Speicher, Pointer & Datenstrukturen

Was tun, wenn Variablen & Arrays nicht mehr reichen?

Möglichkeiten und Grenzen von Variablen & Arrays

Variablen und Arrays müssen immer im Deklarationsbereich des Moduls beziehungsweise der Prozedur fix definiert werden. Man kann in ihnen Daten speichern und später wieder darauf zugreifen; nur, was tut man, wenn ich jetzt zum Beispiel bei der Programmierung noch nicht weiss, wie lange eine Liste werden soll? Ein Array der Länge 500 mag vielleicht unbeschränkt wirken, doch stossen wir bei der Programmierung so oft an die Grenzen. Es wäre also durchaus praktisch während der Laufzeit eine neue Variable zu kreieren oder ein neues Record eines bestimmten Typs zu kreieren.

Neuer, weiterer Speicherplatz während der Laufzeit allozieren

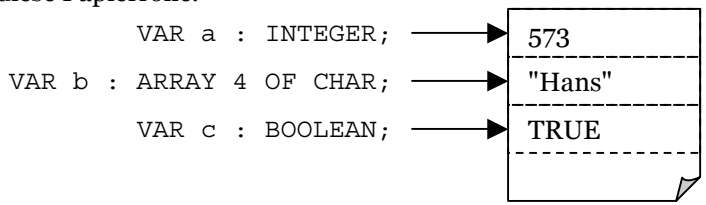
Wie erstellt man ein solches zur Laufzeit erstelltes Objekt, und wie kann ich darauf zugreifen? Wir benötigen dazu zwei Dinge:

- Ein Mechanismus, mit dem man zur Laufzeit neue Objekte im Speicher allozieren kann
- Ein Hilfsmittel, mit dem ich auf dieses neue Objekt zugreifen kann

Diese beiden Aufgaben werden von der Prozedur `NEW()` und von den Pointer zur Verfügung gestellt.

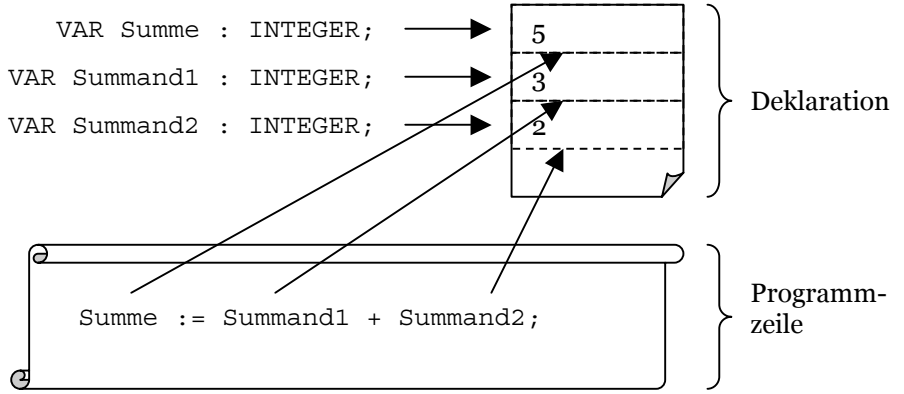
Was ist Speicher?

Ein Programm hat immer Speicher des Computers als Hilfe zur Verfügung. Im Allgemeinen handelt es sich dabei um den *Random Access Memory* - den RAM. Dieser RAM kann man sich als ganz lange Papierrolle vorstellen. Und immer, wenn sich der Computer etwas merken muss, dann schreibt er irgendwo etwas auf diese Papierrolle.



Zum Beispiel sind Variablen fest vergebene Stellen auf dieser Papierrolle. Bei der Deklaration weiss der Computer, dass er im kommenden Programmabschnitt sich Werte gegebenen Typs merken muss; und dafür alloziert er Bereiche im Speicher, das heisst, er reserviert sich Zeilen auf dieser Papierrolle und merkt sich deren Zeilennummern.

Immer, wenn man jetzt im Programmcode zum Beispiel die im Bild zuvor deklarierte Variable `c` verwendet, weiss der Computer, dass die entsprechende Zeile gemeint ist:

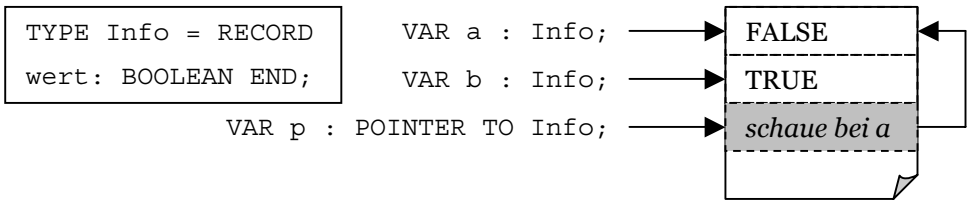


Also macht die Programmzeile Folgendes:

- Schaue, was in der Zeile von `Summand1` steht
- Schaue, was in der Zeile von `Summand2` steht
- Schreibe die Summe der Beiden in die Zeile von `Summe`

Was sind Pointer?

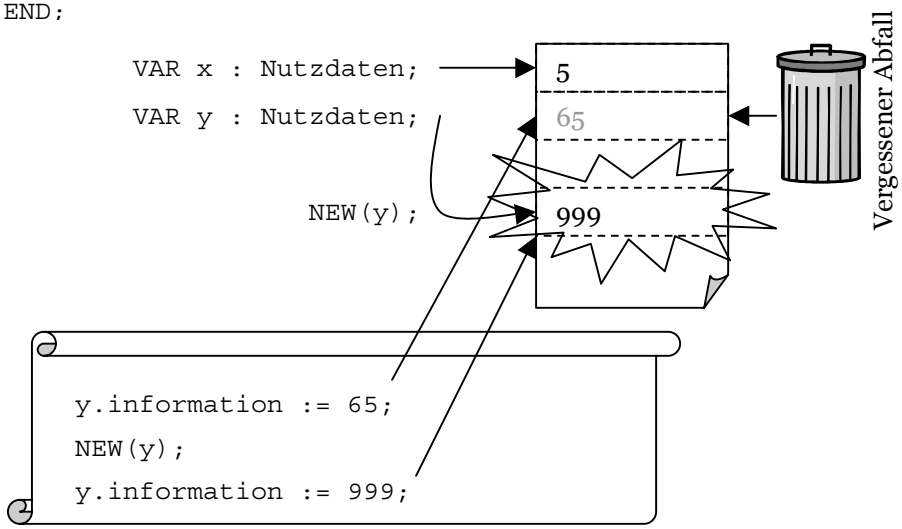
Ein Pointer ist keine Variable im herkömmlichen Sinne. Variablen stehen für eine fixe Zeile im Speicher. Man kann während der Laufzeit keine neuen Zeilen mehr allozieren. Pointer hingegen zeigen auf eine Zeile. Man kann während der Laufzeit des Programmes sagen, auf welche Zeile der Pointer zeigen soll:



Alleine scheint das noch nicht allzuviel zu bringen. Ich möchte ja dynamisch neue Zeilen während der Laufzeit des Programmes erstellen. Dies kann ich aber mit Hilfe der Deklarationen ja nur während dem Programmieren tun.

Wir möchten also neue Bereiche im Speicher während der Laufzeit allozieren. Für diesen Zweck stellt Oberon die Prozedur `NEW()` zur Verfügung:

```
TYPE Nutzdaten = POINTER TO RECORD
      information : INTEGER;
END;
```



Mit NEW(b) wird im Speicher eine neue Zeile alloziert, und b zeigt dann auf diese neue Zeile. In dem Beispiel oben verliert man jeglichen Zugriff auf die zweite Zeile, weil man sich zuvor die Zeilennummer des 65 nicht gemerkt hat. Diese Zeile ist also sozusagen vergessen worden, aber besetzt immer noch den Speicher.

Wenn man einen Pointer deklariert hat, zum Beispiel `VAR x : Nutzdaten`, dann zeigt dieser am Anfang noch ins Nirwana des Speichers beziehungsweise auf `NIL`. `NIL` ist das Schlüsselwort für "ich zeige auf nichts".

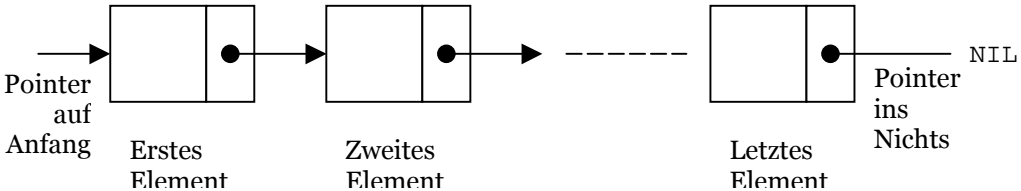
Garbage Collector: Was passiert mit vergessenem alloziertem Speicher?

Im vorherigen Beispiel ist der Fall aufgetreten, dass ein Speicherbereich besetzt ist, aber eigentlich nicht mehr benutzt werden kann. In diesem Fall spricht man von Müll (englisch Garbage) im Speicher. Wenn es zu viel Müll im Speicher hat, dann ist er irgendwann überfüllt, und man kann nicht mehr weiter arbeiten. Deshalb gibt es in Oberon einen kleinen Helfer, der immer im Hintergrund den Abfall entsorgt, indem er immer für jeden Speicherbereich schaut, ob er noch benötigt wird, oder ob er vergessen wurde. Dieser Helfer ist der Garbage Collector. Wenn er einen Bereich findet, der vergessen wurde, dann räumt er ihn weg und macht ihn wieder für neue `NEW()` Aufrufe zur Alloziierung frei.

Listen: Wie kann ich flexiblere Ketten von Variablen machen als Arrays sie bieten?

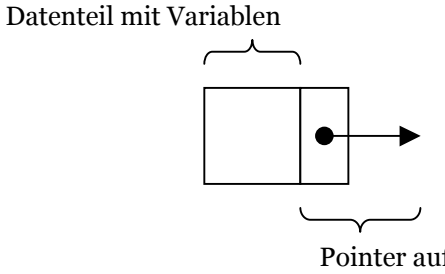
Was sind Listen & was sind deren Vorteile?

Listen sind dynamische Konstrukte, welche Daten in einer Reihenfolge aufnehmen können. Sie bestehen aus einem Anfang, aus Elementen und aus einem Ende.



Listen können, eben weil sie dynamisch sind, die Anzahl Elemente ändern; man kann Elemente einfügen und löschen, und man kann mehrere Listen zusammenhängen oder eine Liste in mehrere kleinere aufsplitten.

Ein einzelnes Element besteht aus einem Datenteil, Variablen, und aus einem Verweis auf das nächste nachfolgende Element, dem Pointer:



Wie erstelle ich Listen in Oberon?

Zuerst muss man einen Typen Element erstellen:

```
TYPE Element = POINTER TO RECORD
   wert: INTEGER;
   naechster: Element;
END;
```

Dieser Typ ist also unser Element, welches in diesem Beispiel Zahlen als Datenteil hat und einen Pointer auf ein nächstes Element. Nun brauchen wir aber noch einen Anfang, und dieser Anfang ist selbst auch ein Element:

```
VAR anfang : Element;
```

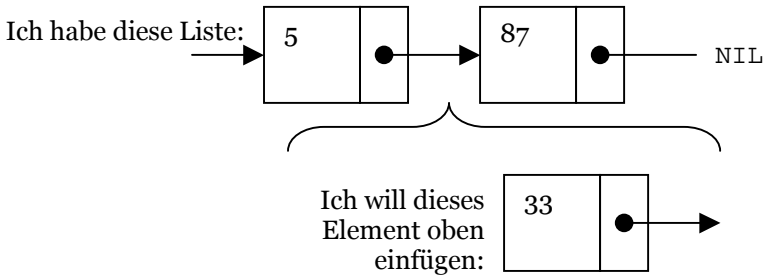
Jetzt haben wir eigentlich die Datenstruktur, die wir zum Arbeiten brauchen. Wir müssen sie jedoch zuerst initialisieren, denn noch ist die Variable `anfang` leer, das heisst, sie zeigt ins Nichts beziehungsweise auf `NIL`. Also schreiben wir eine Prozedur `init()`.

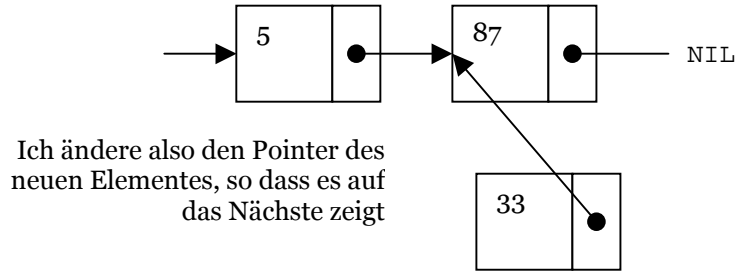
```
PROCEDURE init(VAR diese_liste : Element);
BEGIN
   NEW(diese_liste);
END init;
```

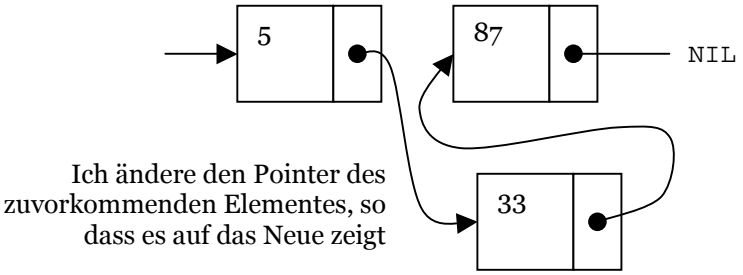
Wenn wir jetzt also als Parameter einer Funktion den Anfang einer Liste bekommen, dann können wir die Liste anschauen, durchlaufen, neue Elemente einfügen, bestehende löschen und die Reihenfolge der Elemente beliebig ändern.

Wie füge ich Elemente ein?

Das Einfügen in eine Liste funktioniert folgendermassen:







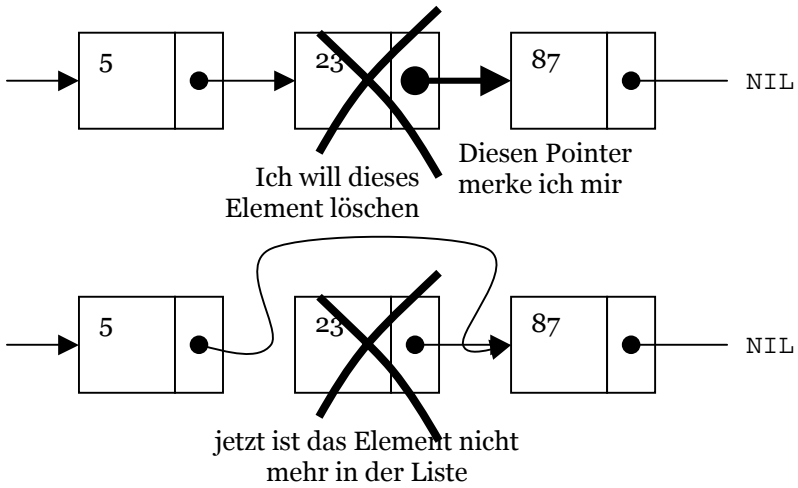
Und so habe ich auf einfachste Weise ein neues Element in die Liste eingefügt. Die Prozedur in Oberon dazu sieht folgendermassen aus:

```
PROCEDURE einfuegen(VAR nach_diesem, einzufuegendes:
Element);
BEGIN
   einzufuegendes.naechster := nach_diesem.naechster;
   nach_diesem.naechster := eunzufuegendes;
END einfuegen;
```

Diese Prozedur nimmt zwei Elemente als Parameter: der erste Parameter sagt, nach welchem Element eingefügt werden soll; der zweite Parameter sagt, welches Element eingefügt werden soll.

Wie entferne ich Elemente?

Das Entfernen passiert auf die gleiche simple Art. Zuerst wird der Pointer des zu löschenden Elementes auf das nächste Element gerettet und dann im vorhergehdenen Element gespeichert:



Durch das Umhängen des Pointers des Elementes mit Wert 5 auf das Element mit der 87, wird das Element mit Wert 23 aus der Liste entfernt. Weil dann nichts mehr auf das Element mit dem Wert 23 zeigt, wird es der Garbage Collector irgendwann einsammeln und den Speicher wieder freigeben.

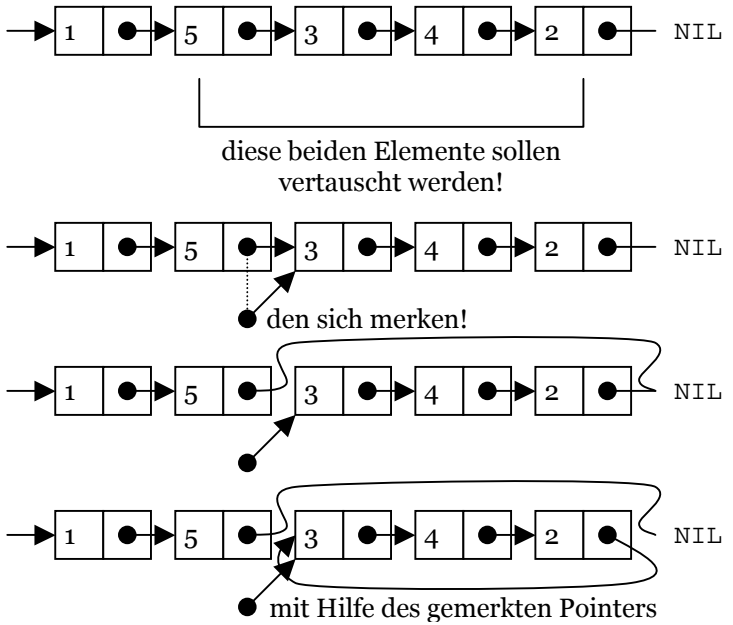
Wir müssen also nur das vorhergehende Element des zu löschenden Elementes herausfinden, und dann können wir umhängen:

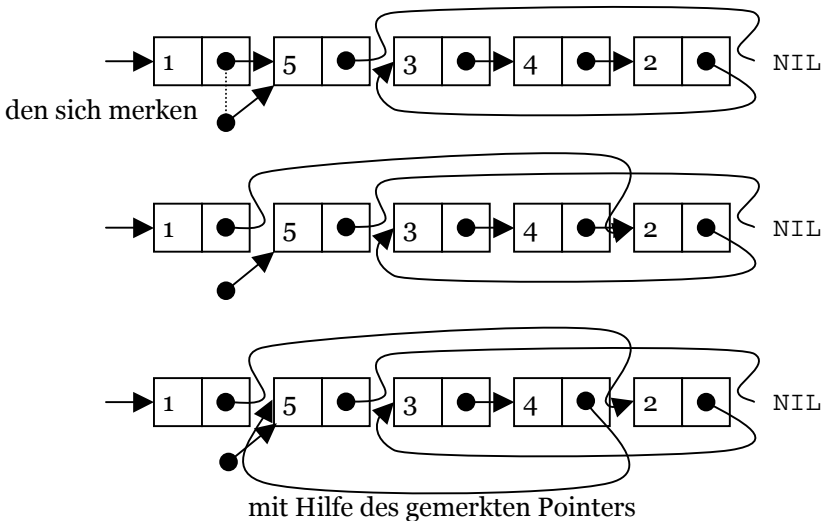
```
PROCEDURE entferne(VAR anfang, zuloeschendes: Element);
VAR aktuell: Element;
BEGIN
   aktuell := anfang;
   WHILE (aktuell.naechster # NIL) &
         (aktuell.naechster # zuloeschendes) DO
         aktuell := aktuell.naechster;
   END;
   IF aktuell.naechster # NIL THEN
         aktuell.naechster := zuloeschendes.naechster;
   ELSE
         (* Fehler! Element ist nicht in *)
         (* der gegebenen Liste *)
   END;
END entferne;
```

Wie Ihr seht, scheint diese Prozedur schon ein bisschen komplexer! Man muss jetzt schon einige Spezialfälle behandeln. Am Anfang reitet man durch die Liste hindurch und man muss immer schauen, ob man schon am Ende ist, oder ob das nächste Element das zu löschende ist. Zudem muss man dann unterscheiden, nachdem man die Liste traversiert hat, ob man am Ende ist, ohne das gegebene zu löschende Element gefunden zu haben.

Wie verändere ich die Reihenfolge?

Um die Reihenfolge einer Liste zu ändern, benutzt man die gleiche Methode wie beim Einfügen und Löschen: einfach die Pointer so vertauschen, dass alles so ist, wie man will. Zum Beispiel das Auswechseln zweier Elemente setzt sich aus folgenden Schritten zusammen:





Bei dieser Operation ist es schon nötig, einzelne Pointer zwischenzuspeichern, bevor sie von einem andern überschrieben werden. Dies ist nötig, damit man dann noch Zugriff darauf hat. Schon bei der Swap-Funktion zweier Variablen ist ein Zwischenspeichern nötig, so auch bei dieser Swap-Funktion zweier Elemente einer Liste.

Den Quellcode zu diesem Swap gebe ich hier nicht an, Ihr werdet das in den Übungen selbst programmieren.

Bäume: Kann ich auch Daten in baumartigen Strukturen organisieren?

Was sind Bäume & was sind deren Vorteile ?

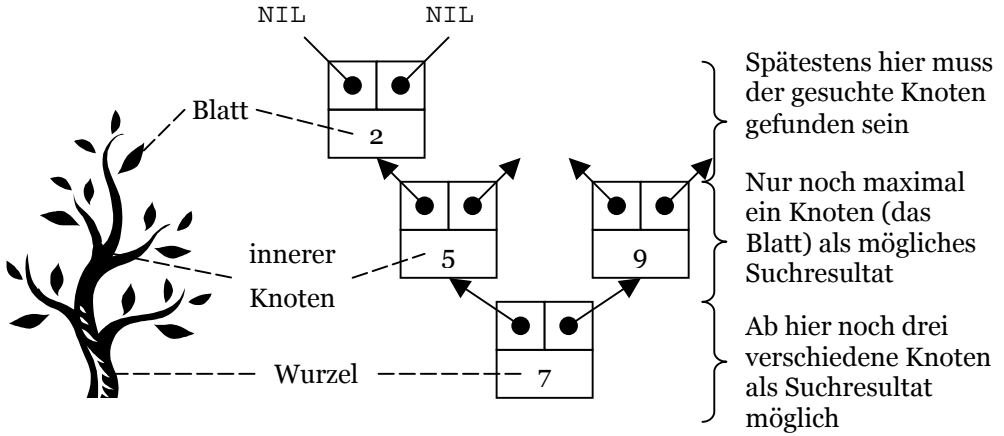
Als Bäume werden in der Informatik alle baumähnlichen Datenstrukturen bezeichnet. Das heisst, die Datenstruktur hat eine **Wurzel** und Astgabeln. An den Astgabeln sind jeweils Nutzdaten gespeichert. Diese Astgabeln werden in der Informatik Knoten genannt. **Knoten** ohne weitere nachfolgende Knoten nennt man **Blätter**, solche mit nachfolgenden Knoten heissen **innere Knoten**.

Die Anzahl Äste an der Astgabel spezifizieren den Baum zusätzlich. Hat jede Astgabel genau zwei Äste, dann spricht man vom binären Baum.

Der Vorteil von Bäumen gegenüber Listen, liegt in der Suche nach solchen Nutzdaten an den Astgabeln in geordneten Bäumen. Sucht man zum Beispiel nach einem Wert in einem geordneten binären Baum, kann man folgende Regel definieren:

> *Immer wenn der Wert an der Astgabel grösser ist als der gesuchte Wert, dann gehe auf dem linken Ast weitersuchen, sonst gehe beim rechten Ast weitersuchen.*

Wenn also der Baum geordnet ist, dann verkürzt sich die Suche logarithmisch, weil ich mit jedem Schritt im Durchschnitt die Hälfte aller noch in Frage kommenden Astgabeln ausschliessen kann, weil ja immer am linken Ast alle kleineren Werte und im rechten Ast alle grösseren Werte stecken. Im Gegensatz zur Liste, wo immer wahllos alle Elemente durchlaufen werden müssen.



Der Schreibarbeit halber werden in der Informatik die Bäume immer von oben nach unten aufgezeichnet, im Gegensatz zu richtigen Bäumen. Also stimmt die obere Abbildung nicht ganz, was die Richtung angeht.

Welche Arten von Bäumen gibt es?

Es gibt unzählige Arten von Bäumen, es handelt sich dabei gar um ein eigenständiges Forschungsgebiet in der Informatik.

Bei einigen Bäumen gibt es mehrere Äste, unterschiedlich viele Äste etc. Bei andern Bäumen werden die Daten nur in den Blättern gespeichert, und die Daten in den inneren Knoten dienen nur der schnelleren Suche. Andere Bäume enthalten an jedem Knoten wieder einen eigenen Baum. Aber im Wesentlichen

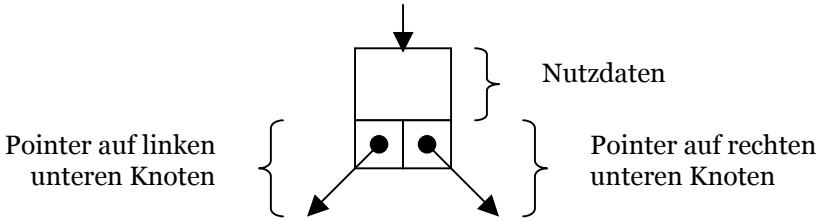
lassen sich all diese Abarten von Bäumen mit dem simplen binären Baum implementieren.

Wie deklariere ich Bäume?

Zuerst muss man einen Typen Knoten erstellen:

```
TYPE Knoten = POINTER TO RECORD
   wert: INTEGER;
   links, rechts: Knoten
END;
```

Dies wäre jetzt der Knotentyp für einen Baum, der Integerzahlen speichern kann. Der `links` Knoten meint den linken unteren Knoten und der rechts Knoten meint den rechten unteren Knoten:



Die Wurzel besteht aus einem Pointer auf den ersten Knoten:

```
VAR wurzel : Knoten;
```

Jetzt müssen wir wie bei der Liste noch ein Exemplar im Speicher erstellen, bevor wir beginnen können. Dazu benutzen wir eine Prozedur `init()`.

```
PROCEDURE init(VAR diesen_baum : Knoten);
BEGIN
   NEW(diesen_baum);
END init;
```

Wie füge ich Elemente ein?

Da Bäume sinnvollerweise immer geordnet sind, benötigt man für das Einfügen in einen Baum lediglich die Angaben, in welchen Baum und welcher Wert.

```
PROCEDURE einfügen(VAR baum : Knoten, wert : INTEGER);
VAR neu : Knoten;
BEGIN
   IF baum = NIL THEN
         NEW(neu);
         neu.wert := wert;
         (* Baum ist leer *)
         baum := neu;
   ELSIF baum.wert = wert THEN
         (* Knoten schon im Baum drinnen *)
   ELSIF baum.wert < wert THEN
         (* suche im rechten Teilbaum weiter *)
         einfuegen( baum.rechts, wert);
   ELSIF baum.wert > wert THEN
         (* suche im linken Teilbaum weiter *)
         einfuegen( baum.links, wert);
   END
END einfuegen;
```

Diese Prozedur mag bei der ersten Betrachtung kompliziert erscheinen; aber die einzige Schwierigkeit ist die Rekursion.

Die If-Schlaufe behandelt vier Fälle:

- Der Baum, in welchen eingefügt werden soll, ist leer:
  Also mache einen neuen Knoten mit dem Wert und setze in als Wurzel
- Die Wurzel des übergebenen Baumes hat bereits den gleichen Wert wie der einzufügende Wert; der Knoten existiert also schon:
  Also muss man nichts tun
- Der einzufügende Wert ist grösser als der Wert der übergebenen Wurzel:
  Also muss man den Wert im rechten Teilbaum einfügen, weshalb wir die Rekursion machen
- Der einzufügende Wert ist kleiner als der Wert der übergebenen Wurzel:

Also muss man den Wert im linken Teilbaum einfügen, weshalb wir die Rekursion machen

Die Rekursion wird solange ausgeführt, bis man zu einem Knoten gelangt, der den Wert schon hat, oder bis man zu einem `NIL` Pointer gelangt und dann dort einen Knoten erstellt.

Mehr Material

Komplette Erklärung und Sammlung von Beispielcode in Oberon für die ganze Vorlesung Informatik I

`http://www.mathematik.uni-ulm.de/sai/ss99/prog/skript/index.html`

Skript der Fachhochschule Mannheim – die verwenden auch Oberon

`http://telematik.fh-mannheim.de/Dokumente/GDI_Algorithmen-DynDS-2.pdf`

Beispiele zu Operationen auf Bäumen in Oberon

`http://www.mindrup.de/atos/online/9605/dyndat2.htm`

Sehr theoretische Definition von Bäumen inkl. Vor- & Nachbedingungen

`http://www.stud.fernuni-hagen.de/q4468120/Skripte/Da/da.doc`

Ausführliches und verständliches Skript zu Bäumen

`http://www.informatik.hu-berlin.de/~weissled/suchbaum/Vortrag.doc`