Einführung in die Informatik I (autip)

Dr. Stefan Lewandowski

Fakultät 5: Informatik, Elektrotechnik und Informationstechnik
Abteilung Formale Konzepte
Universität Stuttgart

24.+31.1.+7.+14.2.2007 / Version 22. Januar 2007

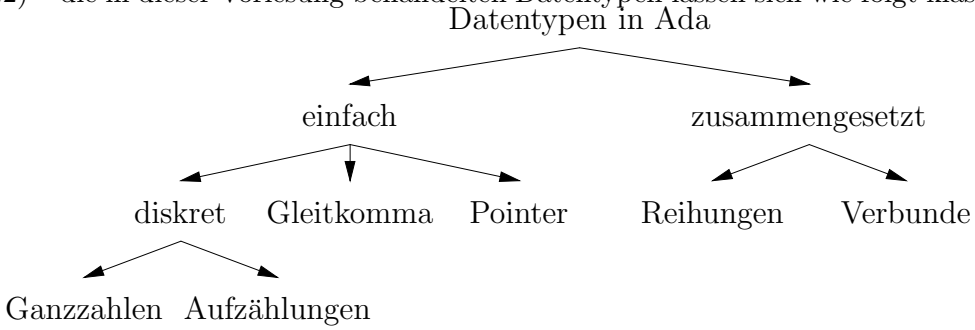
Inhaltsverzeichnis

3 Programmierung – weiterführende Konzepte

- Kritische Bestandsaufnahme unserer bisherigen Ada-Programme:
  - Kontrollstrukturen: Fallunterscheidungen, Schleifen, Rekursion – damit lässt sich prinzipiell alles beschreiben (alles, was berechenbar ist, lässt sich sogar ausschließlich mit Fallunterscheidungen und Rekursion realisieren – Schleifen machen einem das Leben aber deutlich leichter).
  - Datenstrukturen: Wahrheitswerte, Ganzzahltypen, Gleitkommazahlen, Zeichen und Reihungen (Arrays) aus allen diesen – für viele Anwendungen reicht dies zwar, aber ...
- ... wir haben bereits einige Situation gehabt, wo uns diese begrenzten Möglichkeiten einige Kniffe abverlangt haben (z.B. bei Reihungen, deren Index-Bereich erst während der Laufzeit festgelegt wurde). Andere Datenstrukturen konnten und könnten wir hingegen mit den bisherigen Mitteln nicht umsetzen, z.B.
  - Bäume,
  - Reihungen, deren Länge sich dynamisch verändert.

  Ebenso lassen sich Laufzeitfehler bisher zwar zu einem gewissen Grad abfangen (z.B. Überschreitungen der Bereichsgrenzen durch Berücksichtigung der Attribute `'first` und `'last`), andere hingegen nur mit unvertretbar hohem Aufwand und zu Lasten der Lesbarkeit der Programme (z.B. wenn beim Einlesen von Zahlen der Benutzer die Zeichenkette `achtzehn` eingeben würde).

In diesem Rahmen schauen wir uns nochmals die Datentypen an, mit denen wir uns hier beschäftigen (die Ada-Wirklichkeit geht noch darüber hinaus, siehe Ada Reference Manual, Kap. 3.2) – die in dieser Vorlesung behandelten Datentypen lassen sich wie folgt klassifizieren:



3.1 Aufzählungstypen

Bei diesen werden in der Typdefinition alle Werte durch Kommas getrennt und in rundern Klammern eingeschlossen aufgezählt. Werte können dabei Zeichenketten (mit den Einschränkungen wie bei Variablenbezeichnern) oder Zeichen sein, z.B.

- `type Wochentag is (Mon,Die,Mit,Don,Fre,Sam,Son);`
- `type Hexa is ('A','B','C','D','E','F');`

Subtypenbildung ist wie bei Ganzzahlen möglich, also z.B.

- `subtype Werktag is Wochentag range Mon..Fre;`

Folgende Attribute sind bei Aufzählungstypen stets definiert:

- `'first` und `'last` für das erste und letzte Element,
- `'pred` und 'succ für das vorhergehende und nachfolgende Element in der Aufzählung (wir haben somit eine implizite Anordnung der Elemente),
- `'val` und `'pos` zur Umwandlung von und in die natürlichen Zahlen (entsprechend der Position in der Aufzählung beginnend mit der 0)
- sowie die Vergleichsoperationen (basierend auf den zugehörigen `'pos`-Werten) und die Funktionen `'Min` und `'Max`, die über `<Typname>'Min(var1,var2)` (`'Max` analog) verwendet werden können

Beispiel zur Illustration der verwendeten Attribute:

```
for tag in Werktag loop
  Ada.Integer_Text_IO.Put(Wochentag'pos(Werktag'succ(tag)));
end loop;
```

3.2 Verbunde (Records)

Stellen wir uns als Aufgabe, eine Verwaltung der Autip-Studierenden in der Vorlesung Informatik I zu schreiben: Neben dem Namen, Vornamen und der Matrikelnummer sollen die Punkte auf den Aufgabenblättern und in den Testklausuren verwaltet werden. Bisher könnten wir nur einzelne Arrays für die einzelnen Attribute verwenden. Spätestens, wenn wir in dem Programm auch noch andere Studierende verwalten sollen, für die jeweils andere zusätzliche Daten gespeichert werden müssten, wird das Programm unübersichtlich.

Verbunde bieten die Möglichkeit, mehrere Datentypen unter einem Namen zusammenzufassen. Wir könnten so einen Typ `AutipStudi` definieren:

```
type aufgabenblaetter is array (1..14) of natural;
type testklausuren is array (1..3) of natural;

type AutipStudi is record
  name:   string(1..30);
  vorname:string(1..20);
  matrnr: positive;
  punkte: aufgabenblaetter; -- sogenannte "anonymous arrays" sind bei Ada
  tests:  testklausuren;    -- in records nicht erlaubt, daher so ...
end record;
```

Die einzelnen Bezeichner innerhalb eines Verbunds nennt man *Selektoren*, der Zugriff erfolgt über die *Punkt-Notation*. Sei `einStudi:AutipStudi;` mit unserem so definierten Typ deklariert, so können wir z.B. über `einStudi.matrnr:=2581114;` die Matrikelnummer setzen. Definieren wir uns für die Verwaltung ein Array

```
autips : array (1..42) of AutipStudi;
```

um die Leistungen der Studierenden in der Informatik I in diesem Semester zu verwalten, so können wir nun z.B. dem elften Studenten auf dem dritten Aufgabenplatt 17 Punkte zuweisen: `autips(11).punkte(3):=17;`.

Neben der Zusammenfassung von verschiedenen Daten zu einem neuen Datentyp sind Verbunde auch im nun folgenden Abschnitt über Pointer und deren Anwendung notwendig.

3.3 Pointer, Listen, Bäume und Graphen

3.3.1 Einführendes Beispiel in Pointer und Listen

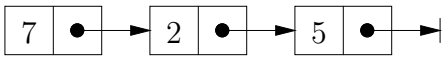
Greifen wir nochmal das Beispiel der Autipsverwaltung auf. Die Feldgröße (also die Anzahl der Studierenden) steht anfangs nicht unbedingt fest. Wir können prinzipiell zwar ein Array mit offenen Indexgrenzen (`range <>`) definieren, aber sowie wir das Array mit Daten füllen wollen, ist die Größe unveränderlich – Nachzügler, Quereinsteiger, Abbrecher können nicht mehr sinnvoll verwaltet werden.

Wir benötigen für solche Fälle flexiblere Möglichkeiten, um solche Mengen, deren Kardinalität sich dynamisch ändern kann, programmiertechnisch in den Griff zu bekommen.

In diesem Zusammenhang betrachten wir Mengen als Listen. Was ist nun eine Liste aus Sicht der Programmierung?

- Ausgehend von einem ersten Element (*Anker* genannt), benötigen wir lediglich einen Verweis, wo wir das nächste Element der Menge finden können.
- Dieses nächste Element braucht wieder einen Verweis auf ein weiteres Element (oder den Hinweis, dass es kein weiteres Element gibt)

Eine Liste aus den Elementen 7, 2 und 5 sieht anschaulich dann so aus:



In Ada ist die Typdefinition sinngemäß wie folgt:

```
type Listenelement is record
  Inhalt : Element_Typ; -- z.B. Integer
  naechstes : Verweis_auf_Listenelement_Typ; -- kommt gleich
end record;
```

Der Verweistyp ist nicht vom Typ `Listenelement`! Verweise werden in Ada mit dem Schlüsselwort `access` gebildet. Versuchen wir eine Integer-Liste zu definieren:

```
type Liste is access Listenelement;
type Listenelement is record
  Inhalt : Integer;
  naechstes : Liste;
end record;
```

Der Versuch dieses zu übersetzen endet mit einem Syntax-Fehler, da `Listenelement` in der ersten Zeile noch nicht bekannt ist. Setzt man die Zeile ans Ende, ist aber im Record der Typ `Liste` noch unbekannt. Hier kann man sich in Ada so behelfen, dass man dem Compiler schon mal mitteilt, dass es einen Typ mit einem bestimmten Namen gibt, dieser aber erst später

genauer definiert wird (dieses geht analog übrigens auch für Prozeduren und Funktionen, wenn diese sich wechselseitig rekursiv aufrufen[1]). Wir fügen also vor die erste Zeile noch ein

```
type Listenelement;
```

ein. Nun haben wir zwar Typen definiert, es fehlen aber noch die Variablen bzw. Elemente der Liste. Das erste Element wäre kein Problem:

```
anker : Listenelement := (42,null); -- null = leerer Verweis = kein weiteres Element
```

... und dann? Da man nicht weiß, wieviele Variablen vom Typ Listenelement denn benötigt werden, geht es so sicherlich nicht (ganz davon abgesehen, dass wir noch nicht wüssten, wie wir zu der Variable den Zeiger bekommen, der auf diese Variable verweist).

Lösung: Variablen, auf die mit Zeigern zugegriffen werden sollen, werden in Ada grundsätzlich mit dem Schlüsselwort `new` erzeugt – dies gilt insbesondere auch für das erste Element der Liste, den Anker (im obigen Bild ist also noch ein Pfeil auf das erste Element hinzuzufügen). Wie man auf „normale Variablen“ auch über Zeiger zugreifen kann, behandeln wir am Ende dieses Abschnitts.

Wir schreiben nun eine kleine Prozedur, die zunächst solange Zahlen zu einer Liste hinzufügt bis der Benutzer eine 0 eingibt und dann die Liste der Zahlen wieder ausgibt.

```
type Listenelement;
type Liste is access Listenelement;
type Listenelement is record
  Inhalt : Integer;
  naechstes : Liste;
end record;

procedure ListenDemo is
  anker : Liste := null; -- zu Beginn ist die Liste leer
  verweis : Liste -- zeigt immer auf das gerade betrachtete Element
  wert : Integer -- für die einzulesenden/auszugebenden Zahlen
begin
  Get(wert);
  if wert /= 0 then
    anker := new Listenelement'(wert,null);
```

[1] Wir geben hier als Beispiel einen etwas künstlichen (und noch dazu ineffizienten) Algorithmus an, der für natürliche Zahlen entscheidet, ob sie gerade oder ungerade sind. Die Idee: eine Zahl `x` ist gerade genau dann, wenn `x-1` ungerade ist. Wir geben zwei boolesche Funktionen `odd` und `even` an, die diese Idee umsetzen.

```
function odd(x:natural) return boolean;

function even(x:natural) return boolean is
begin
  if x=0 then return true; else return odd(x-1); end if;
end;

function odd(x:natural) return boolean is
begin
  if x=0 then return false; else return even(x-1); end if;
end;
```

Wichtig ist dabei, dass die Parameterliste in der Ankündigung der Funktion `odd` dieselben Parameter hat, ansonsten interpretiert Ada dies als eigene Funktion mit gleichem Namen aber anderen Parametern. Machen Sie sich bei diesem Beispiel auch klar, wie hier die rekursiven Aufrufe geschachtelt sind und wie das Ergebnis zurückgegeben wird.

```
    verweis := anker;
    Get(wert);
    while wert /= 0 loop
      verweis.naechstes := new Listenelement'(wert,null);
      verweis := verweis.naechstes;
      Get(wert);
    end loop;
  end if; -- war der erste Wert bereits gleich 0, so bleibt die Liste leer
  verweis := anker;
  while verweis /= null loop
    Put(verweis.Inhalt);
    verweis := verweis.naechstes;
  end loop;
end;
```

Beim Einlesen wird in jedem Schleifendurchlauf jeweils vom letzten Element aus ein Zeiger auf ein neu erzeugtes Element gesetzt, wobei letzteres mit dem eingelesenen Wert und dem leeren Verweis zur Markierung des Endes der Liste initialisiert wird. Die Variable `verweis` ist nötig, da wir sonst später den Zeiger auf das erste Element der Liste nicht mehr rekonstruieren könnten.

3.3.2 Grundoperationen auf Listen

Obiges ist ein Spezialfall für das Einlesen von Listen. Im Allgemeinen werden bei einer Datenverwaltung nicht alle Daten auf einmal eingelesen, sondern es wechseln sich verschiedene Operationen auf dem Datenbestand ab. Wir betrachten hier folgende Grundoperationen auf Listen (`list` ist hierbei stets ein Zeiger auf ein Listenelement, in der Praxis in der Regel auf das erste Element der Liste):

- `function empty_List return Liste` – erzeugt eine leere Liste
- `function is_empty(list : Liste) return Boolean` – gibt `true` zurück, falls die Liste leer ist, `false` sonst
- `procedure add_to_front(list : in out Liste; elem : Element_Typ)` – fügt `elem` am Beginn der Liste ein
- `procedure add_to_end(list : in out Liste; elem : Element_Typ)` – fügt `elem` am Ende der Liste ein
- `procedure add_sorted(list : in out Liste; elem : Element_Typ)` – fügt in eine vorher sortierte Liste `elem` so ein, dass danach die Liste wieder sortiert ist
- `function first_of_list(list : Liste) return Element_Typ` – gibt das erste Element der Liste zurück – bei Aufruf, darf die Liste nicht leer sein (sonst Laufzeitfehler)
- `procedure delete(list : in out Liste; elem : Element_Typ; deleted : out Boolean)` – löscht das erste Vorkommen von `elem` in `list`, der Ausgangsparameter `deleted` wird `false` gesetzt, wenn `elem` nicht in `list` vorkam, sonst auf `true`

- `procedure delete_all(list : in out Liste; elem : Element_Typ; deleted : out Natural)` – löscht alle Vorkommen von `elem` in `list` und zählt in `deleted`, wieviel Elemente gelöscht wurden
- `function copy_List(list : Liste) return Liste` – kopiert die Liste
- `function length_of_List(list : Liste) return Natural` – zählt die Elemente in der Liste
- `function search_in_List(elem : Element_Typ; list : Liste) return Liste` – liefert den Zeiger auf das erste Vorkommen des Elements in der Liste (ggf. `null`-Zeiger, falls das Element nicht vorkommt)

Nun zu den einzelnen Funktionen und Prozeduren:

- Zum Erstellen einer leeren Liste brauchen wir als Rückgabewert einen Verweis auf eine leere Liste – dies ist gerade der `null`-Zeiger, somit ist
  `function empty_List return Liste is begin return null; end;`
- Ähnlich simpel ist die Abfrage, ob eine Liste leer ist oder nicht – entweder ist der Parameter der `null`-Zeiger oder nicht
  ```
  function is_empty(list : Liste) return Boolean is
  begin return list=null; end;
  ```
- Beim Einfügen am Beginn der Liste erhalten wir einen neuen Anker, somit muss `list` ein Durchgangsparameter sein
  ```
  procedure add_to_front(list : in out Liste; elem : Element_Typ) is
  begin
    list := new Listenelement'(elem,list);
  end;
  ```
  – dies ist fast schon etwas tricky – auf die bisherige Liste `list` wird bei der Erzeugung über den `naechstes`-Zeiger verwiesen und somit das neue Element vorne angestellt. Dies funktioniert unabhängig davon, ob `list` vorher leer war oder nicht (selbst überlegen).
- Beim sortierten Einfügen und Einfügen am Ende benötigt man eine Fallunterscheidung, ob die Liste vorher leer war oder nicht – im ersteren Fall erzeugt man eine neue Liste mit nur einem Element, im anderen muss die Liste bis zum letzten Element, das noch kleiner war, bzw. bis zum letzten Element der Liste durchlaufen werden. Dazu merkt man sich entweder über einen Zeiger jeweils das letzte Listenelement oder nutzt den Zugriff über `list.naechstes` – Übungsaufgabe.
- Die Rückgabe des ersten Elements der Liste erfordert, dass die Liste bei Aufruf nicht leer sein darf – man gibt einfach den Inhalt des Records zurück, auf den der Parameter `list` zeigt. Wie man ggf. bei einer leeren Liste den Laufzeitfehler abfangen kann, werden wir uns gegen Ende des Semesters anschauen.
- Das Löschen ist de facto nur ein „Ausklinken“ des Elements aus der Liste (ein explizites Löschen des Elements und Freigeben des Speichers erledigt Ada ggf. selbstständig – wir gehen auf die Details hier nicht weiter ein). Insbesondere beim Löschen aller Vorkommen eines Elements muss man darauf achten, dass man jeweils die richtigen Zeiger neu setzt – Übungsaufgabe.

- Das Kopieren sieht zunächst so aus, als wenn dies wieder sehr einfach ist:
  `function copy_List(list : Liste) return Liste is`
  `begin return list; end;`
  Vorsicht! So kopiert man lediglich den Anker der Liste. Wird die erste Liste verändert, so würde sich jetzt die zweite Liste automatisch mitverändern. Unter Umständen (z.B. Löschen des ersten Elements einer Liste) sind die Folgen noch unübersichtlicher. Man muss also stattdessen die Liste durchlaufen und jeweils Kopien der Elemente neu erzeugen – Übungsaufgabe.

- Zur Bestimmung der Anzahl der Elemente muss man lediglich die Liste analog zum obigen Beispielprogramm durchlaufen und die Anzahl mitzählen. Beim Suchen eines Elements muss man lediglich beim Vergleich des aktuellen Listenelements mit dem gesuchten Element drauf achten, dass man am Ende der Liste nicht auf den Inhalt des leeren Elements zugreift (dies lässt sich elegant mit der Booleschen Verknüpfung `and then` lösen, Abbruchbedingung in der `while`-Schleife `verweis /= null and then verweis.inhalt /= elem` – Details selbst überlegen).

Noch eine Randbermerkung zu Zuweisungen bei Zeigern: Seien `Ref1` und `Ref2` zwei Zeiger auf zwei verschiedene Speicherstellen. Die Semantik einer Zuweisung `Ref1 := Ref2;` birgt ein kleines Problem – in Ada wird hier nur der Zeiger kopiert, d.h. `Ref1` und `Ref2` verweisen danach auf dieselbe Speicherzelle. Wollte man stattdessen die Inhalte kopieren, so schreibt man `Ref1.all := Ref2.all;` – während `Ref1` vom Typ `Liste` ist, so ist `Ref1.all` vom Typ `Listenelement` (also von dem Typ, auf den `Ref1` verweist). Achten Sie also darauf, ob Sie nur die Zeiger kopieren wollen oder die Inhalte, auf die verwiesen werden.

Das gleiche gilt bei Vergleichen: `Ref1 = Ref2` überprüft, ob beide Zeiger auf dieselbe Speicherstelle zeigen, der Ausdruck wird sich also stets zu `false` auswerten, wenn die referenzierten Speicherstellen verschieden sind – auch, wenn die Inhalte dieser Speicherstellen identisch sind. Will man die Inhalte vergleichen, so muss man `Ref1.all = Ref2.all` benutzen.

Und noch ein kleiner hilfreicher Hinweis: Die Erzeugung kurzer Listen zu Testzwecken lässt sich durch Schachtelung recht kurz schreiben – eine Liste mit den drei Elementen 3, 7 und 42 wird z.B. durch
`new Listenelement'(3, new Listenelement'(7, new Listenelement'(42,null)));`
erzeugt – so erspart man sich ggf. das ständige erneute Eintippen der Zahlen.

3.3.3 Varianten der linearen Liste

- Eine Liste, in der jedes Element nur auf seinen Nachfolger verweist, heißt *einfach verkettete Liste* (dies ist die Form der Liste, wie wir sie hier eingeführt haben).

- Eine Liste, bei der das letzte Element nicht auf `null`, sondern auf das erste Element der Liste (zurück) verweist, heißt *zyklische Liste* – es kann nun jedes Element als Anker fungieren.

- Eine Liste, bei der jedes Element zwei Verweise besitzt, einen auf den Vorgänger und einen auf den Nachfolger, heißt *doppelt verkettete Liste* – wird oft verwendet, wenn Listen in beiden Richtungen durchlaufen werden müssen.

- Es gibt natürlich auch die Kombination *zyklische doppelt verkettete Liste*.

Überlegen Sie selbst, wie sich die Datenstrukturen und Prozeduren / Funktionen für obige Varianten verändern. Überlegen Sie Anwendungen, wo die eine oder andere Variante von Vorteil ist.

3.3.4 Datenstruktur Keller (Stack) und Schlange (Queue)

Wir haben gelernt, dass beim Aufruf von Prozeduren lokale Variablen erzeugt und später beim Verlassen der Prozeduren wieder gelöscht werden. Dieses kann man sich als einen Stapel vorstellen, auf den bei jedem Aufruf, die neu erzeugten Variablen oben auf den Stapel gelegt werden und die jeweils zuletzt erzeugten Variablen auch wieder als erste gelöscht werden, also als erste oben vom Stapel wieder verschwinden.

Um dieses Verhalten eines sogenannten *Kellers* (engl. Stack) zu simulieren, benötigen wir exakt folgende 5 Funktionen:

- `"Empty"` – das Leeren des Kellers,
- `"is_Empty"` – Abfragen, ob der Keller leer ist,
- `"Push"` – Hinzufügen eines Elements auf dem Keller,
- `"Top"` – Ausgabe das obersten Elements des Kellers,
- `"Pop"` – Entfernen des obersten Elements des Kellers.

Wir sehen sofort, dass wir dieses mit einfach verketteten Listen implementieren können, indem wir Elemente immer vorne an der Liste einfügen bzw. entfernen.

Wir werden in den Übungen einfache Anwendungsbeispiele behandeln und im Rahmen von Graphen auch nochmal auf den Keller zurückkommen.

Sehr verwandt mit dem Keller ist die *Schlange* (engl. Queue). Sie unterscheidet sich nur dadurch, dass beim Hinzufügen (`"Enter"`) das Element am Ende der Liste eingefügt wird. Das `"Top"` und `"Pop"` nimmt weiterhin die Elemente vom Anfang der Liste – diese beiden Aufrufe werden bei Schlangen in der Regel `"First"` und `"Remove"` genannt.

Auch auf Schlangen werden wir im Rahmen von Graphen nochmals kurz zurückkommen.

3.3.5 Binärbäume und Baumdurchläufe

Mit Zeigern lassen sich nicht nur Listen verarbeiten. Bäume und speziell Binärbäume sind in der Informatik sehr wichtig, wir wollen nun überlegen, wie wir diese programmieren können.

Dazu hier nochmals eine (leicht abgewandelte) Definition für Binärbäume: Ein leerer Baum ist ein Binärbaum. Sind B_1 und B_2 zwei unter Umständen leere Binärbäume, so ist auch der Knoten w_0 als Wurzel mit dem linken Unterbaum B_1 und dem rechten Unterbaum B_2 ein Binärbaum – ein Binärbaum ist also ein Knoten, der zwei (u.U. leere) Nachfolger hat. Die Datenstruktur ist somit sehr ähnlich der unserer Liste:

```
type Knoten;
type BinBaum is access Knoten;
type Knoten is record
  Inhalt : Element_Typ;
```

```
  links,rechts : BinBaum;
end record;
```

Ist `Element_Typ` ein geordneter Datentyp (also einer, auf dem eine "<"-Relation definiert ist, z.B. `Integer`), so heißt solch ein Binärbaum *Suchbaum*, wenn er zusätzlich folgende Eigenschaft erfüllt:

> Für jeden Knoten w gilt, dass alle Knoten des linken Unterbaums einen Wert haben, der kleiner als der Wert von w ist, und alle Knoten des rechten Unterbaums einen Wert haben, der größer als der Wert von w ist.

Zumindest in der Anwendung fordert man bei Suchbäumen, dass die Elemente paarweise verschieden sind. Will man gleiche Elemente zulassen, so müssen alle Knoten des rechten Unterbaums von w einen Wert haben, der größer oder gleich dem Wert von w ist.

Wir überlegen uns nun auch für Suchbäume (mit paarweise verschiedenen Elementen), welche Prozeduren und Funktionen nötig und sinnvoll sind:

- `function empty return BinBaum` – erzeugt einen leeren (Such-)Baum
- `function is_empty(baum : BinBaum) return Boolean` – gibt `true` zurück, falls der Baum leer ist, `false` sonst
- `procedure insert(baum : in out Liste; elem : Element_Typ)` – fügt in einen Suchbaum das Element `elem` gemäß den Eigenschaften eines Suchbaums ein
- `function is_in(elem : Element_Typ; baum : BinBaum) return Boolean` – liefert `true` oder `false` abhängig davon, ob `elem` im `baum` vorkommt
- `procedure delete(baum : in out BinBaum; elem : Element_Typ; deleted : out Boolean)` – löscht `elem` in `baum`, der Ausgangsparameter `deleted` wird `false` gesetzt, wenn `elem` nicht im `baum` vorkam, sonst auf `true`
- `function number_of_elements(baum : BinBaum) return Natural` – zählt die Elemente im Baum

Was ist bei der Implementierung zu beachten?

- Das Erzeugen eines leeren Binärbaums und Überprüfen, ob ein Binärbaum leer ist, sind identisch den Funktionen bei einfach verketteten Listen – also `return null;` bzw. `return baum=null;`.
- Beim Einfügen wird automatisch überprüft, ob das Element schon im Baum vorkommt: Wir beginnen bei der Wurzel – ist das einfügende Element gleich der Wurzel, so ist nichts zu tun – ist es kleiner, so fügen wir das Element (rekursiv) im linken Unterbaum ein (ist es größer, so im rechten Unterbaum). Ist der Unterbaum leer, so wird das Element die Wurzel dieses Unterbaums – Übungsaufgabe.
- Das Überprüfen, ob ein Element im Baum vorkommt, geht analog vor: Ist das Element gleich der Wurzel, so war die Suche erfolgreich – ansonsten sucht man im linken oder rechten Unterbaum weiter bis man entweder das Element findet oder auf einen leeren Unterbaum trifft – Übungsaufgabe.

- Das Löschen in Suchbäumen ist etwas komplizierter, da wir keine Lücken im Baum hinterlassen können (und wollen). Wir werden darauf in der Vorlesung „Einführung in die Informatik II“ noch ausführlich eingehen (dort werden verschiedene Varianten von Suchbäumen mit deren Eigenschaften noch genauer untersucht werden).

- Der Algorithmus zur Bestimmung der Anzahl der Elemente in einem Suchbaum demonstriert wieder einmal die Mächtigkeit und Eleganz der Rekursion. Die Anzahl der Elemente im gesamten Baum ist die Summe der Anzahlen der Elemente in den beiden Unterbäumen plus eins (für die Wurzel), somit:

```
function number_of_elements(baum : BinBaum) return Natural is
begin
  if baum = null then
    return 0; -- leerer Baum
  else
    return 1 + number_of_elements(baum.links) +
               number_of_elements(baum.rechts);
  end if;
end;
```

Randbemerkung: Der Durchlauf durch den Baum (incl. der `null`-Zeiger) ist identisch dem Baum der rekursiven Aufrufe.

Solche Baumdurchläufe sind (speziell bei Binärbäumen) in der Informatik sehr häufig anzutreffen. Betrachten wir nochmals die Reihenfolge, in der in obiger Funktion der Baum durchlaufen wird – als Prozedur formuliert ist der Ablauf wie folgt:

```
procedure baumdurchlauf(baum : BinBaum) is
begin
  if baum /= null then
    baumdurchlauf(baum.links);
    baumdurchlauf(baum.rechts);
  end if;
end;
```

Ohne Beachtung der Knoteninhalte hat so ein Baumdurchlauf natürlich wenig Sinn. Die Prozedur `baumdurchlauf` wird durch die Rekursion für jeden Knoten genau einmal aufgerufen. Wollen wir die Elemente des Suchbaums ausgeben, so gibt es in obiger Prozedur dafür drei Möglichkeiten:

- vor dem Aufruf von `baumdurchlauf(baum.links);`
- nach dem Aufruf von `baumdurchlauf(baum.rechts);`
- zwischen den beiden rekursiven Aufrufen

Die Reihenfolgen, die dadurch erzeugt werden, nennt man

- *Preorder-Durchlauf* – wenn die Ausgabe (oder Bearbeitung) des Knotens vor dem ersten rekursiven Aufruf geschieht,
- *Postorder-Durchlauf* – wenn diese nach dem zweiten rekursiven Aufruf geschieht und

- *Inorder-Durchlauf* – wenn der Knoten zwischen den beiden rekursiven Aufrufen ausgegeben (oder bearbeitet) wird.

Wir werden eine kleine Anwendung dieser Suchbäume und der Baumdurchläufe in der Übung kennenlernen.

Weitere Eigenschaften und wie man z.B. aus einem Preorder-Durchlauf wieder den Suchbaum rekonstruiert, werden wir im kommenden Semester vertiefen.

Eine wichtige Anmerkung zum Schluss dieses Abschnitts: Dass die Rekursion in diesem Abschnitt über Bäume so häufig vorkommt, ist kein Zufall. Grundsätzlich gilt, dass die Art der Definition einer Datenstruktur und deren Bearbeitung im Programm stark korrelieren. Ist die Definition der Binärbäume rekursiv, so wird man auch bei der Bearbeitung von Binärbäumen mit Rekursion genau diese rekursive Definition nachvollziehen. (Bei der Suche nach einem Element in einem Suchbaum kann man zwar die Rekursion leicht durch eine `while`-Schleife umgehen, aber auch da ist die rekursive Implementierung zumindest eleganter und ist näher an der zugehörigen Datenstruktur.)

Analog dazu wird man Arrays in der Regel mit `for`-Schleifen bearbeiten, da die Definition von Arrays ebenso wie die `for`-Schleifen auf festen Zahlenbereichen basieren.

Und ebenso werden Listen in der Regel mit `while`-Schleifen bearbeitet, da die Definition von Listen mit der linearen Anordnung und der nicht vorab festgelegten Anzahl der Elemente dem Wesen nach mit `while`-Schleifen übereinstimmen.

Die Wahl der Programmiertechnik und Kontrollstrukturen hängt also sehr nah mit den verwendeten Datenstrukturen zusammen.

3.3.6 Graphen

"Bäume" sind häufig in der Realität anzutreffen, sei es als Hierarchien in der Universität oder im Beruf, bei realen Bäumen, beim S-Bahn-Netz der Stadt Stuttgart, bei einer Kapiteleinteilung mit Unterkapiteln in einem Buch oder anderswo.

Noch häufiger wird man aber auf Geflechte stoßen, die zusätzlich zu einem Baumgerüst noch Querverweise haben, oder die garkeine Ahnlichkeit mehr mit Bäumen haben, wie z.B. Straßenverbindungen (mit Kreuzungen als Knoten und Straßenstücken als Verbindung zwischen den Knoten) oder das Netzwerk der Verlinkungen des Internets.

Wir wollen uns hier nun ebenfalls überlegen, wie wir solche Geflechte, in der Informatik *Graphen* genannt, im Rechner modellieren können und neben dem nötigen Vokabular uns auch hier – analog zu den Pre-, In- und Postorder-Durchläufen – überlegen, wie man systematisch Graphen durchlaufen kann. Auch dies ist hier in erster Linie als Vorbereitung für das kommende Semester zu sehen, wo dann wichtige Graph-Algorithmen, z.B. die Suche nach kürzesten Wegen, vertieft werden.

Zunächst jedoch noch einige grundlegende Definition zu Graphen: Graphen bestehen aus einer endlichen nicht-leeren Knotenmenge V und einer Kantenmenge E (englisch: Knoten = vertex (oder node), Kante = edge). Die Kantenmenge stellt eine Relation auf V dar. Wir unterscheiden gerichtete und ungerichtete Graphen. Bei gerichteten ist Graphen ist die Kantenmenge $E \subseteq V \times V$. Bei ungerichteten Graphen spielt die Reihenfolge, in der die Knoten einer Kante angegeben werden, keine Rolle, wir fassen Kanten als zweielementige Mengen auf, also

$E \subseteq \{ \{x, y\} \mid x, y \in V,\ x \neq y \}$. Wir betrachten hier Graphen ohne Schlingen (Kanten (x, x), deren Anfangs- und Endpunkt identisch sind; im ungerichteten Fall wären dies einelementige Kanten $\{x\},\ x \in V$).

Die ungerichtete Version eines gerichteten Graphen erhält man, indem man die Orientierung der Kanten ignoriert (jede Kante (x, y) wird zur Kante $\{x, y\}$). Umgekehrt erhält man die gerichtete Version eines ungerichteten Graphen, indem man für jede Kante $\{x, y\}$ beide gerichteten Kanten (x, y) und (y, x) hinzufügt.

Wir betrachten die folgenden Definitionen für gerichtete Graphen (für ungerichtete sind sie sinngemäß zu übertragen).

- Jede Kante (x, y) heißt *inzident* zu ihren Knoten x und y.
- Zwei Knoten x und y mit $(x, y) \in E$ heißen *adjazent* oder *benachbart*.
- Die Endknoten der von einem Knoten x ausgehenden Kanten heißt Menge der *Nachfolger* ($S(x) = \{y \in V \mid (x, y) \in E\}$),
- die Anfangsknoten der in einen Knoten x mündenden Kanten heißt Menge der *Vorgänger* ($P(x) = \{y \in V \mid (y, x) \in E\}$),
- beide Zusammen bilden die Menge der *Nachbarn* ($N(x) = S(x) \cup P(x)$).
- Für einen Knoten x bezeichnet $d^+(x) = |S(x)|$ den Ausgangsgrad und $d^-(x) = |P(x)|$ den Eingangsgrad, die Summe der beiden heißt auch der Grad $d(x) = d^+(x) + d^-(x)$.

Wie können wir nun einen Graphen im Rechner modellieren? Zur Darstellung betrachten wir zwei Möglichkeiten. Es sei $G = (V, E)$ mit $V = \{x_1, \ldots, x_n\}$ ein Graph:

- Die *Adjazenzmatrix* $A = (a_{ij})$ ist definiert durch $a_{ij} = 1$, falls $(x_i, x_j) \in E$, und $a_{ij} = 0$ sonst $(i, j = 1, \ldots, n)$.
- Die *Adjazenzliste* – diese setzt sich zusammen aus einer Knotenliste, wobei es zu jedem Knoten x eine Liste der von ihm ausgehenden Kanten gibt (also eine Liste der Elemente in $S(x)$) – wir werden diese gleich genauer betrachten.
- Die *Inzidenzliste* hat neben der Knotenliste eine (unter Umständen ungeordnete) Kantenliste – sie wird in der Praxis sehr selten verwendet.

Da die Adjazenzmatrix stets n^2 Platz benötigt (auch wenn der Graph nur wenig Kanten hat), wird in der Regel die Adjazenzliste verwendet. Das nötige Wissen für eine Knotenliste haben wir bereits – hier braucht aber jeder Knoten noch die Möglichkeit einer Kantenliste. Hierzu hat der Knoten x einen weiteren Zeiger auf eine erste zu ihm inzidente Kante (x, y). In Ada formuliert:

```
type Knoten; type Kante;
type NextKnoten is access Knoten;
type NextKante is access Kante;

type Knoten is record
  NKn : NextKnoten; -- nächster Knoten in der Liste
  EIK : NextKante; -- Verweis auf eine erste inzidente Kante
end record;
```

```
type Kante is record
  EKn : NextKnoten; -- Endknoten der Kante
  NKa : NextKante; -- Verweis auf weitere Kante mit selbem Anfangsknoten
end record;
```

So sind die Knoten und Kanten sehr nackt, sie haben weder einen Namen, noch einen Inhalt oder ähnliches. Neben diesen benötigt man bei den meisten Graphalgorithmen noch weitere Attribute.

Bei den Pre-, Post- und Inorder-Baumdurchläufen haben wir jede Kante genau einmal betrachtet (und sind an jedem Knoten genau drei Mal vorbei gekommen). Wir wollen nun auch für Graphen einen Algorithmus entwerfen, der einen Graphen durchläuft, so dass jede Kante genau einmal besucht wird (und die Knoten nicht häufiger als unbedingt nötig). Wir fügen dazu eine (mit `false` zu initialisierende) Boolean-Variable `besucht` zu dem `type Knoten` hinzu, in der wir uns merken, ob wir diesen Knoten bereits besucht haben oder nicht.

Üblicherweise formuliert man den Graphdurchlauf-Algorithmus folgendermaßen rekursiv:

- Setze alle `besucht`-Werte auf `false`
- Starte für jeden Knoten `x` in der Knotenliste den Graphdurchlauf, d.h.:
  - Falls `x` schon als besucht markiert ist, tue nichts.
  - Sonst: Markiere `x` als besucht, bearbeite den Knoten und führe für jede von `x` Ausgehende Kante `(x,y)` folgendes aus:
    - Bearbeite die Kante `(x,y)` und rufe den Graphdurchlauf rekursiv mit dem Knoten `y` auf.

Auf diese Weise wird der Graphdurchlauf für jeden Knoten x genau $d^-(x)+1$ mal aufgerufen (1 Mal aus der globalen Schleife heraus und für jede eingehende Kante ein weiteres Mal), die von x ausgehenden Kanten werden jeweils genau einmal betrachtet, da beim ersten Besuch von x dieser Knoten als `besucht` markiert wird.

Wir formulieren hier nur den rekursiven Anteil des Graphdurchlaufs aus:

```
procedure GD (x : NextKnoten) is
  e : NextKante;
begin
  if not x.besucht then
    x.besucht := true;
    -- "bearbeite" den Knoten x
    e := x.EIK; -- erste von x ausgehende Kante
    while e /= null loop -- Schleife über alle weiteren Kanten (x,.)
      -- "bearbeite" die Kante (x,e.EKn)
      GD (e.EKn);
      e := e.Nka;
    end loop;
  end if;
end;
```

Machen Sie sich den Ablauf an einigen Beispielen klar. Ausgehend vom ersten Knoten der Knotenliste wird man in dem Graphdurchlauf sich also zunächst entlang miteinander verbundener Kanten soweit wie möglich vom Startknoten entfernen, bis man an einem Knoten angelangt ist, dessen ausgehende Kanten ausschließlich zu Knoten inzident sind, die schon besucht worden sind. Man geht dann in der Rekursion zurück bis zum letzten Knoten, der noch nicht betrachtete ausgehende Kanten hat. Man geht sozusagen zunächst in die Tiefe (möglichst weit weg vom Startknoten) – daher heißt diese Art des Graphdurchlaufs auch *Tiefendurchlauf* oder *Tiefensuche* (englisch: depth first search).

Der Zeitaufwand: Wie oben schon gesagt, wird jede Kante genau einmal bearbeitet (und die Prozedur rekursiv für den jeweiligen Endpunkt einmal aufgerufen) und jeder Knoten zusätzlich einmal aus der Knotenliste heraus aufgerufen). Die wesentliche Anzahl der Schritte ist damit $n + 2m$, wenn n die Anzahl der Knoten und m die Anzahl der Kanten ist. In O-Notation bedeutet das also einen Aufwand von $O(n + m)$ (oder etwas genauer $\Theta(n + m)$).

Randbemerkung: Man kann diese Tiefensuche auch ohne Rekursion formulieren: Dazu benötigt man einen Keller, auf den man zu Beginn den Startknoten legt. Der Ablauf ist dann: Solange noch Knoten auf dem Keller liegen, nimm den obersten Knoten und falls dieser noch nicht besucht ist, markiere ihn und lege dann alle über eine Kante erreichbaren Knoten auf den Keller (Ende der Schleife). Ersetzt man in dieser Betrachtungsweise die Datenstruktur Keller durch eine Schlange, so erhält man eine weitere Art des Graphdurchlaufs, den *Breitendurchlauf*, auch *Breitensuche* genannt (englisch: breath first search). Der Zeitaufwand ist bei beiden Varianten gleich.

Wir betrachten noch einige weitere Definitionen (hier für gerichtete Graphen formuliert):

- Eine Folge von Knoten $(u_0, u_1, \ldots, u_k)$ mit $k \geq 0$ heißt *Weg* im Graphen G, wenn für alle $i \in \{1, \ldots, k\}$ gilt $(u_{i-1}, u_i) \in E$.
- Die *Länge* des Weges ist durch die Anzahl der Kanten, also k, definiert.
- Zwei Knoten u und v heißen verbunden, wenn es einen Weg von u nach v gibt.
- Der Weg heißt *doppelpunktfrei* oder *einfach*, wenn die u_i paarweise verschieden sind, also $u_i \neq u_j$ für $i \neq j$,
- er heißt *geschlossen*, wenn $u_0 = u_k$ gilt – ist $k \geq 2$ und $(u_1, \ldots, u_k)$ doppelpunktfrei, so heißt solch ein Weg *Zyklus* (oder *Kreis*). Manchmal wird hier auch $k \geq 3$ gefordert.
- Der Graph ist stark zusammenhängend, wenn für alle Knoten u und v es einen Weg von u nach v gibt. Der Graph ist schwach zusammenhängend, wenn in der ungerichteten Version u und v verbunden sind für alle u und v.
- Dem entsprechend sind starke und schwache Zusammenhangskomponenten eines Graphen definiert als die Teilmengen der Knoten, die paarweise durch einen gerichteten bzw. ungerichteten Weg verbunden sind.
- Zu einem Graphen G heißt $G_{tH} = (V, E_{tH})$ mit $E_{tH} = \{(x, y) | \text{es gibt einen Weg von } x \text{ nach } y\}$ die *transitive Hülle* des Graphen G.

Wir werden uns mit den Zusammenhangskomponenten und transitiven Hüllen in der Übung beschäftigen.

3.4 Ein Nachtrag zu Zeigern

Wir hatten schon erwähnt, dass sich der Typ einer Zeigervariable `Ref` (ein `access`-Typ auf den referenzierten Datentyp) und der Typ von `Ref.all` (der referenzierte Datentyp) unterscheiden. Neben der Zuweisung ist dieser Unterschied auch bei Vergleichen wichtig:

`Ref1 = Ref2` überprüft, ob beide Zeiger auf dieselbe Speicherstelle zeigen, der Ausdruck wird sich also stets zu `false` auswerten, wenn die referenzierten Speicherstellen verschieden sind – auch, wenn die Inhalte dieser Speicherstellen identisch sind. Will man die Inhalte vergleichen, so muss man `Ref1.all = Ref2.all` benutzen.

Zeiger auf Variablen Die Zeiger, die wir bisher kennen gelernt haben, können nur als Verweise auf mittels `new` erzeugter Variablen benutzt werden.

Im Allgemeinen dient es der Lesbarkeit von Programmen, wenn auf Variablen nicht gleichzeitig auf verschiedene Arten und Weisen zugegriffen werden kann (also insbesondere sollte man vermeiden auf eine Variable sowohl direkt als auch indirekt über Zeiger zuzugreifen).

Manchmal ist dies jedoch sinnvoll. Wir rufen uns das Beispiel der Maximum-Berechnung in einem Array in Erinnerung: Wir hatten seinerzeit eine rekursive Variante dieser Prozedur vorgestellt:

```
type vektor is array (integer range <>) of integer;

function maximum (feld:vektor) return integer is
begin
  if feld'length=1 then
    return feld(feld'first);
  else
    return integer'max(feld(feld'first),maximum(feld(1+feld'first..feld'last)));
  end if;
end;
```

Ermittelt man experimentell die Laufzeit des Algorithmus, so stellt man fest, dass diese nicht linear sondern quadratisch mit der Anzahl der Elemente wächst. Dies liegt daran, dass bei dem rekursiven Aufruf das Feld kopiert wird und somit der Aufwand durch die Summe der Feldgrößen dominiert wird. Sinnvoll wäre es hier, wenn man lediglich einen Verweis auf die dem Teilfeld entsprechende Speicherstelle übergibt (Ada sorgt sich dann implizit um die korrekten Feldgrenzen).

Um es nicht gleich mit Arrays zu kompliziert zu machen, betrachten wir zunächst ein kleines Beispiel, das demonstriert, wie man auf normale Variablen auch über Zeiger zugreifen kann. Folgendes ist zu beachten:

- Die entsprechende Variable muss als ein `aliased` Typ definiert werden – dies ist auch zugleich eine Art kleiner Warnung bzw. Erinnerung an den Programmierer, dass man auf die Variable auch über Zeiger zugreifen kann.
- Die Zeigervariable muss vom Typ `access all` <`Datentyp`> sein.

- Solch einer Zeigervariablen darf kein Verweis auf eine Variable zugewiesen werden, deren Lebensdauer vor der der Zeigervariablen endet – dies stellt sicher, dass die Zeigervariable stets auf eine noch gültige Speicherstelle verweist.

```
declare
  type ref_int is access all integer;
  int_ptr : ref_int; -- unbenannte Zeigertypen mag Ada nicht
  int_var : aliased integer := 37;
begin
  put (int_var);
  int_ptr := int_var'access; -- 'access liefert den Verweis auf die Variable
  put (int_ptr.all);
  int_ptr.all := 23;
  put (int_var);
end;
```

Bei folgender Variante würden wir jedoch schon bei der Übersetzung eine Fehlermeldung erhalten!

```
declare
  type ref_int is access all integer;
  int_ptr : ref_int;
begin
  -- irgendwelche Anweisungen
  declare
    int_var : aliased integer := 37;
  begin
    put (int_var);
    int_ptr := int_var'access; -- Fehler! Globaler Zeiger auf lokale Variable
    put (int_ptr.all);
    int_ptr.all := 23;
    put (int_var);
  end;
  -- irgendwelche Anweisungen
end;
```

Kommen wir zurück zum Beispiel der Maximumbestimmung. Mit solchen `aliased`-Typen lässt sich das Programm nun wie folgt schreiben:

```
Programm verschollen -- wird nachgereicht
```

Die Laufzeit ist hier nun linear und nicht mehr quadratisch, da nur ein Zeiger auf ein Array übergeben wird und nicht mehr das entsprechende Teilfeld bei der Parameterübergabe kopiert wird. Überprüfen Sie dies, indem Sie experimentell die Laufzeit für verschiedenen Arraygrößen miteinander vergleichen.

Zum Abschluss nochmals die Warnung: Wann immer es geht, sollte man solche `aliased`-Konstrukte vermeiden. Auf eine Variable sollte immer nur über eine Möglichkeit zugegriffen werden, um die Lesbarkeit und Wartbarkeit der Programme nicht zu beeinträchtigen.

3.5 Packages und Abstrakte Datentypen

Eine Datenstruktur zusammen mit den darauf definierten Operationen und ihren Eigenschaften nennt man in der Informatik einen **Abstrakten Datentyp** (ADT). Solche Datenstrukturen mit den zugehörigen Operationen bilden oft Einheiten, die in vielen Programmen wiederverwendet werden können. Ada bietet zum Einbinden von solchen Einheiten das Konzept der *Packages* an – wir haben dieses schon lange verwendet (z.B. `Ada.Text_IO`, etc.).

Die Alternative – die direkte Unterbringung der Deklarationen von Prozeduren und Funktionen im Deklarationsteil anderer Prozeduren und Funktionen (oder im Hauptprogramm) durch copy-&-paste ist nicht immer schön:

- Zum einen werden die Deklarationen lang und unübersichtlich,
- zum anderen wollen wir die Programmteile oft auch in anderen Prozeduren oder anderen (Haupt-)Programmen verwenden – das geht nicht, solange sie im Innern einer Prozedur „versteckt“ sind.

Eine Abhilfe bieten in Ada *Pakete*, die Prozeduren und Funktionen (und ggf. auch Variablen) zusammenfassen und zur Verfügung stellen – Ada.Text_IO ist z.B. ein solches Paket. In der Praxis verwendet man Pakete zwar in der Regel zum Einbinden von Abstrakten Datentypen, prinzipiell lassen sich mit Paketen aber beliebige Sammlungen von Datentypen und Operationen einbinden.

Um selber ein Paket zu schreiben, brauchen wir zwei Dateien: die Spezifikation und den Rumpf:

Die Spezifikation enthält nur die Spezifikationen der Prozeduren und Funktionen, also alles, was vor dem `is` steht, sowie als Kommentar alles vom Kontrakt, was wir hier erwähnen wollen.

Eingerahmt wird das mit

```
package <Name> is
  <Spezifikationen>
end Name;
```

Blicken wir zurück auf das Beispiel eines Kellers (englisch: Stack), so könnte solch eine Spezifikationsdatei folgendermaßen aussehen:

```
  with Ada.Text_IO; use Ada.Text_IO;
package Stack is
  type StElement;
  type Stack is access StElement;
  type StElement is record
    Inhalt : Integer;
    Next : Stack;
  end record;
  function empty return Stack; -- erzeugt einen leeren Stack
  function is_empty(S : Stack) return Boolean; -- ist true, falls S leer ist
  procedure Push(S : in out Stack; elem : Integer);
  function Top(S : Stack) return Integer;
  procedure Pop(S : in out Stack);
end Stack;
```

Solche Spezifikationsdateien haben die Endung `.ads` (Ada Specification) im Gegensatz zur Implementierung mit Endung `.adb` (Ada Body). Die Implementierung sähe dann so aus (die Parameterlisten der Prozeduren und Funktionen müssen natürlich übereinstimmen).

```
package body Stack is

  function empty return Stack is
  begin
    return null;
  end;

  function is_empty(S : Stack) return Boolean is

...

end Stack;
```

Im Hauptprogramm kann solch ein Paket dann wie auch bei den Standardpaketen mit `with` und `use` eingebunden und benutzt werden. Damit der Ada-Compiler die Dateien findet, sollte man diese am einfachsten im aktuellen Verzeichnis speichern (in dem auch das zu übersetzende Programm steht).

In diesem und anderen Beispielen mag es wünschenswert sein, dass man nicht direkt auf die Datenstrukturen zugreifen kann (sondern nur über die zur Verfügung gestellten Funktionen). Dies hält die Möglichkeit für den Programmierer offen, die Implementierung des Stacks verändern zu können (z.B. mit Hilfe eines ausreichend großen Arrays), ohne die Programme verändern zu müssen, die dieses Stack-Paket benutzen. Dazu müssen wir die Datentypen als `private` deklarieren.

```
package Stack is
  with Ada.Text_IO; use Ada.Text_IO;
  type Stack is private;
  function empty return Stack; -- erzeugt einen leeren Stack
  function is_empty(S : Stack) return Boolean; -- ist true, falls S leer ist
  procedure Push(S : in out Stack; elem : Integer);
  function Top(S : Stack) return Integer;
  procedure Pop(S : in out Stack);
  private
    type StElement;
    type Stack is access StElement;
    type StElement is record
      Inhalt : Integer;
      Next : Stack;
    end record;
end Stack;
```

Für mit dem Schlüsselwort `private` deklarierte Variablen steht nur ein sehr eingeschränkter Satz von Operationen zur Verfügung:

- Zuweisungen (`s1 := s2;`),
- Test auf (Un-) Gleichheit (`s1 = s2` und `s1 /= s2`),
- und die Operationen, die das Paket zur Verfügung stellt.

In manchen Situationen will man sogar Zuweisungen verbieten (sonst könnten bei dem Stack-Beispiel mit Listenimplementierung durch Zuweisung von Zeigern z.B. zwei Stackvariablen auf denselben Stack zeigen, die sich dann durch weitere Operationen auf den Stacks zu einem undurchschaubaren Zeigerwirrwarr entwickeln). Will man Zuweisungen verbieten, so kann dies durch die Deklaration als `limited private` geschehen. Dies ist eine Steigerung von `private`, bei der es dann weder Zuweisung noch Gleichheitsoperator gibt. Nun stünde einer nachträglichen Änderung der Stackimplementierung als Array nichts mehr im Wege – Programme, die ein Stack-Paket mit als `limited private` deklarierten Datentypen verwendet, können nun nicht mehr sehen, wie der Stack implementiert ist. Probieren Sie es aus.

3.6 Ausnahmen im Programmablauf – Exceptionhandling

Heute steht noch einmal ein Thema aus dem Bereich der Programmierung an: wir werden Mechanismen kennen lernen, mit denen unsere Programme Fälle behandeln können, in denen der normale Programmablauf nicht fortgesetzt werden kann: beim Auftreten von *Ausnahmen*

3.6.1 Beispiele für Ausnahmen

Eine Ausnahme tritt z.B. ein, wenn wir eine arithmetische Operation mit ungeeigneten Operanden ausführen wollen, etwa versuchen, durch 0 teilen oder die Wurzel aus einer negativen Zahl zu ziehen.

Ein gutes Programm sollte eine solche Ausnahme soweit wie möglich abfangen (u.U. führt ein anderer Algorithmus ans Ziel), wenigstens aber im Ausnahmefall das Programm kontrolliert zu Ende bringen (z.B. Daten sichern...).

Ein anderes typisches Beispiel für Ausnahmen entsteht bei der Eingabe von Werten durch den Benutzer: wenn der z.B. statt einer ganzen Zahl einen Buchstaben eingibt, sollte das Programm nicht „abstürzen“ und die Arbeit der letzten Stunde zunichte machen, sondern z.B. die Eingabe wiederholen.

Ausnahmen gar nicht erst entstehen lassen? Eine einfache Lösung (d.h. ohne neue Sprachelemente lernen zu müssen) wäre, mögliche Ausnahmen vorab durch Fallunterscheidungen abzufangen.

```
with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;
procedure addieren1 is
   function plus (a, b : Integer) return Integer is
   --  liefert a+b (bei Bereichsueberschreitung jeweilige Grenze)
   begin
      if b > 0 then
         if Integer'Last - b >= a then return a + b;
```

```
         else return Integer'Last;
         end if;
      else
         if Integer'First - b <= a then return a + b;
         else return Integer'First;
         end if;
      end if;
   end plus;
   x, y : Integer;
begin
   Get (x); Get (y);
   Put (plus (x, y));
end addieren1;
```

- Schon beim primitiven Additions-Beispiel wird klar: dieser Weg ist sehr aufwändig.
- Schlimmer noch als die Arbeit, die das macht, ist, dass das Resultat undurchschaubar ist; die Ausnahmebehandlung verdeckt den eigentlichen „Algorithmus“.
- Wenn das ausartet, können wir u.U. sogar in Effizienzprobleme kommen, wenn die Fallunterscheidungen mehr Arbeit machen als die eigentliche Rechnung.
- Also:

  Man soll den Regelfall nicht mit den Kosten des Spezialfalls belasten!

Ausnahmen innerhalb von Unterprogrammen

- Noch gar nicht gelöst ist im Moment das Problem, das eintritt, wenn der Benutzer beim Eingeben einer Integerzahl (im Programm: Aufruf von Get) „Unsinn“ eingibt, z.B. einen Buchstaben.
- Auf den Ablauf der Prozedur Get haben wir „von außen“ keinen Einfluss.
- In verschiedenen Situationen sind aber verschiedene Reaktionen notwendig (Eingabe wiederholen, mit Standardwert weiterrechnen, Programm abbrechen,...).
- Müssen wir für jede dieser Möglichkeiten unser eigenes Get mit passenden Maßnahmen schreiben?!
- Zum Glück ist das nicht so: der Mechanismus in Ada zur Ausnahmebehandlung erlaubt es uns, jederzeit die reguläre Programmausführung zu unterbrechen und an geeigneten Stellen (das kann auch außerhalb der Prozedur sein, in der das Problem auftrat) Anweisungen zum Verhalten im Ausnahmefall zu ergreifen.

3.6.2 Ausnahmen erzeugen

- Das Auftreten einer Ausnahme unterbricht den normalen Programmablauf; damit unser Programm auf den Ausnahmefall passend reagieren kann, stellt Ada spezielle Objekte zur Verfügung, diese Objekte nennt man, da eine Verwechslung hier kaum möglich ist, ebenfalls Ausnahmen (englisch: exceptions).

- Wir werden der Reihe nach lernen, wie man solche Objekte deklariert, sie erzeugt (d.h., den Ausnahmefall herstellt) und schließlich, wie man sie behandelt, d.h., wie man merkt, dass der Ausnahmefall eingetreten ist, und wie man, wenn man das Problem beheben konnte, wieder zum regulären Programmablauf zurückkehrt.

Vordefinierte Ausnahmen Ada95 stellt standardmäßig folgende Ausnahmen bereit:

- Constraint_Error: unsere „Standardausnahme“, die bei Bereichsüberschreitungen während arithmetischer Operationen, bei Indexüberschreitungen beim Zugriff auf Reihungen etc. auftritt.
- Program_Error, Storage_Error, Tasking_Error: eher technische Ausnahmen, z.B. bei nicht ausreichendem Speicher etc.

Einige Pakete stellen weitere Ausnahmen bereit, wichtig sind dabei für uns vor allem die für Ein- und Ausgabe (ein besonders ausnahmeträchtiges Feld!)

Eigene Ausnahmen definieren

- Wir können (und sollen) eigene, aussagekräftige Ausnahmen deklarieren.
- Formal sieht das aus wie eine Variablendeklaration, wobei der Datentyp durch das Wort `exception` ersetzt wird.
- Auch wenn es so aussieht: es werden hier keine Variablen vereinbart (man kann sich diese Deklarationen eher wie Typdeklarationen vorstellen).
- Angenommen, unsere Additionsprozedur soll im Fehlerfall nicht stillschweigend die jeweilige Bereichsgrenze zurückliefern, sondern eine Ausnahme auslösen, die spezifischer sein soll als der `CONSTRAINT_ERROR`.

  Dazu könnten wir uns je eine Ausnahme für „Summe zu groß“ und für „Summe zu klein“ deklarieren:

  ```
  ObenRaus, UntenRaus : exception;
  ```

- Es gelten die üblichen Sichtbarkeitsregeln für Bezeichner (auch wenn die Ausnahme über diesen Bereich hinaus Auswirkungen haben kann).

Ausnahmen erzeugen mittels raise Und wie ruft man nun den Ausnahmezustand aus? Mittels einer Anweisung

```
raise Name_der_Aunnahme;
```

```
with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;
procedure addieren2 is
   ObenRaus, UntenRaus : exception;
   function plus (a, b : Integer) return Integer is
   --  liefert a+b (oder loest passende Ausnahme aus)
```

```
   begin
      if b > 0 then
         if Integer'Last - b >= a then return a + b;
         else raise ObenRaus;
         end if;
      else
         if Integer'First - b <= a then return a + b;
         else raise UntenRaus;
         end if;
      end if;
   end plus;
   x, y : Integer;
begin
   Get (x); Get (y);
   Put (plus (x, y));
end addieren2;
```

Die Ausführung der Anweisung `raise` bewirkt folgendes:

- Die normale Programmausführung wird unterbrochen.
- Präziser gesagt (das wird gleich wichtig werden): der derzeit bearbeite *Block* von Anweisungen ist damit beendet.
- Ein Block ist dabei eine Folge von Anweisungen zwischen `begin` und `end`, also (bei uns bisher) der Rumpf einer Prozedur oder einer Funktion.
- Damit sind wir (einen Augenblick lang) an der Stelle, von der aus der Block betreten wurde, also beim Prozedur- bzw. Funktionsaufruf.
- Allerdings herrscht immer noch der Ausnahmefall, daher wird auch dieser Block sofort beendet...
- ...und so weiter, bis es nichts mehr zu beenden gibt, weil der äußerste Block (die Prozedur, die das Hauptprogramm bildet), beendet ist.
- Dann wird eine Fehlermeldung gedruckt (Name der Ausnahme und der Ort, wo sie aufgetreten ist):

  ```
  raised ADDIEREN2.OBENRAUS : addieren2.adb:10
  ```

- Wozu diese komplizierte Betrachtung der ineinander geschachtelten Blöcke? Weil wir gleich lernen, wie wir am Ende eines Blocks den Ausnahmefall behandeln und zum normalen Programmablauf zurückkehren können, so dass die äußeren Blöcke „nichts merken“.

3.6.3 Ausnahmen behandeln

- Natürlich kann man mit diesem Mechanismus mehr machen als nur eine etwas aussagekräftigere Fehlermeldung drucken.

- Dazu kann man an das Ende jedes Blocks (unmittelbar vor dem `end`) Anweisungen zur Ausnahmebehandlung anfügen.
- Diese bestehen
  - Aus dem reservierten Wort `exception` (wie in der Deklaration von Ausnahmen, hier aber solo)
  - und einem oder mehreren *Ausnahmezweigen*.
- Ein Ausnahmezweig seinerseits hat (meistens) die Form

  ```
  when Ausnahme => Anweisungen
  ```

  oder

  ```
  when others => Anweisungen
  ```

  (letztere Form ist nur als letzte in einer Folge von Anweisungszweigen erlaubt).
- Wenn der Block wegen einer Ausnahme vorzeitig beendet wird und es einen Ausnahmezweig mit der passenden Ausnahme (z.B. `Constraint_Error`) gibt, werden dessen Anweisungen ausgeführt. Der Ausnahmefall ist damit beendet, die Programmausführung geht regulär im nächstäußeren Block weiter.
- Ein ggf. vorhandener Zweig mit `others` behandelt alle Arten von Ausnahmen, die keinen eigenen Zweig bekommen haben.
- Wenn es keinen passenden Ausnahmezweig für die eingetretene Ausnahme gibt, bleibt der Ausnahmefall bestehen und pflanzt sich in den nächstäußeren Block fort.
- Wenn (was der Regelfall sein sollte...) keine Ausnahme während der Abarbeitung des Blocks auftritt, wird auch keiner der Ausnahmezweige ausgeführt.

In unserem Beispiel ist es schwer, sich eine sinnvolle Fehlerbehandlung vorzustellen, also drucken wir nur eine etwas nettere Meldung:

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure addieren3 is
   ObenRaus, UntenRaus : exception;
   function plus (a, b : Integer) return Integer is
   --  liefert a+b (oder loest passende Ausnahme aus)
   begin
      if b > 0 then
         if Integer'Last - b >= a then return a + b;
         else raise ObenRaus;
         end if;
      else
         if Integer'First - b <= a then return a + b;
         else raise UntenRaus;
         end if;
      end if;
```

```
   end plus;
   x, y : Integer;
begin
   Get (x); Get (y);
   Put (plus (x, y));
exception
   when ObenRaus => Put_Line ("Summe zu gross");
   when UntenRaus => Put_Line ("Summe zu klein");
end addieren3;
```

- Der Unterschied zum vorigen Programm ist vermutlich in diesem kleinen Beispiel noch nicht hinreichend deutlich geworden.
- Schließlich bekommen wir in beiden Fällen bei Bereichsüberschreitung eine Fehlermeldung,

  ```
  raised ADDIEREN2.OBENRAUS : addieren2.adb:10
  ```

  und einmal

  ```
  Summe zu gross
  ```

- Wichtig ist, dass letzteres keine Fehlermeldung ist, sondern eine reguläre Ausgabe unseres Programms, das hinterher noch viele weitere Anweisungen ausführen könnte (die es im Minibeispiel halt nicht gibt).
- Erwähnen sollte man noch, dass in einem Ausnahmezweig natürlich auch Ausnahmen auftreten können — automatisch oder mittels `raise`. Das ist gelegentlich nützlich, wenn sich herausstellt, dass in der aktuellen Situation das Problem doch nicht zu beheben ist...

3.6.4 Blöcke

- Wir haben die Bedeutung der Blöcke bei der Behandlung von Ausnahmen kennen gelernt. Da die Ausnahmebehandlung nur am Ende eines Blocks stattfinden kann, ist — solange wir nur Prozedur- und Funktionsrümpfe als Blöcke kennen — die Prozedur bzw. die Funktion nach Bearbeitung des Fehlerfalls auf jeden Fall beendet.
- Das ist unpraktisch — z.B. hätten wir in der Prozedur addieren3 vielleicht noch einige Worte zum Abschied ausgedruckt, unabhängig davon, ob ein Fehler aufgetreten ist oder nicht.
- Daher ist es nützlich zu wissen, dass überall dort, wo eine Anweisung erlaubt ist, auch ein Block stehen darf.
- Er besteht
  - optional aus dem reservierten Wort `declare` und einer Folge von lokalen Deklarationen (in der Regel, wenn überhaupt, einige Variablen)
  - dem reservierten Wort `begin`,
  - einer Folge von Anweisungen,

- optional Anweisungen zur Fehlerbehandlung
- und aus dem reservierten Wort `end` und einem Semikolon.

Unser nächster Versuch, ein Additionsprogramm zu schreiben, ignoriert die Möglichkeit der Bereichsüberschreitung, kümmert sich dafür um den Fall, dass etwas eingegeben wird, das keine Integerzahl ist:

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure addieren4 is
   x, y : Integer;
   procedure Get_neu (x : out Integer) is
      OK : Boolean := False;
   begin
      while not OK loop
         begin
            Get (x);
            OK := True;
         exception
            when Data_Error =>
               Put_Line ("Falsche Eingabe, nochmal!");
               Skip_Line;  -- Ganze Eingabezeile wegwerfen
         end;
      end loop;
   end Get_neu;
begin
   Get_neu (x); Get_neu (y);
   Put (x + y);
end addieren4;
```

3.6.5 Programmablauf im Fall von Ausnahmen

- Prinzipiell wissen wir bereits, wie sich eine Ausnahme durch die Blöcke hindurchfrisst, bis sie auf eine geeignete Ausnahmebehandlung trifft (oder nicht). Da man daran aber das wichtige Konzept der *Blockstrukturierung* studieren kann, sehen wir uns das noch mal genauer an.
- Ein Block besteht aus
  - (optional) einem Satz lokaler Variablen,
  - einer Folge von Anweisungen
  - und (optional) Anweisungen zur Ausnahmebehandlung
- Ein Block wird betreten
  - im Fall von Prozeduren und Funktionen durch deren Aufruf
  - im Fall von Block-Anweisungen durch deren Ausführung.

- Bei Betreten des Blocks wird Platz für die lokalen Variablen geschaffen (sofern es welche gibt), anschließend werden die Anweisungen nacheinander ausgeführt.
- Der Block wird verlassen
  - bei Erreichen des `end` oder
  - bei Auftreten einer Ausnahme.

  In letzteren Fall wird ggf. noch ein Ausnahmezweig abgearbeitet (womit der Ausnahmefall erstmal wieder aufgehoben ist); die Block ist aber so oder so beendet.
- Alle betretenen, aber noch nicht wieder verlassenen Blöcke stellen wir uns ineinander geschachtelt vor: das Hauptprogramm außen, die derzeit ausgeführten Anweisungen ganz innen (im rekursiven Fall kann das schon bei kurzen Programmen eine erhebliche Schachtelungstiefe geben).
- Die Kontrolle im Falle einer Ausnahme bekommt der innerste Block, der einen passenden Ausnahmezweig vorsieht (wenn es keinen gibt, wird das Programm mit Fehlermeldung abgebrochen).
- Alle weiter innen liegenden Blöcke sind mit Auftreten der Ausnahme sofort beendet; alle weiter außen liegenden Blöcke merken von der Ausnahme nichts (außer, die Ausnahmebehandlung löst ihrerseits eine Ausnahme aus).
- Dieser Mechanismus erlaubt es uns, zuverlässige und gleichzeitig übersichtliche Programme zu schreiben!

3.7 Übungsaufgaben

1. **Rechnen mit Brüchen:** Gleitkommazahlen bringen stets automatisch gewisse Rundungsfehler mit sich - so lassen sich z.B. selbst einfache Zahlen wie 0.2 oder 23.1 mit dem Datentyp `float` nicht exakt im Rechner darstellen. Wir wollen hier nun Abhilfe schaffen.

   Wir definieren uns einen Typ für Rationale Zahlen (Brüche), also ein Paar (Zähler,Nenner), wobei der Nenner stets positiv ist. Die Darstellung ist nicht eindeutig, da wir den Bruch 2/3 z.B. auch als 4/6 schreiben können, ohne den Wert dabei zu verändern. Um diese Mehrdeutigkeiten zu vermeiden, werden wir die Brüche stets soweit möglich kürzen. Den dazu benötigten größten gemeinsamen Teiler (ggT) von Zähler und Nenner können wir mit Euklids Algorithmus effizient berechnen. Die Zahl 0 soll durch 0/1 dargestellt sein.

   Haben wir die Grundrechenarten Plus, Minus, Mal und Geteilt implementiert, können wir ein paar Beispielprogramme schreiben.

   Nun ans Werk ... und vergessen Sie nicht, Ihre Programme zu kommentieren.

   - Definieren Sie einen Datentyp für Brüche, benutzen Sie dafür einen Verbund (Record).
   - Schreiben Sie Ein- und Ausgaberoutinen (Put und Get) für Ihren neu definierten Datentyp (Sie können Zähler und Nenner einzeln erfragen). Achten Sie darauf, dass Sie den Bruch nach Eingabe kürzen – nach Eingabe des Bruches 4/6, soll die Ausgabe der eingelesenen Zahl also 2/3 sein.

     Der Euklidische Algorithmus basiert auf den zwei Eigenschaften

  - ggT(a,b) = ggT(b,a mod b) für b größer 0 und
  - ggT(a,0) = a

  Achten Sie (bei negativen Zahlen) bei der ggT-Berechnung auf die Vorzeichen.

  Zusatzaufgabe: Zeigen Sie, dass der Aufwand zur Berechnung des ggT mit Euklids Algorithmus nur $O(\log(b))$ Schritte beträgt.
- Implementieren Sie die 4 Grundrechenarten und die Negation.
- Implementieren Sie die Vergleiche <=, <, =, / =, >, >=.
- Berechnen Sie die Harmonische Funktion $h(n) := \sum_{i=1}^{n} 1/i$
  – **Zusatzaufgabe:** Hier kommt es schon bei relativ kleinem n zu `constraint errors`, weil die Nenner recht schnell recht groß werden. Verbessern Sie Ihre Algorithmen dahingehend, indem die Produkte soweit möglich schon vorab gekürzt werden, bevor die Multiplikation zu große Zahlen generiert. Bis zu welchem n können Sie nun die Harmonische Funktion berechnen?
- Demonstrieren Sie die Korrektheit der Implementierung durch geeignete Testfälle im Hauptprogramm.

2. **Listen:** Implementieren Sie die fehlenden Prozeduren und Funktionen aus der Vorlesung, Sie können hier als `Element_Typ` den Typ `Integer` verwenden.

   - `procedure add_to_end(list : in out Liste; elem : Element_Typ)` – fügt `elem` am Ende der Liste ein
   - `procedure add_sorted(list : in out Liste; elem : Element_Typ)` – fügt in eine vorher sortierte Liste `elem` so ein, dass danach die Liste wieder sortiert ist
   - `procedure delete(list : in out Liste; elem : Element_Typ; deleted : out Boolean)` – löscht das erste Vorkommen von `elem` in `list`, der Ausgangsparameter `deleted` wird `false` gesetzt, wenn `elem` nicht in `list` vorkam, sonst auf `true`
   - `procedure delete_all(list : in out Liste; elem : Element_Typ; deleted : out Natural)` – löscht alle Vorkommen von `elem` in `list` und zählt in `deleted`, wieviel Elemente gelöscht wurden
   - `function copy_List(list : Liste) return Liste` – kopiert die Liste

   Demonstrieren Sie die Korrektheit Ihrer Implementierungen mit entsprechenden Testfällen, die Sie in Ihr Hauptprogramm einbauen. Achten Sie auf ausreichend Kommentare und Beschreibung der Algorithmen in Ihrem Ada-95-Code.

   Beachten Sie auch die Hinweise zu diesen Prozeduren und Funktionen im Skript.

3. **Listen II:** Schreiben Sie zwei **rekursive** Prozeduren zur Verarbeitung von einfach verketteten Listen (als Element-Typ können Sie `integer` verwenden):

   - `Put_revers(anker : Liste)` – gibt die Liste in umgekehrter Reihenfolge aus. Die Liste selbst soll dabei unverändert bleiben.
   - `Revers_List(anker : Liste)` – diese Prozedur soll die Richtung der Verzeigerung innerhalb der Liste umkehren. Nach dem Aufruf soll der Anker also auf das vormals letzte Element zeigen, dieses auf das vorletzte, usw. Von welcher Parameterart sollte `anker` sein? Was passiert, wenn Sie `Revers_List` zwei Mal aufrufen?

   Geben Sie jeweils den Aufwand (als Kommentarzeilen eingefügt) in O-Notation an.

4. **Binärbäume / Suchbäume:** Implementieren Sie die Prozeduren und Funktionen aus der Vorlesung, Sie können hier als `Element_Typ` den Typ `Integer` verwenden.

   - `function empty return BinBaum` – erzeugt einen leeren (Such-)Baum
   - `procedure insert(baum : in out Liste; elem : Element_Typ)` – fügt in einen Suchbaum das Element `elem` gemäß den Eigenschaften eines Suchbaums ein
   - `function is_in(elem : Element_Typ; baum : BinBaum) return Boolean` – liefert `true` oder `false` abhängig davon, ob `elem` im `baum` vorkommt
   - `function number_of_elements(baum : BinBaum) return Natural` – zählt die Elemente in der Liste

   Beachten Sie auch die Hinweise zu diesen Prozeduren und Funktionen im Skript.

   Schreiben Sie ein Hauptprogramm, das zunächst eine Zufallszahl z aus dem Bereich 5..20 ermittelt und dann z Zahlen (aus einem ausreichend großen Integer-Bereich) zufällig erzeugt und diese in einen Suchbaum einfügt. Geben Sie den Suchbaum dann jeweils in Pre-, Post- und Inorder aus. Eine dieser drei Ausgaben hat eine auffällige Eigenschaft. Begründen Sie diese (fügen Sie die Begründung als Kommentar in Ihrem Programm mit ein).

   Schreiben Sie auch eine Prozedur, die bestimmt, wieviel Vergleiche man höchstens benötigt, um mit der Funktion `is_in` festzustellen, ob ein Element im Suchbaum vorhanden ist oder nicht. **Hinweis:** Sie müssen dazu nicht für alle möglichen Eingaben die Anzahl der Vergleiche bestimmen (zumal dies so auch nicht durchführbar wäre) – überlegen Sie sich eine rekursive Lösung.

   Achten Sie unbedingt auf ausreichend Kommentare und Beschreibung der Algorithmen in Ihrem Ada-95-Code.

5. **Ein Binärbaumalgorithmus:** Betrachten Sie folgenden Algorithmus (`BinBaum` ist ein `access` Typ auf einen Binärbaumknoten `Knoten`):

```
procedure wastutdas (v : BinBaum) is
  r : BinBaum := new Knoten;
  p : BinBaum := r;
  c : BinBaum := v;
  h : BinBaum;
begin
  r.left:=v;
  while c /= r loop
    if c /= null then
      Put (c.inhalt);
      h := c.links;
      c.links := c.rechts;
      c.rechts := p;
      p := c;
      c := h;
    else
      h := c;
      c := p;
      p := h;
    end if;
  end loop;
end wastutdas;
```

Was tut diese Prozedur (wie wird der Unterbaum von `v` verändert) und welche Ausgabe liefert sie (vergleichen Sie die Ausgabe mit anderen Ihnen bereits bekannten Baumalgorithmen)?

Hinweis: Eine dringende Empfehlung: Die Aufgabe ist an sich recht leicht – Sie müssen aber sehr genau aufpassen, wohin welcher Zeiger jeweils gerade zeigt – man vertut sich beim ständigen Umsetzten der Verweise sehr leicht.
Sie sollten das Programm unbedingt von Hand durchführen, um mit Zeigern vertraut zu werden – nehmen Sie (zunächst) Beispiele mit höchstens 3 oder 4 Knoten. Wenn Sie dann eine Idee haben, was das Programm tut, können Sie es immer noch abtippen und Ihre Vermutung an größeren Beispielen verifizieren.

6. **Binärbäume und Suchbäume:** Wir betrachten hier Bäume mit `integer`-Werten als Knoteninhalt.

   - Schreiben Sie eine Funktion, die in einem Binärbaum den Wert des Knotens mit größtem Inhalt bestimmt und diesen zurückgibt (ist der Baum leer, so soll der Wert 0 zurückgegeben werden).
   - Schreiben Sie eine effiziente Funktion, die in einem Suchbaum den Wert des Knotens mit größtem Inhalt bestimmt und diesen zurückgibt (ist der Suchbaum leer, so soll der Wert 0 zurückgegeben werden).

   Geben Sie jeweils den Aufwand (als Kommentarzeilen eingefügt) in O-Notation an.

7. **Eine kleine Graph-Eigenschaft:** Ein Kommillitone bietet Ihnen eine Wette an: Bei der nächsten Vorlesung soll beobachtet werden, wieviel Studierende sich gegenseitig begrüßen (das Ergebnis der Beobachtung ist dann z.B.: Student A begrüßte 23 Studenten, Student B begrüßte 18 Studenten, usw. – wenn Student A Student B begrüßt, dann begrüßt Student B auch stets Student A). Wenn die Anzahl der Studierenden, die eine ungerade Anzahl von Studierenden begrüßt haben, ungerade ist, sollen Sie 100 Euro von ihm erhalten, wenn die Anzahl jedoch gerade ist, erhält er nur 10 Euro.

   Gehen Sie diese Wette ein? Begründen Sie Ihre Antwort! (**Hinweis:** Modellieren Sie das obige Szenario als Graphen)

8. **Datenstrukturen für Graphen:** Graphen liegen manchmal nicht in dem gewünschten Datenformat vor. Schreiben Sie eine Funktion, die aus einer Adjazenzmatrix die Adjazenzlisten-Darstellung des Graphen erzeugt. (Um Testfälle zu erzeugen, können Sie die Matrix zufällig mit 0 und 1 füllen – um Schlingen zu vermeiden, sollten dabei die Elemente in der Diagonalen 0 sein.)

Geben Sie den Aufwand in O-Notation an.

9. **Graphdurchläufe:** In der Vorlesung haben wir die rekursive Fassung der Tiefensuche kennengelernt. Dort wurde auch skizziert, wie man dieses iterativ formulieren kann, indem man den Keller der noch zu bearbeitenden Knoten selbst verwaltet.

   (a) Schreiben Sie eine iterative Version der Prozedur `Tiefensuche`.

   (b) Ersetzen Sie den Keller in der iterativen Version durch eine Schlange (Sie erhalten so eine Breitensuche). Fügen Sie dabei eine zusätzliche Zahl als Inhalt zu den Knoten hinzu. Diese Zahl soll für den Startknoten (mit dem die Breitensuche gestartet wird) mit 0 initialisiert werden. Wenn die ausgehenden Kanten (u, v) eines Knotens u betrachtet werden, so soll der Wert des jeweils adjazenten Knotens v auf einen eins höheren Wert als der des Knotens u gesetzt werden.

   (c) Bei der Breitensuche geben die Nummern, die den Knoten zugewiesen wurden, jeweils die Länge des kürzesten Wegs zu diesen Knoten an. Begründen Sie, warum dies so ist.

10. **Transitive Hülle:** Die Transitive Hülle G_{tH} eines Graphen G erweitert die Kantenmenge so, dass es zwischen zwei Knoten u und v genau dann eine Kante (u, v) gibt, wenn es im Graphen G einen Weg von u nach v gibt. Folgender Algorithmus ändert die Adjazenzmatrix `A` eines Graphen so ab, dass diese am Ende der Transitiven Hülle des Graphen entspricht.

```
for i in 1..n loop A(i,i):=1 end loop; -- der leere Weg verbindet i mit i
for k in 1..n loop
  for i in 1..n loop
    for j in 1..n loop
      if A(i,k)=1 and A(k,j)=1 then -- es existiert ein Weg über den Knoten k
        A(i,j):=1;
      end if;
    end loop;
  end loop;
end loop;
```

   (a) Begründen Sie, warum der Algorithmus die Transitive Hülle eines Graphen korrekt berechnet (Hinweis: nach dem ersten Durchlauf durch die `k`-Schleife sind alle Kanten `(i,j)` im durch die Adjazenzmatrix $\mathtt{A}_{\mathtt{ij}}$ dargestellten Graphen vorhanden (d.h. `A(i,j)=1`), die entweder bereits im Ausgangsgraphen vorhanden sind oder für die es einen Weg gibt, der nur über den Zwischenknoten `1` führt – setzen Sie den Beweis induktiv fort).

   (b) Geben Sie in O-Notation an, welchen Aufwand der Algorithmus in Abhängigkeit von der Knotenanzahl `n` hat.

   (c) In der Regel liegt der Graph als Adjazenzliste vor, sodass die Abfrage, ob `A(i,j)=1`, nicht in einem Schritt erfolgen kann. Will man direkt auf der Adjazenzliste arbeiten, so muss man zunächst die Knotenliste durchsuchen, um den Knoten `i` zu finden und in dessen Kantenliste überprüfen, ob es dort eine Kante zum Knoten `j` gibt.

   Beschreiben Sie, wie die Knoten- und Kantenlisten angeordnet sein sollten, um unnötige Arbeit zu ersparen.

Schätzen Sie ab, wie groß der Aufwand zur Berechnung der Transitiven Hülle ist, wenn man direkt auf der Adjazenzliste arbeitet.

Wie groß wäre der Aufwand, wenn man zunächst die Adjazenzliste in eine Adjazenzmatrix umwandelt, dann mit der Matrix die Transitive Hülle berechnet und danach die Matrix wieder in eine Adjazenzliste zurückwandelt?

Geben Sie Ihre Abschätzungen jeweils in O-Notation an und begründen Sie Ihre Antworten.

11. **Zusammenhangskomponenten:** Wir wollen hier nun die Tiefensuche verwenden, um die schwachen Zusammenhangskomponenten eines Graphen zu bestimmen. Zum ersten Knoten sollen also alle weiteren Knoten bestimmt werden, die über eine oder mehrere Kanten mit dem ersten Knoten verbunden sind. Sind diese bestimmt, so führen wir den Algorithmus erneut für die verbleibenden Knoten durch. Der Algorithmus läuft in zwei Schritten:

    - Schreiben Sie eine Funktion, die aus einem Graphen G in Adjazenzlistendarstellung einen neuen Graphen G' erzeugt, bei dem für jede Kante (i, j) sowohl (i, j) als auch (j, i) im Graphen vorkommt (achten Sie darauf, dass Sie Kanten nicht doppelt einfügen).
    - Verwenden Sie die Tiefensuche, um die Knoten zu ermitteln, die durch Wege miteinander verbunden sind. Überlegen Sie sich, warum wir im ersten Aufgabenteil dazu die Kanten in beiden Richtungen im Graphen einfügen – was würde sich am Ergebnis ändern, wenn man direkt auf dem ursprünglichen Graphen arbeitet?

    Sie können als Testfälle die Zufallsgraphen von Aufgabe 8 verwenden, vermutlich müssen Sie die Wahrscheinlichkeit, mit der eine Kante im Graphen vorkommt, senken, damit nicht stets alle Knoten über Wege miteinander verbunden sind.

    Geben Sie den Aufwand (als Kommentarzeilen eingefügt) in O-Notation an.

12. **Rückblick auf das erste Semester:** Schreiben Sie (für sich) eine kurze Zusammenfassung des Stoffs des ersten Semesters. Geben Sie eine Liste von 10 Punkten an, von denen Sie denken, dass sie besonders wichtig sind. Diese Punkte können sich sowohl auf den Stoff der Informatik im ersten Semester beziehen als auch allgemein sein (zum Beispiel in Hinblick auf die kommenden Semester).

    Stellen Sie einige Fragen zusammen, die aus Ihrer Sicht noch nicht geklärt wurden.

13. Noch eine kleine Programmier- und Knobelaufgabe zum Abschluss ... **Das ist der Rest ...** Es gibt Zahlentripel (a, b, c), so dass folgende Eigenschaft gilt: $a \cdot b \mod c = 1$ und $b \cdot c \mod a = 1$ und $c \cdot a \mod b = 1$.

    Schreiben Sie ein Ada 95 Programm, das alle Zahlentripel (a, b, c) mit $a \leq b \leq c$ und $a, b, c \leq 100$ auf obige Eigenschaft überprüft.

    Zusatzaufgabe für Mathefreaks und Zahlenknobler: Beweisen Sie, dass es keine weiteren Zahlentripel mit dieser Eigenschaft als die oben gefundenen gibt.