Deckblatt Kurseinheit 2 Kurs 1663 Datenstrukturen und Kurs 1661 Datenstrukturen I:

Datenstrukturen

Kurseinheit 2

Grundlegende Datentypen

Autoren: Ralf Hartmut Güting und Stefan Dieker

algebra $list_1$
sorts *list*, *elem* {*bool* ist im folgenden implizit immer dabei}

ops	*empty*	:	$\rightarrow$ *list*
	first	: *list*	$\rightarrow$ *elem*
	rest	: *list*	$\rightarrow$ *list*
	append	: *list* $\times$ *elem*	$\rightarrow$ *list*
	concat	: *list* $\times$ *list*	$\rightarrow$ *list*
	isempty	: *list*	$\rightarrow$ *bool*
sets	*list*	$= \{ <a_1, ..., a_n> \mid n \geq 0, a_i \in elem\}$	
functions			
	empty		$= \Diamond$
	first $(a_1 \ldots a_n)$		$= \begin{cases} a_1 & \text{falls } n > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$
	rest $(a_1 \ldots a_n)$		$= \begin{cases} a_2 \ldots a_n & \text{falls } n > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$
	append $(a_1 \ldots a_n, x)$		$= x\ a_1 \ldots a_n$
	concat $(a_1 \ldots a_n, b_1 \ldots b_m)$		$= a_1 \ldots a_n \circ b_1 \ldots b_m$
	isempty $(a_1 \ldots a_n)$		$= (n = 0)$

end $list_1$.

Inhalt der Kurseinheit 2

Lehrziele

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- wissen, was man unter den Begriffen *Liste*, *Stack*, *Queue*, *Mapping* und *Baum* versteht;
- die unterschiedlichen Operationen auf diesen Datenstrukturen kennen;
- den Begriff des *parametrisierten Datentyps* definieren können;
- beurteilen können, wann man welche der obigen Datenstrukturen sinnvoll einsetzt;
- für jede der Datenstrukturen verschiedene Implementierungen sowie deren Vor- und Nachteile bei unterschiedlichen Anwendungen angeben können;
- den Zusammenhang zwischen Stacks und Rekursion bzw. Möglichkeiten zur Umwandlung von rekursiven Algorithmen in iterative kennen;
- einige Eigenschaften von Bäumen benennen (und ggf. beweisen können), z.B. die minimale (maximale) Baumhöhe etc.;
- die unterschiedlichen Durchlaufarten von binären Bäumen kennen.

3 Grundlegende Datentypen

In diesem Kapitel führen wir einige elementare Datentypen ein, die Bausteine für die Implementierung komplexer Algorithmen und Datenstrukturen bilden, nämlich *Listen*, *Stacks*, *Queues*, *Abbildungen* und *Bäume*. Für *Listen* werden beispielhaft zwei verschiedene Modelle (Algebren) definiert. Die erste Algebra ist einfach und bietet die Grundoperationen an. Die zweite Algebra ist komplexer, da explizit Positionen in einer Liste verwaltet werden, entspricht dafür aber eher der üblichen Benutzung von Listen in einer imperativen oder objektorientierten Programmiersprache. Anschließend wird eine Reihe von Implementierungsoptionen vorgestellt (einfach und doppelt verkettete Listen, Einbettung in einen Array). *Stacks* und *Queues* sind Listentypen mit eingeschränkten Operationssätzen; ein Stack ist eine nur an einem Ende zugängliche Liste, eine Queue eine Liste, bei der an einem Ende eingefügt, am anderen Ende entfernt wird. Der spezielle Operationssatz legt jeweils eine bestimmte einfache Implementierung nahe. Der Datentyp *Abbildung* stellt unter anderem die Abstraktion eines Arrays auf der programmiersprachlichen Ebene dar. Schließlich werden in diesem Kapitel *Bäume* als allgemeine Struktur eingeführt; *Suchbäume* betrachten wir im vierten Kapitel im Kontext der Datentypen zur Darstellung von Mengen. Hier untersuchen wir zunächst den einfacheren und wichtigeren Spezialfall der *binären* Bäume und danach Bäume mit beliebigem Verzweigungsgrad.

3.1 Sequenzen (Folgen, Listen)

Wir betrachten Datentypen, deren Wertemenge jeweils die endlichen Folgen (*Sequenzen*) von Objekten eines gegebenen Grundtyps umfaßt. Solche Datentypen unterscheiden sich in bezug auf die angebotenen Operationen. Im Unterschied zu Mengen ist auf Sequenzen eine *Ordnung* definiert, es gibt also ein erstes, zweites, ... Element und zu jedem Element (außer dem ersten und letzten) einen Vorgänger und Nachfolger. Weiterhin dürfen Sequenzen *Duplikate*, also mehrfach auftretende Elemente der Grundmenge, enthalten.

Um mit Sequenzen auch formal umgehen zu können, führen wir einige Notationen ein. Es bezeichne

$$a_1 \ldots a_n$$

eine Sequenz mit n Elementen. Wir schreiben auch

$$<a_1, \ldots, a_n>,$$

wobei die spitzen Klammern explizit ausdrücken, daß es sich um eine Sequenz handelt. Insbesondere muß man eine einelementige Sequenz $<a>$ so von dem Objekt a unterscheiden. Weiterhin stellt das Symbol

$\Diamond$

die leere Sequenz dar (entsprechend $a_1 \ldots a_n$ mit $n = 0$) und es bezeichnet

$\circ$

einen Konkatenationsoperator für Sequenzen, definiert als

$$<a_1, \ldots, a_n> \circ <b_1, \ldots, b_m> = <a_1, \ldots, a_n, b_1, \ldots, b_m>.$$

Implementierungen, also Datenstrukturen für Sequenzen, werden meistens *Listen* genannt. Listen haben vielfältige Anwendungen in Algorithmik bzw. Programmierung.

3.1.1 Modelle

Wir entwerfen beispielhaft zwei Datentypen für Listen. Der erste stellt einige einfache Grundoperationen bereit, erlaubt allerdings nur die Behandlung einer Liste in ihrer Gesamtheit und die rekursive Verarbeitung durch Aufteilung in erstes Element und Restliste. Man kann also beispielsweise nicht direkt auf das vierte Element einer Liste zugreifen. Der zweite Datentyp entspricht mehr vielen praktischen Anwendungen, in denen explizite Zeiger auf Listenelemente verwaltet werden und z.B. Elemente in der Mitte einer Liste eingefügt oder entfernt werden können. Dafür ist bei diesem Datentyp die Modellierung etwas komplexer.

(a) Listen mit first, rest, append, concat

Dieser Datentyp bietet zunächst zwei Operationen *empty* und *isempty* an, die eine leere Liste erzeugen bzw. eine gegebene Liste auf Leerheit testen. Derartige Operationen braucht man bei praktisch jedem Datentyp, wir haben deshalb in der Überschrift nur die eigentlich interessanten Operationen erwähnt. Mit *first* erhält man das erste Element einer Liste, mit *rest* die um das erste Element reduzierte Liste. Die Operation *append* fügt einer Liste “vorne” ein Element hinzu, *concat* verkettet zwei Listen. Das wird durch die folgende Algebra spezifiziert.

algebra $list_1$
sorts *list*, *elem* {*bool* ist im folgenden implizit immer dabei}

ops	*empty*	:	$\rightarrow$ *list*
	first	: *list*	$\rightarrow$ *elem*
	rest	: *list*	$\rightarrow$ *list*
	append	: *list* $\times$ *elem*	$\rightarrow$ *list*
	concat	: *list* $\times$ *list*	$\rightarrow$ *list*
	isempty	: *list*	$\rightarrow$ *bool*
sets	*list*	$= \{ <a_1, ..., a_n> \mid n \geq 0, a_i \in elem\}$	
functions			
	empty	$= \Diamond$	
	first $(a_1 \ldots a_n)$	$= \begin{cases} a_1 & \text{falls } n > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$	
	rest $(a_1 \ldots a_n)$	$= \begin{cases} a_2 \ldots a_n & \text{falls } n > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$	
	append $(a_1 \ldots a_n, x)$	$= x\ a_1 \ldots a_n$	
	concat $(a_1 \ldots a_n, b_1 \ldots b_m)$	$= a_1 \ldots a_n \circ b_1 \ldots b_m$	
	isempty $(a_1 \ldots a_n)$	$= (n = 0)$	

end $list_1$.

(b) Listen mit expliziten Positionen

Hier werden explizit "Positionen" innerhalb einer Liste verwaltet, die in der Implementierung typischerweise durch Zeigerwerte realisiert werden.

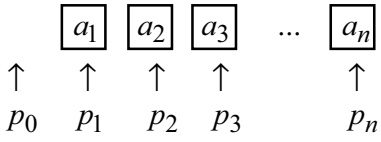


Abbildung 3.1: Zeiger als Positionswerte

Es ist zweckmäßig, auch eine gedachte Position p_0 "vor" einer Liste einzuführen. Wir werden neue Elemente immer hinter einer angegebenen Position einfügen, und so ist es möglich, ein Element am Anfang einer Liste oder in eine leere Liste einzufügen. Eine leere Liste enthält lediglich die Position p_0.

Neben den bereits bekannten Operationen *empty*, *isempty* und *concat* bietet die folgende Algebra zunächst einige Operationen zur Zeigermanipulation an. *Front* und *last* liefern

die Position des ersten[1] und letzten Listenelementes, *next* und *previous* zu einer gegebenen Position die Nachfolger- bzw. Vorgängerposition, *bol* ("begin of list") und *eol* ("end of list") stellen fest, ob die gegebene Position die des ersten oder letzten Elementes ist (nützlich für Listendurchläufe). Hinter einer gegebenen Position kann ein Element eingefügt werden (*insert*) oder das Element an der Position entfernt werden (*delete*).

Die Funktion *find* liefert die Position des ersten Elementes der Liste, das die Parameterfunktion wahr macht, also die dadurch gegebenen "Anforderungen erfüllt". Beispielsweise kann $elem = int$ und die Parameterfunktion von *find* die Überprüfung eines Elements auf Gleichheit mit einer Konstanten, z.B. $f(x) = (x = 17)$ sein. Schließlich kann man das an einer gegebenen Position vorhandene Element mit *retrieve* bekommen. Dies alles wird in der folgenden Algebra präzise spezifiziert.

algebra $list_2$

sorts *list*, *elem*, *pos*

ops

empty	:	$\rightarrow list$
front, *last*	: *list*	$\rightarrow pos$
next, *previous*	: $list \times pos$	$\rightarrow pos \cup \{null\}$
bol, *eol*	: $list \times pos$	$\rightarrow bool$
insert	: $list \times pos \times elem$	$\rightarrow list$
delete	: $list \times pos$	$\rightarrow list$
concat	: $list \times list$	$\rightarrow list$
isempty	: *list*	$\rightarrow bool$
find	: $list \times (elem \rightarrow bool)$	$\rightarrow pos \cup \{null\}$
retrieve	: $list \times pos$	$\rightarrow elem$

sets

Sei *POS* eine nicht näher spezifizierte unendliche Menge (Adressen, wobei wir annehmen, daß es beliebig viele gibt). Dabei gilt $null \notin POS$; *null* ist ein spezieller Wert für eine nicht existente Position.

$$list = \{(a_1 \ldots a_n, p_0 \ldots p_n) \mid n \geq 0, a_i \in elem, p_i \in POS, p_i \neq p_j \text{ falls } i \neq j\}$$

$$pos = POS$$

1. genauer: die Position vor dem ersten Listenelement

functions

$empty \quad = (\Diamond, <p_0>)$

Sei für die restlichen Funktionsdefinitionen $l = (a_1 \ldots a_n, p_0 \ldots p_n)$.

$front\ (l) \quad = p_0$

$$last\ (l) \quad = \begin{cases} p_n & \text{falls } n > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

$$next\ (l, p) \quad = \begin{cases} p_{i+1} & \text{falls } \exists i \in \{0,\ldots,n-1\} : p = p_i \\ null & \text{sonst} \end{cases}$$

$$previous\ (l, p) \quad = \begin{cases} p_{i-1} & \text{falls } \exists i \in \{1,\ldots,n\} : p = p_i \\ null & \text{sonst} \end{cases}$$

$bol\ (l, p) \quad = (p = p_0)$

$eol\ (l, p) \quad = (p = p_n)$

Für *insert* sei $p = p_i \in \{p_0, ..., p_n\}$. Sonst ist *insert* undefiniert. Sei $p' \in$ POS $\setminus \{p_0, ..., p_n\}$.

$insert\ (l, p, x) \quad = (<a_1, ..., a_i, x, a_{i+1}, ..., a_n>,$
$<p_0, ..., p_i, p', p_{i+1}, ..., p_n>)$

Für *delete* sei $p = p_i \in \{p_1, ..., p_n\}$. Sonst ist *delete* undefiniert.

$delete\ (l, p) \quad = (<a_1, ..., a_{i-1}, a_{i+1}, ..., a_n>,$
$<p_0, ..., p_{i-1}, p_{i+1}, ..., p_n>)$

Für *concat* sei $\{p_0,...p_n\} \cap \{q_1,...q_m\} = \varnothing$. Sonst ist *concat* undefiniert.

$concat\ ((a_1 \ldots a_n, p_0 \ldots p_n), (b_1 \ldots b_m, q_0 \ldots q_m))$
$= (< a_1,...,a_n,b_1,...,b_m >,< p_0,...,p_n,q_1,...,q_m >)$

$isempty\ (l) \quad = (n = 0)$

$$find\ (l, f) \quad = \begin{cases} p_i & \text{falls } \exists i : f(a_i) = \text{true} \wedge \\ & \quad \forall j \in \{1,\ldots,i-1\} : f(a_j) = \text{false} \\ null & \text{sonst} \end{cases}$$

Für *retrieve* sei $p = p_i \in \{p_1, ..., p_n\}$. Sonst ist *retrieve* undefiniert.

$retrieve\ (l, p) \quad = a_i$

end $list_2$.

Der Datentyp $list_2$ ist ebenso wie $list_1$ parametrisiert mit dem Elementtyp *elem* und (dadurch auch) dem Funktionstyp $elem \rightarrow bool$. Der Anwender kann also den Typ *elem* noch spezifizieren und beliebige Werte von *elem* und Instanzen des Funktionstyps (also Funktionen) verwenden. Aufgrund der Parametrisierung sollte der Typ strenggenommen *list* (*elem*) heißen. *list* ist damit ein *Typkonstruktor* (wie *array*, *class* auf der programmiersprachlichen Ebene); mit ihm konstruierte Typen könnten etwa *list* (*integer*), *list* (*Person*) oder *list* (*list* (*Person*)) sein.

Warum verwenden wir nicht einfach natürliche Zahlen 0, ..., *n* als Positionen für eine Liste $<a_1, ..., a_n>$? Manches wäre einfacher, z.B. könnte man die Operation *next* spezifizieren durch next (p) = p + 1. Aber das Modell würde dann verschiedene Aspekte nicht ausdrücken, z.B. daß man Positionen tatsächlich nur durch Nachsehen in einer Liste erhalten kann, oder daß beim Einfügen oder Entfernen die Positionen der Folgeelemente gültig bleiben (die vielleicht vorher in "Positionsvariablen" übernommen worden sind).

Wir haben damit eine komplette Spezifikation einer noch zu implementierenden Datenstruktur und somit eine saubere Schnittstelle zwischen Anwender und Implementierer. Über die Realisierung möchte der Anwender eigentlich nichts weiter wissen, höchstens noch die Laufzeiten der Operationen und den Platzbedarf der Struktur. Der Implementierer ist frei, irgendeine Struktur zu wählen, die das Modell möglichst effizient realisiert. Dafür gibt es durchaus eine Reihe von Alternativen, von denen wir einige jetzt besprechen.

3.1.2 Implementierungen

(a) Doppelt verkettete Liste

Diese Struktur unterstützt effizient sämtliche vom Datentyp $list_2$ angebotenen Operationen. Von jedem Listenelement geht dabei jeweils ein Zeiger auf das Nachfolger- (*succ*) sowie auf das Vorgänger-Element (*pred*) aus.

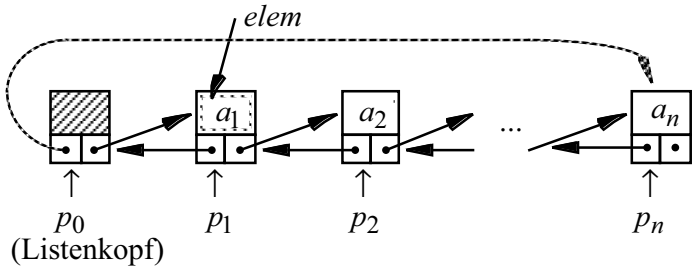


Abbildung 3.2: Doppelt verkettete Liste

Die leere Liste sieht bei dieser Darstellung so aus:

Abbildung 3.3: Leere Liste

Der nicht als solcher benötigte Vorgängerzeiger im Listenkopf wird eingesetzt, um auf das letzte Element zu zeigen. Dadurch lassen sich die Operationen *last* und *concat* effizient implementieren.

Die gezeigte Darstellung läßt sich etwa mit folgenden Deklarationen realisieren:

```
interface Elem
{
   String toString();
   ...
}
```

Mit dieser *Interface*-Deklaration legen wir fest, welche Methoden eine Klasse mindestens implementieren muß, deren Instanzen als Listenelemente verwendbar sein sollen. Die Algebra fordert keine solche "Pflichtfunktionalität" für Elementtypen. Dennoch wird eine Implementierung sie aus praktischen Gründen häufig enthalten. Die oben angegebene Methode *toString* liefert beispielsweise eine Darstellung des Elementwertes als Textstring.

```
class Pos
{
   Elem value;
   Pos pred, succ;
}
```

Die Klasse *Pos* definiert ein Listenelement. Eine Variable vom Typ *Pos* ist ein Zeiger auf ein solches Listenelement (vgl. Abschnitt 2.2.2) und erfüllt damit genau den Zweck der Sorte *pos* unserer Algebra: eindeutige Identifizierung eines Listenelementes.

```
public class List extends Pos
{
      ...   (Methoden der Klasse List)
}
```

Die Klasse *List* erweitert die Klasse *Pos* um Methoden, die die Operationen des Datentyps $list_2$ realisieren. Eine *List*-Instanz ist das Kopfelement einer Liste, deren Elemente aus *Pos*-Instanzen bestehen.

Wir zeigen im folgenden die Implementierung einiger ausgewählter Operationen. Die Operation *empty* wird durch den Konstruktor der Klasse *List* realisert. Tatsächlich brauchen wir diesen Konstruktor gar nicht zu implementieren, da Java im Falle eines fehlenden Konstruktors automatisch einen Default-Konstruktor erzeugt, der alle Zeiger auf *null* setzt und damit genau unseren Anforderungen genügt. Eine leere Liste wird durch die Anweisung

```
List l = new List();
```

erzeugt.

Die Methode *front* ist schnell programmiert:

```
public Pos front()
{
  return this;
}
```

Hier wird also ein Zeiger auf die *List*-Instanz selbst zurückgegeben. Dies ist möglich, da *Pos* ein Supertyp von *List* ist.

Als nächste Operation betrachten wir *insert*, das Einfügen eines Elementes hinter einer gegebenen Position *p*. Dabei sind zwei Fälle zu unterscheiden:

(a) das Element soll in der Mitte oder am Anfang der Liste eingefügt werden, oder

(b) es soll an das Ende der Liste angehängt werden.

Bei der Manipulation verzeigerter Strukturen muß man sehr darauf achten, die einzelnen Änderungen in der richtigen Reihenfolge durchzuführen, um zu verhindern, daß plötzlich irgendwelche Teile nicht mehr zugreifbar sind. Die Reihenfolge dieser Schritte ist in Abbildung 3.4 und in der Methode *insert* durch Numerierung gezeigt.

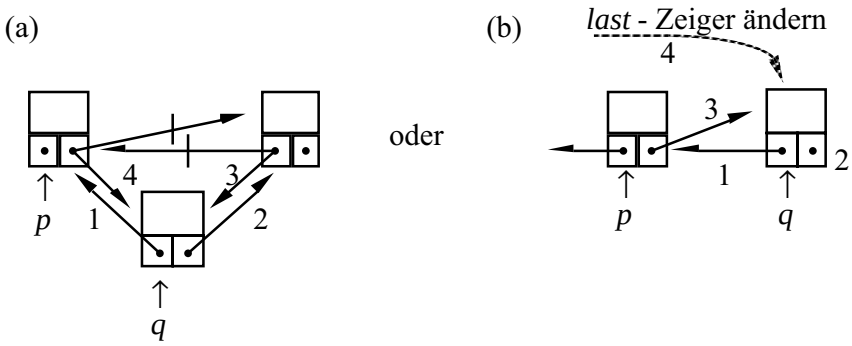


Abbildung 3.4: (a) Einfügen am Anfang oder in der Mitte, (b) Anhängen am Ende

```
public List insert(Pos p, Elem e)
{
  Pos q = new Pos();
  q.value = e;
  if (!(eol(p) || isempty()))
  {
    q.pred = p;              // 1a
    q.succ = p.succ;         // 2a
    p.succ.pred = q;         // 3a
    p.succ = q;              // 4a
  }
  else
  {
    q.pred = p;              // 1b
    q.succ = null;           // 2b
    p.succ = q;              // 3b
    pred = q;                // 4b
  }
  return this;
}
```

Die Verkettung zweier Listen, *concat*, ist in Abbildung 3.5 und der entsprechenden Methode gezeigt.

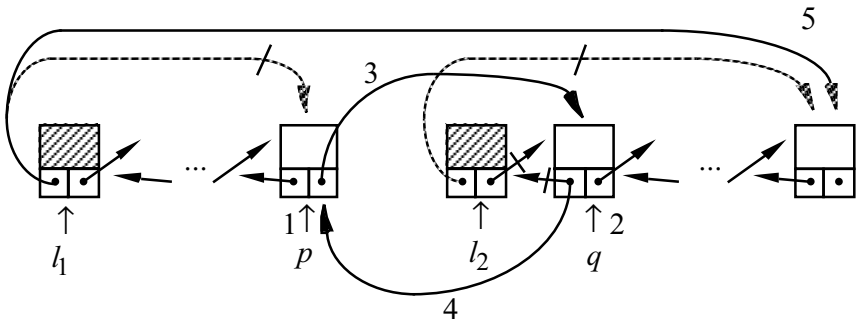


Abbildung 3.5: Verkettung zweier Listen

Die Liste l_1 in Abbildung 3.5 erscheint in der Methode *concat* nicht explizit als Argument, sondern ist durch die Listeninstanz (*this*) selbst realisiert. Zur Verdeutlichung haben wir an den entsprechenden Stellen *this* in den Code aufgenommen.

```
public List concat(List l2)
{
  Pos p, q;
```

```
    if (this.isempty()) return l2;
    else if (l2.isempty()) return this;
    else
    {
      p = this.pred;          // 1
      q = l2.next(l2.front()); // 2
      p.succ = q;             // 3
      q.pred = p;             // 4
      this.pred = l2.pred;    // 5
      l2.reset();
      return this;
    }
  }
```

Man beachte, daß manche Methoden *ihre Argumente verändern*; insofern stimmt das funktionale Modell der Algebra nicht ganz mit der Realität überein. Das Ergebnis von *concat* enthält physisch die Argumentlisten, die dann nicht mehr eigenständig existieren; der Listenkopf der zweiten Liste wird durch die hier zusätzlich eingeführte Methode *reset* zum Kopf einer leeren Liste.

Suchen eines Elementes (*find* - Operation): Hier liegt die Besonderheit einer Parameterfunktion *f*: *elem* → *bool* vor, die als Argument an *find* übergeben wird. Im Gegensatz zu Sprachen wie Modula-2, Pascal, C oder C++ kann man in Java Funktionen (bzw. Methoden) nicht direkt als Parameter übergeben. Die von Java angebotene Alternative besteht aus der Definition eines *Interfaces*, das die gewünschte Parameterfunktion als Methode definiert, und der Verwendung dieses *Interfaces* als Parameter der Methode *find*. Innerhalb von *find* kann dann über diesen Parameter die gewünschte Methode aufgerufen werden.

```
interface ElemTest
{
  boolean check(Elem l);
}

public Pos find(ElemTest test)
{
  Pos p = this;
```

```
    while (!eol(p))
    {
      p = next(p);
      if (test.check(p.value)) return p;
    }
    return null;
  }
```

Als Beispiel betrachten wir einen Elementtyp *Int*:

```
class Int implements Elem
{
  int value;
  public String toString() {return new String(Integer.toString(value,10));};
};
```

Sei *l* nun eine Liste mit Elementen vom Typ *Int*. Der Aufruf der Methode *find* zum Finden eines Listenelementes mit dem Wert 100 lautet wie folgt:

```
Pos p = l.find(new ElemTest()
                 {
                   public boolean check(Elem le)
                   {
                     return (((Int)le).value) == 100;
                   }
                 }
               );
```

Nach dieser Anweisung zeigt *p* nun entweder auf das Listenelement mit dem Wert 100 oder enthält den Wert *null*, falls in *l* kein passendes Element vorhanden ist.

Selbsttestaufgabe 3.1: Implementieren Sie die Operation *delete* für doppelt verkettete Listen. □

(b) Einfach verkettete Liste

Diese Struktur (Abbildung 3.6) unterstützt effizient einen Teil der Operationen des Datentyps $list_2$; sie ist einfacher, braucht etwas weniger Speicherplatz (für jedes Listenelement nur *einen* Zeiger auf den Nachfolger) und ist daher bei Anwendungen, die mit diesen Operationen auskommen, vorzuziehen.

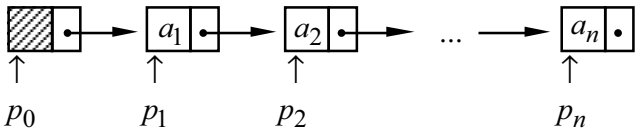


Abbildung 3.6: Einfach verkettete Liste (Prinzip)

Bei dieser Repräsentation scheinen zunächst die Operationen *previous*, *last*, *concat* und *delete* nicht effizient realisierbar zu sein. *Delete* läßt sich nicht durchführen, weil man einen Zeiger im Vorgänger umsetzen muß. Einige dieser Mängel lassen sich aber leicht beheben. Um *delete* zu unterstützen, benutzt man als Zeiger auf Elemente grundsätzlich Zeiger auf ihre Vorgänger. Also dient die bisherige Position p_{i-1} als Zeiger auf a_i. Wir werden im folgenden die Positionen entsprechend umnumerieren, so daß dann wieder p_i zu a_i gehört (Abbildung 3.7).

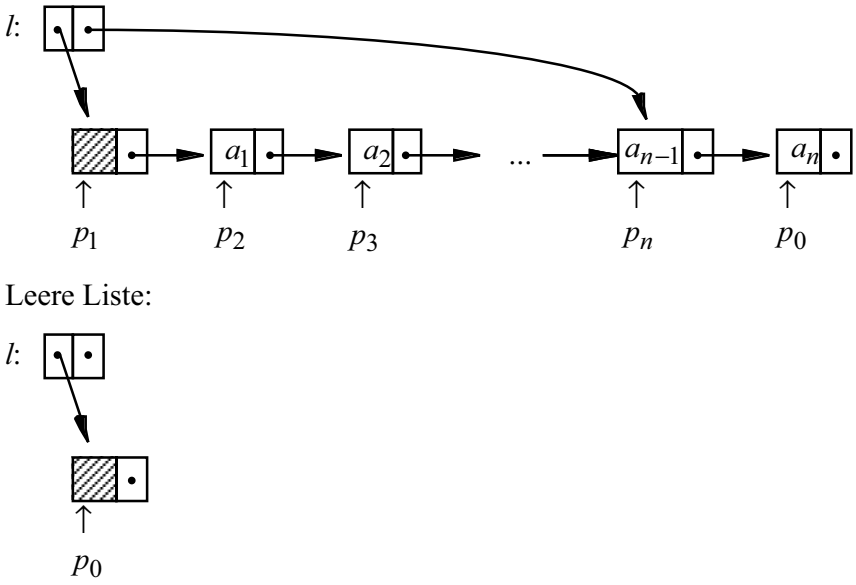


Abbildung 3.7: Einfach verkettete Liste (Implementierung)

Der Zeiger p_i zeigt dann nicht mehr direkt auf a_i, sondern auf den Vorgänger. Das Element a_i wird also über $p_i.succ$ erreicht. Um *last* und *concat* effizient zu unterstützen, d.h. eine O(1)-Laufzeit durch einfaches Umsetzen einer konstanten Anzahl von Zeigern zu erzielen, muß man noch zusätzlich irgendwo einen Zeiger auf das letzte Listenelement aufbewahren. Dies kann z.B. so geschehen:

```
class Pos
{
  Elem value;
  Pos succ;
}

public class List
{
   Pos head, last;
   … (Methoden der Klasse List)
}
```

Der *last*-Zeiger im Record *l* muß auf das vorletzte Element zeigen, da die Operation *last* die Position p_n zurückliefern soll.

Ein Aufruf von *insert* mit p_0 wird aufgefaßt als Auftrag, ein Element hinter dem Listenkopf einzufügen; p_0 selbst wird dazu nicht gebraucht. Ob eine Position p gleich p_0 ist, läßt sich leicht feststellen:

$$p = p_0 \Leftrightarrow p.succ = null.$$

Leider sind die Positionszeiger nicht stabil unter Änderungsoperationen. Beispielsweise zeigt der Zeiger p_i nicht mehr auf a_i, wenn hinter der Position p_{i-1} ein Element eingefügt wird.

Die Methode *delete* ist etwas kompliziert, da mehrere Fälle (a) - (c) unterschieden werden müssen, die in den Abbildungen 3.8 bis 3.10 gezeigt sind.

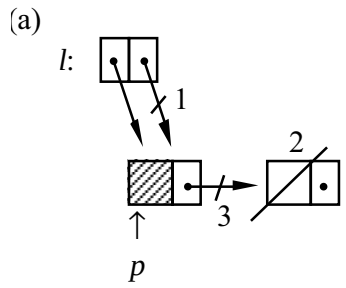


Abbildung 3.8: *p* zeigt auf das letzte Element einer einelementigen Liste.

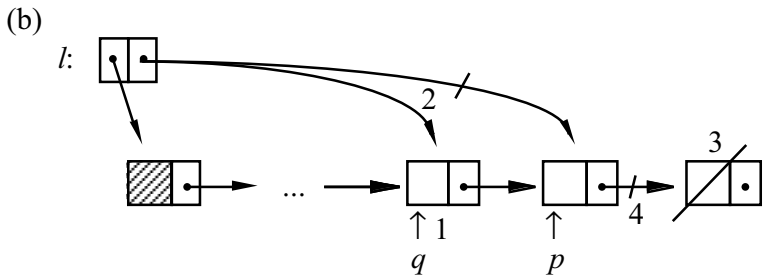


Abbildung 3.9: *p* zeigt auf das letzte Element einer mehrelementigen Liste.

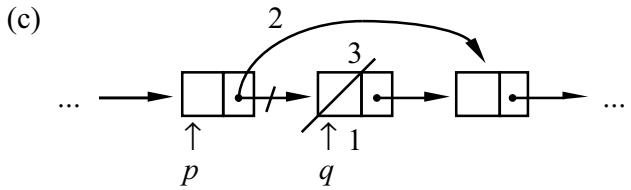


Abbildung 3.10: *p* zeigt nicht auf das letzte Element.

```
public List delete(Pos p)
{
   Pos q;
   if (isempty()) { Fehlerbehandlung (Löschen in leerer Liste unmöglich) }
   else
   {
      if (eol(p))                          // p = p_n
      {
         if (p == head)                    // n = 1
         {
            last = null;                   // 1a
            p.succ = null;                 // 3a
         }
         else
         {
            q = head;
            while (q.succ != p) q = q.succ;   // 1b
            last = q;                      // 2b
            p.succ = null;                 // 4b
         }
      }
      else
```

```
        {
          q = p.succ;                          // 1c
          if (q == last) last = p;
          p.succ = p.succ.succ;                // 2c
        }
      }
      return this;
    }
```

Die Aktionen 2a, 3b und 3c (Freigabe des durch das zu löschende Element belegten Speicherplatzes) fehlen in dieser Implementierung, denn Java führt eine automatische Garbage Collection durch, so daß Speicherplatzfreigabe durch den Programmierer nicht nötig (und auch nicht möglich) ist.

Die einzige noch fehlende Operation bei dieser Realisierung des Datentyps $list_2$ als einfach verkettete Liste ist *previous*; es ist also nicht möglich, eine Liste rückwärts zu durchlaufen. Hier werden *last* und *concat* in O(1) Zeit unterstützt, *delete* im allgemeinen auch. Das Entfernen des letzten Elementes kostet leider O(n) Zeit, da man die Liste von Anfang an durchlaufen muß, um den *last*-Zeiger umzusetzen.

Eine alternative Implementierung würde den *last*-Zeiger in l jeweils auf das letzte statt auf das vorletzte Element der Liste zeigen lassen. Dann könnte man *concat* und *delete* (auch für das letzte Element) in O(1) Zeit realisieren; es gäbe aber für *last* keine effiziente Implementierung mehr. Im allgemeinen wird die erste vorgestellte Implementierung vorzuziehen sein, da nur beim Löschen des letzten Elements Nachteile entstehen. Insgesamt muß man zu dieser Implementierung natürlich sagen, daß das ständige Arbeiten mit Zeigern auf den Vorgänger des "gemeinten" Elementes ziemlich unnatürlich ist.

Selbsttestaufgabe 3.2: Schreiben Sie für die Operation

$$swap\colon list \times pos \to list$$

eine Methode, die in einer einfach verketteten Liste l die Elemente an den Positionen p und $next\ (l, p)$ vertauscht. Positionen sind dabei wie in Abbildung 3.7 zu interpretieren, d.h. bei Angabe von p_3 sind a_3 und a_4 zu vertauschen. Denken Sie daran, daß ggf. auch der Zeiger $l.last$ umgesetzt werden muß! Angabe von p_0 als Position ist nicht zulässig (da es keine Position eines Elementes ist) und muß auch nicht abgefangen werden. □

(c) Sequentielle Darstellung im Array

Dies entspricht der Technik der Mengendarstellung des einleitenden Beispiels in Kapitel 1. Die Listenelemente werden in aufeinanderfolgenden Zellen eines Arrays gespeichert, am Ende verbleibt ein ungenutztes Stück des Arrays.

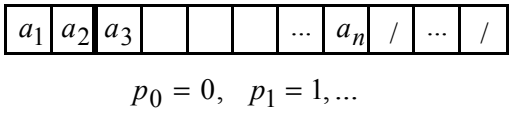


Abbildung 3.11: Listendarstellung im Array

Diese Darstellung hat einige offensichtliche Nachteile:

- Das Einfügen und Entfernen ist aufwendig wegen des notwendigen Verschiebens der Folgeelemente.
- Die Maximalgröße der Liste muß vorher festgelegt werden (Speicherplatzverschwendung). Falls diese nicht von vornherein bekannt ist, kann man diese Darstellung nicht verwenden, oder man muß bei "Überlauf" einen größeren Array anlegen, alles dorthin kopieren, und dann den alten Array freigeben.
- Die Positionszeiger sind beim Einfügen bzw. Entfernen nicht stabil.

Vorteile liegen in der Einfachheit und der Tatsache, daß keine expliziten Zeiger benötigt werden. Deshalb kann diese Darstellung gelegentlich Anwendung finden zur Speicherung einer annähernd statischen Objektmenge.

(d) Einfach oder doppelt verkettete Liste im Array

Dynamische Datenstrukturen, die aus einer Menge *gleichartiger*, durch Zeiger verketteter Elemente bestehen, können ganz allgemein auch in Arrays eingebettet werden. Die Zeiger werden dabei ersetzt durch "Adressen" im Array, also durch Array-Indizes. Diese allgemeine Technik kann natürlich auch für den Spezialfall einfach oder doppelt verketteter Listen verwendet werden.

Beispiel 3.1: Die Liste L

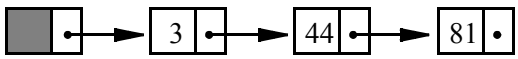


Abbildung 3.12: Beispielliste

könnte im Array so dargestellt sein:

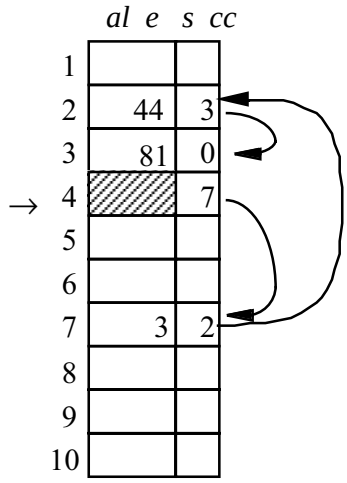


Abbildung 3.13: Beispielliste im Array

□

Innerhalb eines Arrays können sogar beliebig viele verschiedene Listen dargestellt werden (soweit der Gesamtplatz für ihre Elemente ausreicht). Der Programmierer muß allerdings selbst den freien Speicherplatz, also die unbelegten Array-Zellen, verwalten. Effektiv übernimmt er damit Aufgaben, die sonst das Laufzeitsystem (über *new* und *dispose* bzw. per *garbage collection*) löst. Die folgende Abbildung zeigt zwei Listen $L = <a, b, c>$ und $M = <x, y>$, die sich einen Array teilen:

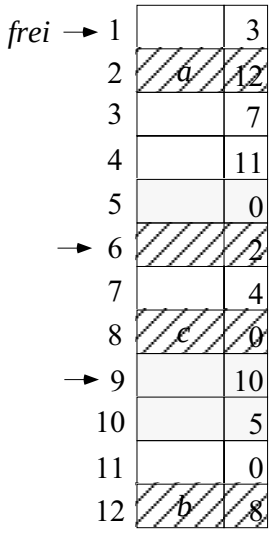


Abbildung 3.14: Zwei Listen in einem Array

Hier sind die freien Felder des Arrays auch noch in einer dritten Liste verkettet (siehe unten). Eine solche Struktur könnte etwa so deklariert werden:

```
class Pointer
{
   public int pos;
   public static final int nil = -1;
   public Pointer(int p) { pos = p; }
}
```

Der Positionswert -1 wird als *nil*-Pointer interpretiert. Zur Steigerung der Übersichtlichkeit der Programme, die die Klasse *Pointer* benutzen, haben wir die Konstante *Pointer.nil* definiert.

```
class ListElem
{
   Elem value;
   Pointer succ = new Pointer(Pointer.nil);
};

public class List
{
   int size;
   ListElem[] space;
   Pointer frei = new Pointer(0);
   … (Konstruktor und Methoden der Klasse List)
}
```

Hier enthält Zelle 0 den Listenkopf der Freiliste. Betrachten wir nun den Konstruktor der Klasse *List*. Er legt zum einen Speicherplatz für die als Parameter übergebene maximale Anzahl von Listenelementen an, zum anderen verkettet er für die Freispeicherverwaltung alle Elemente initial zu einer einzigen Liste:

```
public List(int sz)
{
   size = sz;
   space = new ListElem[size];
   for (int i=0; i < size; i++)
   {
      space[i] = new ListElem();
      if (i < size - 1)
         space[i].succ.pos = i + 1;
      else
         space[i].succ.pos = Pointer.nil;
   }
}
```

Bei einer Anforderung (*alloc*) wird der Freiliste das erste Element entnommen; es kann dann in eine andere Liste eingefügt werden.

```
public void alloc (Pointer p)
{
  if (frei.pos == Pointer.nil) { /* Error (kein Platz mehr) */ }
  else
  {
    p.pos = space[frei.pos].succ.pos;                  // 1
    space[frei.pos].succ.pos = space[p.pos].succ.pos;  // 2
    space[p.pos].succ.pos = Pointer.nil;               // 3
  }
}
```

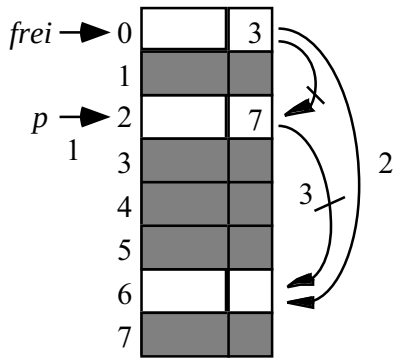


Abbildung 3.15: Freispeicherverwaltung

Ein freigegebenes Element wird an den Anfang der Freiliste gehängt.

```
public void dispose(Pointer p)
{
  if (p.pos == Pointer.nil) { /* Error (Zeiger undefiniert)*/ }
  else
  {
    space[p.pos].succ.pos = space[frei.pos].succ.pos;
    space[frei.pos].succ.pos = p.pos;
  }
}
```

Nachteil der im Array eingebetteten Listen-Implementierung ist natürlich wieder die Festlegung des Gesamtplatzbedarfs, die zu Beschränkung der Listengröße bzw. zu Speicherplatzverschwendung führt. Es gibt aber auch wichtige *Vorteile*:

(a) Aufrufe des Laufzeit- bzw. Betriebssystems (mit entsprechendem Verwaltungsaufwand) werden vermieden, dadurch sind *alloc* und *dispose* sehr schnell, und

(b) die komplette im Array dargestellte Struktur kann *extern gespeichert* werden. Beim Neustart des Programms kann der Array-Inhalt wieder eingelesen werden; die Zeiger (Array-Indizes) sind danach wiederum gültig. Explizite Zeiger hingegen "überleben" das Programmende nicht; sie haben beim nächsten Aufruf keine Bedeutung mehr.

Zusammenfassend läßt sich sagen, daß die Implementierungen mit doppelt verketteten Listen alle Operationen der Datentypen $list_1$ und $list_2$ bis auf *find* in O(1) Zeit realisieren. *find* benötigt O(n) Zeit. Abgesehen von *find* unterstützen einfach verkettete Listen entweder alle Operationen außer *last*, *concat* und *previous* in O(1) Zeit (Implementierung ohne last-Zeiger) oder aber auch *last* und *concat* in O(1), wobei aber *delete* im Einzelfall O(n) Zeit braucht (Implementierung mit last-Zeiger). Der Platzbedarf ist bei den explizit verketteten Strukturen O(n).

3.2 Stacks

Stacks sind Spezialfälle von Listen, nämlich Listen mit den Operationen *first*, *rest*, *append* (und wie immer, *empty* und *isempty*). Sie entsprechen also exakt unserem Datentyp $list_1$ ohne die Operation *concat.* Die Operationen werden hier traditionell anders genannt:

top = *first* *stack* = *list*
push = *append*
pop = *rest*

Stack heißt Stapel und die Idee dabei ist, daß man nur oben etwas darauflegen oder herunternehmen kann und daß auch nur das oberste Element sichtbar ist.

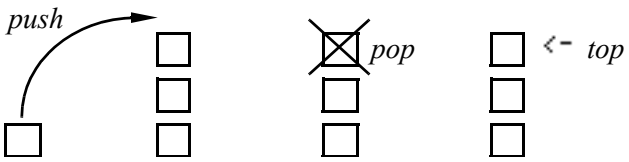


Abbildung 3.16: Stack-Operationen

Stacks sind *das* Standardbeispiel für die Spezifikation mit abstrakten Datentypen, und so sei hier noch einmal eine entsprechende Spezifikation gezeigt:

adt *stack*			
sorts	*stack*, *elem*		
ops	*empty*	:	$\rightarrow$ *stack*
	push	: *stack* $\times$ *elem*	$\rightarrow$ *stack*
	pop	: *stack*	$\rightarrow$ *stack*
	top	: *stack*	$\rightarrow$ *elem*
	isempty	: *stack*	$\rightarrow$ *bool*
axs	*isempty* (*empty*)		= *true*
	isempty (*push* (*s*, *e*))		= *false*
	pop (*empty*)		= error
	pop (*push* (*s*, *e*))		= *s*
	top (*empty*)		= error
	top (*push* (*s*, *e*))		= *e*
end *stack*.			

Eine Spezifikation als Algebra findet sich in Abschnitt 3.1 als Teil des Datentyps $list_1$.

Die vielleicht beliebteste Implementierung benutzt die sequentielle Darstellung im Array. Da nur an einem Ende der Liste angefügt werden kann, entfällt das ineffiziente Verschieben der Folgeelemente.

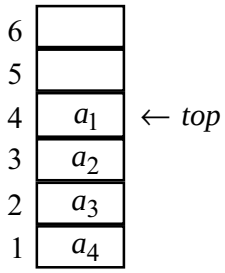


Abbildung 3.17: Stack-Implementierung im Array

Wir zeigen die Deklarationen und die Implementierung der Operation *push*:

```
class Stack
{
   int size;
   int top = 0;
   Elem[] a;

   public Stack(int sz)
   {
      size = sz;
      a = new Elem[size];
   }
```

```
    public Stack Push(Elem e)
    {
      if (top == size - 1) { /* Error (kein Platz mehr)*/ }
      else a[top++] = e;
      return this;
    }
}
```

Da in einem Stack das zuletzt eingefügte Element als erstes entnommen wird, werden Stacks auch als LIFO (last-in-first-out)-Strukturen bezeichnet. Es gibt vielfältige Anwendungen; z.B. kann man eine Zeichenfolge invertieren, d.h. in umgekehrter Reihenfolge erhalten. Stacks eignen sich ganz allgemein zur Bearbeitung von *Klammerstrukturen*. Wir betrachten einige Beispiele.

Beispiel 3.2: Auswertung arithmetischer Ausdrücke

Es sollen vollständig geklammerte arithmetische Ausdrücke, wie etwa

((6 * (4 * 28)) + (9 - ((12 / 4) * 2)))

zeichenweise gelesen und dabei ausgewertet werden. Bei genauerer Betrachtung besteht ein Ausdruck aus vier Arten von Symbolen, nämlich öffnenden und schließenden Klammern, Zahlen und Operationssymbolen. Mit Hilfe eines Stacks kann man einen solchen Ausdruck symbolweise lesen und gleichzeitig auswerten (solche Methoden werden z.B. im Compilerbau eingesetzt). Die Grundidee ist die folgende: Jede öffnende Klammer beginnt einen neuen Teilausdruck; wir merken uns auf einem Stack unvollständige Teilausdrücke. Ein vollständiger Teilausdruck wird ausgewertet, vom Stack entfernt und sein Ergebnis auf der nächsttieferen Stackebene angefügt. Wir zeigen die Auswertung des obigen Ausdrucks mit den verschiedenen Stackebenen, bis zum Einlesen des dritten Multiplikationssymbols. Der Stackinhalt zu diesem Zeitpunkt ist:

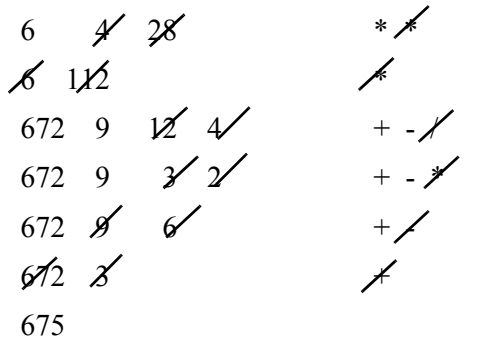
```
(       672       +
~~( 6 *   112 )~~   ( 9 -
   ~~(4 * 28)~~          (   3   *
                        ~~( 12 / 4)~~
```

Eine einfachere Implementierung ist vielleicht mit getrenntem Operanden- und Operatorenstack möglich. Der Ausdruck wird symbolweise eingelesen, wobei folgende Aktionen durchgeführt werden:

- öffnende Klammer: ignorieren
- Operand: auf Operandenstack legen

- Operator: auf Operatorenstack legen
- schließende Klammer: obersten Operator vom Operatorstack nehmen; soviele Operanden wie nötig vom Operandenstack nehmen (hier immer zwei, da alle Operatoren dyadisch sind); Ausdruck auswerten; Ergebnis auf Operandenstack schreiben.

Beim Abarbeiten des obigen Ausdrucks entstehen folgende Stackzustände (wir zeigen Stackübergänge jeweils bei der Auswertung eines Operators):



□

Beispiel 3.3: Verwaltung geschachtelter Prozeduraufrufe

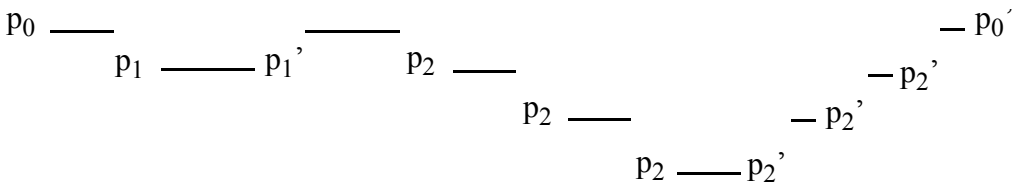
Es seien drei Prozeduren p_0, p_1 und p_2 einer imperativen Programmiersprache wie Pascal oder Modula-2 gegeben, wobei p_2 sich selbst rekursiv aufruft.

```
procedure p0;
begin ...; p1; ...; p2 (z); ... end p0;

procedure p1;
begin ... end p1;

procedure p2 (x : integer);
var y : integer;
begin ...; p2 (y); ... end p2;
```

Eine Folge von Prozeduraufrufen mit verschiedenen Rekursionstiefen könnte folgendermaßen aussehen (bezeichne p_i einen Aufruf, p_i' die Terminierung der Prozedur p_i, die Laufzeit sei durch waagerechte Striche dargestellt):



Zur Laufzeit eines Programms muß dafür gesorgt werden, daß für jede Prozedurinkarnation die Rücksprungadresse, Parameter (hier x) und lokale Variablen (hier y) gemerkt werden. Dies geschieht üblicherweise in sog. *Aktivierungs-Records*, die auf einem Stack abgelegt werden. Wenn also eine Prozedur P aufgerufen wird, wird ein neuer Aktivierungs-Record auf den Stack gelegt, unabhängig davon, ob bereits ein anderer für P existiert. Wenn P beendet ist, muß sich der zugehörige Aktivierungs-Record oben auf dem Stack befinden, da alle innerhalb von P aufgerufenen Prozeduren ebenfalls beendet sind. Damit kann dieser Record vom Stack genommen und so die Kontrolle an die Aufrufstelle von P zurückgegeben werden. □

Wenn man aus irgendwelchen Gründen in einer Sprache programmieren muß, die keine rekursiven Prozeduren zur Verfügung stellt, kann man diese Art der Prozedurverwaltung mit Hilfe eines Stacks selbst implementieren. Die gegebene rekursive Formulierung des Algorithmus oder Programms kann auf folgende systematische Art in eine nicht-rekursive Version umgewandelt werden.[2]

1. Es werden Sprungmarken eingeführt, und zwar eine am Anfang des Programms und eine weitere direkt hinter jedem rekursiven Aufruf. Die erste Marke wird dazu benutzt, einen rekursiven Aufruf zu simulieren, die weiteren für entsprechende Rücksprünge.
2. Man definiert einen Stack, dessen Elemente jeweils einen kompletten Parametersatz (und evtl. vorhandene lokale Variablen) sowie eine Sprungmarke (die Rücksprungadresse) aufnehmen können.
3. Vor der ersten Sprungmarke – also vor dem Anfang des ursprünglichen Programms – wird der Stack als leer initialisiert.
4. Jeder rekursive Aufruf wird durch folgende Aktionen ersetzt:
   a. Lege die aktuellen Parameterwerte sowie die auf den Aufruf folgende Sprungmarke auf den Stack.
   b. Weise den Parametern neue Werte zu, die sich durch Auswertung der entsprechenden Parameterausdrücke des rekursiven Aufrufs ergeben.

2. Voraussetzung für die direkte Implementierung des dargestellten Verfahrens sind Sprungmarken und *goto*-Anweisungen, die von vielen imperativen sowie "systemnahen" objektorientierten Sprachen, einschließlich C++, unterstützt werden, allerdings nicht von Java.

c. **goto** <erste Sprungmarke>

Das Programm wird daraufhin von Anfang an mit den Parameterwerten des rekursiven Aufrufs ausgeführt und simuliert so diesen Aufruf.

5. Wenn das Ende des bisherigen Programms erreicht wird, kann es sich entweder um die tatsächliche Terminierung oder um eine Rückkehr aus einem rekursiven Aufruf handeln. Das hängt davon ab, ob noch Aktivierungsrecords auf dem Stack liegen. Bei Rückkehr aus einem rekursiven Aufruf muß an die entsprechende Folgemarke gesprungen werden. Deshalb wird noch angefügt:

   if der Stack ist nicht leer
   then nimm den obersten Aktivierungsrecord vom Stack und weise seine Parameterwerte den Parametervariablen zu, die Rücksprungmarke einer Variablen *return_label*;
   goto *return_label*
   end if

Als Beispiel betrachten wir die Umwandlung eines rekursiven Algorithmus zur Lösung des Problems "Türme von Hanoi" in eine nicht-rekursive Version. Vermutlich ist Ihnen dieses Problem bereits bekannt:

Beispiel 3.4: Türme von Hanoi

Gegeben seien drei Stäbe A, B und C und n Scheiben verschiedener Größen. Die Scheiben sitzen zu Anfang auf dem Stab A, von unten nach oben nach abnehmender Größe geordnet, so daß jeweils eine kleinere Scheibe auf einer größeren liegt. Die Aufgabe besteht nun darin, die n Scheiben von Stab A nach Stab B zu bringen. Dabei gelten folgende Einschränkungen:

- Bei jedem Schritt wird genau eine Scheibe von einem Stab zu einem anderen bewegt.
- Eine Scheibe darf nie auf einer kleineren Scheibe liegen.

Der folgende rekursive Algorithmus erzeugt als Ausgabe eine entsprechende "Zugfolge":

```
type tower = (A, B, C);

algorithm move (n : integer; source, target, other : tower);
{Bewege n Scheiben vom Turm source zum Turm target, wobei other der verbleibende Turm ist.}
```

```
if n = 1
then output (source, "->", target)
else
      move (n - 1, source, other, target);
      output (source, "->", target);
      move (n - 1, other, target, source)
end if
```

Die einzelnen Schritte des oben genannten Verfahrens sehen hier so aus:

1. Wir führen folgende Sprungmarken ein:

```
1: if n = 1
   then output (source, "->", target)
   else
        move (n - 1, source, other, target);
     2: output (source, "->", target);
        move (n - 1, other, target, source);
     3:
   end if
```

2. Der Stack wird so definiert:

```
type activation   = record n                    : integer;
                           source, target, other : tower;
                           return_address        : label
                    end;

var s : stack (activation)
```

3. - 5. Schritte 3 - 5 erzeugen folgendes Programm:

```
algorithm move (n : integer; source, target, other : tower);
{Bewege n Scheiben vom Turm source zum Turm target, wobei other der
verbleibende Turm ist.}

var return_label : label;
begin
   s := empty;

1: if n = 1
   then output (source, "->", target)
   else  {Ersetze "move (n - 1, source, other, target)"}
      push (s, (n, source, target, other, 2));                    (*)
      (n, source, target, other) := (n - 1, source, other, target); (**)
      goto 1;
```

```
    2: output (source, "->", target);

       {Ersetze "move (n - 1, other, target, source)"}
       push (s, (n, source, target, other, 3));
       (n, source, target, other) := (n - 1, other, target, source);
       goto 1;
3:
    end if;
    if not  isempty (s)
    then    (n, source, target, other, return_label) := top (s);
            pop (s);
            goto return_label
    end if
end move.
```

□

Um die Struktur des Algorithmus nicht zu verschleiern, haben wir uns einige notationelle Freiheiten erlaubt, die es z.B. in Java nicht gibt, nämlich eine Notation für Record-Konstanten (*) und eine "kollektive Zuweisung" (**). Bei letzterer wird angenommen, daß alle Ausdrücke auf der rechten Seite zunächst ausgewertet werden und dann seiteneffektfrei den Variablen in gleicher Position auf der linken Seite zugewiesen werden (siehe z.B. [Bauer und Wössner 1984, Kapitel 5]).

3.3 Queues

Queues sind ebenfalls spezielle Listen, in denen Elemente nur an einem Ende ("vorne") entnommen und nur am anderen Ende ("hinten") angehängt werden. Sie heißen deshalb FIFO (first-in-first-out)-Strukturen. Der deutsche Ausdruck ist (Warte-)*Schlange*. Queues entsprechen unserem Datentyp $list_1$ ohne die Operationen *concat* und *append*; anstelle von *append* gibt es eine Operation *rappend* (rear - *append*), die ein Element ans Ende einer Liste hängt. Die Algebra $list_1$ wäre entsprechend zu erweitern:

ops $\quad rappend : list \times elem \rightarrow list$
functions $\quad rappend\ (a_1 \ldots a_n, x) = a_1 \ldots a_n\ x$

Traditionell gibt es auch hier wieder andere Bezeichnungen:

front	= *first*
enqueue	= *rappend*
dequeue	= *rest*
queue	= *list*

Somit erhalten wir die Signatur

sorts	*queue*, *elem*		
ops	*empty*	:	$\rightarrow$ *queue*
	front	: *queue*	$\rightarrow$ *elem*
	enqueue	: *queue* $\times$ *elem*	$\rightarrow$ *queue*
	dequeue	: *queue*	$\rightarrow$ *queue*
	isempty	: *queue*	$\rightarrow$ *bool*

Natürlich eignen sich wieder sämtliche Möglichkeiten, Listen zu implementieren, um Queues zu implementieren. Wie bei Stacks gibt es eine besonders beliebte sequentielle Implementierung im Array.

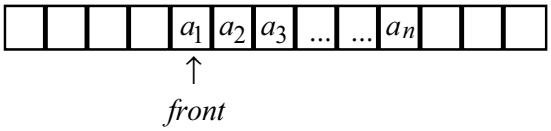


Abbildung 3.18: Queue-Implementierung im Array

Durch das Einfügen und Entfernen an verschiedenen Enden "durchwandert" die Queue den Array und stößt bald an einem Ende an, obwohl die Gesamtgröße des Arrays durchaus ausreicht (dies natürlich nur, wenn die Queue höchstens soviele Elemente enthält wie der Array lang ist). Der Trick, um dieses Problem zu lösen, besteht darin, den Array als zyklisch aufzufassen, d.h. wenn die Queue das Array-Ende erreicht hat, wird wieder von vorn angefangen, sofern dort wieder Plätze frei sind:

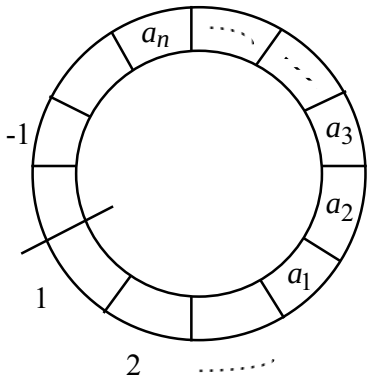


Abbildung 3.19: Zyklische Queue-Implementierung im Array

Queues haben viele Anwendungen, insbesondere zur Realisierung von Warteschlangen (z.B. Prozesse, Nachrichten, Druckaufträge) in Betriebssystemen.

Selbsttestaufgabe 3.3: Außer einer Implementierung von Queues in einem zyklischen Array ist auch eine denkbar, die auf zwei Stacks basiert, zwischen denen die Objekte hin- und hergeschaufelt werden. Zur Vereinfachung sei angenommen, daß die Stacks beliebig groß werden können, d.h. ein Stack-Overflow bei *push* tritt nicht auf.

Implementieren Sie die Queue-Operationen für diese Repräsentation. Die Operationen dürfen dabei nur die als gegeben anzunehmenden Stack-Operationen verwenden, die auf zwei Stacks arbeiten. Definieren Sie also

```
class Queue
{
  Stack s1, s2;
  … (Methoden der Klasse Queue)
}
```

□

3.4 Abbildungen

Eine Abbildung *f*: *domain* → *range* ordnet jedem Element des Argumentwertebereichs (*domain*) ein Element des Zielwertebereichs (*range*) zu. In manchen Fällen kann man diese Zuordnung prozedural auswerten, in anderen Fällen muß man sie speichern (z.B. eine Funktion, die jedem Mitarbeiter einer Firma sein Gehalt zuordnet). Für den zweiten Fall führen wir einen Datentyp *mapping* ein (Abbildung 3.20), dessen wesentliche Operationen die *Definition* der Abbildung für ein Element des Domains mit einem Wert und das *Anwenden* der Funktion auf ein Element des Domains sind. Bei der letzten Operation wird als Ergebnis ein Wert des Range oder ein spezieller Wert "⊥" (undefiniert) zurückgeliefert.

Ein *mapping*-Wert ist also eine Menge S von Paaren, die für jedes Element in *domain* genau ein Paar enthält. Offensichtlich sind alle mit den Operationen erzeugten Mappings nur an endlich vielen Stellen definiert. An allen anderen Stellen liefert *apply* den Wert ⊥.

Implementierungen: Falls der Domain-Typ endlich und skalar (z.B. ein Unterbereichs- oder Aufzählungstyp) ist und genügend kleine Kardinalität besitzt, gibt es eine direkte Implementierung mit Arrays:

```
type mapping = array [domain] of range
```

Daher ist der Datentyp *mapping* die natürliche *Abstraktion des Typs bzw. Typkonstruktors Array der programmiersprachlichen Ebene*.

algebra *mapping*

sorts	*mapping*, *domain*, *range*		
ops	*empty*	:	$\rightarrow$ *mapping*
	assign	: *mapping* $\times$ *domain* $\times$ *range*	$\rightarrow$ *mapping*
	apply	: *mapping* $\times$ *domain*	$\rightarrow$ *range* $\cup$ $\{\perp\}$

sets *mapping* = die Menge aller S, für die gilt:

(i) $S \subseteq \{ (d, r) \mid d \in domain, r \in range \cup \{\perp\} \}$

(ii) $\forall\, d \in domain\ \exists r \in range \cup \{\perp\} : (d, r) \in S$

(iii) $\forall\, (d_1, r_1) \in S, (d_2, r_2) \in S : d_1 = d_2 \Rightarrow r_1 = r_2$

functions

empty = $\{(d, \perp) \mid d \in domain\}$

Sei für die folgenden Definitionen $(d, x) \in m$.

assign $(m, d, r) = (m \setminus \{(d, x)\}) \cup \{(d, r)\}$
apply $(m, d) = x$

end *mapping*.

Abbildung 3.20: Algebra *mapping*

Falls der Domain-Typ große Kardinalität besitzt und/oder die Abbildung nur an wenigen Stellen definiert ist, ist die Array-Darstellung nicht mehr effizient. Man kann dann z.B. Implementierungen mit Listen aller definierten Paare wählen. Wir werden später noch viele Techniken zur Mengendarstellung mit effizientem Auffinden eines Elementes kennenlernen; diese Methoden eignen sich natürlich auch, um Mengen von Paaren und damit Abbildungen darzustellen.

3.5 Binäre Bäume

Bäume erlauben es, hierarchische Beziehungen zwischen Objekten darzustellen. Solche Hierarchien treten vielfach in der "realen Welt" auf, z.B.

- die Gliederung eines Unternehmens in Bereiche, Abteilungen, Gruppen und Angestellte;

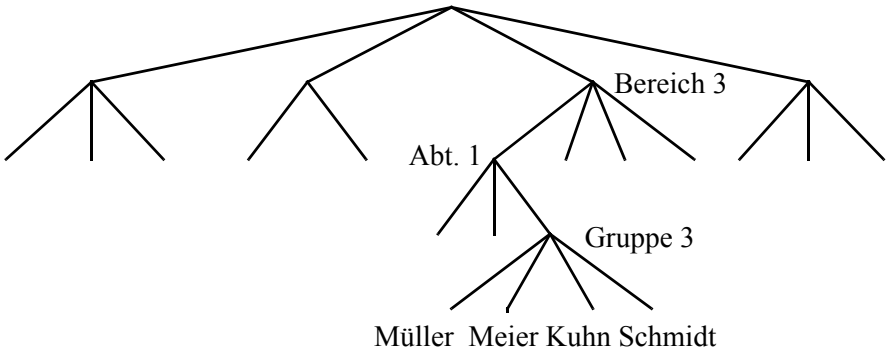


Abbildung 3.21: Baumdarstellung einer Unternehmenshierarchie („Organigramm“)

- die Gliederung eines Buches in Kapitel, Abschnitte, Unterabschnitte;
- die Aufteilung Deutschlands in Bundesländer, Kreise, Gemeinden, Bezirke;
- die Nachkommen, also Kinder, Enkel usw. eines Menschen.

Klammerstrukturen beschreiben ebenfalls Hierarchien, sie sind im Prinzip äquivalent zu Bäumen. So können wir den arithmetischen Ausdruck aus Abschnitt 3.2 auch als Baum darstellen:

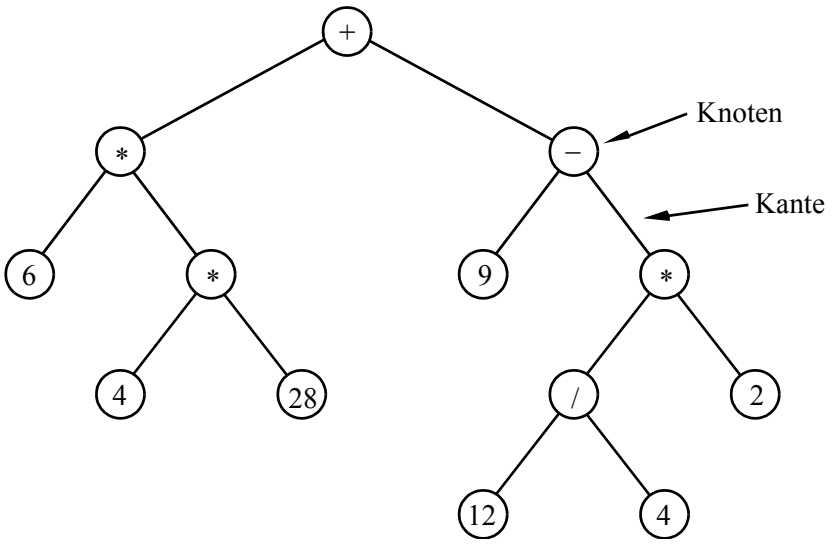


Abbildung 3.22: Operatorbaum

Ein *Baum* besteht aus einer Menge von Objekten, die *Knoten* genannt werden zusammen mit einer Relation auf der Knotenmenge (graphisch durch *Kanten* dargestellt), die die Knotenmenge hierarchisch organisiert. Die Knoten, die mit einem Knoten p durch eine Kante verbunden sind und die unterhalb von p liegen, heißen *Söhne* (oder *Kinder*) von p.

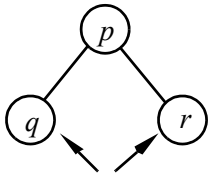


Abbildung 3.23: Vater-Sohn-Beziehung

p heißt *Vater* für diese Knoten. Wir betrachten in diesem Abschnitt zunächst den Spezialfall *binärer* Bäume. Das sind Bäume, in denen jeder Knoten *zwei* Söhne hat, die allerdings jeweils fehlen dürfen, so daß jeder Knoten entweder keinen Sohn hat, oder einen linken Sohn, oder einen rechten Sohn, oder einen linken und rechten Sohn. Die Darstellung des arithmetischen Ausdrucks ist ein binärer Baum. Binäre Bäume sind etwas einfacher zu behandeln als allgemeine Bäume, die in Abschnitt 3.6 besprochen werden.

Definition 3.5: (Binäre Bäume)

(i) Der leere Baum ist ein binärer Baum.

(Bezeichnung: $\Diamond$; graphisch: ■)

(ii) Wenn x ein Knoten ist und T_1, T_2 binäre Bäume sind, dann ist auch das Tripel (T_1, x, T_2) ein binärer Baum T.

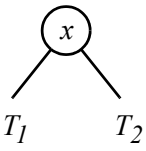
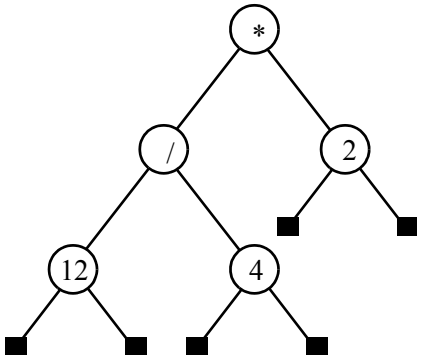


Abbildung 3.24: Graphische Darstellung des Tripels (T_1, x, T_2)

x heißt *Wurzel*, T_1 *linker Teilbaum*, T_2 *rechter Teilbaum* von T. Die Wurzeln von T_1 und T_2 heißen *Söhne* von x, x ist ihr *Vaterknoten*. Ein Knoten, dessen Söhne leer sind (andere Sprechweise: der keine Söhne hat) heißt *Blatt*.

Beispiel 3.6: Teilausdruck in graphischer und linearer Notation



(((◊, 12, ◊), /, (◊, 4, ◊)), *, (◊, 2, ◊))

Abbildung 3.25: Graphische vs. lineare Notation eines Baumes

Gewöhnlich läßt man allerdings in der graphischen Darstellung die leeren Bäume weg und zeichnet Blätter als Rechtecke, die anderen Knoten als Kreise. □

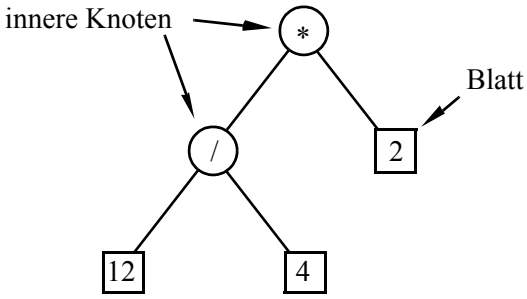


Abbildung 3.26: Innere Knoten und Blätter

Ein Knoten, der nicht Blatt ist, heißt *innerer Knoten*. Im Beispiel entsprechen also Blätter den Operanden, innere Knoten den Operatoren. Zwei Knoten, die Söhne desselben Vaters sind, heißen *Brüder* (oder Geschwister).

Ein *Pfad* in einem Baum ist eine Folge von Knoten $p_0, ..., p_n$, so daß jeweils p_i Vater von p_{i+1} ist. Meist betrachtet man Pfade, die von der Wurzel ausgehen. Für einen Knoten p heißen die Knoten auf dem Pfad von der Wurzel zu p *Vorfahren* (ancestors) und alle Knoten auf Pfaden von p zu einem Blatt (Blätter eingeschlossen) *Abkömmlinge* oder *Nachfahren* (descendants) von p. Jeder *Teilbaum* eines Baumes besteht aus einem Knoten zusammen mit allen seinen Abkömmlingen. Die *Länge eines Pfades* $p_0, ..., p_n$ ist n (Anzahl der Kanten auf dem Pfad). Die *Höhe eines Baumes* T ist die Länge des längsten Pfades in T, der offensichtlich von der Wurzel zu einem Blatt führen muß. Die *Tiefe eines Knotens* p im Baum ist die Länge des Pfades von der Wurzel zu p.

Die folgenden Eigenschaften binärer Bäume lassen sich recht einfach herleiten:

Beobachtung 3.7: Die maximale Höhe eines Binärbaumes mit n Knoten ist n - 1, also O(n).

Beweis: Dieser Fall tritt ein, wenn der Baum zu einer Liste entartet (Abbildung 3.27). □

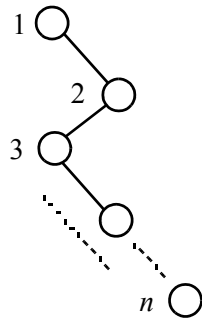


Abbildung 3.27: Zur Liste entarteter Baum

Beobachtung 3.8: Die minimale Höhe eines Binärbaumes mit n Knoten ist O($\log_2 n$).

Beweis: Ein *vollständiger* binärer Baum ist ein binärer Baum, in dem alle Ebenen bis auf die letzte vollständig besetzt sind. Offensichtlich hat ein solcher Baum unter allen binären Bäumen mit n Knoten minimale Höhe.

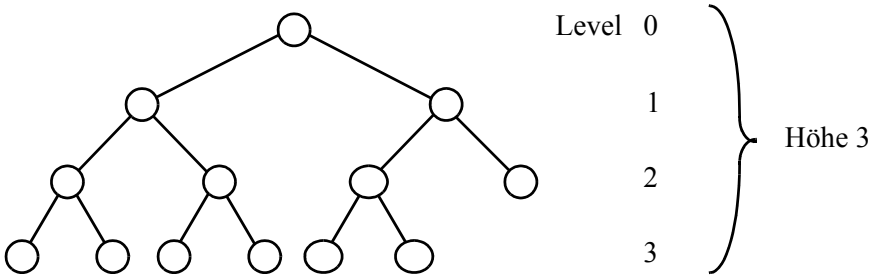


Abbildung 3.28: Vollständiger binärer Baum

Ohne viel zu rechnen, läßt sich die Aussage mit dem folgenden einfachen intuitiven Argument zeigen: Nehmen wir an, auch der letzte Level sei voll besetzt. Wenn man dann einem Pfad von der Wurzel zu einem Blatt folgt, wird in jedem Schritt die Größe des zugehörigen Teilbaumes *halbiert*. log n gibt an, wie oft man n halbieren kann. □

Satz 3.9: Die minimale Höhe eines Binärbaumes mit n Knoten ist exakt $\lfloor \log n \rfloor$. Sei $N(h)$ die maximale Knotenzahl in einem Binärbaum der Höhe h. Dann gilt $N(h) = 2^{h+1} - 1$.

Beweis: Ein Baum der Höhe h hat maximal 2^i Knoten auf dem Level i, also insgesamt

$$N(h) = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Ein vollständiger Baum mit n Knoten hat minimale Höhe. Deshalb besteht zwischen n und h der Zusammenhang:

$$N(h-1)+1 \le n < N(h)+1$$

Damit läßt sich h ausrechnen:

$$\left(2^h - 1\right)+1 \le n < \left(2^{h+1} - 1\right)+1$$

$$2^h \le n < 2^{h+1}$$

$$h = \lfloor \log n \rfloor$$

□

Beobachtung 3.10: Falls alle inneren Knoten zwei Söhne haben, so hat ein binärer Baum mit n+1 Blättern genau n innere Knoten.

Selbsttestaufgabe 3.4: Geben Sie einen Beweis für Beobachtung 3.10 an. □

Einen Datentyp für binäre Bäume könnte man etwa entwerfen, wie in Abbildung 3.29 gezeigt.

Wegen der rekursiven Mengendefinition bezeichnet man Bäume auch als *rekursive* Strukturen. Rekursiven Strukturen entsprechen auf der programmiersprachlichen Ebene *Zeigerstrukturen* bzw. *verkettete* Strukturen.

Listen lassen sich ebenfalls als rekursive Strukturen auffassen:

$$list = \{\Diamond\} \cup \{(x, l) \mid x \in elem,\ l \in list\}$$

Es gibt drei interessante Arten, eine lineare Ordnung auf den Knoten eines Baumes zu definieren, genannt *inorder*, *preorder* und *postorder*. Wir können diese Ordnungen als Operationen auffassen, die einen Baum auf eine Liste abbilden:

---

algebra *tree*

sorts	*tree*, *elem*		
ops	*empty*	:	$\rightarrow$ *tree*
	maketree	: *tree* $\times$ *elem* $\times$ *tree*	$\rightarrow$ *tree*
	key	: *tree*	$\rightarrow$ *elem*
	left, *right*	: *tree*	$\rightarrow$ *tree*
	isempty	: *tree*	$\rightarrow$ *bool*
sets	*tree* = $\{\Diamond\} \cup \{(l, x, r) \mid x \in elem,\ l, r \in tree\}$		
	{man beachte die *rekursive* Mengendefinition}		

functions

empty $= \Diamond$

maketree $(l, x, r) = (l, x, r)$

Sei $t = (l, x, r)$. Sonst sind *key*, *left*, *right* undefiniert.

key $(t) = x$

left $(t) = l$

right $(t) = r$

$$isempty\ (t) = \begin{cases} true & \text{falls } t = \Diamond \\ false & \text{sonst} \end{cases}$$

end *tree*.

Abbildung 3.29: Algebra *tree*

ops *inorder*, *preorder*, *postorder* : *tree* $\rightarrow$ *list*

functions

inorder $(\Diamond)$	
$=$ *preorder* $(\Diamond)$	
$=$ *postorder* $(\Diamond) =$	$\Diamond$
inorder $((l, x, r)) =$	*inorder* $(l) \circ \langle x \rangle \circ$ *inorder* (r)
preorder $((l, x, r)) =$	$\langle x \rangle \circ$ *preorder* $(l) \circ$ *preorder* (r)
postorder $((l, x, r)) =$	*postorder* $(l) \circ$ *postorder* $(r) \circ \langle x \rangle$

Beispiel 3.11: Abbildung 3.30 zeigt die resultierenden Folgen beim Durchlaufen eines Baumes in *inorder*, *preorder* und *postorder*-Reihenfolge.

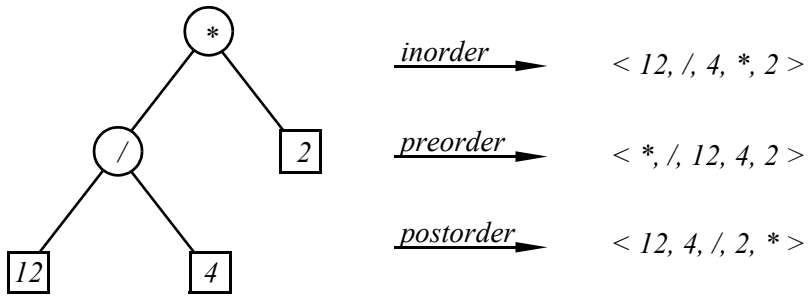


Abbildung 3.30: Baumdurchläufe

Bei arithmetischen Ausdrücken wird das Ergebnis eines *inorder*-, *preorder*- oder *postorder*-Durchlaufs auch Infix-, Präfix-, Postfix-Notation genannt. □

Selbsttestaufgabe 3.5: Schreiben Sie eine Methode für die Operation

height: *tree* → *integer*,

die die Höhe eines binären Baumes berechnet. □

Implementierungen

(a) mit Zeigern

Diese Implementierung ist ähnlich der von einfach oder doppelt verketteten Listen. Hier besitzt jeder Knoten zwei Zeiger auf die beiden Teilbäume. Das führt zu folgenden Deklarationen:

```
class Node
{
   Elem key;
   Node left, right;
   … (Konstruktor und Methoden der Klasse Node)
}
```

Eine damit aufgebaute Struktur ist in Abbildung 3.31 gezeigt. Die Implementierung der Operation *maketree* würde hier in Form eines Konstruktors so aussehen:

```
public Node(Node l, Elem x, Node r)
{
   left = l; key = x; right = r;
}
```

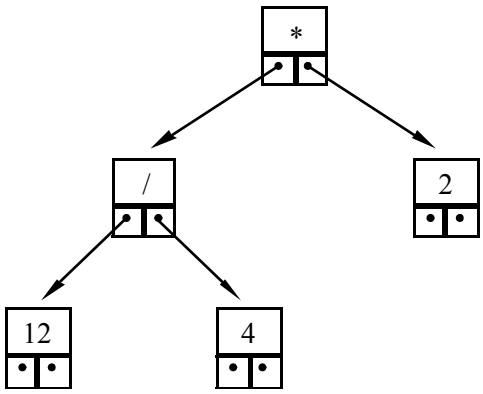


Abbildung 3.31: Binärbaum-Implementierung mit Zeigern

Komplettiert wird die Definition des Datentyps *tree* durch die Definition einer entsprechenden Klasse:

```
class Tree
{
   Node root;
   … (Konstruktor und Methoden der Klasse Tree)
}
```

Die Klasse *Tree* enthält also einen Zeiger auf die Wurzel eines Baumes von Knoten, die durch die Klasse *Node* implementiert werden.

Anmerkung. An dieser Stelle ist die Notwendigkeit einer zusätzlichen Klasse (*Tree*), die lediglich auf die Wurzel des eigentlichen Baumes (Klasse *Node*) verweist, noch nicht recht einzusehen. Im nächsten Kapitel werden wir jedoch sehen, wie Knoten aus einem binären Suchbaum gelöscht werden. In einigen Fällen wird dabei die Wurzel entfernt, und ein Sohn der gelöschten Wurzel wird zur neuen Wurzel. Eine Änderung der Wurzel kann aber nur dann realisiert werden, wenn ein *Tree*-Objekt die Wurzel des Baumes als Attribut enthält.

(b) Array - Einbettung

Neben der stets möglichen Einbettung einer Zeigerstruktur in einen Array mit Array-Indizes als Zeigern (Abschnitt 3.1) gibt es eine spezielle Einbettung, die völlig ohne (explizite) Zeiger auskommt.

Man betrachte einen vollständigen binären Baum der Höhe h.

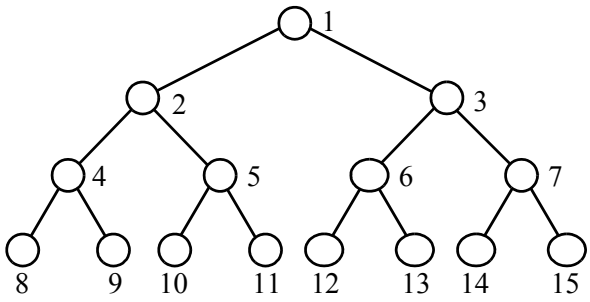


Abbildung 3.32: Vollständiger binärer Baum

Dieser wird in einen Array der Länge 2^{h+1} - 1 eingebettet, so daß die Knoten mit Nummer i in Abbildung 3.32 in Feld i des Arrays dargestellt werden. Es gilt für jeden Knoten p des Baumes:

$$\begin{aligned} Index\,(p) = i \quad &\Rightarrow Index\,(p.left) \;\; = 2 \cdot i \\ &\quad\; Index\,(p.right) = 2 \cdot i + 1 \end{aligned}$$

Das ist für spezielle Anwendungen mit vollständigen oder fast vollständigen Bäumen eine gute Darstellung, die wir in den folgenden Kapiteln noch benutzen werden.

Zum Schluß dieses Abschnitts sei noch eine Implementierung der Funktion *inorder* als Methode der Klasse *Node* auf der Basis der Datentypen *tree* und $list_1$ (deren Implementierungen hier gar nicht bekannt sein müssen) gezeigt:

```
public List inorder()
{
   List l, x, r;
   if (left  == null) l = new List(); else l =  left.inorder();
   if (right == null) r = new List(); else r = right.inorder();
   x = new List(); x.Insert(x.Front(), key);
   return l.Concat(x.Concat(r));
}
```

3.6 (Allgemeine) Bäume

In allgemeinen Bäumen ist die Anzahl der Söhne eines Knotens beliebig. Im Gegensatz zum binären Baum gibt es keine “festen Positionen” für Söhne (wie left, right), die auch unbesetzt sein könnten (ausgedrückt durch leeren Teilbaum). Deshalb beginnt die rekursive Definition bei Bäumen, die aus einem Knoten bestehen.

Definition 3.12: (Bäume)

(i) Ein einzelner Knoten x ist ein Baum.

Graphische Darstellung:

(ii) Wenn x ein Knoten ist und T_1, ..., T_k Bäume sind, dann ist auch das (k+1)-Tupel (x, T_1, ..., T_k) ein Baum. Graphische Darstellung:

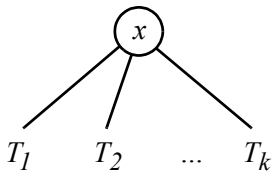


Abbildung 3.33: Graphische Darstellung des (k+1)-Tupels (x, T_1, ..., T_k)

Alle bei binären Bäumen eingeführten Begriffe (außer denen, die sich auf die Positionen beziehen, wie z.B. *linker Teilbaum*) lassen sich übertragen. Zusätzlich gilt:

- Der *Grad eines Knotens* ist die Anzahl seiner Söhne,
- der *Grad eines Baumes* ist der maximale Grad eines Knotens im Baum.
- Ein *Wald* ist eine Menge von m ($m \geq 0$) Bäumen mit disjunkten Knotenmengen. Beispielsweise erhält man einen Wald, indem man die Wurzel eines Baumes entfernt.

Satz 3.13: Die maximale Höhe eines Baumes vom Grad d mit n Knoten ist n-1, die minimale Höhe ist $O(\log_d n)$.

Beweis: Die Aussage über die maximale Höhe ist klar, da auch hier der Baum zu einer Liste entarten kann.[3] Sei $N(h)$ die maximale Knotenzahl in einem Baum der Höhe h.

Wie man in Abbildung 3.34 erkennt, gilt:

$$N(h) = 1 + d + d^2 + \ldots + d^h = \sum_{i=0}^{h} d^i$$

3. Die Sichtweise bei dieser Aussage ist die, daß man eine Liste von n Knoten hat, die jeweils potentiell d Söhne haben könnten. Dies entspricht einer Implementierung mit einem Knotenrecord fester Größe, wie in Abbildung 3.35 gezeigt. Strenggenommen ist das dann allerdings kein Baum vom Grad d, sondern ein Baum vom Grad 1.

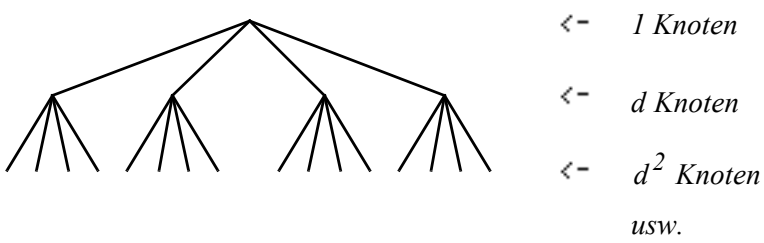


Abbildung 3.34: Baum mit maximaler Knotenanzahl

An dieser Stelle sollten Sie sich mit dem Abschnitt Grundlagen I des Anhangs "Mathematische Grundlagen" vertraut machen, dessen Inhalt im folgenden vorausgesetzt wird. Damit ergibt sich hier:

$$N(h) = \frac{d^{h+1} - 1}{d - 1}$$

Wie bereits bei der Bestimmung der minimalen Höhe von Binärbäumen haben maximal gefüllte Bäume minimale Höhe und legen einen Zusammenhang zwischen n und h fest:

$$N(h-1) < n \leq N(h)$$

Damit kann man ausrechnen:

$$\frac{d^h - 1}{d - 1} < n \leq \frac{d^{h+1} - 1}{d - 1}$$

$$d^h < n \cdot (d-1) + 1 \leq d^{h+1}$$

$$h < \log_d \left(n(d-1) + 1 \right) \leq h + 1$$

$$h = \left\lceil \log_d \left(n(d-1) + 1 \right) \right\rceil - 1$$

Wegen $n \cdot (d-1) + 1 < n \cdot d$ gilt weiterhin

$$h \leq \lceil \log_d nd \rceil - 1 = \lceil \log_d n \rceil + \log_d d - 1 = \lceil \log_d n \rceil$$

□

Ein Datentyp für Bäume ergibt sich analog zu dem bei binären Bäumen.

Implementierungen

(a) über Arrays

Vorraussetzung für diese Implementierung ist, daß alle Knoten höchstens einen festen Grad *d* haben. Dann können Zeiger auf die Söhne eines Knotens in einem Array gespeichert werden, so daß *sons*[*i*] einen Zeiger auf den *i*-ten Sohn enthält.

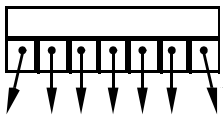


Abbildung 3.35: Array-Implementierung allgemeiner Bäume

```
type   node = record key  : elem;
                     sons: array [1..d] of ↑node
              end
```

Diese Implementierung ist schlecht, wenn der Knotengrad stark variiert, da dann Platz verschwendet wird.

(b) über Binärbäume

Die Struktur eines Knotens ist hierbei gleich der in einem Binärbaum. Allerdings werden die Zeiger anders interpretiert. Es gibt jeweils statt eines Zeigers auf den linken und rechten Teilbaum einen auf den am weitesten links stehenden Sohn und einen auf den rechten Bruder. Dadurch ist ein direkter Zugriff auf den *i*-ten Sohn nicht möglich, man muß sich über die Liste von Geschwistern hangeln. Andererseits wird kaum Speicherplatz für nicht benutzte Zeiger verschwendet. In Abbildung 3.36 ist eine solche Struktur gezeigt; dabei sind die tatsächlichen Zeiger durchgezogen, die gedachten gestrichelt dargestellt.

Eine entsprechende Typdeklaration dazu sieht so aus:

```
type   node= record key           : elem;
                    leftmostChild :↑node;
                    rightSibling  :↑node
             end
```

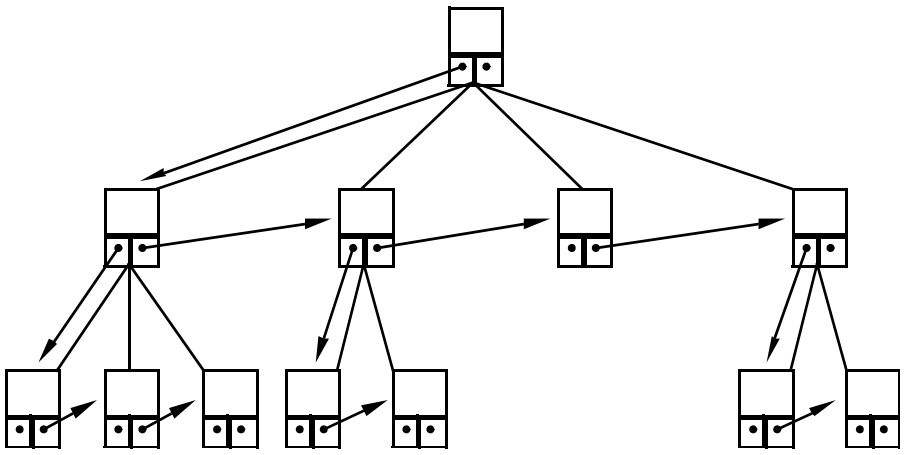


Abbildung 3.36: Binärbaum-Implementierung allgemeiner Bäume

3.7 Weitere Aufgaben

Aufgabe 3.6: In einer *zyklischen Liste (Ring)* ist das erste Element der Nachfolger des letzten Elementes. Für derartige Listen soll ein Datentyp *cycle* definiert werden, der folgende Operationen anbietet:

- *empty*: liefert einen leeren Ring.
- *append*: fügt ein neues letztes Element ein.
- *first*: liefert die Position des ersten Elementes.
- *next*
- *isempty*
- *isfirst*: prüft, ob ein gegebenes Element das erste ist.
- *split*: Gegeben seien zwei Elemente a und b innerhalb eines Ringes r. Diese Operation zerlegt r in zwei Ringe. Der erste Ring enthält die Elemente von a bis b, der zweite Ring die übrigen Elemente.
- *merge*: verschmilzt zwei Ringe und ist mit einem Element a im ersten Ring r und einem Element b im zweiten Ring s parametrisiert. Beginnend mit dem Element b wird der Ring s hinter dem Element a in den Ring r eingebaut.

Spezifizieren Sie eine entsprechende Algebra!

Hinweis: Die Operation *split* liefert zwei Ergebnisse. In leichter Erweiterung der Definition einer mehrsortigen Algebra erlauben wir die Angabe einer entsprechenden Funktionalität, z.B.

$$alpha\colon a \times b \rightarrow c \times d$$

Das Ergebnis der Operation wird als Paar notiert. Eine derartige Operation findet sich auch im Abschnitt 4.3 (Operation *deletemin* in der Algebra *pqueue*).

Aufgabe 3.7: Implementieren Sie die Operationen *empty*, *front*, *next*, *bol*, *eol* und *insert* des Datentyps $list_2$ in Java. Die Listenelemente sollen das Interface *Elem* realisieren.

Aufgabe 3.8: Unter Benutzung des Datentyps $list_1$ sollen Polynome der Form

$$p(x) = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_0, \qquad \text{mit } n \geq 0$$

dargestellt werden. Ein Listenelement enthält dabei den Koeffizienten c_i sowie den Exponenten i. Schreiben Sie einen Algorithmus, der Polynome dieser Form differenziert.

Aufgabe 3.9: Eine mögliche Datenstruktur zur Darstellung von Polynomen ist eine einfach verkettete Liste mit folgendem Elementtyp:

```
class Summand implements Elem { int coeff, exp; }
```

Die einzelnen Summanden eines Polynoms seien dabei nach absteigenden Exponenten geordnet. Schreiben Sie eine Methode, die zwei Polynome addiert, wobei das Ergebnis eine möglichst einfache Form haben soll, d.h. "Summanden" mit dem Koeffizienten 0 nicht vorkommen.

Aufgabe 3.10: Angenommen, Sie hätten eine vollständige Java-Implementierung des im Text angegebenen Datentyps $list_2$ zur Verfügung. Geben Sie eine Implementierung der Datentypen

(a) *stack* und
(b) queue

auf dieser Basis an.

Aufgabe 3.11: Erweitern Sie die algebraische Spezifikation von *tree* um die Funktionen

(a) *contains*: $tree \times elem \rightarrow bool$
(b) *ancestors*: $tree \times elem \rightarrow elemset$

Die Operation *contains* (t, x) liefert *true*, wenn x im Baum t auftritt. Die Operation *ancestors* (t, x) liefert die Menge aller Schlüsselelemente aus den Vorfahrknoten des Knotens mit dem Schlüssel x.

Aufgabe 3.12: Die Knoten eines binären Baumes seien durchnumeriert von 1 bis n. Angenommen, wir haben Arrays *preorder*, *inorder* und *postorder*, die zu jedem Knoten i die Position des Knotens in der zugehörigen Ordnung enthalten. Beschreiben Sie einen Algorithmus, der für zwei Knoten mit Nummern i und j bestimmt, ob i Vorfahr von j ist.

Erklären Sie, warum Ihr Algorithmus funktioniert. Sind tatsächlich alle drei Ordnungen notwendig?

Aufgabe 3.13: Entwerfen Sie jeweils einen nicht-rekursiven Algorithmus für den

(a) *preorder*-Durchlauf
(b) *postorder*-Durchlauf

durch einen binären Baum unter Verwendung eines Stacks. Die Ergebnisliste soll unter Verwendung des Datentyps $list_2$ aufgebaut werden.

3.8 Literaturhinweise

Die Datenstrukturen dieses Kapitels sind so elementar, daß sie von den Anfängen der Programmierung an eingesetzt wurden und die Ursprünge vielfach im Dunkeln liegen. Viele dieser Konzepte sind wohl unabhängig voneinander von verschiedenen Personen entwickelt worden. Eine genaue Bestimmung der Erfinder z.B. der einfach verketteten Liste oder des Binärbaumes ist daher nicht möglich. Ein recht ausführlicher Versuch einer "historischen" Aufarbeitung findet sich bei Knuth [1997] (Abschnitt 2.6).

Es gibt viele Varianten von Techniken, Listen zu implementieren. Ottmann und Widmayer [2012] zeigen die Benutzung eines "Listenschwanz"-Elementes (*tail*) neben einem Listenkopf; dadurch können einige der Listenoperationen einfacher implementiert werden. [Horowitz, Sahni und Anderson-Freed 2007] enthält eine Fülle von Material zu Listen, Stacks und Queues, insbesondere viele Anwendungsbeispiele, etwa zur Speicherverwaltung, Addition von Polynomen oder Darstellung spärlich besetzter Matrizen. Die Technik zur Umwandlung von rekursiven in iterative Programme ist angelehnt an [Horowitz und Sahni 1990] (Abschnitt 4.8), sie ist dort detaillierter beschrieben.

Die Idee zur Aufnahme eines speziellen Datentyps für Abbildungen (*mapping)* stammt von Aho, Hopcroft und Ullman [1983].

Lösungen zu den Selbsttestaufgaben

Aufgabe 3.1

```
public List delete(Pos p)
{
   if (this.isempty()) return null;
   else
   {
      if(eol(p))
      {
         this.pred = p.pred;
         p.pred.succ = null;
      }
      else
      {
         p.pred.succ = p.succ;
         p.succ.pred = p.pred;
      }
   }
   return this;
}
```

Aufgabe 3.2

```
public List swap(Pos p)
{
   if (this.eol(p)) return null;
   else
   {
      Pos q = p.succ;
      Pos r = q.succ;
      Pos s = r.succ;
      /* q und r sind zu vertauschen, s koennte null sein. */
      if(q == this.last) this.last = r;
      else if(r == this.Last) this.last = q;
      p.succ = r;
      r.succ = q;
```

```
        q.succ = s;
      }
      return this;
    }
```

Diese Lösung ist korrekt, da für p die Position p_0 nicht zulässig ist und $p = p_n$ zu Anfang ausgeschlossen wird (vgl. Abbildung 3.7). Folglich sind q und r beide von *null* verschieden.

Aufgabe 3.3

Die Idee bei dieser Implementierung ist, einen Stack s_1 nur für das Einfügen, den anderen (s_2) nur für das Entnehmen von Elementen der Queue zu verwenden. Beim Einfügen werden die Elemente einfach auf den ersten Stack gelegt, beim Entnehmen vom zweiten Stack genommen. Wann immer der zweite Stack leer ist, werden alle gerade im ersten Stack vorhandenen Elemente auf den zweiten übertragen, wodurch sie in die richtige Reihenfolge für das Entnehmen geraten. Dieses "Umschaufeln" wird von der Hilfsmethode *shovel* übernommen.

```
import Stack;

public class Queue
{
  Stack s1 = new Stack();
  Stack s2 = new Stack();

  public boolean isempty(){ return (s1.isempty() && s2.isempty());}

  public void enqueue(Elem e) { s1.push(e); }

  private void shovel(Stack from, Stack to)
  {
    while(!from.isempty())
    {
      to.push(from.top());
      from.pop();
    }
  }
```

```
    public Elem front()
    {
       if(s2.isempty()) shovel(s1, s2);
       if(s2.isempty()) return null; else return s2.top();
    }

    public void dequeue()
    {
       if(s2.isempty()) shovel(s1, s2);
       if(s2.isempty()) return; else s2.pop();
    }
}
```

Aufgabe 3.4

Die Behauptung ist äquivalent zu der Aussage, daß die Differenz zwischen der Anzahl der Blätter und der Anzahl der inneren Knoten gleich 1 ist. Wir definieren eine Funktion Δ, die zu einem Baum diese Differenz berechnet. Da jeder innere Knoten zwei Nachfolger hat, ergeben sich für die Anzahl der Blätter B, die Anzahl der inneren Knoten I sowie für die Differenz $\Delta = B - I$ die beiden folgenden Fälle:

(1) Blatt x:

$$B((\Diamond, x, \Diamond)) = 1,\ I((\Diamond, x, \Diamond)) = 0$$
$$\Rightarrow \Delta((\Diamond, x, \Diamond)) = 1$$

(2) Baum (T_1, x, T_2):

$$B((T_1, x, T_2)) = B(T_1) + B(T_2),$$
$$I((T_1, x, T_2)) = I(T_1) + I(T_2) + 1$$
$$\Rightarrow \Delta((T_1, x, T_2)) = B(T_1) + B(T_2) - (I(T_1) + I(T_2) + 1)$$
$$= \Delta(T_1) + \Delta(T_2) - 1$$

Nun können wir durch Induktion über die Anzahl m der gesamten Blätter im Baum zeigen, daß Δ immer 1 liefert:

Im Fall $m = 1$ ist der Baum ein Blatt, und wir wissen bereits, $\Delta = 1$.

Angenommen, $\Delta = 1$ gilt für alle Bäume mit nicht mehr als m Knoten. Dann ergibt sich für m+1 (für den zu betrachtenden Baum ist Fall (2) anzuwenden):

$$\Delta((T_1, x, T_2)) = \Delta(T_1) + \Delta(T_2) - 1.$$

Aus der Induktionsannahme folgt (da T_1 und T_2 weniger als m Knoten besitzen), daß $\Delta(T_1) = 1$ und $\Delta(T_2) = 1$. Damit ergibt sich

$$\Delta((T_1, x, T_2)) = 1,$$

womit die Behauptung bewiesen ist. □

Aufgabe 3.5

```
public int height()
{
   int l = 0, r = 0;
   int max;
   if(left  != null) l = 1 + left.height();
   if(right != null) r = 1 + right.height();
   if(l > r) max = l; else max = r;
   return max;
}
```

Literatur

Aho, A.V., J.E. Hopcroft und J.D. Ullman [1983]. Data Structures and Algorithms. Addison-Wesley Publishing Co., Reading, Massachusetts.

Bauer, F. L. und H. Wössner [1984]. Algorithmische Sprache und Programmentwicklung. 2.Aufl., Springer-Verlag, Berlin.

Horowitz, E. und S. Sahni [1990]. Fundamentals of Data Structures in Pascal. 3rd Ed., Computer Science Press, New York.

Horowitz, E., S. Sahni und S. Anderson-Freed [2007]. Fundamentals of Data Structures in C. 2nd Ed., Silicon Press, Summit, NJ.

Knuth, D.E. [1997]. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. 3rd Ed., Addison-Wesley Publishing Co., Reading, Massachusetts.

Ottmann, T., und P. Widmayer [2012]. Algorithmen und Datenstrukturen. 5. Aufl., Spektrum Akademischer Verlag, Heidelberg.

Index