, 0, 1, 2, 3, ... } sind abzählbar unendlich und selbstverständlich aufzählbar.

Die Menge der rationalen Zahlen Θ = {n/m | n ganze Zahl, m positive ganze Zahl, n und m teilerfremd} ist abzählbar unendlich und zugleich aufzählbar (wie zeigt man dies?).

Die Menge der reellen Zahlen 3 und die Menge der komplexen Zahlen ∀ sind überabzählbar (Beweis über das Cantorsche Diagonalverfahren, vgl. Mathematikvorlesungen).

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 (z.B. nur aus Binärziffern). Genauer: Es gibt beschränkt viele Zeichen $\{a_1, ...,a_k\}$, so dass jedes Datum eine Folge dieser Zeichen ist.

b) Die Operationen sind "diskret" aufgebaut. Genauer: Es gibt beschränkt viele Zeichen $\{b_1, ...,b_m\}$, so dass jede Operation einschließlich ihrer Operanden hiermit beschrieben werden kann.

c) Die Vorschrift ist eine endliche Folge von Operationen. Die Vorschrift wird schrittweise abgearbeitet (diskrete Zeitskala).

d) Eine der Operationen ist als Startoperation ausgezeichnet.

e) Für jede Operation ist unmittelbar nach ihrer Ausführung bekannt, welches die Folgeoperation ist oder ob der Algorithmus abbricht (terminiert).

f) Die Eingabe für die Vorschrift ist eine (eventuell unendliche oder auch leere) Folge von Daten.

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: Der 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.

Hinweis: Obige Ausführungen sind keine richtige Definition, sondern nur eine Liste von umgangssprachlich formulierten Anforderungen.

Turingmaschinen erfüllen diese Anforderungen,

aber auch andere "Rechenmaschinen" oder Grammatiken.

Auch Programme beschreiben Algorithmen. Prüfen Sie die Anforderungen an der Ihnen geläufigen Programmiersprache und ihren Programmen nach.

Man beachte, dass Algorithmen beliebige Algorithmen als Eingabe besitzen können.

Zusatz: Wünschenswerte Anforderungen an Algorithmen:

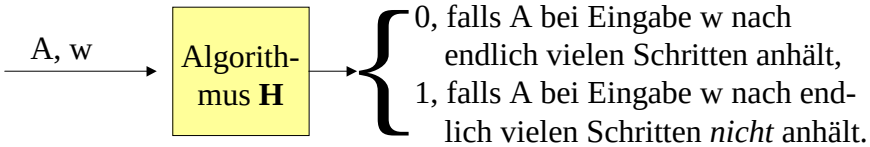
- Nichtdeterminismus: Oft weiß man nicht, welches die nachfolgende Operation sein soll, und möchte eine Menge möglicher Folgeoperationen angeben. Solange diese Menge bei jeder Operation endlich ist, lässt sich das Problem durch einen normalen Algorithmus simulieren (siehe später: BFS-Verfahren mit BFS = Breadth-First-Search). Im Falle unendlicher Mengen liegt jedoch kein Algorithmus mehr vor.
- Terminierung = ein Algorithmus muss für alle Eingaben nach endlich vielen Schritten anhalten.
  Diese Bedingung ist für Algorithmen *nicht notwendig* und wird auch nicht für alle Algorithmen gewünscht. Z.B. sollte ein Betriebssystem möglichst unendlich lange arbeiten ...

1.2. Grenzen der Algorithmen

Was kann man mit Algorithmen ***nicht*** beschreiben?

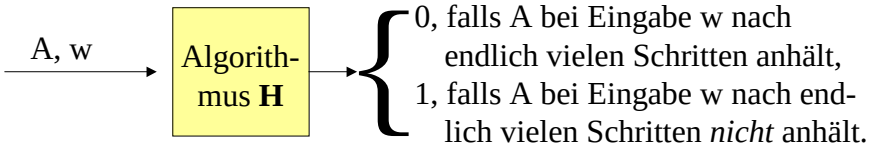
Betrachte folgende Aufgabe:
Konstruiere einen Algorithmus, der beliebige Algorithmen und Daten einlesen kann und der zu jedem beliebigen Algorithmus A und zu jeder Folge von Daten w in endlich vielen Schritten feststellt, ob der Algorithmus A mit den Eingabedaten w nach endlich vielen Schritten anhält.

Gesucht wird also ein Algorithmus H

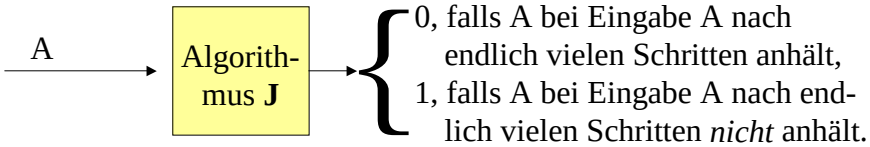


für alle Algorithmen A und für alle Eingabefolgen w.

Angenommen, es gibt solch einen Algorithmus H mit ∀A ∀w

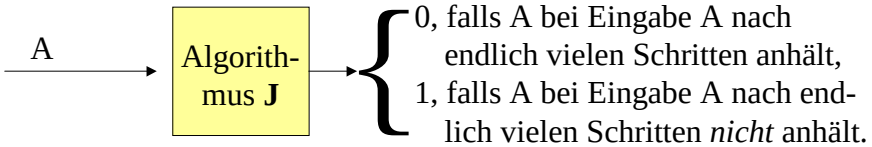


Speziell gibt es nun auch einen Algorithmus J mit ∀A

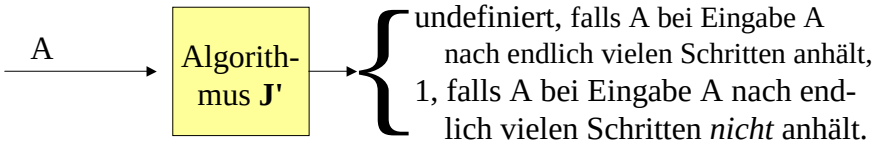


für alle Algorithmen A, indem man in H nur den Algorithmus A eingibt und dann H mit der Eingabe A und A (=w) laufen lässt.

Zu diesem Algorithmus J mit ∀A

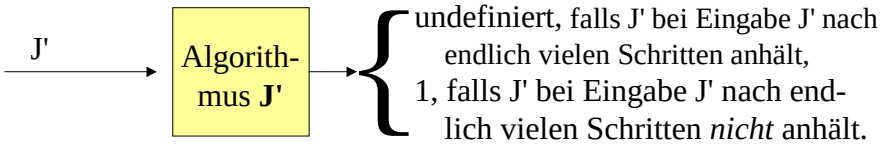


gibt es dann auch einen Algorithmus J' mit ∀A



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.
Widerspruch!

Beide möglichen Fälle führen also auf einen Widerspruch. Folglich muss die Annahme, dass es den Algorithmus H gibt, falsch gewesen sein. Es gilt daher der

Satz 1.1:

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 mit den Eingabedaten w nach endlich vielen Schritten anhält.

Man nennt die Suche nach einem solchen Algorithmus H auch das Halteproblem und sagt kurz:

Das Halteproblem ist algorithmisch nicht lösbar.

Historischer Hinweis:
Dieses Resultat kennt man seit 1931. In der hier vorgestellten Form wurde es 1936 unabhängig voneinander von Church, Kleene und Turing bewiesen.

Aus diesem Resultat lässt sich eine Fülle von weiteren Problemen ableiten, die alle algorithmisch nicht lösbar sind, z.B.
- entscheide, ob Programme für *alle* Eingaben nicht anhalten,
- entscheide, ob zwei beliebige Programme dasselbe berechnen,
- entscheide, ob eine kontextfreie Grammatik alle Wörter erzeugt

usw.
Weitere Aussagen hierzu in der Vorlesung "Theorie I" (Satz von Rice) und in Büchern.

1.3. Korrektheit von Algorithmen

Wie lässt sich beweisen, dass ein Algorithmus (= ein Programm) genau das realisiert, was es realisieren soll?

Dazu muss man zweierlei kennen:
- die Spezifikation, was zu realisieren ist,
- die Ermittlung dessen, was ein Programm realisiert.

Zu realisieren ist eine Zuordnung von Eingabedaten zu Ausgabedaten oder die Erfüllung von Anforderungen.

Ein Programm realisiert eine Abbildung von der Menge der Eingabedaten in die Menge der Ausgabedaten. (Dies kann implementierungsabhängig sein!)

Spezifikation:
$f: E \rightarrow A$

oder

Spezifikation:
Anfangs-Formel ....
Am Ende Formel....

```
x, z: Integer;
   Get(x);
  z:= x + x;
   Put(z);
```

Realisierung
$f'': 9 \rightarrow 9$ mit
$f''(n) = 2n$ für alle $n \in 9$

Zu beweisen: f = f'' oder f'' erfüllt die Formeln ....

Beispiel 1.2:
Spezifikation: Die Funktion $f: \mathbb{9} \rightarrow \mathbb{9}$ mit $f(n) = 2n$ für alle $n \in \mathbb{9}$ soll realisiert werden.

Folgendes Programmstück wird vorgeschlagen:

```
x: Integer;
Get(x);
for i in 1..x loop x := x+1; end loop;
Put(x);
```

Dieses Programm liest eine ganze Zahl a in die Variable x ein, addiert a mal eine 1 zu ihr und druckt sie dann aus. Für $a \geq 0$ ist dieses Vorgehen sicher richtig, doch für $a < 0$ wird nur der Eingabewert wieder ausgegeben. Also wird die Multiplikation mit 2 nicht vom Programm realisiert.

Beispiel 1.3:
Spezifikation: Die Funktion $f: \mathbb{9} \rightarrow \mathbb{9}$ mit $f(n) = 2n$ für alle $n \in \mathbb{9}$ soll realisiert werden.

Folgendes Programmstück wird vorgeschlagen:

```
x: Integer;
Get(x);
Put(x, Base=>2);
Put("0");
```

Dieses Programm liest eine ganze Zahl ein, druckt sie zur Basis 2 aus und fügt die Ziffer 0 am Ende an. Diese eventuell richtige Idee wird aber dadurch zunichte gemacht, dass Ada.Integer_Text_IO die Zahlen zu einer Basis, die nicht 10 ist, in "#" einschließt. Bei Eingabe von 13 wird also 2#1101#0 statt 11010 ausgegeben.

Beispiel 1.4:
Spezifikation: Die Funktion f: $9 \rightarrow 9$ mit f(n) = 2n für alle $n \in 9$ soll realisiert werden.

Folgendes Programmstück wird vorgeschlagen:

```
x: Integer;  Negativ: Boolean;
Get(x);
Negativ := x<0;
if Negativ then x:=-x; end if;
for i in 1..x loop x:=x+1; endloop;
if Negativ then x:=-x; end if;
Put(x);
```

In diesem Programm wird ein negativer Wert zunächst ins Positive umgewandelt, dann wird der Wert verdoppelt und im negativen Fall wieder negiert. Man vermutet, dieses Programm sei nun richtig?! Ist das schon ein Beweis?

Es gibt mehrere Methoden, die Bedeutung eines Programms (also die realisierte Abbildung) zu ermitteln. Wir wollen hier nur eine betrachten, nämlich die prädikatenlogische Methode; sie wird in der Literatur *"axiomatische Semantik"* genannt.

Man geht davon aus, dass zu jedem Zeitpunkt des Programmablaufs gewisse Bedingungen erfüllt sind; diese beschreibt man als prädikatenlogische Formeln; man nennt sie ***Zusicherungen*** oder ***assertions*** und schreibt sie in geschweifte Klammern.

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

$$\{A\}\ c\ \{B\}.$$

Wir betrachten zunächst ein triviales Beispiel, wobei wir die prädikatenlogischen Formeln umgangssprachlich formulieren:

```
x: Integer;
{Der Wert von x ist noch undefiniert}
Get(x);
{Der Wert von x ist eine ganze Zahl a}
x := x+2;
{Der Wert von x ist a+2}
Put(x);
{Der Wert a+2 wurde ausgegeben}
```


Als Modell für die Interpretation prädikatenlogischer Formeln müssen wir hier die ganzen Zahlen betrachten. Hinzu kommt der Fall "undefiniert", den wir durch das Zeichen $\perp$ darstellen.

Die Aussage "Der Wert von x ist noch undefiniert" beschreiben wir nun durch die Zusicherung

$$x = \perp$$

Die Eingabe Get(x) beschreiben wir durch die Zusicherung

$$a \in 9 \wedge x = a$$

Schließlich wird "der Wert von x ist a+2" dargestellt durch

$$x = a + 2$$

Damit erhalten wir folgendes Programm mit zusätzlichen Zusicherungen zwischen je zwei aufeinander folgenden Anweisungen:

```
x: Integer;
{x = ⊥}
Get(x);
{a ∈ 9 ∧ x = a}
x := x+2;
{x = a + 2}
Put(x);
```

Die Ausgabe lässt sich nicht beschreiben; man erhält sie, indem man den Wert aller Variablen, die ausgegeben werden, in der Anweisung davor betrachtet.

Die realisierte Abbildung dieses Programms lässt sich nun schrittweise an jeder Anweisung beweisen (bitte selbst durchführen).

Dies sieht so aus, als ob der Mensch als Beweiser benötigt wird. Dies ist aber nur teilweise richtig.

Denn bis auf die while-Schleife (und die Rekursion) kann man alle anderen Anweisungen weitgehend automatisch nachprüfen lassen.

Es gelten nämlich folgende sechs Regeln. Hierbei werden die Ein- und Ausgabe nicht berücksichtigt; sie müssen zusätzlich hinzugefügt werden.

Vorbemerkung: Jede Regel $\frac{X}{Y}$ bedeutet:

Wenn X zutrifft, dann trifft auch Y zu.

Die sechs Regeln von Hoare:

Für alle Zusicherungen A, B, C, für alle Wertzuweisungen `x:=ß` , für alle Booleschen Ausdrücke b und für alle Anweisungen `c`, `d`:

1 $\dfrac{\text{true}}{\{A\}\ \varepsilon\ \{A\}}$,wobei ε die leere Anweisung 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.

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 $\dfrac{\{A \wedge b\}\ \mathbf{c}\ \{B\}\ \wedge\ \{A \wedge \text{not}(b)\}\ \mathbf{d}\ \{B\}}{\{A\}\ \mathbf{if\ b\ then\ c\ else\ d\ end\ if}\ \{B\}}$

5 $\dfrac{\{A \wedge b\}\ \mathbf{c}\ \{A\}}{\{A\}\ \mathbf{while\ b\ loop\ c\ end\ loop}\ \{A \wedge \text{not}(b)\}}$ 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

Beispiel 1.5: Was macht folgendes Programmstück?

```
x, y, z: Natural;
Get (x, 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);
```

Realisiert dieses Programmstück die Multiplikation?
Füge Zusicherungen ein und führe einen Beweis.


```
x, y, z: Natural;
{x=⊥ ∧ y=⊥ ∧ z=⊥}
Get (x, y);
{a∈ 9 ∧ a≥0 ∧ x=a ∧ b∈ 9 ∧ b≥0 ∧ y=b ∧ z=⊥}
z:=0;
{a∈ 9 ∧ a≥0 ∧ x=a ∧ b∈ 9 ∧ 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)
```

Wir betrachten nun nur die while-Schleife.

```
{a∈ 9 ∧ a≥0 ∧ x=a ∧ b∈ 9 ∧ 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;
Put(z)
```

then-Zweig betrachten


```
{a∈ 9 ∧ a≥0 ∧ x=a ∧ b∈ 9 ∧ 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;
Put(z)
```

else-Zweig betrachten

Es gilt also

```
{x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
if (y mod 2 = 0) then
     y := y div 2;
     x := x + x;
{x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
else
     y:=y-1;
     z:=z+x;
{x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
end if;
```


Mit der Konsequenzregel folgt hieraus

```
{x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
if (y mod 2 = 0) then
     y := y div 2;
     x := x + x;
{x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
else
     y:=y-1;
     z:=z+x;
{x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
end if;
```

Also gilt nach der vierten Regel:

```
{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}
```

Wir setzen dies ein:


```
{a∈ 9 ∧ a≥0 ∧ x=a ∧ b∈ 9 ∧ 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
          y:=y-1;
          z:=z+x;
     end if;
     {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
end loop;
{x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b ∧ y≤0}  ⇒
     {y=0 ∧ z=a·b}
Put(z)
```

```
x, y, z: Natural;
Get (x, 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);
```

Also wird das Produkt zweier nicht-negativer Zahlen berechnet.

Beispiel 1.6: *Plateau-Problem*

Gegeben: n ≥ 1 und ein sortiertes Feld natürlicher Zahlen

a: array (1..n) of Natural;

Gesucht ist die maximale Anzahl gleicher Zahlen, also

Max {d≥1 | ∃ i mit a(i) = a(i+1) = ... = a(i+d-1)}.

Lösungsvorschlag:

" Lies n und das array a ein. Sortiere a. "

```
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;
```

" Ergebnis ist p. "

Der Beweis erfolgt in der Vorlesung an der Tafel. Hierzu definiere man das Prädikat m(j,p) als "p ist die maximale Anzahl gleicher Zahlen im Teil-array a(1..j)". Dann zeige man, dass m(j,p) eine Schleifeninvariante der while-Schleife ist. Der Rest ist einfach.

Vorgehensweise beim Nachweis der Korrektheit:

Schreibe zwischen je zwei Anweisungen eine prädikatenlogische Formel und versuche mit Hilfe der Hoareschen Regeln zu bewiesen, 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 hier nicht, weil es uns ja nur um das Prinzip geht und weil diese Regeln rasch recht kompliziert werden.)

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 (Sie ahnen es schon) auch dieses Problem, die Korrektheit eines Programms zu beweisen, ist algorithmisch nicht lösbar.

(Anschaulich ist diese Aussage klar: Denn wer die Korrektheit nachweisen will, muss auch beweisen können, ob ein Programm überhaupt anhält; und damit ist dieses Korrektheitsproblem mindestens so schwer wie das Halteproblem.)

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 könnte ein Beweis der Korrektheit ermöglicht und teilweise sogar automatisiert werden können.

Wie muss man hierbei vorgehen?
Der wichtigste Schritt ist:

Ausgehend von einer Zusicherung und der davor stehenden Anweisung muss man versuchen die Zusicherung zu konstruieren, die vor der letzten Anweisung einzufügen ist.

Also: Finde zu der Situation

c {B}

eine Zusicherung A, so dass

{A} **c** {B}

gilt.

(Das geht in vielen Fällen tatsächlich mittels folgender "wp".)

Triviales Beispiel: Gegeben sei

```
x := x +1;
```
{x > 7}

Dann wird eine Zusicherung vor der Wertzuweisung gesucht:

{x > 6}
```
x := x +1;
```
{x > 7}

Dies entspricht der Hoareschen Regel: {A'} **x:=ß** {A}, wobei A' aus A durch Ersetzen von x durch ß hervorgeht.

Man hätte auch die Zusicherung {x > 20} nehmen können:

{x > 20}
```
x := x +1;
```
{x > 7}

Welche Zusicherung soll man nehmen?

Man denke an die Konsequenzregel. Es gilt:

{x > 20} ⇒ {x > 6}
```
x := x +1;
```
{x > 7}

Die Zusicherung {x>6} ist "schwächer" als die Zusicherung {x>20}. "Schwächer" bedeutet: Die Zusicherung {x>6} trifft auf mehr Werte zu als die Zusicherung {x>20}.

{x>0} wäre eine noch schwächere Bedingung. Leider gilt aber die Formel

{x > 0}
```
x := x +1;
```
falsch
{x > 7}

nicht mehr, da x ja einen der Werte 1, 2, 3, 4, 5 oder 6 haben könnte. Gibt es zu **c** und B vielleicht eine "schwächste" Zusicherung A, für die {A} **c** {B} zutreffend ist? **Ja!**

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. Wenn es mehrere solche Vorbedingungen gibt, dann suche die "schwächste" Vorbedingung, d.h., suche ein A mit:

1. {A} **c** {B}
2. wenn für irgendeine Zusicherung A' gilt: {A'} **c** {B}, dann ist A eine Folgerung aus A', d.h., dann gilt A' $\Rightarrow$ A.

Definition:

Eine solche Zusicherung A heißt "schwächste Vorbedingung" zu **c** und B (englisch: *weakest precondition*, abgekürzt wp).

Man schreibt dann: **A = wp(c,B)**.

Als erstes müssen wir fragen, ob unsere Definition in sich stimmig ist (man sagt auch, ob sie "wohldefiniert" ist). Es könnte ja sein, dass es zu **c** und B in der Regel keine schwächste Vorbedingung gibt oder dass es mehrere gibt.

Ohne Beweis notieren wir hier:
Es gibt stets eine schwächste Vorbedingung zu **c** und B.

Dass diese stets eindeutig ist, ist leicht zu sehen: Wenn es zwei solche schwächsten Vorbedingungen A und A' gäbe, dann müssen gleichzeitig $A' \Rightarrow A$ und $A \Rightarrow A'$ gelten. Zwei solche Zusicherungen sind aber gleichwertig, d.h., es gilt dann $A' \Leftrightarrow A$.

Beispiel 1.7:
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 der Variablen X, für die nach Durchführung der Anweisung **c** die Zusicherung B gilt:
{a∈ • | falls a ungerade ist, dann muss a+3 durch 6 teilbar sein;
falls a gerade ist, dann muss a durch 6 teilbar sein}
= {a∈ • | a ist durch 3 teilbar}.

Eine Zusicherung, die genau für die Werte dieser Menge erfüllt ist, muss zur schwächsten Vorbedingung gehören (also zur größten Menge, die die Zusicherung erfüllt). Also gilt hier:

wp(**c**,B) ≡ X mod 3 = 0

Wenn **c** eine Wertzuweisung ist, dann ist wp(**c**,B) in der Regel leicht zu berechnen und dies kann auch automatisiert werden. Die Hintereinanderausführung und die Alternative sind auch noch gut zu handhaben. Als Beispiel betrachte man:

c ≡ if X<0 then X:=-X+1; else X:=X-1; end if;
B ≡ X > 5

Betrachte zunächst den then-Zweig:
$\mathbf{c_1}$ ≡ **X := -X + 1;**
B ≡ X > 5
wp($\mathbf{c_1}$, B) ≡ X < -4

Dies muss man noch koppeln mit der Bedingung b ≡ X < 0 für den then-Zweig. Dies ergibt die Zusicherung
(X < -4) ∧ (X < 0) , also X < -4.

c ≡ if X<0 then X:=-X+1; else X:=X-1; end if;
B ≡ X > 5

Betrachte nun den else-Zweig:
$\mathbf{c_2}$ ≡ **X := X - 1;**
B ≡ X > 5
wp($\mathbf{c_2}$, B) ≡ X > 6

Dies muss man koppeln mit der Bedingung not(b) ≡ X ≥ 0. So erhält man die Zusicherung
(X > 6) ∧ (X ≥ 0) , also X > 6 für den else-Zweig.

Aus beiden Vorbedingungen erhält man die schwächste Vorbedingung der gesamten Alternative als Disjunktion:
wp(**c**, B) ≡ (X < -4) ∨ (X > 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
wp(**c**, B) ≡ ?

"X mod 3 = 0" ist 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, sie steht auch auf der nächsten Folie).

Dagegen sind sowohl "X mod 2 = 0" als auch "X mod 2 = 1" Schleifeninvarianten.

Diese Information nützt uns aber nichts. Betrachten wir stattdessen noch mal die 5. Hoaresche Regel:

$$\frac{\{A \wedge b\}\ \mathbf{c}\ \{A\}}{\{A\}\ \texttt{while b loop c end loop}\ \{A \wedge \text{not}(b)\}}$$

In unserem Fall ist not(b) ≡ (X = 0), so dass B ≡ X mod 3 = 0 stets erfüllt ist, egal mit welchem Wert von X die Schleife begonnen wurde. Die Frage, ob die Schleife terminiert, spielt nach der Definition von {A} **c** {B} keine Rolle: Denn es soll B nach der Ausführung von **c** erfüllt sein (vgl. Folie 32); wenn jedoch die Schleife nicht endet, "dann ist danach alles erfüllt".

Somit erhalten wir als die schwächste Vorbedingung obiger Schleife die Zusicherung: X∈ •, d.h. die Nachbedingung ist für jede ganze Zahl, mit der man startet, erfüllt.

Im Allgemeinen lässt sich die schwächste Vorbedingung für eine Schleife jedoch algorithmisch nicht berechnen. Diejenigen, die das Programm entwickelt haben, kennen in der Regel aber mindestens eine Schleifeninvariante, nämlich die, die sie bei der Formulierung der Schleife im Sinn hatten. Mit einem interaktiven System kann diese Zusicherung eingegeben werden und das System kann versuchen zu beweisen, dass sie tatsächlich eine Schleifeninvariante ist.

Hier brechen wir die Erläuterung der Korrektheit mit Hilfe der axiomatischen Semantik ab. Es sollten nur die Ideen vorgestellt und ein mögliches Unterstützungssystem angedeutet werden. (Vertiefungen hierzu erfolgen in Vorlesungen über Semantik und z.T. in Vorlesungen über Interaktive Systeme, Wissensverarbeitung sowie Programmiersprachen/Übersetzerbau.)

Historischer Hinweis:
Das Problem, die Richtigkeit von Programmen nachzuweisen, besteht seit dem Beginn 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 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, konkrete Methoden wie model checking usw.).

1.4. Komplexität von Algorithmen

Die Idee: Komplexität ist der Aufwand an Zeit $t_A(w)$ oder an Platz $s_A(w)$, den ein Algorithmus A bei der Eingabe w benötigt, um seine Berechnung durchzuführen. (t kommt von "time complexity" und s von "space complexity".)

Genauer: Wenn ein Algorithmus A eine Eingabe w erhält, dann führt er eine Berechnung durch. Eine Berechnung ist eine Folge von elementaren Anweisungen (leere Anweisung, Wertzuweisung) und Abfragen, die durch die aktuelle Eingabe w festgelegt ist.

$t_A(w)$ = Anzahl der elementaren Anweisungen und Abfragen, die A bei Eingabe w durchläuft, bis er anhält,

$s_A(w)$ = Anzahl der Speicherplätze, auf die A bei Eingabe von w zugreift, bis er anhält.

In der Praxis interessiert man sich nicht für die einzelnen Eingabefolgen w, sondern nur für deren Länge. Daher definiert man:

Etwas konkretere Idee:
Es sei A ein Algorithmus, w sind Eingabefolgen.

$t_A(n)$ = Max $\{t_A(w) \mid$ die Länge von w ist n$\}$,

$s_A(n)$ = Max $\{s_A(w) \mid$ die Länge von w ist n$\}$.

Man beachte: Diese Definition setzt indirekt voraus, dass der Algorithmus A für alle Eingaben anhält; denn falls der Algorithmus A für eine Eingabefolge w der Länge n *nicht* anhält, dann wären $t_A(n)$ [und eventuell auch $s_A(n)$] unendlich groß und somit für Anwendungen uninteressant.

Definition:

Es sei A ein Algorithmus, der für alle Eingaben w anhält.
Dann definieren wir:
$\mathbf{t_A(n)}$ = Max $\{t_A(w) \mid$ die Länge von w ist n$\}$,
$\mathbf{s_A(n)}$ = Max $\{s_A(w) \mid$ die Länge von w ist n$\}$.
t_A und s_A sind also Abbildungen von $\bullet_0$ nach $\bullet_0$.

Hinweis: Legt man ein Maschinenmodell zugrunde, z.B. eine Turing- oder eine Registermaschine, dann lässt sich diese Definition exakt definieren als die Anzahl der durchlaufenen Konfigurationen oder als die Anzahl der besuchten Speicherplätze. Siehe Vorlesung "Theoretische Informatik II", Kapitel Komplexität.

Obige Definition ist aber für unsere Zwecke ausreichend.

Beispiel 1.8: Wir betrachten erneut Beispiel 1.5:

```
x, y, z: Natural;
Get (x, 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);
```

Dieses Programmstück berechnet die Multiplikation zweier natürlicher Zahlen. Wie groß ist der Aufwand?

```
Get (x, y);                    2 Schritte
z:=0;                          1 Schritt
while y > 0 loop               1 Schritt                m+1 mal
     if (y mod 2 = 0) then     1 Schritt
          y:=y div 2;
          x:=x+x;              2 Schritte
     else                                               m mal
          y:=y-1;              oder:
          z:=z+x;              2 Schritte
     end if;
end loop;
Put(z);                        1 Schritt
```

Zusammen: 4m + 5 Schritte. Aber was ist m?

Die Schleife wird durchlaufen, bis y = 0 ist. Spätestens in jedem zweiten Schritt wird y halbiert. Also wird die Schleife mindestens log(y) mal und höchstens 2 log(y) mal durchlaufen. Also gilt $\log(y) \leq m \leq 2\log(y)$.

Die Eingabewerte seien die natürlichen Zahlen a und b. Die Eingabelänge ist dann n = log(a) + log(b) + 2. (Das "+2" kommt daher, dass man ein Trennzeichen zwischen a und b und ein Zeichen für das Ende der Eingabe benötigt.)
m kann daher durch n nach oben abgeschätzt werden.

Für unseren Multiplikationsalgorithmus A gilt also wegen $m \leq 2 \cdot \log(y) \leq 2 \cdot n$:
$\mathbf{t_A(n)} = \text{Max}\ \{t_A(w) \mid \text{die Länge von w ist n}\} \leq 8n + 5$.

Man braucht drei Speicherplätze für x, y und z, folglich gilt:
$\mathbf{s_A(n)} = \text{Max}\ \{s_A(w) \mid \text{die Länge von w ist n}\} = 3$

Aber: Ist dies eine "richtige" Aufwandsberechnung?
Warum geht zum Beispiel nur die Länge der zweiten Zahl b, die y zugeordnet wird, und nicht auch die der ersten Zahl a, die in x steht, in die Komplexität ein??

..... weil wir jede Wertzuweisung und jede Abfrage so behandeln, als ob sie **in einem Schritt** ausgeführt werden kann.

In der Praxis würde die Anweisung x:=x+x nicht einen Schritt, sondern so viele Schritte benötigen, wie der Wert von x lang ist (ungefähr log(x)); denn so lange dauert die ziffernweise Addition, wenn man sie wie in der Schule erlernt durchführt.

Auch die Frage, ob y durch 2 teilbar ist (y mod 2 = 0), oder die Division durch 2 benötigen in der Regel einen Aufwand proportional zur Länge der jeweiligen Zahl (das hängt aber von der Zahldarstellung ab; bei der Basis 2 oder 10 braucht man nur die letzte Ziffer zu prüfen).

Also noch mal rechnen:

```
Get (x, y);        log(a) + log(b) = n  Schritte
z:=0;              1 Schritt
while y > 0 loop   ≤ log(b) Schritte   2 log(b)+1 mal
     if (y mod 2 = 0) then ≤ log(b)  Schritte
          y:=y div 2;  ≤ log(b) Schritte
          x:=x+x;      log(a)+log(b) Schritte
     else                                          2 log(b)
          y:=y-1;      log(b) Schritte               mal
          z:=z+x;      log(a)+log(b) Schritte
     end if;
end loop;
Put(z);   log(a) + log(b) = n  Schritte
```

Zusammen: ≤ (2log(b)+1) (4log(b)+log(a))+2n+1 ≤ ca. 8n(n+1)+1 Schritte.

Bezeichnung:
Berechnet man die Komplexität $t_A(n)$ bzw. $s_A(n)$ so, dass jede elementare Anweisung und jede Abfrage genau einen Schritt dauern, bzw. dass jede Zahl genau einen Speicherplatz belegt, so spricht man von der **uniformen Komplexität**.
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**.

In der Praxis berechnet man stets zunächst die uniforme Komplexität: Sie gibt meist eine hinreichend genaue Orientierung über den zu erwartenden Aufwand. Erst für eine genaue Abschätzung bzw. bei der Programmierung betrachtet man die logarithmische Komplexität, die den tatsächlichen Aufwand wesentlich genauer wiedergibt.

Die hier definierten Komplexitätsfunktionen $t_A(n)$ und $s_A(n)$ erfassen den schlechtesten Fall, der bei der Länge n auftreten kann. Man bezeichnet diese $t_A(n)$ und $s_A(n)$ daher auch als worst case complexity.

Eine andere Möglichkeit, die Komplexität zu definieren, besteht in der Mittelung über die Komplexitätswerte für alle Eingaben der Länge n. Es sei E die Menge aller Eingaben und für jede natürliche Zahl i sei E_i die Menge der Eingaben der Länge i. Insbesondere gilt
$E = E_0 \cup E_1 \cup E_2 \cup E_3 \cup E_4 \cup .... = \bigcup_{i=0}^{\infty} E_i$.
Der mittlere Wert, die average case complexity, lautet dann:

$$\bar{t}_A(n) = \frac{1}{|E_n|} \sum_{w \in E_n} t_A(w) \qquad \text{und} \qquad \bar{s}_A(n) = \frac{1}{|E_n|} \sum_{w \in E_n} s_A(w).$$

Es ist klar, dass der Aufwand an Zeit, an Platz usw., den ein Programm benötigt, besonders wichtig für die Informatik ist. Eine zentrale Aufgabe ist es daher, zu einer Funktion (bzw. zu einer Spezifikation) einen realisierenden Algorithmus (bzw. ein Programm) zu schreiben, dessen Komplexität möglichst gering ist.

Als Beispiel betrachten wir die Multiplikation zweier natürlicher Zahlen. Wir haben oben gesehen, dass es einen Algorithmus gibt, der die Multiplikation mit einer Zeitkomplexität von höchstens $8(n+1)^2$ realisiert, wobei n die Länge der eingegebenen Zahlen ist.

Gibt es einen schnelleren Algorithmus?

Oder kann man beweisen, dass es kein schnelleres Verfahren geben kann?

Vorbemerkung: 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) besitzt.

Da also das Ergebnis der Multiplikation nur so lang sein kann, wie die beiden Multiplikanten zusammen, könnte es eventuell sogar einen Algorithmus geben, der die Multiplikation proportional zu n durchführt.

Doch 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 keinem gelungen zu beweisen, dass die Multiplikation deutlich mehr Zeit als die Addition erfordert.

Nun wollen wir ein Verfahren vorstellen, welches deutlich schneller als mit der Zeitkomplexität $4(n+1)^2$ multipliziert.

Gegeben seien also zwei k-stellige natürliche Zahlen x und y; die Länge k sei eine gerade Zahl. Setze p=k/2. Dann lassen sich x und y schreiben in der Form:

$$\mathbf{x = A \cdot 2^p + B} \quad \text{und} \quad \mathbf{y = C \cdot 2^p + D},$$

wobei A, B, C, D mindestens 0 und echt kleiner als 2^p sind. Dann gilt

$$\mathbf{x \cdot y = (A \cdot 2^p + B) \cdot (C \cdot 2^p + D) = A \cdot B \cdot 2^{2p} + (A \cdot D + B \cdot C) \cdot 2^p + B \cdot D}.$$

Man kann also die Multiplikation zweier k-stelliger Zahlen durchführen, indem man sie auf 4 Multiplikationen von halber Länge k/2 zurückführt. Dies ergibt jedoch wiederum 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:

$$\begin{aligned} x \cdot y &= A \cdot C \cdot 2^{2p} + (A \cdot D + B \cdot C) \cdot 2^p + B \cdot D \\ &= A \cdot C \cdot 2^{2p} + ((A-B) \cdot (D-C) + A \cdot C + B \cdot D) \cdot 2^p + B \cdot D \end{aligned}$$

wobei A, B, C, D mindestens 0 und echt kleiner als 2^p sind.

Nun haben wir es geschafft: Auf der rechten Seite stehen nur noch drei verschiedene Multiplikationen, nämlich:

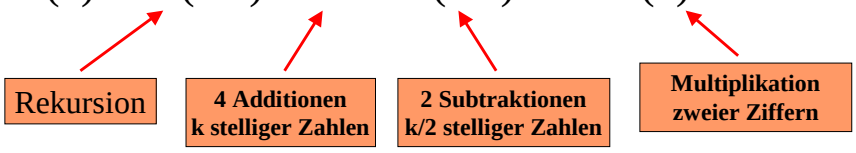
$A \cdot C$, **$B \cdot D$** und **$(A-B) \cdot (D-C)$**

Beachte: Die Multiplikation mit 2^p bzw. 2^{2p} ist keine echte Multiplikation, sondern nur ein Anhängen von p bzw. 2p Nullen.

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 zwei zusätzliche 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 nun 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:

$$\begin{aligned}
t(k) &= 3 \cdot t(k/2) + 5 \cdot k \\
&= 3 \cdot (3 \cdot t(k/4) + 5 \cdot k/2\ ) + 5 \cdot k \\
&= 3 \cdot 3 \cdot t(k/4) + 3 \cdot 5 \cdot k/2 + 5 \cdot k \\
&= 3 \cdot 3 \cdot t(k/4) + 5 \cdot k \cdot (1 + 3/2) \\
&= 3 \cdot 3 \cdot (3 \cdot t(k/8) + 5 \cdot k/4) + 5 \cdot k \cdot (1 + 3/2) \\
&= 3 \cdot 3 \cdot 3 \cdot t(k/8) + 5 \cdot k \cdot (1 + 3/2 + 9/4) \\
&= 3 \cdot 3 \cdot 3 \cdot (3 \cdot t(k/16) + 5 \cdot k/8) + 5 \cdot k \cdot (1 + 3/2 + 9/4) \\
&= 3 \cdot 3 \cdot 3 \cdot 3 \cdot t(k/16) + 5 \cdot k \cdot (1 + 3/2 + 9/4 + 27/8) \\
&= \ldots
\end{aligned}$$

Die allgemeine Form nach i Schritten lautet offensichtlich:

$$t(k) = 3^i \cdot t(k/2^i) + 5 \cdot k \cdot (1 + 3/2 + 9/4 + \ldots + 3^{i-1}/2^{i-1})$$
$$= 3^i \cdot t(k/2^i) + 10 \cdot k \cdot ((3/2)^i - 1)$$

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:

$$\mathbf{t(k)} = 3^{\log(k)} \cdot t(k/2^{\log(k)}) + 10 \cdot k \cdot ((3/2)^{\log(k)} - 1)$$
$$= k^{\log(3)} + 10 \cdot k \cdot (k^{\log(1,5)} - 1)$$
$$= 11 \cdot k^{\log(3)} - 10 \cdot k \approx \mathbf{11 \cdot k^{1,585} - 10 \cdot k} \in \mathbf{O(k^{1,585})}$$

Beachte hierbei die Formeln:
log ist hier der Logarithmus zur Basis 2,

$a^{\log(b)} = b^{\log(a)}$ und

$k \cdot k^{\log(1,5)} = k^{1+\log(1,5)} = k^{\log(2)+\log(1,5)} = k^{\log(2 \cdot 1,5)} = k^{\log(3)}$

Satz 1.2:

Die Multiplikation zweier k-stelliger Zahlen lässt sich in proportional zu $k^{1,585}$ Schritten durchführen.

(Die Schulmethode benötigt k^2 Schritte, siehe nächste Folie.)

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 damit ungefähr so schnell wie eine Addition - durchführt?

Anmerkung: Ab wann lohnt sich dieses rekursive Verfahren? Hierfür müssen wir wissen, wie aufwändig eine normale Multiplikation in Wahrheit ist. Der Algorithmus aus Beispiel 1.5 benötigt höchstens $8 \cdot (k+1)^2$ Schritte (vgl. Folie 72; setzen Sie dort log(a)=log(b)=k und rechnen nochmals nach). Doch bei einer genaueren Untersuchung erkennt man, dass jener Algorithmus im Wesentlichen in k^2 Schritten arbeitet; denn nur die Anweisung z:=z+x kostet "richtig" Zeit (x:=x+x bedeutet: hänge eine 0 an x an, bei y:=y div 2 wird nur die letzte 0 gestrichen usw.).

Dann läuft die Frage, ab wann sich die rekursive Methode lohnt, auf die Frage hinaus, ab welchem k gilt: $11 \cdot k^{1,585} - 10 \cdot k < k^2$? Dies trifft leider erst für Werte von k zu, die größer als 290 sind. Es müssen also schon spezielle Probleme sein, bei denen es sich lohnt, diese Methode einzusetzen.
Dies war aber nur eine überschlägige Berechnung. Eine genaue Analyse müsste *alle* auftretenden Operationen einbeziehen, z.B. auch die Funktionsaufrufe und die Speicherung von aktuellen Parametern.

Auf den letzten Folien trat das Symbol O(...) auf, das Sie aus der "Einführung in die Informatik I" bereits kennen. Dies ist das so genannte *Landau-Symbol* (nach dem deutschen Mathematiker Edmund Landau, 1877-1938). Es beschreibt die Größenordnung einer Funktion. Hierbei werden multiplikative und additive Konstanten vernachlässigt und nur der Term in Abhängigkeit von k, der für $k \to \infty$ 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.

Um die Größenordnung von Funktionen zu beschreiben, verwendet man folgende fünf Klassen O, o, Ω, ω und Θ:

Definition: "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).

$$\mathbf{O(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0: g(n) \leq c \cdot f(n)\},$$

$$\mathbf{o(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \forall c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0: c \cdot g(n) \leq f(n)\},$$

$$\mathbf{\Omega(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0: f(n) \leq c \cdot g(n)\},$$

$$\mathbf{\omega(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \forall c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0: c \cdot f(n) \leq g(n)\},$$

$$\mathbf{\Theta(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0: c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}.$$

Erläuterungen: 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 kleiner für gleiche Werte von n wächst als f, 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): 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 Ω(f) liegen also die Funktionen, die mindestens so stark wachsen wie f, und in ω(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: 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:
$3 \cdot n^3 + 12 \cdot n^2 + 8 \cdot n \cdot \log(n) + 6/n \in O(3 \cdot n^3) \cup O(n^2 + n \cdot \log(n)) = O(n^3)$.

Einige Klassen:

O(1) ist die Klasse der Funktionen, die höchstens wie ein Vielfaches der konstanten Funktion *f(n) = 1 für alle* $n \in \mathbb{N}$ 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 nur sehr schwach wachsen, also deutlich langsamer als f(n)=n? Aus der Schule und dem ersten Semester kennen Sie den Logarithmus. 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

Aufgabe für Sie: Untersuchen Sie diese Funktion log*.

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 $n \geq 1$ 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: $\log(n) \in O(n)$. Aber auch für die Potenzen des Logarithmus gilt $\log^m(n) \in O(n)$ für alle natürlichen Zahlen m. Hierfür beachte:

Für jedes $m \geq 1$ gilt ab einem hinreichend großen n:
$\log(n) < n^{\frac{1}{m}}$ wegen $n < 2^{\sqrt[m]{n}}$; 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 $n \geq 1$ 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 $\Theta(f) = O(f) \cap \Omega(f)$ 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 $n \geq 1$ 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*.
$\Omega(n^2)$ 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 in der Regel folgende Funktionen:

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.

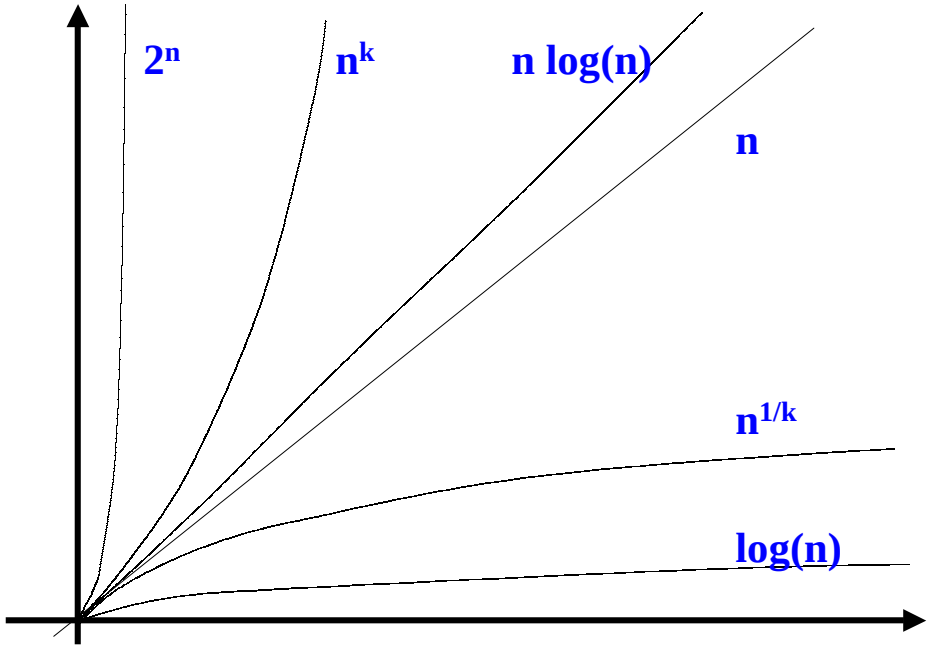
O(n): lineare Funktionen.

O(n log(n)): Das sind Funktionen, die "fast unmerklich" stärker als linear wachsen.

$O(n^2)$: höchstens quadratische Funktionen.

$O(n^k)$: höchstens polynomiell vom Grad k wachsende Funktionen.

$O(2^n)$: höchstens exponentiell (zur Basis 2) wachsende Funktionen.



Hilfssatz: Es gelten folgende Aussagen (der Beweis ist 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 $\Theta(f+g) = \Theta(f) = \Theta(g)$.

Aufgabe: 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 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 uns nämlich nur die Größenordnung der Komplexität und nicht der genaue Wert von Konstanten.

Historischer Hinweis:

Seit es (mathematische) Verfahren gibt, möchte man diese so schnell wie möglich ausführen. Beispiele sind die Lösung algebraischer (etwa: quadratischer) Gleichungen und die Berechnung von Wurzeln sowie die Lösung linearer Gleichungssysteme (Gaußsches Eliminationsverfahren). Bei vielen Verfahren kannte man ziemlich exakt die Größenordnung des Zeitaufwands in Abhängigkeit von den Parametern. Mit der Entwicklung von Großrechenanlagen in den 1960er Jahren analysierte man sehr genau die Komplexität von Algorithmen (abhängig vom jeweils benutzten Maschinenmodell). Hieraus entstand etwa Mitte der 1960er Jahre die Komplexitätstheorie.

Heute ist es üblich, mit jedem Algorithmus auch seine Komplexität zu berechnen, meist in Abhängigkeit zur Länge der Eingabe oder zur Zahl der bearbeiteten Objekte. Das Problem sind hierbei vor allem die *unteren* Schranken, also der Nachweis, dass ein Problem zu seiner Lösung mindestens einen gewissen Aufwand erfordert. Es gibt nur wenige Probleme, für die man nichttriviale untere Schranken kennt. *Obere* Schranken findet man dagegen meist leicht, da jeder Algorithmus eine solche obere Schranke für das von ihm realisierte Problem liefert.

1.5 Gegenläufigkeiten

Wenn es für ein Problem eine algorithmische Lösung gibt, so gibt es grundsätzlich auch unendlich viele Lösungsalgorithmen. Unter dieser Vielzahl möchte man einen möglichst guten finden. Dabei ist die "Zielfunktion", mit der die Algorithmen bewertet werden, wichtig. Sucht man beispielsweise einen Algorithmus, der möglichst wenig Speicherplatz verbraucht, so wird man einen anderen Algorithmus erhalten, als wenn man ein möglichst schnell arbeitendes Verfahren haben möchte.

Unterschiedliche Ziele sind in der Regel nicht gleichzeitig zu optimieren. Man spricht dann von Gegenläufigkeiten (engl.: Trade-Offs). Diese treten oft als "Zeit gegen Platz" (time versus space) auf.

Typische Situation: Ein Problem kann wesentlich einfacher gelöst werden, wenn man zuvor Hilfsberechnungen (z.B. eine Entfernungstabelle für gewisse "markante" Stellen bei der Berechnung kürzester Wege) durchführt und deren Ergebnis in arrays speichert; hat man keinen zusätzlichen Speicherplatz zur Verfügung, so muss man andererseits länger rechnen.

Wir geben hierfür keine konkreten Fragestellungen an, da wir bei Such- und Sortierverfahren mehrere solcher Beispiele kennen lernen werden. Es sollte hier nur auf dieses ständig wiederkehrende Problem hingewiesen werden.

Einführung in die Informatik II

Sommersemester 2002

2. Datenstrukturen

Manches in diesem Kapitel kennen Sie bereits aus der Vorlesung Einführung in die Informatik I von Prof. Lagally!

2.1 Mengen und elementare Typen

Die meisten Probleme kann man mit Hilfe von Mengen und auf ihnen definierten Operationen und Relationen beschreiben.

"Elementare" Mengen: Zeichen, Wahrheitswerte, natürliche Zahlen, ganze Zahlen, reelle Zahlen.

Symbol	Menge	Ada-Bezeichnung
A	Menge der (Tastatur-) Zeichen	Character
IB	{false, true}	Boolean
$\mathbb{N}_0$	{0, 1, 2, 3, 4, ...}	(zum Teil:) Natural
9	{..., -2, -1, 0, 1, 2, 3, ...}	(zum Teil:) Integer
IR	Menge der reellen Zahlen	(zum Teil:) Float

Hinzu kommen selbstdefinierte Mengen (Aufzählungstyp).

Auf diesen Mengen gibt es Operationen. In der Programmiersprache Ada werden u.a. folgende verwendet:

Menge der Zeichen (Character): **A**
Die Menge umfasst 256 Zeichen (darstellbar durch ein Byte). Die Elemente sind (entsprechend der 8-Bit-Reihenfolge) angeordnet. Daher gibt es die Vergleichsoperatoren:
=, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist.

Prinzipiell können auch bei Character die für Aufzählungstypen gültigen Funktionen verwendet werden:

pred(X) = das Zeichen vor X in der Character-Reihenfolge
succ(X) = das Zeichen nach X in der Character-Reihenfolge
pos(X) = das wievielte Zeichen ist X in der Character-Reihenfolge? (beginnend mit 0)
val(n) = n-tes Zeichen (beginnend mit 0)

Endliche selbstdefinierte Mengen: Aufzählungen
Wir erläutern dies am Beispiel. Es soll die Menge
D = {a, A, Do, 365, 7, xyz, Quadrat} eingeführt werden:
type mengeD **is** (a, A, Do, 365, 7, xyz, Quadrat);

Die Elemente sind entsprechend der Auflistung angeordnet, also a < A < Do < 365 < 7 <xyz < Quadrat. Zugleich sind die Elemente durchnummeriert, beginnend mit 0.
Es gibt die sechs Vergleichsoperatoren:
=, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist.

Für Aufzählungstypen gibt es weiterhin folgende Funktionen:
pred(X) = das Element vor X in der Auflistung
succ(X) = das Element nach X in der Auflistung
pos(X) = das wievielte Element ist X in der Auflistung?
val(n) = n-tes Element (beachte: beginnend mit 0)

Menge der Wahrheitswerte (Boolean): $\mathbb{B}$

Konstanten sind **true** und **false**.
Es gibt die üblichen logischen Operatoren:
Negation: **not**,
Konjunktion: **and**, Disjunktion: **or**, Exklusives Oder: **xor**.

Zusätzlich stehen and und or auch als „Kaskadenoperationen“ **and then** und **or else** zur Verfügung, die man mit Vorsicht verwenden sollte (z.B. wegen Fehlern und exceptions):

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

Menge der ganzen Zahlen (Integer): 9

Einschränkung: Es gibt eine größte und eine kleinste darstellbare Zahl, die implementierungsabhängig sind, aber mindestens $\pm 2^{15}$ betragen. Es kann also nur dieser endliche Bereich verwendet werden. Damit sind dann auch die Operationen nicht mehr assoziativ, wenn man die Bereichsgrenzen überschreitet.

Die üblichen Operationen: Negatives eines Zahl -, Absolutbetrag **abs**, Addition +, Subtraktion -, Multiplikation *, ganzzahlige Division /, Rest bei Division (**mod** oder **rem**), Exponentiation ** (rechts muss eine natürliche Zahl stehen).
Es gibt die sechs Vergleichsoperatoren:
=, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist.

Menge der natürlichen Zahlen (Natural): $\mathbb{N}_0$

In Ada werden die natürlichen Zahlen als Untertyp der ganzen aufgefasst, d.h., es gilt:

subtype Natural **is** Integer **range** 0..Integer‘Last;
Somit gibt es auch eine größte darstellbare natürliche Zahl.

Addition +, Subtraktion - (sofern das Ergebnis nicht negativ ist), Multiplikation *, ganzzahlige Division /, Rest bei Division (**mod**), Exponentiation **. Es gibt die sechs Vergleichsoperatoren wie bei Integer.

Menge der reellen Zahlen (Float): $\mathbb{R}$

Da es nur endlich lange Zahldarstellungen im Rechnern gibt, lässt sich nur eine Teilmenge der rationalen Zahlen darstellen. Diese Zahlen werden durch die jeweilige Zahldarstellung (z.B. Vorzeichen, Mantisse, Exponent) und die Anzahl der verwendeten Dezimalstellen festgelegt. (Eine Darstellung zu einer beliebigen Basis m>2 ist möglich.)

Kann hierbei der Dezimalpunkt an unterschiedlichen Stellen stehen, so spricht man von der floating point (Gleitpunkt- oder Gleitkomma-) Darstellung. bei fester Position des Dezimalpunkts liegt die fixed point (Festkomma-) Darstellung vor.

In Ada kann man auch die Zahl der Ziffern und die minimal erforderliche Genauigkeit vorgeben (Sprachelemente: digits und delta).

Für Gleitkomma- und Festkommadarstellungen gibt es die Operationen unäres + und -, Absolutbetrag, Addition +, Subtraktion -, Multiplikation *, Division / und Exponentiation ** (rechts von ** muss eine ganze Zahl stehen).
Es gibt die sechs Vergleichsoperatoren: =, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist.

Hinweis: Verwenden Sie "=" oder "/=" nicht bei Gleitkommazahlen, da wegen der Rundungsfehler kein korrektes Ergebnis erwartet werden kann.

Hinweis: Beachten Sie, dass das Ergebnis der Operationen vom Typ der Operanden abhängt und in der Regel keine automatische Typanpassung erfolgt! (Stichwörter: Überladen, Typkonversion.)

2.2. Datenkonstruktoren

Aus Basisdaten lassen sich neue Datenbereiche zusammensetzen. Typische "Konstruktoren" sind:

Kartesisches Produkt und Vektorräume bilden.

Mengen disjunkt vereinigen.

Potenzmenge bilden.

Menge der endlichen Folgen bilden ("freies Monoid").

Menge der Abbildungen zwischen Mengen bilden.

Zusammengesetzte Mengen: kartesisches Produkt, Vereinigung, Vektoren, freie Monoide (Wortmengen), Potenzmenge

Operation	Ada-Bezeichnung
$M_1 \times M_2$	**type** <name> **is record** <sel_1> : <Typ_1>; <sel_2> : <Typ_2>; **end record**;
$M_1 \cup M_2$	**type** <name> **is record** **case** <name> **is** **when** <name1> => <sel_1> : <Typ_1>; **when** <name1> => <sel_2> : <Typ_2>; **end case**; **end record**; (sog. varianter record)
M^n	**type** <name> **is array** (1..n) **of** <Typ>;
M^*	(das freie Monoid wird über Pointer dargestellt)
2^M	(die Potenzmenge lässt sich nicht direkt darstellen)

2.3. Aufbau von Programmiersprachen

Wenn Sie eine Programmiersprache definieren müssten ...

1. Elementare Datentypen:

   Festlegung der "atomaren" Mengen, also derjenigen Mengen, die in der Sprache zur Verfügung stehen und die nicht mehr aus kleineren Mengen aufgebaut werden sollen.
   Festlegung der auf ihnen zulässigen Operationen. Für jede dieser Mengen formuliere man einen elementaren Datentyp einschl. möglicher Beschränkungen durch die Implementierung.

   Beispiel: Menge der natürlichen Zahlen. Operationen +, *, >, =, mod, ... Typname: Natural. Die maximal darstellbare Zahl ist Natural'Last; die Operationen sind daher auf den Bereich (0..Natural'Last) einzuschränken. Zusätzlich kann man elementare Datentypen durch Aufzählung ihrer Elemente einführen; Operationen müssen dann über Programmstücke (Prozeduren, Funktionen) festgelegt werden.

2. Datenstrukturen:

   Festlegung der zulässigen Konstruktoren, um aus Mengen neue Mengen zu gewinnen, z.B. record, case, array, access, Subtyp.
   Festlegung, wie man auf die einzelnen Komponenten zugreifen kann. Lege weiterhin fest, welche neuen Operationen für diese neuen Strukturen definiert sein sollen. Gib die erforderlichen Beschränkungen an.

   Beispiel: Bilde zu einer Menge seine Potenzmenge, also die Menge aller Teilmengen. Mengenkonstruktor: set of <Menge>. Neue Konstanten und Operationen: leere Menge, in (Element-Beziehung), Enthalten-Sein, Komplement, Vereinigung, Durchschnitt, Anzahl der Elemente, ...
   Beschränkung: Es sind nur Teilmengen der Grundmenge <Menge> zugelassen, die höchstens ... Elemente besitzen.
   [Diese Datenstruktur "Potenzmenge" ist in Ada nicht zugelassen; sie kann aber über Bitvektoren, Bäume usw. simuliert werden.]

3. Ausdrücke:
   Festlegung der zulässigen Ausdrücke und ihrer Ergebnistypen. In der Regel baut man diese (wie logische oder arithmetische Ausdrücke) rekursiv aus den bereits festgelegten Operationen auf, wobei man Verträglichkeiten genau definieren muss. Die erforderlichen Beschränkungen sind anzugeben.

   Beispiel: Ganzzahlige Ausdrücke (Integer Expression, abgekürzt *IntExp*):

   a. Jede Variable und Konstante vom Typ Integer oder vom Typ Natural ist ein *IntExp*.
   b. Jeder Funktionsaufruf mit dem Ergebnistyp Integer oder Natural ist ein *IntExp*.
   c. Wenn P ein *IntExp* ist, dann auch (P), +P, -P, **P und abs(P).
   d. Wenn P und Q *IntExp* sind, dann auch (P+Q), (P-Q), (P*Q), (P/Q), (P mod Q) und (P rem Q).
   e. Alle Ausdrücke in *IntExp* müssen durch die Regeln a. bis d. herleitbar sein.

   Ergebnistyp ist Integer; er ist jedoch Natural, falls nur Natural-Typen an den Operationen beteiligt waren und der Natural-Bereich während der Berechnungen nicht verlassen wurde.

4. Elementare Anweisungen:

   In imperativen Sprachen sind dies:

- Leere Anweisung ("skip" oder ein allein stehendes Semikolon).
- Wertzuweisung "X := ß;" wobei ß ein Ausdruck ist, dessen Ergebnistyp gleich dem Typ der Variablen X ist.
- Prozeduraufruf.
- Sprung (sofern in der Sprache zugelassen; "goto").

Hinzu kommen spezielle Anweisungen wie exit (Beendigung von Schleifen), return (expliziter Rückkehr aus Einheiten), raise (Aktivieren einer Ausnahmebehandlung), delay, code, abort usw.

5. Kontrollstrukturen:

Die elementaren Anweisungen werden mit Konstruktoren zu neuen Anweisungen zusammengesetzt.
Es seien a, a_1, a_2 Anweisungen und b eine Bedingung (= Ausdruck vom Ergebnistyp Boolean).

- Hintereinanderausführung: a_1 **;** a_2 (Semikolon).
- Alternative: **if** b **then** a_1 **else** a_2 **end if**;
- Auswahl: **case** <ausdruck> **is**
  when <auswahl$_1$> => a_1;
  when <auswahl$_2$> => a_2; ...
  when <auswahl$_n$> => a_n; **others** => a_{n+1}; **end case**;
- Laufschleife: **for** K **in** (1..n) **loop** a **; end loop;**
- Schleife: **while** b **loop** a **;end loop;**
- Ausnahme: **raise** ..., bezogen auf ein **exception** ... **end**;
- Nichtdeterminismus: **select** ... **or** ... **else** ... **end select**;
- Nebenläufigkeit, Synchronisation (Taskaufruf, entry, accept, ...)

Achten Sie bei der Definition der Kontrollstrukturen darauf, dass mit ihnen die bereits definierten Datenstrukturen möglichst optimal durchlaufen werden können. Mit obigen Kontrollstrukturen wird genau dies angestrebt:

- Die Auswahl *case ... end case* korrespondiert mit der Datenstruktur record .... end record und zugleich mit dem "varianten" record (Vereinigung von Mengen).
- Die Laufschleife *for K in (1..n) loop a ; end loop* erlaubt das Durchlaufen von arrays.
- Die Schleife *while b loop a ;end loop* ermöglicht den Durchlauf durch eine Folge von Elementen (Listen).

6. Zusammenfassung zu Einheiten:

Algorithmen und Daten können zu einer Einheit zusammengefügt und anschließend als Ganzes genutzt werden. *Parametrisierungen* sind erlaubt; diese können Daten, Typen, Einheiten, ... sein.

Einheiten können sein: Prozeduren und Funktionen, Moduln (Pakete: Spezifikation und Rumpf), Prozesse (Tasks), Objekte und Klassen mit Vererbungen und Interaktionen. Hinzu kommen Bibliotheken, in denen ausformulierte Einheiten stehen, die in andere Programme eingebunden werden können.

Einheiten haben oft die grobe Struktur: Zweiteilung in Spezifikation (auch Schnittstelle genannt) und Implementierungsteil (Rumpf). Der Rumpf besteht meist aus einer Deklarationen gefolgt von Anweisungen. Hierbei ist festzulegen, was deklariert werden muss und was nicht und was wie gekapselt (= nach außen hin verborgen) wird.

7. Programme:

Programme sind in der Regel Folgen von Einheiten, wobei ein Startpunkt anzugeben ist.

Für die konkrete Ausführung müssen noch viele zusätzliche Angaben gemacht werden, z.B.:

Einbindung in die Datenverwaltung des jeweiligen Systems: Angabe der Ein- und Ausgabebereiche, der Filesysteme, Anschluss an Netze, Nutzung von Geräten ...

Hinweise an den Compiler, Nutzung von Gegebenheiten (pragmas). Angabe von Übersetzungseinheiten (separate).

Einbinden fremder Programmstücke und vordefinierte Klassen mittels **with** und **use**.

Und dann gibt es noch viele weitere "Features" .....

Anregung an Sie:

Definieren Sie sich Ihre eigene Programmiersprache für einen bestimmten Bereich, z.B.

- Sprache zum Rechnen mit (beliebigen) Mengen,
- Sprache für die Geometrie,
- Sprache für den Bau eines Eigenheims,
- Sprache zur Beschreibung von Schaltwerken,
- Sprache für das Kochen,
- Sprache für das Schachspiel,
- Sprache für Arbeitsprozesse

usw. Machen Sie sich klar, wie einfach dies *im Prinzip* ist, wie komplex die konkrete Ausgestaltung wird und welche Schwierigkeiten beim Zusammenwirken der verschiedenen Sprachelemente und bei der Implementierung in einer Programmiersprache entstehen.

Überlegen Sie auch, wie Ihre Sprache umgestaltet werden muss, um beispielsweise folgende Punkte behandeln zu können:

Gute Erlernbarkeit. Nähe zu Anwendungen.
Gute Bedienoberflächen. Visualisierbarkeit.
Erstellen einer guten Dokumentation.
Zuverlässigkeit und Robustheit der Programme.
Wartung, Pflege, Austauschbarkeit.
Wiederverwendbarkeit von Programmstücken.
Zusammenwirken mit anderen Programmen über Netze.
Verteilung eines Programms auf viele verschiedene Rechner.
Beachtung von Standards und Normen.
Und was die Kunden sonst noch alles gerne hätten ....

2.4. Felder (arrays)

Wiederholung: arrays (auf deutsch: Felder) beschreiben n-Tupel über einer Menge, also Folgen über dem gleichen Datentyp. Der Zugriff auf die einzelnen Komponenten erfolgt direkt über einen Index. Der Wert von n kann flexibel sein; der Indexbereich ist ein zusammenhängender Subtyp eines geordneten Datentyps. Grundsätzliche Form:

type <name> **is array** <Indexbereich> **of** <Datentyp>

Beispiele:
type Hvektor **is array** (1..100) **of** float;
type Hmatrix **is array** (1..50) **of** Hvektor;
type Bundesligatabelle **is array** (1..18) **of** fussballverein;
type codierung **is array** (Character) **of** Character;

Die iterierte array-Bildung kürzt man ab, indem alle Indexbereiche in eine Definition geschrieben werden:

Statt
type Hvektor **is array** (1..100) **of** float;
type Hmatrix **is array** (1..50) **of** Hvektor;
type Hvolume **is array** (1..80) **of** Hmatrix;

schreibt man also kurz:
type Hvolume **is array** (1..100,1..50,1..80) **of** float;

Die Zahl der hierbei verwendeten Indexbereiche heißt die *Dimension* des Feldes. Hvolume ist also ein 3-dimensionales Feld.

Statt Konstanten dürfen in den Indexbereichen auch Ausdrücke verwendet werden, sofern jeder Ausdruck in dem Augenblick, in dem die Deklaration erreicht wird, auch tatsächlich ausgerechnet werden kann.

Beispiele (wobei f und g Funktionen vom Ergebnistyp integer sein sollen):
type Hvektor **is array** (1..N, x..I*J) **of** Boolean;
type ausschnitt **is array** (f(unten)..g(oben)) **of** integer;
type sonstiges **is array** ((oben-unten) div 2..f(g(unten*oben))) **of** float;

Ein Feld heißt *statisch*, wenn zur Übersetzungszeit die Feldgrenzen alle bekannt sind. Wird mindestens eine Feldgrenze erst zur Laufzeit des Programms berechnet, so heißt das Feld *dynamisch*.

Auch *unspezifizierte* Feldgrenzen sind zulässig. Man trägt dann nur den Datentyp des Index ein und fügt ein
range <> ("<>" sprich "box")
hinzu. Solche unspezifizierten array-Deklarationen muss man bei ihrer Verwendung spezifizieren, z.B. bei der Deklaration von Variablen und beim konkreten Parameteraufruf.

Beispiele:
type text **is array** (natural **range** <>) **of** Character;
type raster **is array** (integer **range** <>, integer **range** <>) **of** pixel;
type ganzzahlvektor **is array** (integer **range** <>) **of** integer;
procedure Sort (A: **in out** ganzzahlvektor; unten, oben: integer) **is** ...

Bei der Deklaration von Variablen gibt man dann die konkreten Grenzen statisch oder dynamisch ein, z.B.:
X: ganzzahlvektor (-10..10) oder Y: ganzzahlvektor (I..J)

Ein einfaches Problem ist die Einbettung mehrdimensionaler Felder in eindimensionale. Diese Aufgabe muss jeder Compiler lösen können, da heutige Speicher in der Regel eindimensional sind. Die übliche Einbettung lautet:

Wenn das n-dimensionale Feld ($n \geq 2$) von der Form

array ($u_1..o_1$, $u_2..o_2$, ..., $u_n..o_n$) **of** ...

und das eindimensionale Feld von der Form

array (unten..oben) **of** ...

und alle Indextypen ($u_j..o_j$) und (unten..oben) ganzzahlige Intervalle sind, dann kann man den Index (i_1, i_2, ..., i_n) abbilden auf

$$f(i_1, i_2, ..., i_n) = unten + \sum_{k=1}^{n} d_k \cdot (i_k - u_k) \quad \text{mit} \quad d_k = \prod_{j=k+1}^{n} (o_j - u_j + 1)$$

Nebenbedingung: oben-unten+1 = d_0. (beachte: $d_n = 1$)

f heißt Speicherabbildungsfunktion.

Hinweis 1: Wenn ein Compiler ein mehrdimensionales Feld in die Maschinensprache übersetzt, dann legt er zugleich ein Feld für die Werte d_k an, k = n, n-1, ..., 1. Bei jedem Zugriff auf das Feld wird dann $f(i_1, i_2, ..., i_n)$ berechnet.

Ein "optimierender" Compiler wird bei **for**-Schleifen die Differenz zwischen den Indices zweier aufeinander folgender Zugriffe ermitteln und versuchen, das Fortschalten des Index durch Addition eines Differenzterms zu erledigen, so dass f nur beim ersten Mal berechnet wird und anschließend nicht mehr. Allerdings wird hierbei eine eventuelle Bereichsüberschreitung bei den Grenzen nicht erkannt, weshalb der Compiler mehrere Möglichkeiten bei der Übersetzung hat, die i.A. der Benutzer auswählen kann (in Ada: pragma).

Hinweis 2: Die oben angegebene Funktion f speichert das mehrdimensionale Feld "zeilenweise" (machen Sie sich dies an einer Matrix klar!).

Will man spaltenweise speichern (FORTRAN macht das so), dann muss man die Funktion leicht modifizieren. (Wie?)

Hinweis 3: Die Funktion f lässt sich auch schreiben als:

$$f(i_1, i_2, ..., i_n) = \text{unten} - \sum_{k=1}^{n} d_k \cdot u_k + \sum_{k=1}^{n} d_k \cdot i_k = u_{reduz} + \sum_{k=1}^{n} d_k \cdot i_k$$

mit der "reduzierten Anfangsadresse" $u_{reduz} = \text{unten} - \sum_{k=1}^{n} d_k \cdot u_k$

Ein Compiler speichert daher u_{reduz} und die d_k-Werte sowie eventuell alle o_j und u_j, um die Bereichsüberschreitung zu testen.

Beispiel 2.1: Intervallschachtelung

Ein Feld A: **array** (1..n) **of** integer sei gegeben. Das Feld sei sortiert, d.h.: $A(i) \leq A(i+1)$ 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 bekannte Intervallschachtelung: Teste, ob s genau in der Mitte A(mitte) von A liegt, falls nein und ist $A(\text{mitte}) < s$ ist, suche rechts von der Mitte weiter, sonst links.

Vergleiche Manuskript Plödereder, Abschnitt 4.1.2; dort heißt die Intervallschachtelung "Binäre Suche".

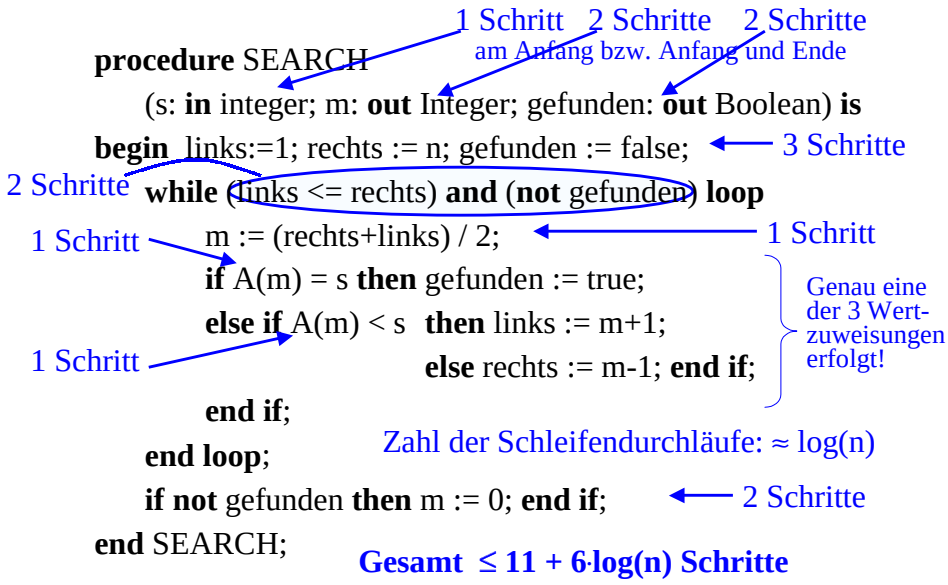
Programm 1: Intervallschachtelung oder binäre Suche

```
procedure SEARCH
        (s: in integer; m: out Integer; gefunden: out Boolean) is
procedure Intervallschachtelung (links, rechts: in integer) is
begin
    if links <= rechts then
        m := (rechts+links) / 2;
        if A(m) = s then gefunden := true;
        else if A(m) < s  then Intervallschachtelung (m+1,rechts);
                          else Intervallschachtelung (links, m-1); end if;
    else gefunden := false; m:=0;
    end if;
end Intervallschachtelung;
begin  Intervallschachtelung (1,n)  end SEARCH;
```


Programm 2: Man kann auch eine iterative Version angeben:

```
procedure SEARCH
    (s: in integer; m: out Integer; gefunden: out Boolean) is
begin  links:=1; rechts := n; gefunden := false;
    while (links <= rechts) and (not gefunden) loop
        m := (rechts+links) / 2;
        if A(m) = s then gefunden := true;
        else if A(m) < s  then links := m+1;
                          else rechts := m-1; end if;
        end if;
    end loop;
    if not gefunden then m := 0; end if;
end SEARCH;
```

Aufwand (für beide Programme), uniforme Zeitkomplexität:



Der schlechteste Fall kann auch tatsächlich eintreten, wenn nämlich das gesuchte Element s nicht im Feld A enthalten ist. Die *uniforme worst case time-complexity* lautet daher

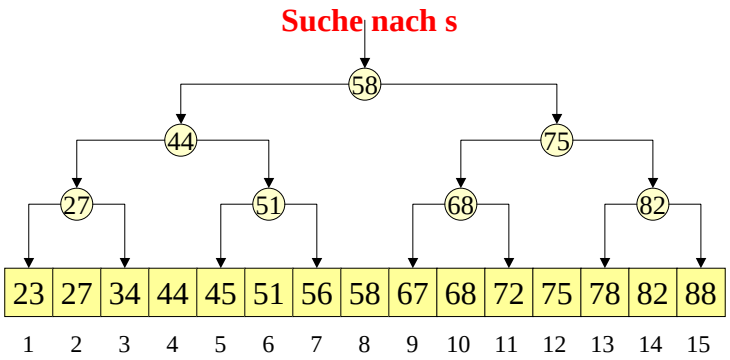
$t(n) = 11 + 6 \cdot \log(n)$.

Beachten Sie: n ist hier nicht die Länge der Eingabe, sondern die Anzahl der Elemente im Feld A.

Was ist der beste Fall? In diesem Fall wird s im ersten Durchgang der while-Schleife gefunden, d.h. nach 17 Schritten ist die Prozedur beendet.

Mit wievielen Schritten muss man im Mittel rechnen? Hierzu nehmen wir an, dass sich das gesuchte Element s tatsächlich im Feld A befindet (sonst kann man nur die 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 $\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^j$

$$\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^{j} = \frac{1}{2}\sum_{j=1}^{k} (j-1)\cdot 2^{j} + \frac{1}{2}\sum_{j=1}^{k} 2^{j}$$

$$= \sum_{j=1}^{k} (j-1)\cdot 2^{j-1} + \frac{1}{2}(2^{k+1} - 2)$$

$$= \sum_{j=0}^{k-1} j\cdot 2^{j} + (2^{k} - 1) = \sum_{j=1}^{k} j\cdot 2^{j} - k\cdot 2^{k} + (2^{k} - 1),$$ 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 time-complexity* der Intervallschachtelung ziemlich genau **11 + 6·(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(m) = 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(m) = s gewesen ist; hierzu muss man im Falle A(m) < s in dem rechten Teil des Feldes weitersuchen (links:=m+1), anderenfalls im linken Teil einschließlich des gerade betrachteten Feldes m (rechts:=m).

Programm 3: Verbesserte iterative Version:

```
procedure SEARCH
    (s: in integer; m: out Integer; gefunden: out Boolean) is
begin  links:=1; rechts := n;
    while (links < rechts) loop
        m := (rechts+links) / 2;
        if A(m) < s then links := m+1;
                    else rechts := m; end if;
        end if;
    end loop;
    gefunden := A(m) = s;
    if not gefunden then m := 0; end if;
end SEARCH;
```


Weisen Sie nun nach, dass für diese Version gilt:

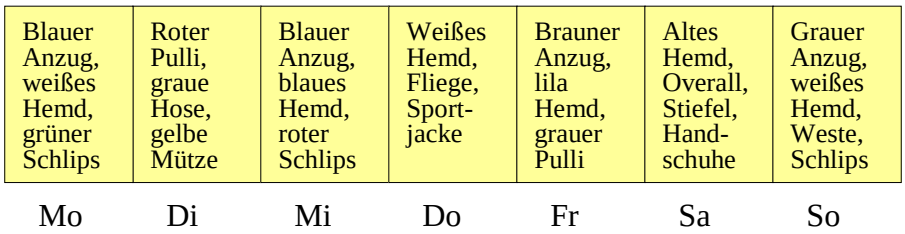
Die *uniforme worst case time-complexity* beträgt $11 + 4 \cdot \log(n)$; die *uniforme average case time-complexity* besitzt nun genau den gleichen Wert, da ja die Entscheidung, ob A(m)=s ist, in jedem Fall erst nach dem log(n)-maligen Durchlaufen der Schleife gefällt wird!

Hierbei ist n die Zahl der Elemente im array.

Weitere Suchverfahren auf Feldern betrachten wir in Kapitel 4.

Wie kann man Felder nutzen? Hier sind drei Möglichkeiten:

1. Direkte Speicherung der Information, also: das Feld als Speicher - und Suchstruktur:



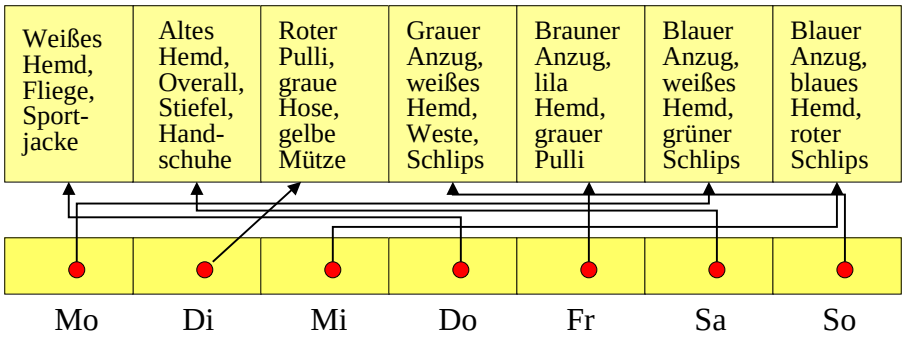
type wochentag **is** (Mo, Di, Mi, Do, Fr, Sa, So);

type kleidung **is array** (1..100) **of** Character;

type anziehen **is array** (wochentag) **of** kleidung;

2. Nur Speicherung der Adressen, also nur als Suchstruktur:

In der Halde können die Daten beliebig irgendwo liegen:



type kleidung **is array** (1..100) **of** Character;
type zugriff **is access** kleidung;
type anziehen **is array** (wochentag) **of** zugriff;

3. Getrennte Felder für Speicherung und Sortieren/Suche:

Weißes Hemd, Fliege, Sport-jacke	Altes Hemd, Overall, Stiefel, Hand-schuhe	Roter Pulli, graue Hose, gelbe Mütze	Grauer Anzug, weißes Hemd, Weste, Schlips	Brauner Anzug, lila Hemd, grauer Pulli	Blauer Anzug, weißes Hemd, grüner Schlips	Blauer Anzug, blaues Hemd, roter Schlips
1	2	3	4	5	6	7

6	3	7	1	5	2	4
Mo	Di	Mi	Do	Fr	Sa	So

```
type kleidung is array (1..100) of Character;
type index is range 1..7;
type anziehsachen is array (index) of kleidung;   -- Speicherung
type sortierfeld is array (wochentag) of index;   -- Zugriff
```


Beispiel 2.2: Noch ein kurzes Standardbeispiel:
Codieren durch Buchstabenverschiebung um N Stellen:

```
type Codierung is array (Character) of Character;
X: Codierung; N: Natural; A: Character;

procedure EinsVerschieben is
begin  for A in Character loop
          if X(A) = Character'Last then X(A):=Character'First
          else X(A) := succ(X(A)); end if; end loop;
end;     .....

for A in Character loop X(A) := A; end loop;
for I in (1..N) loop EinsVerschieben; end loop;   ....
--  Die Codierung durch Verschieben um N Stellen im Alphabet
    --     erfolgt anschließend durch:
get (A); put(X(A)); ...
```

Bemerkung: Wir verwenden öfters den Logarithmus log(n) wie eine ganze Zahl. Was genau ist damit gemeint?

Mathematische Definition: für $x > 0$, $b > 1$ gilt
$\log_b(x) = y \Leftrightarrow x = b^y$.

Der ganzzahlige Anteil des Logarithmus von x zur Basis b ist für x>1 im Wesentlichen die Länge der Zahlendarstellung von x im Stellenwertsystem zur Basis b.

Beispiele (man schreibt auch lg, ld, ln bei Basis 10, 2 bzw. e = 2,7182818...):
$\log_{10}(17635) = \lg(17635) = 4{,}2464... \approx 5 = \lceil \log_{10}(17635) \rceil$
$\log_2(3{,}8) = \mathrm{ld}(3{,}8) = 1{,}9260... \approx 2 = \lceil \log_2(3{,}8) \rceil$
$\log_e(8{,}2) = \ln(8{,}2) = 2{,}1041... \approx 3 = \lceil \log_e(8{,}2) \rceil$

Für uns ist der Logarithmus log die Länge der Darstellung zur jeweiligen Basis, wobei wir stets die Basis 2 annehmen, sofern nichts anderes gesagt wird. Also definieren wir:

Unsere Definition: log: $\mathbb{N}_0 \to \mathbb{N}_0$ für alle $n > 0$ definiert durch
$\log(n) = y \Leftrightarrow$ y ist die kleinste natürliche Zahl mit $n < b^y$.
Weiterhin sei ist $\log(0) = 1$. Einige Werte:

n	log(n)	n	log(n)	n	log(n)
0	1	8	4	128	8
1	1	9	4	500	9
2	2	10	4	512	10
3	2	15	4	1000	10
4	3	16	5	9000	14
5	3	31	5	10^6	20
6	3	32	6	10^9	30
7	3	64	7	10^{12}	40

Hinweis 1: Wenn $\log_2$ der mathematisch exakt definierte reellwertige Logarithmus ist, dann gilt $\log(0) = 1$ und für alle $n > 0$: $\log(n) = \lceil \log_2(n+1) \rceil$.
Da sich unser Logarithmus und der exakte Logarithmus für $n \geq 1$ immer höchstens um 1 unterscheiden, werden wir diese beiden Funktionen als "im Wesentlichen gleich" auffassen.

Hinweis 2: Beachten Sie, dass sich die Logarithmen zu zwei verschiedenen Basen a und b immer nur um die Konstante $\log_a(b)$ unterscheiden,
denn es gilt mathematisch für alle $x>0$:

$$\log_a(x) = \log_b(x) \cdot \log_a(b).$$

Speziell gilt daher für alle Basen $a>1$ und $b>1$:

$$O(\log_a(n)) = O(\log_b(n)) = O(\log(n)).$$

2.5. Pointer, Listen

Listen sind Folgen von Elementen des gleichen Datentyps. Sie werden durch *Zeiger (pointer)* realisiert. Die Verkettung kann einfach oder doppelt sein. Das erste und/oder das letzte Element sind von außen über einen Zeiger erreichbar (auch *Anker* der Liste genannt). Der Zugriff erfolgt *sequentiell*; man durchläuft also die Liste von vorne nach hinten bzw. von hinten nach vorne, um nach einem Element zu suchen oder ein Element einzufügen.

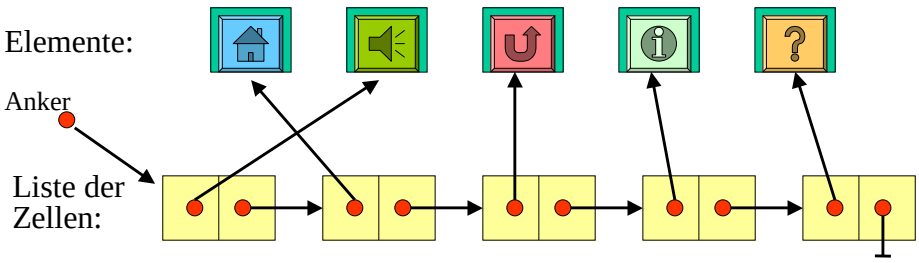
Vgl. Informatik I (Prof. Lagally) und Skript Plödereder Abschnitt 1.5.2

Das Schlüsselwort für Zeiger lautet in Ada ***access***.
Typische Definition einer Liste in Ada:

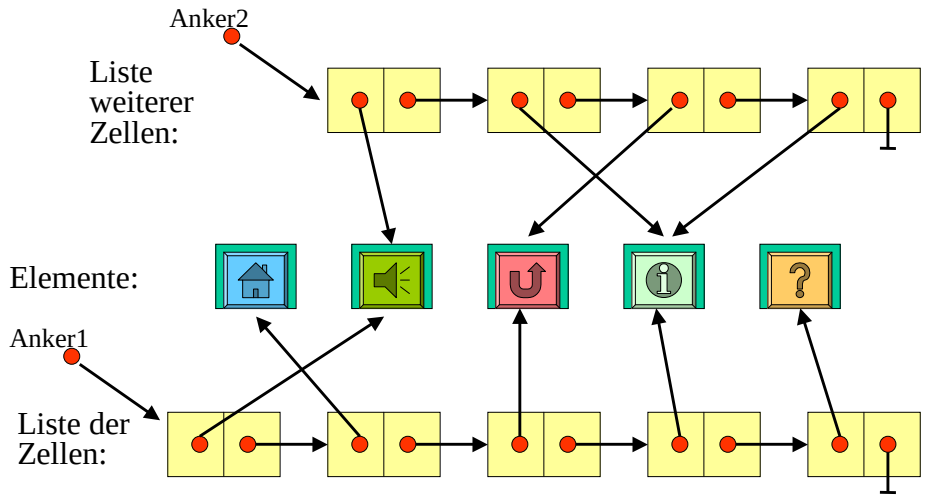
```
type Element is ....;    -- Definiere den Datentyp der Listenelemente
type List_Element;    -- Vorwärtsverweis
type List_Element_Zeiger is access List_Element;
type List_Element is  record
                          Inhalt: Element;
                          Next: List_Element_Zeiger;
                      end record;
```

Dies ist eine Liste, in der die Elemente direkt stehen. In der Praxis kann dies lästig sein (vgl. Folien 141-143 über Felder), da ein Element in vielen Listen auftreten kann und dann auch in allen Listen gespeichert und geändert werden muss. 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;
```



Will man nun Elemente in mehreren Listen gleichzeitig verwenden, so kann dies ohne Kopien geschehen:



Vorteile dieser Zellen-Darstellung:

- Elemente können in verschiedenen Listen sein.
- Das Ändern von Elementen geschieht synchron überall.
- Das Einfügen in andere Listen ist einfach.
- Es lassen sich weitere Zugriffsstrukturen leicht aufbauen.

Nachteile dieser Zellen-Darstellung:

- Der Zugriff auf Elemente dauert etwas länger.
- Man braucht etwas mehr Speicherplatz.
- Die Speicherverwaltung kann deutlich schwerer werden!

Hinweis: Listen werden in der Halde abgelegt. Nur die Anker stehen im statischen Bereich des Programms, sofern sie deklarieret Variablen sind.

Bitte wiederholen: Informatik I (Prof. Lagally) S. 135 und 138, WS 01/02

2.6. Stapel, Warteschlangen

Stapel und Warteschlangen sind Listen mit speziellen zulässigen Operationen (also (abstrakte) Datentypen).

Stapel oder Stack oder Keller oder Pushdown:

newstack	-- Leeren des Stacks
isempty	-- Abfrage, ob der Stack leer ist
top	-- Oberstes Element im Stack
push	-- Füge ein Element oben an
pop	-- Lösche das oberste Element
length	-- aktuelle Länge des Stacks

Warteschlange oder Queue:

newqueue	-- Leeren des Schlange
isempty	-- Abfrage, ob Schlange leer ist
first	-- Vorderstes Element der Schlange
rest	-- Schlange ohne erstes Element
enqueue	-- Füge Element hinten an
dequeue	-- Entferne erstes Element
length	-- Länge der Schlange

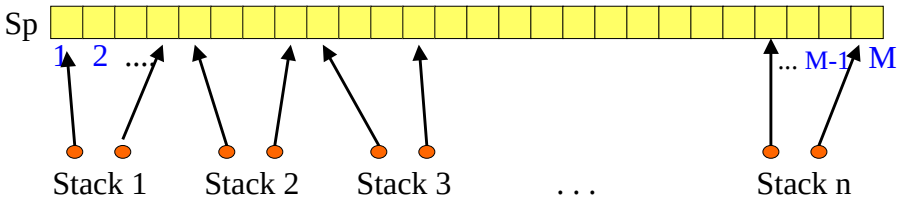
Zu Ringlisten und Doppelschlangen siehe WS 01/02, Einführung in die Informatik I, S. 145 ff

Beispiel 2.3: Implementierung von mehreren Stapeln / Stacks

Aufgabe: Wir wollen eine ***Multistapelverwaltung*** in einem eindimensionalen Feld durchführen, d.h.:

Es sollen n Stacks verwaltet werden. Insgesamt steht hierfür ein linearer Speicher Sp(1..M) zur Verfügung.

Spontane Idee: Jeder Stack erhält gleichviele Speicherplätze M/n:



Diese Verweise werden wir durch Indizes realisieren.

Einfachster Fall: **n = 1**. Es liegt ein einzelner Stack vor. Die Ada-Formulierung hierfür ist ein generisches Paket, z.B.:

```
generic
   M: natural := 2002;                  -- Initialisierung willkürlich
   type element is private;
package stack is
   procedure newstack;                  -- Leeren des Stacks
   function isempty return Boolean;     -- Ist der Stack leer?
   function isfull return Booelan;      -- Ist der Stack voll?
   function top return element;         -- Oberstes Stackelement
   procedure push (x: in element);      -- Füge Element x oben an
   procedure pop;                       -- Lösche oberstes Element
   function length return natural;      -- Aktuelle Stacklänge
   unterlauf, ueberlauf: exception;     -- Ausnahmebehandlung
end stack;
```

Hieran schließt sich der Modulrumpf an:

```
package body stack is
   type speicher is array (1..M) of element;
   Sp: speicher;
   index: integer range 0..M := 0;
   procedure newstack is begin index := 0; end;
   function isfull return Booelan 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 stack;
```


Eine Instanz kann nun lauten:

```
package Ganzzahlkeller is
        new stack (M => 50000, element => integer);
```

Nächster Fall: **n = 2**. Wenn man zwei Stacks auf einem linearen Speicher Sp der Größe 1 .. M unterbringen möchte, so wird man den ersten Stack von 1 an aufwärts und den zweiten Stack mit M beginnend abwärts implementieren.

Aufgabe: Realisieren Sie diesen Fall selbst!

Allgemeiner Fall: **n ≥ 3**. Vorhandener Speicher Sp(1..M).

Hier gibt es mindestens zwei Varianten:

- *Variante 1:* Jeder Stack 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

$$\sum_{i=1}^{n} \text{Max}(i) = M.$$

Dieser Fall ist wie der Fall n=1 zu behandeln, indem überall die Nummer des Stacks hinzugefügt wird und jeder Stack unabhängig von allen anderen ist. Beispielsweise muss es dann zwei Felder Base, Index: **array** (1..n) **of** natural geben mit $0 \le \text{Index}(i)-\text{Base}(i) \le \text{Max}(i)$ für i = 1, 2, ..., n. (Details siehe unten.)

- *Variante 2:* Die Größe der einzelnen Stacks ist nicht vorab beschränkt und alle Stacks zusammen sollen den Speicherplatz der Größe M möglichst gut nutzen. Hier muss es ebenfalls zwei Felder Base, Index: **array** (1..n) **of** natural geben, für die zu jedem Zeitpunkt gilt

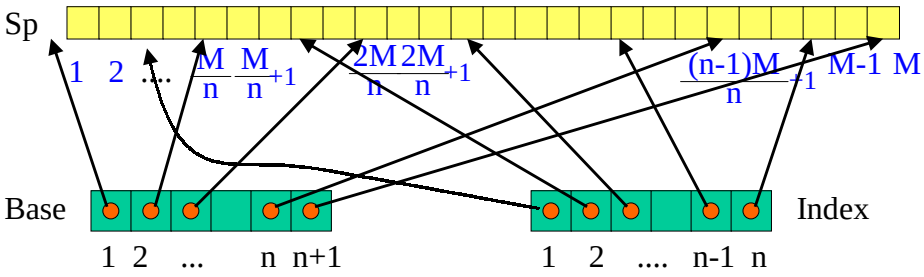
$$\sum_{i=1}^{n} \text{Index}(i)-\text{Base}(i) \le M.$$

Wenn also einer der Stacks überläuft (d.h.: Index(i) = Base(i+1)) und andere Stacks nutzen den ihnen zugewiesenen Bereich noch nicht voll aus, so muss der Speicherplatz neu auf die Stacks verteilt werden.

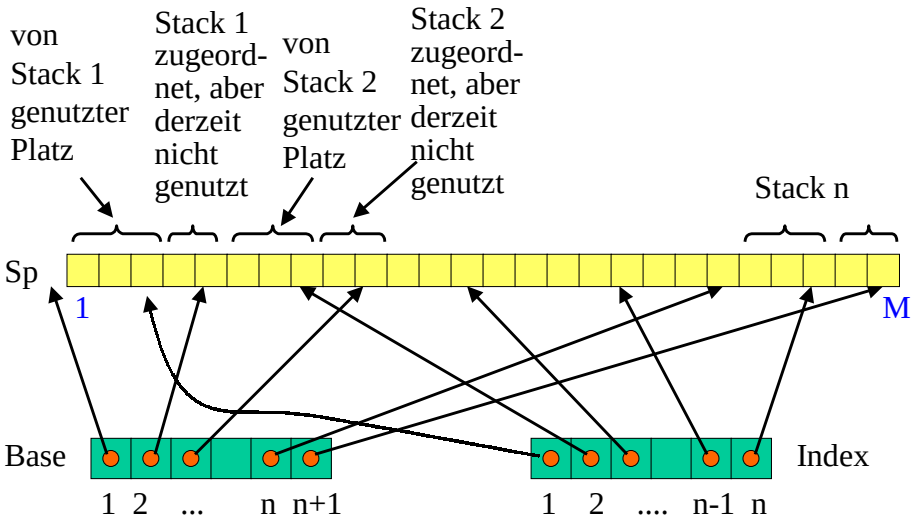
Hier sind wieder mehrere Untervarianten möglich, siehe unten.

Variante 1: Jeder Stack erhält den gleichen Platz der Größe M/n.

Wir realisieren dies über zwei Zeiger bzw. Indizes:
Base(i) zeigt auf den Speicherplatz, der unmittelbar vor dem Bereich für den i-ten Stack liegt;
Index(i) zeigt auf den Speicherplatz, auf dem sich das oberste Element des i-ten Stacks befindet.



Nochmals zur Illustration: Aufteilung des Speichers Sp



Ada Deklarationen hierzu: (MSV = Multistackverwaltung)

```
generic
   M: natural := 20000;            -- willkürlicher Default-Wert
   n: natural;                     -- Anzahl der Stacks, n ≥ 2.
   type Element is private;        -- Datentyp der Stackelemente
package MSV1 is
   type NN is natural range (1..n+1);  -- für Zugriff auf Stacks
   procedure newstack (i:NN);              -- Leeren des Stacks
   function isempty (i:NN) return Boolean; -- Ist der Stack leer?
   function isfull (i:NN) return Booelan;  -- Ist der Stack voll?
   function top (i:NN) return element;  -- Oberstes Stackelement
   procedure push (i: in NN; x: in element);  -- Füge x oben an
   procedure pop (i:NN);           -- Lösche oberstes Element
   function length (i:NN) return natural;  -- Aktuelle Stacklänge
   unterlauf (i:NN), ueberlauf (i:NN): exception;
end MSV1;
```


Paketrumpf hierzu: (MSV = Multistackverwaltung)

```
package body MSV1 is
type Adressen is range 0..M+1;   -- Speicher“adressen“
Sp: array (Adressen) of Element; -- Speicher
Base: array (NN) of Adressen;    -- Beginn der Stacks
Index: array (NN) of Adressen;   -- Aktueller Stand jedes Stacks
procedure newstack (i:NN) is
   begin  Index(i) := Base(i);  end newstack;
function isempty (i:NN) return Boolean is
  begin  return  Base(i) = Index(i);  end isempty;
function isfull (i:NN) return Booelan is
  begin  return  Base(i+1) = Index(i);  end isfull;
function top (i:NN) return element is
  begin  return Sp(Index(i));  end top;
```

Paketrumpf MSV1(Fortsetzung)

```
procedure push (i: in NN; 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:NN) is
  begin  if isempty(i) then raise unterlauf(i);
           else Index(i) := Index(i) - 1; end if; end pop;
function length (i:NN) return natural is
  begin return Index(i)-Base(i); end length;
exception    when ... =>.....
end MSV1;
```


Eine konkrete Instanz könnte dann sein:

```
package Zahlenkellerei is
        new MSV1(M => 50000; n => 10; Element => integer);
use Zahlenkellerei;
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 Multistackverwaltung zusammenbricht, falls irgendein Stack 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:

```
        procedure umordnen (i:NN);  ...
```

Möglichkeit 1:
Schaue nach, ob der rechte oder linke Nachbar des Stacks i noch genügend freien Platz hat und tritt dann die Hälfte dieser Plätze an den Stack i ab.

Möglichkeit 2:
Suche denjenigen Stack j mit maximal viel freiem Platz, d.h., Index(j)-Base(j) ist maximal, und tritt dann die Hälfte dieser Plätze an den Stack i ab. Konkret muss dann der Speicherbereich zwischen den Stacks i und j um q Speicherplätze verschoben werden, wenn q die Hälfte der freien Plätze von Stack j ist.

Möglichkeit 3:
Berechne den Speicherplatz, den jeder Stack bekommen soll, neu, indem jedem Stack eine Mindestzahl an Plätzen und weitere Plätze entsprechend seines bisherigen Wachstums zugewiesen wird, und ordne den Speicher dann komplett um.

Möglichkeit 1:

```
procedure umordnen (i: NN) is
    k: NN;  j, q: Adressen;
begin  k :=  i;
    if (i=1) and (Index(2) < Base(3)) then k := 2;
    elsif (i=n) and (Index(n-1) < Base(n)) then k := n-1;
    elsif Base(i) - Index(i-1) < Base(i+2) - Index(i+1)
        then k := i+1; else k := i-1; end if;
                -- Stack k dient nun als Platz-Lieferant
    if k=i then raise ueberlauf;
    elsif k<i then
        q := (Base(k+1)-Index(k)+1)/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, nur nach oben verschieben; selbst einfügen > end if;
end umordnen;
```

Möglichkeit 2:

```
procedure umordnen (i: NN) is
   k: NN;  j, q: Adressen;
begin  k := 1;          -- Suche Stack 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;
   if Base(k+1) = Index(k) then raise ueberlauf;
   elsif k<i then
       q := (Base(k+1)-Index(k)+1)/2;   -- Hälfte des freien Platzes
       for j in k..i loop
          Base(j) := Base(j) - q; Index(j) := Index(j) - q; end loop;
       for j in Base(k)..Index(i) loop Sp(j) := Sp(j+q); end loop;
   else  < das Gleiche, nur nach oben verschieben; selbst einfügen > end if;
end umordnen;
```


Nachteil der Möglichkeit 1:
Ein Abbruch kann geschehen, obwohl noch irgendwelche anderen Stacks ihren Platz kaum benötigen. Denn man prüft ja nur die benachbarten Stacks ab. Auch kann "umordnen" relativ rasch wieder aufgerufen werden.

Nachteil von Möglichkeit 2:
Eventuell wird die Prozedur "umordnen" nach q Schritten erneut aufgerufen.

Vorteil:
Die Prozedur "umordnen" wird meist schnell abgearbeitet.

Möglichkeit 3: (Garwick-Algorithmus)

- Berechne den insgesamt freien Platz aller Stacks ("sum").
- Berechne den gesamten Zuwachs seit dem letzten Umordnen.
- Verteile 10% des freien Platzes gleichmäßig an alle Stacks.
- 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)), aber man darf nur die positiven Werte hierbei aufaddieren. NewBase(j) ergibt sich aus den Newbase-Werten der darunter liegenden Stacks erhöht um den festen Anteil u, der jedem Stack zusteht, und dem Zuwachs-Anteil.
Dies ergibt folgende Prozedur "umordnen":

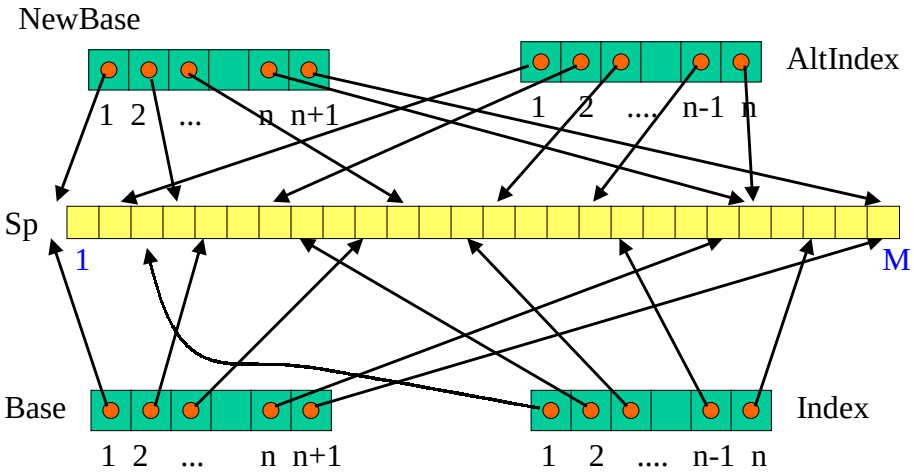
Garwick-Algorithmus: Füge zum "package body MSV1" hinzu:

```
NewBase: array (NN) of Adressen;  -- Neuer Beginn der Stacks
AltIndex: array (NN) of Adressen;  -- Alter Stand jedes Stacks

procedure umordnen (i: NN) is
   k: NN;  j: Adressen;  sum, zuwachs, u: integer; v: float;
   Delta: array (NN) of Adressen;   -- Zuwachs jedes Stacks
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
                -- sum=0 wäre zu knapp wegen Rundungsfehlern
   else  zuwachs := 0;   -- ermittle Zuwächse seit letztem "umordnen"
      for j in NN loop Delta(j) := Index(j) - AltIndex(j);
        if Index(j)>Altindex(j) then Delta(j):=Index(j)-AltIndex(j);
                                     zuwachs:=zuwachs+Delta(j);
        else Delta(j) := 0; end if;      end loop;
```

```
if zuwachs >= 1 then
   u := INTEGER(0.1*FLOAT(sum)/FLOAT(n)  +  0.5);
         -- 10% Anteil (gleichmäßig für alle Stacks)
   v := INTEGER(FLOAT(sum) -
                FLOAT(u)*FLOAT(n))/FLOAT(zuwachs));
else u := INTEGER(FLOAT(sum)/FLOAT(n)); v:=0; end if;
         -- dieser else-Fall darf eigentlich nicht eintreten
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 + Delta (j-1)*v;     end loop;
speicherumordnen;
for j in NN loop AltIndex(j) := Index(j); end loop;
end if;
end umordnen;
```


Garwick-Algorithmus: Verwaltung des Speichers Sp



Unterprozedur zu "umordnen":

```
procedure speicherumordnen is
    m, j, k: NN;
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;
```



```
procedure verschieben (i: in NN) is  -- Unterprozedur zu speicherumordnen
    a: Adressen; d: integer;
begin d := NewBase(i) - Base(i);
        -- d gibt an, um wieviel Stellen Stack i verschoben werden muss
    if (d /= 0) then
        if d > 0 then
           for a in reverse Base(i) .. 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 d<0
        end if;
        Index(i) := Index(i) + d;
        Base(i) := NewBase(i);
    end if;
end verschieben;
```

2.7. Geflechte, die Halde und Freispeicher

Daten oder Objekte lassen sich durch Zeiger miteinander verflechten. Früher sprach man dann von "Geflechten", die im Speicher aufzubauen sind. Heute bezeichnet man diese Strukturen meist als Vernetzungen oder - mathematisch - als *Graphen* (vgl. das folgende Kapitel 3).

Um solche Geflechte oder Vernetzungen aufzubauen, muss in jedem Datenobjekt mindestens ein Zeiger existieren.

Existiert genau ein Zeiger, so kann man nur Listen aufbauen. Ab zwei Zeigern lassen sich stark vernetzte Strukturen realisieren.

Beispiel: Binäre Bäume. Siehe hierzu "Einführung in die Informatik I", WS 01/02, S. 149-152.

Grundsätzlicher Hinweis:

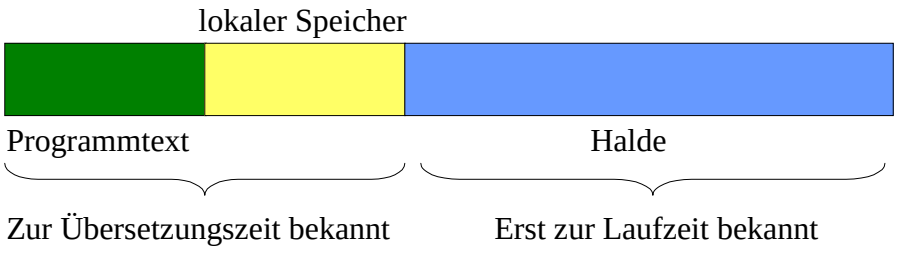
Wir identifizieren in diesem Kapitel 2 stets "Zeiger" und "Adresse eines Speicherplatzes".

Grund: Man beachte, dass heutige Rechner in der Regel einen ein-dimensionalen Speicher besitzen, auf dessen Speicherplätze über einen Index von 0 bis 2^s-1 (für eine natürliche Zahl s) zugegriffen wird. Wenn man Zeiger daher in einem solchen Rechner realisiert, so geschieht dies durch Angabe der Adresse derjenigen Speicherzelle, ab der das Objekt, auf das verwiesen wird, steht.

Wir haben in der Vorlesung bereits mehrfach angedeutet, wie man Programme im eindimensionalen Speicher des Rechners anlegt. Man unterteilt jedes Programm in drei Teile:

Zwei statische Teile (*Programmtext* und *lokaler Speicher*) für alle Bestandteile des Programms, deren Größe (Speicherplatzbedarf) während der Berechnung unverändert bleibt.

Dynamischer Teil (*Halde*) für alle Bestandteile, deren Größe oder Lage im Speicher sich ändern können.



Programmtext: Üblicherweise wird der Programmtext nach der Eingabe in einen Rechner nicht mehr verändert.

Dennoch findet eine Veränderung des Textes statt, sobald ein Unterprogramm aufgerufen wird. Die Bedeutung eines Unterprogrammaufrufs ist nämlich die "Kopierregel"; sie lautet: Ersetze den Aufruf durch den Rumpf des Unterprogramms; modifiziere den Rumpf abhängig von den Parametern, führe den modifizierten Rumpf aus, ersetze ihn am Ende wieder durch den Aufruf und fahre mit der nächsten Anweisung fort.

Es gibt jedoch eine Realisierung dieser Kopierregel, bei der der Programmtext nicht geändert wird, sondern bei der die korrekte Ausführung des Aufrufs durch einen Stack von Daten simuliert wird. In der Praxis wird daher der Programmtext während des Programmablaufs nicht geändert (vgl. Vorlesung Compilerbau).

Lokaler Speicher: Alle vom Programm benutzten Daten müssen letztlich über die Namen der Variablen (bzw. über die "Anker" von Listen und Geflechten) erreichbar sein. Genau diese Variablen, deren benötigter Speicherplatz zur Übersetzungszeit bekannt ist, werden in einem festen Speicherbereich, dem lokalen Speicher des Programms, abgelegt.

Felder mit festen Grenzen werden in der Regel ebenfalls hier gespeichert. Felder, deren Grenzen erst zur Laufzeit berechnet werden, kommen dagegen in die Halde (s.u.), jedoch wird für den Namen und die künftig einzutragende Größe und Lage jedes solchen Feldes Speicherplatz im lokalen Speicher reserviert, über den zur Laufzeit die Verbindung zum Feld hergestellt wird. Weiterhin müssen die Strukturen der deklarierten Datentypen im lokalen Speicher notiert werden (um die Zugriffe zu realisieren).

Halde (engl.: Heap): Teil 1: Dies ist ein Speicherbereich, dessen Größe hinreichend groß ist, um die dynamischen Felder und alle mittels **new** erzeugten Datenobjekte (Zugriff über Zeiger!) abzulegen. Wenn diese Datenobjekte im Laufe der Rechnungen nicht mehr gebraucht werden, sollten sie wieder frei gegeben (in Ada mit Hilfe des pragmas "Controlled" und der Prozedur FREE) und von anderen, neu erzeugten Datenobjekten genutzt werden. Die Verwaltung erfolgt auch über eine Freispeicherliste.

Werden die nicht mehr benötigten Speicherplätze nicht wieder frei gegeben, so liegen nach einiger Zeit in der Halde viele unnütze Datenobjekte herum (= Daten, auf die nicht mehr von irgendeinem lokalen Speicher aus zugegriffen werden kann). So kann die Halde rasch voll werden. Um dann weiterarbeiten zu können, müssen die nicht mehr benötigten Datenobjekte erkannt, ihre Speicherplätze frei gegeben und die Halde in geeigneter Weise umorganisiert werden (Speicherbereinigung).

Halde (engl.: Heap): Teil 2: Die Halde kann/muss ebenfalls zur Unterprogrammverwaltung dienen. Bei jedem solchen Aufruf muss für die lokalen Variablen des Unterprogramms Speicher zur Verfügung gestellt werden, der dynamisch mit weiteren (geschachtelten) Aufrufen wächst oder schrumpft.

Dieses Problem der beiden Speicher für dynamische Daten und für die Unterprogrammverwaltung kann man behandeln wie die Stackverwaltung für 2 Stacks: Von vorne nach hinten legt man in der Halde die dynamischen Daten ab, von hinten nach vorne lässt man die geschachtelten Aufrufe der Unterprogramme aller Programme (=Multistackverwaltung) wachsen.

Wir erläutern dies nun an einem Beispiel mit nur einem Programm.

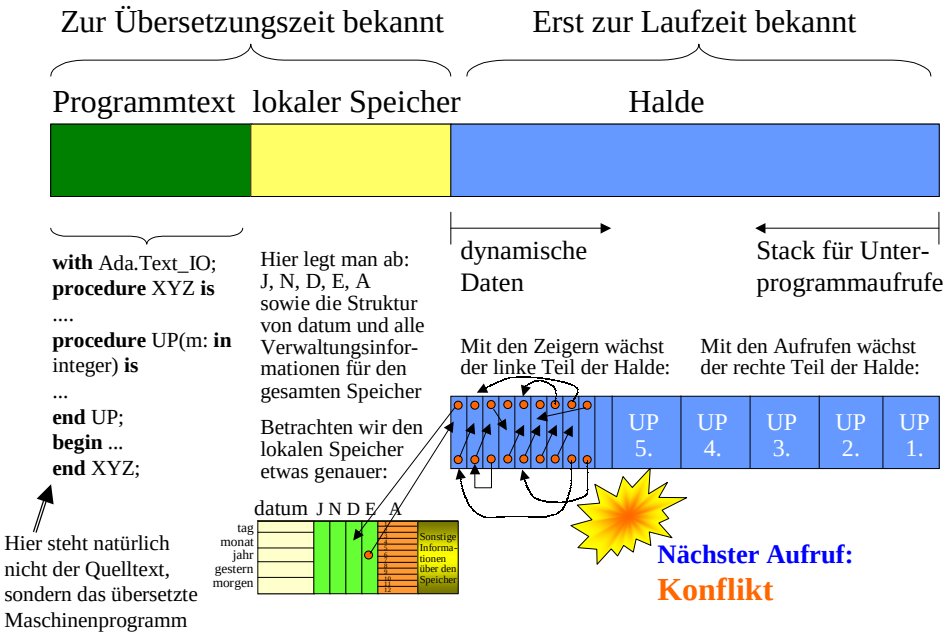

```
with Ada.Text_IO;

procedure XYZ is
type datum is
    record tag: 1..31; monat: 1..12; jahr: 1900..2100;
        gestern, morgen: access datum; end record;
J, N: integer; D: datum; E: access datum;
A: array (1..12) of character;
procedure UP (m: in integer) is
    B: array (1..m) of character;
    begin  ... UP(m+1); ...
    end UP;

begin  get(N); J := 2; D := (13, 5, 2002, null, null);
    E := D; E.gestern := D; E := new datum ...
    UP(N); ...
end XYZ;
```

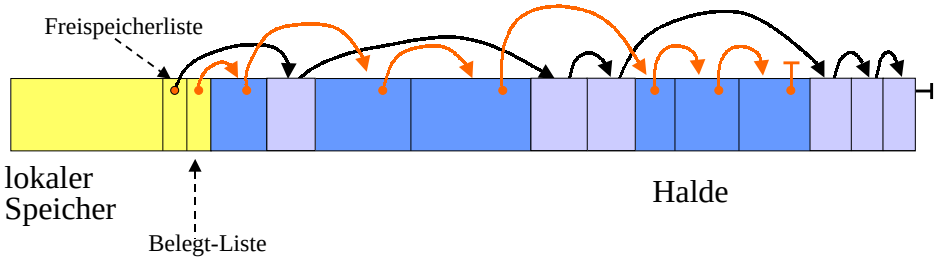
Lokaler Speicher

Halde



2.7.1 Freispeicherverwaltung

Die dynamischen Daten der Halde (linker Teil im vorigen Bild) sollen nun verwaltet werden. Hierzu tragen wir die freien Speicherplätze in eine "Freispeicherliste" ein, aber nicht jede Speicherzelle einzeln, sondern immer ganze "Datenblöcke". Diese können eine variable Größe besitzen. Skizze (hier *nur ein Programm*; prinzipiell nutzen viele Programme die Halde):



Benutzen mehrere Programme die Halde, so werden 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 "größe" 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). Füge hierbei alle Datenblöcke, die nicht mehr benutzt werden, in die Freispeicherliste ein (siehe 2.7.2).

5. Falls auch dies nicht erfolgreich ist, führe einen Austausch der Speicherinhalte mit dem Hintergrundspeicher durch (Stichwort: Seitenaustauschstrategien, Paging; siehe Vorlesungen im Bereich "Betriebssysteme").

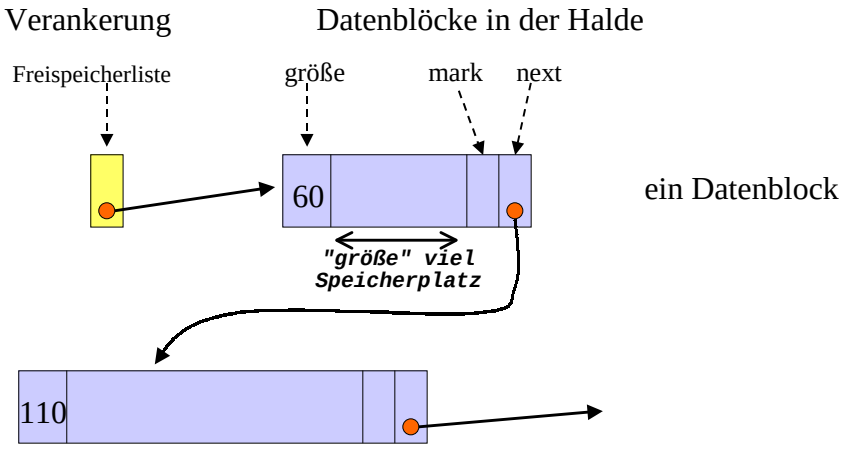
Um diese Aufgaben durchzuführen, muss die Freispeicherliste oft durchlaufen werden, wobei wir die Datenblöcke, die zur Freispeicherliste gehören, markieren, um sie später "erkennen" zu können. Ein Datenblock muss also neben dem Inhalt, den das jeweilige Programm hineinschreibt, mindestens seine Größe, ein Markierungsfeld und den Verweis auf den nächsten freien Datenblock enthalten.

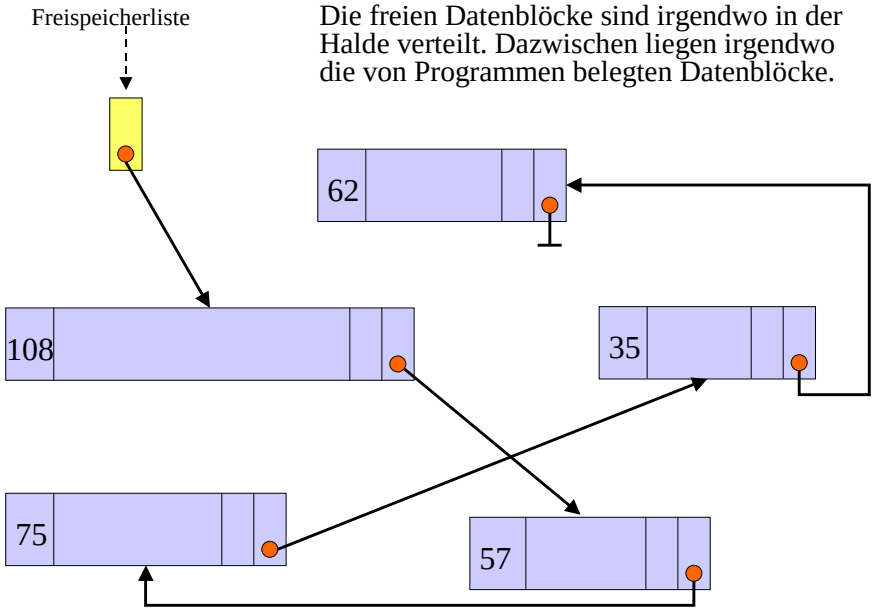
Wir legen daher folgenden Datentyp "Block" für Datenblöcke fest. Es sei eine natürliche Zahl "maxgröße" (= die maximale Größe an Speicherplätzen je Datenblock) vorgegeben:

```
type Block;
type Blockzeiger is access Block;
type Block is record
        größe: 1..maxgröße;
        ... < Komponenten, die insgesamt genau "größe" viele
                Speicherplätze belegen > ...
        mark: Boolean;
        next: Blockzeiger;
        end record;
```

Jeder Block belegt also größe + x viele Speicherplätze in der Halde, wobei x die Zahl der Speicherplätze für "größe", "mark" und "next" bezeichnet. (generic-Formulierung in Ada? Selbst!)

Skizze:





1. Aufgabenstellung: Ein Programm fordert einen Datenblock mit m Speicherplätzen an.

Algorithmus 1: **First Fit**
Gehe die Freispeicherliste durch, bis ein Datenblock D mit größe ≥ m gefunden ist.
Mache hieraus zwei Datenblöcke: Einen mit m+x und einen mit größe-m-x Speicherplätzen (x = Speicherplatz für größe, mark und next, siehe Datentyp Block auf Folie Seite 189).
Füge diese beiden Datenblöcke in die Freispeicherliste anstelle des Blocks D ein.
Klinke den ersten dieser beiden Datenblöcke aus der Freispeicherliste aus und weise ihn dem Programm zu.
Hinweise: Falls der zweite Block "zu klein" ist, vermeide die Aufspaltung in zwei Datenblöcke und weise D dem Programm zu. Falls kein Block D existiert, rufe die Speicherbereinigung auf, vgl. Abschnitt 2.7.2.

Algorithmus 2: **Best Fit**
Gehe die gesamte Freispeicherliste durch und ermittele den kleinsten Datenblock D mit größe ≥ m.
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" 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 zeitaufwändige Speicherbereinigung durchgeführt werden, die zu Wartezeiten bei den "Kunden" führt.

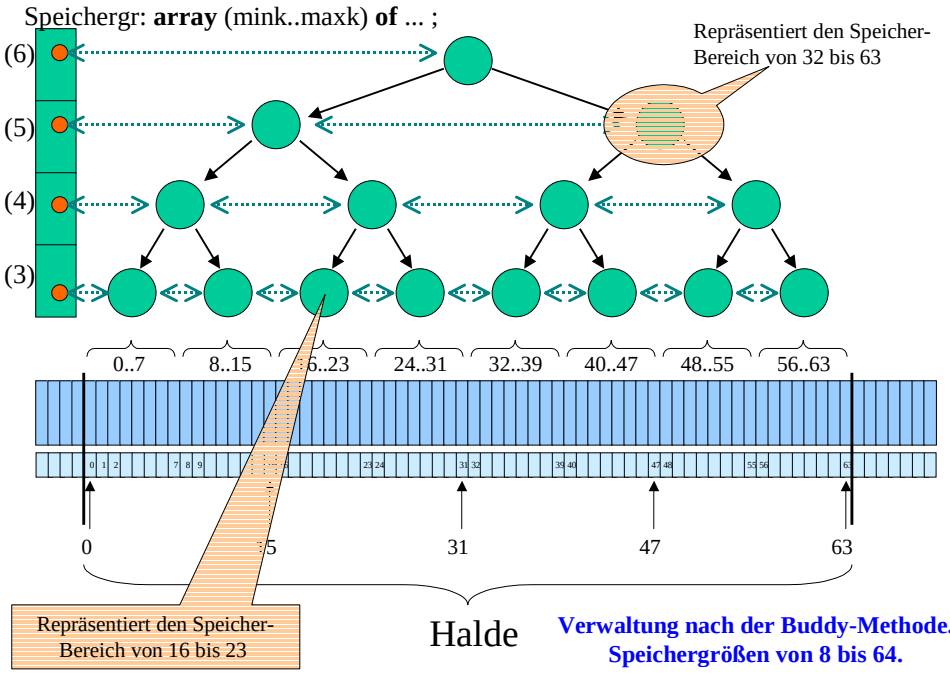
Recht nachteilig ist auch die Zeit, die beim Durchlaufen der Liste verstreicht. Zwei Ideen zur Verbesserung:

- Halte die Freispeicherliste stets nach der Größe der Datenblöcke sortiert. Nachteil: Das Einfügen freigegebener Speicherbereiche ist dann aufwändig.
- Lege einen binären Suchbaum über die Freispeicherliste. Nachteil: Dies kostet relativ viel Speicherplatz. Da solche Suchbäume selbst wieder dynamische Datenstrukturen sind, sollte dieser Platz in der Halde bereit gestellt werden.

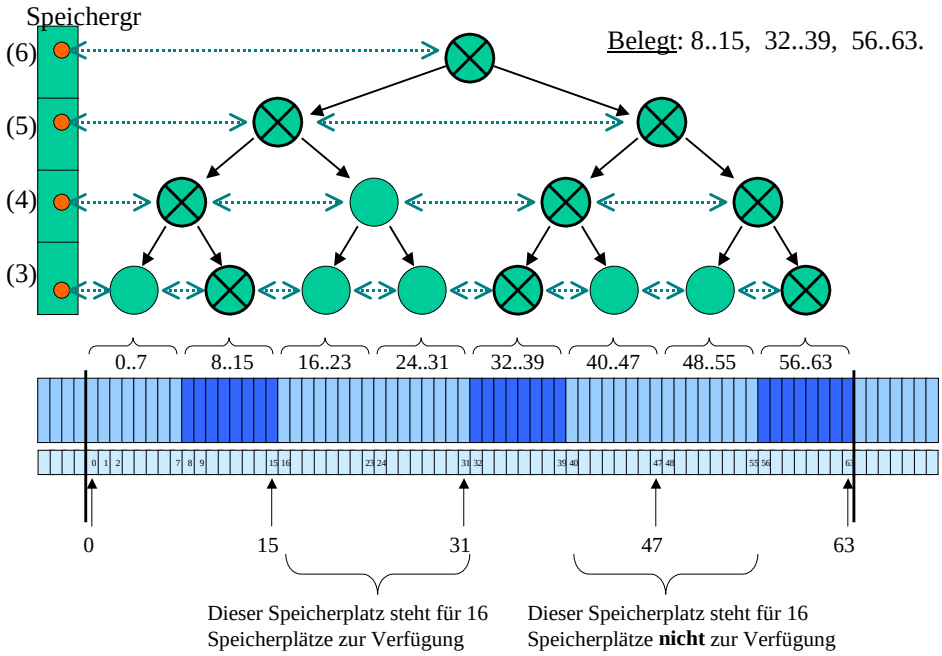
Lassen sich die Vor- und Nachteile gegeneinander auswägen? Wir stellen kurz einen alten Vorschlag vor, der die Situation einigermaßen verbessert, aber oft einige Nachteile beibehält, die sog. Buddy-Methode. Buddy = (engl.) Kamerad, Kumpel.

Idee: Das Verfahren legt einen gleichverzweigten binären Baum über den Speicher (also über die Halde). Das Aufspalten und das Verschmelzen 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`** ≤ k ≤ **`maxk`**. (Man schränkt in der Praxis k ein z.B. zwischen **`mink`** = 9 und **`maxk`** = 20.) 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 die Freispeicherverwaltung nicht aus, sondern geben nur die Vorgehensweisen an.

Anfangs werden alle Speicherbereiche als frei markiert.

Speicheranforderung: Ein Programm fordert einen Datenblock der Größe m an. Es sei $2^{k-1} < m \leq 2^k$.
Die Speicherverwaltung durchläuft dann die Liste, die über Speichergr(k) erreichbar ist.
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.
Wird kein freier Speicherbereich gefunden, so lege die Speicheranforderung in einer Warteschlange des Systems ab, sende dem Programm einen "Wartehinweis" und prüfe später erneut.

Beispiel: Wird in der Situation der Folie auf Seite 197 ein Bereich der Größe 14 angefordert, so wird ab Speichergr(4) die Liste der Speicherblöcke der Größe $2^4 = 16$ durchsucht. Bereits der zweite Block ist frei, so dass der Bereich 16..31 zugewiesen wird.

Speicherfreigabe: Ein Programm gibt einen Datenblock der Größe 2^k wieder frei. 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 Folie auf Seite 197 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 das Verfahren bewährt sich in der Praxis hinreichend gut.
Nachteile: Benachbarte Bereiche, die nicht Buddys sind, können nicht verschmolzen werden, und es entsteht eine interne Fragmentierung, da immer nur Blöcke von der Länge einer Zweierpotenz zugewiesen werden können.

2.7.2 Speicherbereinigung (garbage collection)

Wir nehmen nun an, ein Programm schreibt die Halde mit dynamischen Datenstrukturen, und zwar mit verzeigerten Strukturen, voll. 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.

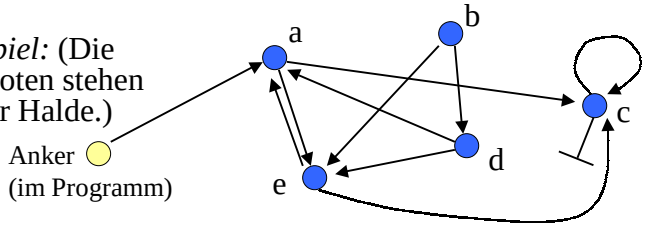
Hierfür verfolgt man alle Zeiger, die vom lokalen Speicher des Programms ausgehen und markiert alle Datenobjekte, die auf diese Weise erreichbar sind. Die nicht-markierten kann man löschen.

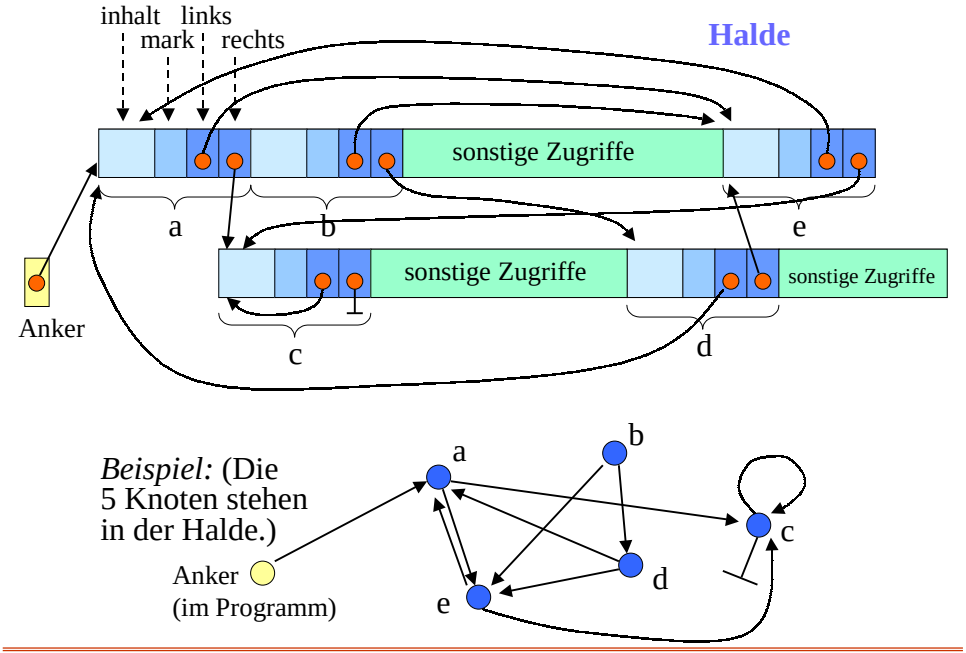
Um das Vorgehen zu erläutern, genügt es, Datenobjekte mit zwei Zeigern zu betrachten, also Objekte des Typs

```
type Knoten;
type Kante is access Knoten;
type Knoten is record
        inhalt:  ...
        mark: Boolean;
        links, rechts: Kante;
end record;
```

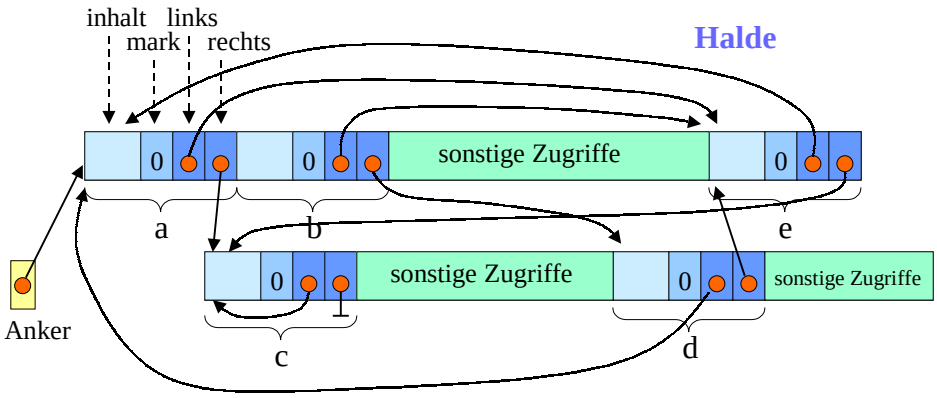
Die Knoten b und d sind vom Programm aus nicht erreichbar.

Beispiel: (Die 5 Knoten stehen in der Halde.)

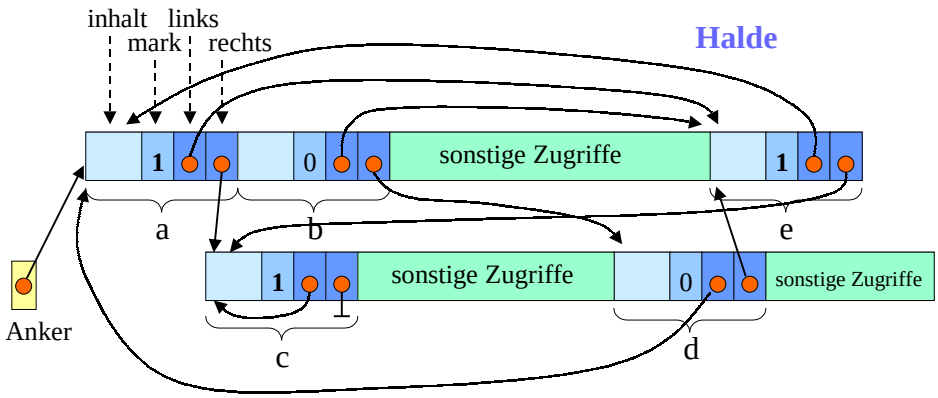




Beispiel: (Die 5 Knoten stehen in der Halde.)



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 hierbei erreichten Knoten mit true (oder einer "1"). Dies geschieht für alle "Anker", die aus dem lokalen Speicher in die Halde verweisen. Dann löscht man alle mit "0" markierten Datenobjekte und schiebt den Speicher geeignet zusammen.



Ergebnis des Durchlaufs: ausgehend von "Anker" wurden alle Zeiger nachverfolgt und die hierbei erreichten Knoten mit einer "1" und die nicht erreichbaren mit einer "0" markiert.

Anschließend eventuell den Speicherbereich neu organisieren.

Hinweis: Wenn man weiß, dass 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 zwei Zeigern hinzugefügt, so müssen die Verweiszähler der Knoten, auf die diese Zeiger zeigen, jeweils um 1 erhöht werden. Wird ein Knoten gelöscht, so muss man die Verweiszähler in den beiden Objekten, auf die die Zeiger links und rechts zeigten, um jeweils 1 erniedrigen. Wird ein Verweiszähler hierbei 0, dann muss man auch in allen nachfolgenden Knoten den Verweiszähler um 1 erniedrigen. Bei dieser Methode 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. (Selbst durchdenken.)

Wir kommen nun zum
Algorithmus zur Markierung der erreichbaren und der unerreichbaren Knoten:

Schritt 1:
Markiere alle Knoten in der Halde mit "false" (bzw. mit 0).

Schritt 2:
Markiere alle Knoten in der Halde, die vom 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 besitzt mindestens einen
         Nachfolger (d.h.: (links /= null) or (rechts /= null))
       and  dieser Knoten ist mit "true" markiert
   then  if  (links /= null) and (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 (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 Verfahrens: 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.

Aufwand dieses Verfahrens?

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.

Nach dem zweiten Durchlauf geschieht das Gleiche mit dem vorletzten Knoten usw.

Wenn n die Zahl der Knoten in der Halde ist, so würde man also $n + (n-1) + (n-2) + ... + 2 + 1 = n \cdot (n+1)/2$ $= O(n^2)$ Schritte ausführen müssen. Wegen $n \approx M/r$ erhält man also ein $O(M^2)$-Verfahren mit konstanter Speicherkomplexität (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.

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 Stack S nutzen kann, dann empfiehlt sich folgender deutlich schnellerer Algorithmus, der die weiter zu verfolgenden Zeiger im Stack S ablegt:

Schritt 1: wie oben.
Schritt 2: Markiere alle Knoten in der Halde, die vom lokalen Speicher direkt erreicht werden können, mit "true" und lege sie (bzw. ihre Adressen) im Stack S ab.
Schritt 3:

```
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 Stack-Verfahrens?

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).

Allerdings bezahlt man diese Schnelligkeit mit dem benötigten Speicherplatz für den Stack S. Dieser kann bis zu n/2 Knoten groß werden. Im Mittel wird man aber deutlich weniger Platz brauchen.

Die uniforme Zeitkomplexität dieses Stack-Verfahren wird also vor allem durch Schritt 1 bestimmt, welcher 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:

Solange noch genügend Platz für den Stack S vorhanden ist, arbeite nach dem Stackverfahren. Sobald der Stack aber überläuft, schalte auf das andere Verfahren um, bis wieder Platz für den Stack da ist.

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 an vorgegebenen Stellen notiert werden müssen.

Nachdem wir nun die erreichbaren Knoten bzw. Datenblöcke mit "true" markiert haben, kann man alle diese in die Freispeicherliste (oder die Buddy-Verwaltung) eintragen und normal weitermachen.

Oft möchte man jedoch den Speicher "zusammenschieben" ("kompaktifizieren") und dabei auch die im Laufe der Rechnungen entstandenen kleinen Fragmente (nicht nutzbaren Speicherbereiche) beseitigen. Dieser Kompaktifizierungs-Algorithmus ist bei beliebiger Verzeigerung einigermaßen aufwändig.

Diese und weitere Fragen zur Verwaltung von Programmen und Daten lernen Sie in Vorlesungen über Betriebssysteme oder auch in speziellen Praktika.

2.8. Zusammenfassung

Behandelte Datenstrukturen (und korrespondierende Kontrollstrukturen):
Feld (array; Vektoren, Matrizen, ...),
Verbund (record; kartesisches Produkt),
Vereinigung (varianter record, union),
Potenzmengen (set of ...)
Folgenbildung (Listen, Zeigertypen; freies Monoid)
Geflechte (Graphen, siehe Kap. 3)

Konkrete Verfahren:
n-dimensionale Felder auf eindimensionale abbilden:
Speicherabbildungsfunktion.
n Stacks möglichst gut verwalten (u.a.: Garwick-Algorithmus)
Freispeicherverwaltung
Speicherbereinigung (garbage collection, ohne Kompaktifizierung)

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). Das gesamte Konzept der Datenstrukturen wurde komplett in Algol 68 (Standard: 1975) zusammengeführt. Dessen "gut verständlicher" Anteil wurde von Nikolaus Wirth in die Sprache PASCAL (1972) eingebracht, die bis heute als "didaktisches Vorbild" für Programmiersprachen gilt.

Mit den ersten Compilern Anfang der 1960er Jahre wurden Speicherabbildungsfunktionen und Optimierungen bei for-Schleifen eingeführt.
Dass sehr allgemeine Datenstrukturen korrekt übersetzbar sind, demonstrierten die Compiler der Sprache SIMULA 67 (ab 1965) und etwas später von PL/1. Probleme bereiteten aber die ganz allgemeinen Datenstrukturen von Algol 68, bei denen kartesische Produkte, Vereinigungen, Potenzmengen, Funktionenbildung usw. beliebig miteinander verknüpft werden können: Die Laufzeitsysteme wurden derart kompliziert, dass jeder Algol-68-Compiler gewisse Einschränkungen machen musste.
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 Multistack, Freispeicher, Bereinigung usw.).