Einführung in die Informatik I (autip)

Dr. Stefan Lewandowski

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

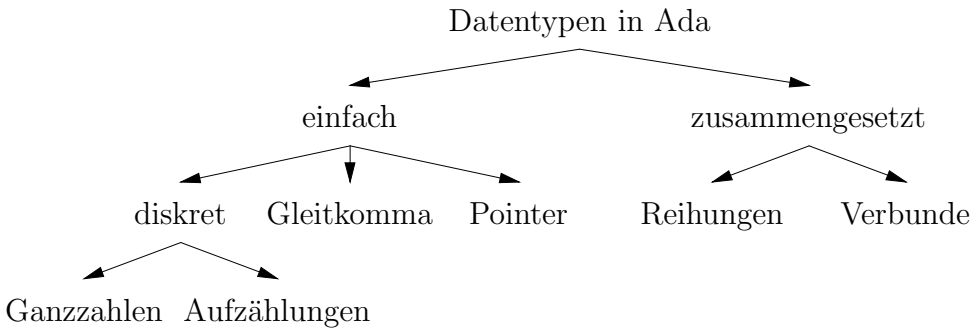
18.+25.1.+1.2.2006 / Version 2. Februar 2006

5 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:



5.1 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;
```

5.2 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..55) 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.

5.3 Pointer, Listen, Bäume und Graphen

5.3.1 Einführendes Beispiel in Pointer und Listen

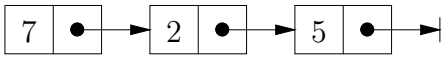
Greifen wir nochmal das Beispiel der Autipsverwaltung auf. Die Feldgröße (also die Anzahl der Studierenden) steht anfangs nicht unbedingt fest. Wir können prinzipiell zwar ein Array mit offenen Indexgrenzen (`range <>`) definieren, aber sowie wir das Array mit Daten füllen wollen, ist die Größe unveränderlich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

[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.

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);
    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.

5.3.2 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.

5.3.3 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.

5.3.4 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.

5.3.5 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.

5.3.6 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.