

# Abschnitt 11ad

8. Dezember 1999

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>8</b>
1.1	Was ist Informatik? . . . . .	8
1.2	Geschichte der Programmierung . . . . .	16
<b>2</b>	<b>Induktive Definitionen</b>	<b>20</b>
2.1	Natürliche Zahlen . . . . .	20
2.2	Wortmengen . . . . .	27
2.3	Syntaktische Beschreibungsmittel . . . . .	30
2.3.1	Backus-Naur-Form . . . . .	31
2.3.2	Erweiterte BNF . . . . .	37
2.4	Terme . . . . .	42
2.5	Aufgaben . . . . .	50
<b>3</b>	<b>Algorithmen und Programme</b>	<b>55</b>
3.1	Verifikation von Algorithmen . . . . .	79
3.2	Termination und Rechenaufwand . . . . .	80
3.3	Aufgaben . . . . .	89
<b>4</b>	<b>Abstrakte Datentypen</b>	<b>93</b>
4.1	Einführung . . . . .	93
4.2	Boolesche Werte . . . . .	95
4.3	Zähler . . . . .	98
4.4	Listen . . . . .	101
4.5	Bäume . . . . .	108
4.6	Aufgaben . . . . .	116

# Kapitel 4

## Abstrakte Datentypen

### 4.1 Einführung

Beim Entwurf eines Programms treten neben den Algorithmen auch (sozusagen gleichberechtigt) problemspezifische *Datenstrukturen* bzw. *Datentypen* auf. In der Tat kann man sagen, daß sich die Lösung eines Problems auf Datenstrukturen und Algorithmen verteilt. Wir haben dies am Beispiel der „Türme von Hanoi“ und des Nim-Spiels gesehen. In Abhängigkeit von der Programmiersprache müssen wir solche Datenstrukturen immer irgendwie *darstellen*, d.h. wir müssen sie auf Vorrichtungen abbilden, welche die Programmiersprache anbietet. So haben wir etwa bei den „Türmen von Hanoi“ Züge durch Paare und Zugfolgen durch Listen dargestellt. Die Praxis der Programmierung hat gezeigt, daß eine solche Darstellung immer etwas andere Eigenschaften hat, als wir uns „abstrakt“ von der gewünschten Datenstruktur wünschen würden. Viele Fehler im Programmieren werden dadurch begangen, daß wir uns an der Stelle, wo wir die Datenstruktur verwenden, auf Eigenschaften der Darstellung verlassen. Wird dann später die Darstellung aus irgendwelchen Gründen geändert, z.B. zur Effizienzsteigerung oder wegen der Einführung neuer Leistungen, so müssen alle Verwendungsstellen aufgespürt werden, an denen auf die alte Darstellung bezug genommen wurde. Wird dabei eine Verwendungsstelle übersehen, so wird das Programm später aus unerklärlichen Gründen abstürzen.

Deshalb ist der Gedanke vernünftig, Datenstrukturen in einer abstrakten Weise zu verwenden, so daß wir uns an der Verwendungsstelle auf keine Details der Darstellung beziehen können. Die Trägermenge und die genaue Darstellung der Operationen ist an der Verwendungsstelle nämlich dann gar nicht maßgeblich, wenn wir uns auf die zulässigen Operationen der Datenstruktur und deren spezifizierte Eigenschaften beschränken. Diese Einschränkung hat den Vorteil, daß man während der Algorithmen- bzw. Programmentwicklung die interne Darstellung des Datentyps sowie die Implementierung der zulässigen Operationen nach Belieben (z.B. aus

Effizienzgründen) ändern kann, ohne daß dies einen Einfluß auf die Verwendungsstellen hat. David L. PARNAS hat hierfür das Wort „Geheimnisprinzip“ („*information hiding principle*“) eingeführt. In der Softwaretechnik nennt man dieses wichtige Konzept *Datenkapselung* oder *Datenabstraktion*.

Im Zusammenhang mit Programmiersprachen verwendet man das Wort „Datenstruktur“ häufig für komplexere Gebilde, die stärker strukturiert sind als die Elemente der sogenannten „primitiven Datentypen“. Wir machen diesen Unterschied hier nicht, sondern verwenden die Wörter „Datentyp“ und „Datenstruktur“ synonym.

Datentypen sind für uns nicht bloß Mengen, sondern es interessiert in der Informatik vor allem, welche *Operationen* man auf den Daten ausführen kann. Mit anderen Worten: Datentypen sind *Algebren* im Sinne der im vergangenen Kapitel eingeführten Terminologie.

**Definition 4.1 (Datentyp)** Ein *Datentyp*  $\mathcal{D}$  ist eine  $\Sigma$ -Algebra. Wir schreiben einen Datentyp in der Form  $\mathcal{D} = \langle M; \Sigma \rangle$  bzw.  $\mathcal{D} = \langle M; f_1, \dots, f_n \rangle$ , wenn  $\Sigma = f_1, \dots, f_n$ .

Wir werden sehen, daß wir gegenüber Definition 2.34 einen allgemeineren Begriff von Algebra brauchen. Die dort vorgestellten Algebren sind sog. *homogene* oder *einsortige* Algebren: Alle Elemente der Trägermenge sind von gleicher Art. Für die Beschreibung von Datentypen brauchen wir *heterogene* oder *mehrsortige* Algebren, bei denen die Trägermenge in Teilmengen unterschiedlicher Bauart zerfällt. Zunächst machen wir uns jedoch an einem einsortigen Beispiel klar, wieso die Zuordnung von bestimmten Operationen zu einer Menge für die Informatik besonders wichtig ist:

**Beispiel 4.2** Sei  $\mathbf{Q}$  die Teilmenge der rationalen Zahlen, die sich in Dezimaldarstellung mit maximal 10 Ziffern schreiben lassen. Der zugrundeliegende Datentyp eines einfachen Taschenrechners ist dann

$$\mathcal{Q}_1 \stackrel{\text{def}}{=} \langle \mathbf{Q}; +, -, \times, / \rangle$$

(Dabei sind alle Operationen Funktionen von  $\mathbf{Q} \times \mathbf{Q}$  nach  $\mathbf{Q}$ .) Wenn wir auf diesem Taschenrechner etwa Prozentrechnung durchführen wollen, so müssen wir diese zuerst auf die Operationen des eingebauten Datentyps reduzieren, d.h. also z.B. „ $\times 5\%$ “ ersetzen durch „ $\times 0,05$ “ und „ $+5\%$ “ durch

„×1,05“. Ein etwas komfortablerer Taschenrechner bietet uns vielleicht den folgenden Datentyp an:

$$\mathcal{Q}_2 \stackrel{\text{def}}{=} \langle \mathbf{Q}; +, -, \times, /, \%, \sqrt{x}, x^2 \rangle$$

Beide Datentypen haben die gleiche Trägermenge, aber die Existenz der „ $\sqrt{x}$ “-Funktion in  $\mathcal{Q}_2$  stellt ein schwerwiegendes Problem dar, wenn wir eine Rechnung des  $\mathcal{Q}_2$ -Rechners auf dem  $\mathcal{Q}_1$ -Rechner ausführen wollen. Noch komplizierter wird es, wenn wir den folgenden „Luxus-Datentyp“ betrachten:

$$\mathcal{Q}_3 \stackrel{\text{def}}{=} \langle \mathbf{Q}; +, -, \times, /, \%, \sqrt{x}, x^2, \sin, \cos, \tan, \exp, \ln \rangle$$

Hier ist es für den Normalbürger schlicht unmöglich, eine Rechnung eines  $\mathcal{Q}_3$ -Rechners auf dem  $\mathcal{Q}_1$ -Rechner nachzuvollziehen.

Wir sehen, daß die bloße Angabe der Trägermenge keine besondere Aussagekraft hat, wenn wir an Rechenprozesse denken; hier sind die operativen Fähigkeiten eines Datentyps wesentlich wichtiger.

Auf den ersten Blick scheint es zunächst unmöglich, über Datentypen unabhängig von ihrer Darstellung reden zu können: jede Bezeichnung, die wir uns für Elemente abstrakter Datentypen ausdenken, ist selbst schon eine Darstellung. Die *Terme* zusammen mit dem Erzeugungsprinzip und Satz 2.35 geben uns eine ziemlich neutrale Möglichkeit der Bezeichnung an die Hand, wenn wir uns auf induktiv erzeugte Datentypen beschränken:

Jedes Element eines induktiv erzeugten Datentyps läßt sich durch einen (variablenfreien) Term bezeichnen. Die Menge  $T_\Sigma \stackrel{\text{def}}{=} T_\Sigma(\emptyset)$  der variablenfreien Terme bietet sich also als Trägermenge eines abstrakten Datentyps an.

## 4.2 Boolesche Werte

Als trivialstes Beispiel wollen wir die schon bekannte Boolesche Algebra als abstrakten Datentyp definieren. Wir verwenden dabei eine Art von Datentyp-Spezifikationsprache, die wir hier nicht formal einführen.

Wenn wir abstrakte Datentypen spezifizieren, abstrahieren wir von der *Trägermenge* und können dann natürlich auch die Operationen nicht explizit

angeben. Der Schlüssel zur präzisen Definition von abstrakten Datentypen ohne Angabe der Trägermenge liegt in der Idee, die Trägermenge selbst mit Hilfe von sogenannten *erzeugenden Operationen* zu definieren.

```

datatype Boolean;
sorts Boolean;
constructors
    true  : → Boolean;
    false : → Boolean;
operations
    not   : Boolean → Boolean;
    and   : Boolean × Boolean → Boolean;
    or    : Boolean × Boolean → Boolean;
end

```

Dies ist ein besonders trivialer Fall einer induktiven Definition, denn sie besteht nur aus der Induktionsverankerung: die einzigen Elemente der Sorte *Boolean*, die wir überhaupt konstruieren können, sind *true* und *false*. Die zusätzlichen Operationen *not*, *and* und *or* erzeugen keine weiteren Datenelemente.

Durch eine solche Definition wird nur ein Operationsalphabet  $\Sigma$  in einer programmiersprachenähnlichen Notation festgelegt. Spezifikationen dieser Art sind reine *Syntax*; wir müssen separat erklären, was denn die *Semantik* davon sein soll, d.h. was solche Spezifikationen bedeuten. In der mathematischen Logik spricht man davon, *Modelle* für Definitionen zu finden; Informatiker würden wahrscheinlich eher von *Implementierungen* reden. In erster Näherung würden wir jede  $\Sigma$ -Algebra als ein Modell des oben vorgestellten Datentyps betrachten. Betrachten wir also einige Modelle:

1.  $A = \{W, F\}$  mit den Operationen

$$\begin{aligned}
 \text{true}_A &= W \\
 \text{false}_A &= F \\
 \text{not}_A(x) &= \text{if } x = W \text{ then } F \text{ else } W \\
 \text{and}_A(x, y) &= \text{if } x = W \text{ then } y \text{ else } F \\
 \text{or}_A(x, y) &= \text{if } x = W \text{ then } W \text{ else } y
 \end{aligned}$$

Dies ist bis auf Isomorphie sicher das Modell, das wir im Auge gehabt haben; es entspricht dem, was wir im Abschnitt „Aussagenlo-

gik“ beschrieben haben. Zu beachten ist aber, daß Namen nur Schall und Rauch sind: hätten wir `and` und `or` jeweils mit vertauschten Definitionen aufgeschrieben oder hätten wir etwa definiert

$$\begin{aligned}\text{and}_{\mathcal{A}}(x, y) &= x \\ \text{or}_{\mathcal{A}}(x, y) &= y\end{aligned}$$

so wäre unsere Algebra  $\mathcal{A}$  nicht die Boolesche Algebra, aber immer noch eine ganz legitime  $\Sigma$ -Algebra. Man kann aber auch ganz andere Modelle definieren:

2.  $B = \{\text{maybe}\}$  mit den Operationen

$$\begin{aligned}\text{true}_B &= \text{maybe} \\ \text{false}_B &= \text{maybe} \\ \text{not}_B(x) &= \text{maybe} \\ \text{and}_B(x, y) &= \text{maybe} \\ \text{or}_B(x, y) &= \text{maybe}\end{aligned}$$

Dieses Modell ist offensichtlich zu klein, um etwas sinnvolles damit anstellen zu können; in der Mathematik nennt man dies die „einpunktige Algebra“. Es ist aber genau so leicht, Modelle zu definieren, die zu groß sind:

3.  $C = \mathbb{Z}$  mit den Operationen

$$\begin{aligned}\text{true}_C &= -1 \\ \text{false}_C &= 0 \\ \text{not}_C(x) &= \text{if } x \neq 0 \text{ then } 0 \text{ else } -1 \\ \text{and}_C(x, y) &= x \cdot y \\ \text{or}_C(x, y) &= \text{if } x = 0 \text{ then } y \text{ else } x\end{aligned}$$

Dieses Modell ist deshalb zu groß, weil es außer den geforderten Wahrheitswerten noch viele andere Werte enthält, für die wir die Operationen ebenfalls definieren müssen, wenn wir totale Operationen haben wollen. Bei induktiven Definitionen werden solche „Nicht-Standard-Elemente“ durch das Prinzip des induktiven Abschlusses („Induktionsaxiom“) ausgeschlossen. Wir haben uns hier die Freiheit genommen, bei der „Implementierung“ eine Ausnahme vom Induktionsaxi-

om zu machen; solche Freiheiten nimmt man sich beim implementieren häufiger. Man kann übrigens mit den hier angegebenen Operationen sinnvoll rechnen, wie die folgenden Beispiele zeigen:

$$\begin{aligned}\text{not}_C(\text{true}_C) &= \text{not}_C(-1) = 0 = \text{false}_C \\ \text{not}_C(\text{false}_C) &= \text{not}_C(0) = -1 = \text{true}_C \\ \text{not}_C(4711) &= 0 = \text{false}_C\end{aligned}$$

Wir können dies so interpretieren, daß in diesem Modell die 0 die Rolle des Wahrheitswerts F spielt, während jede andere Zahl implizit als W interpretiert wird. Eine ähnliche Interpretation liegt in der Programmiersprache C zugrunde, die eigentlich gar keine Wahrheitswerte kennt. Betrachten wir noch einige weitere Beispiele:

$$\begin{aligned}\text{and}_C(\text{true}_C, \text{false}_C) &= (-1) \cdot 0 = 0 = \text{false}_C \\ \text{and}_C(\text{true}_C, 4711) &= (-1) \cdot 4711 = -4711 \\ \text{and}_C(\text{false}_C, 4711) &= 0 \cdot 4711 = 0 = \text{false}_C\end{aligned}$$

Bei der Operation or wollen wir uns auf die „Nicht-Standard-Elemente“ konzentrieren:

$$\begin{aligned}\text{or}_C(\text{true}_C, 4711) &= \text{true}_C \\ \text{or}_C(4711, \text{true}_C) &= \text{true}_C \\ \text{or}_C(5110, 4711) &= 5110 \\ \text{or}_C(4711, 5110) &= 4711\end{aligned}$$

Wir erkennen, daß die Operation or in dieser Algebra nicht kommutativ ist.

### 4.3 Zähler

Der folgende abstrakte Datentyp soll einen Zähler darstellen, den man auf einen definierten Wert (vermutlich 0) zurücksetzen, herauf- und herunterzählen kann:

```
datatype Counter;  
sorts Counter;  
constructors
```

```

        reset  :  → Counter;
        inc    :  Counter → Counter;
operations
        dec   :  Counter → Counter;
end

```

Auch dazu wollen wir verschiedene Modelle betrachten:

1.  $A = \mathbb{N}$  mit den Operationen

$$\begin{aligned}
 \text{reset}_A &= 0 \\
 \text{inc}_A(n) &= n + 1 \\
 \text{dec}_A(n) &= \text{if } n = 0 \text{ then } 0 \text{ else } n - 1
 \end{aligned}$$

Dieses Modell können wir in der Tat dazu verwenden, z.B. über die Anzahl von Menschen in einem Raum Buch zu führen. Die augenblickliche Anzahl („der Wert des Zählers“) läßt sich direkt ermitteln, und man sieht, daß in diesem Modell für alle  $x \in \mathbb{N}$  gilt

$$\text{dec}_A(\text{inc}_A(x)) = x$$

Die „umgekehrte“ Gleichung

$$\text{inc}_A(\text{dec}_A(x)) = x$$

gilt nur für  $x \neq 0$ , denn wir haben das Herunterzählen in diesem Modell einfach bei 0 begrenzt.

2.  $B = \mathbb{N}$  mit den Operationen

$$\begin{aligned}
 \text{reset}_B &= 4711 \\
 \text{inc}_B(n) &= 2n + 1 \\
 \text{dec}_B(n) &= 2n
 \end{aligned}$$

In diesem Modell haben wir willkürlich dafür gesorgt, daß der Wert des Zählers nicht direkt ablesbar ist. Ist in der Algebra  $\mathcal{A}$  etwa der Wert des Terms

$$\text{dec}(\text{dec}(\text{inc}(\text{inc}(\text{reset}))))$$

gleich 0, so ist er in der Algebra  $\mathcal{B}$  gleich 75388. Hieraus die von dem Zähler festgehaltene Anzahl (= 0) zu ermitteln, ist nicht ganz einfach. Dafür hat die Algebra  $\mathcal{B}$  aber eine interessante Eigenschaft: wir

können hier nämlich die komplette Geschichte des Zählers rekonstruieren. Es entstehen durch `inc` nur ungerade Zahlen, durch `dec` nur gerade. Finden wir also eine gerade Zahl vor, so wissen wir, daß die letzte ausgeführte Operation des Zählers eine `dec`-Operation gewesen ist. Finden wir eine ungerade Zahl vor, die verschieden von 4711 ist, so muß die letzte Operation eine `inc`-Operation gewesen sein. Aus der Zahl 75388 können wir also rekonstruieren

$$\begin{aligned}
 75388 &= \text{dec}_B(37694) \\
 &= \text{dec}_B(\text{dec}_B(18847)) \\
 &= \text{dec}_B(\text{dec}_B(\text{inc}_B(9423))) \\
 &= \text{dec}_B(\text{dec}_B(\text{inc}_B(\text{inc}_B(4711)))) \\
 &= \text{dec}_B(\text{dec}_B(\text{inc}_B(\text{inc}_B(\text{reset}_B))))
 \end{aligned}$$

Wir haben mit diesem Modell also eine Methode konstruiert, wie man Terme in Zahlen codieren kann.

3.  $C = \{0, \dots, p\}$  mit den Operationen

$$\begin{aligned}
 \text{reset}_B &= 0 \\
 \text{inc}_B(n) &= \text{if } n = p \text{ then } 0 \text{ else } n + 1 \\
 \text{dec}_B(n) &= \text{if } n = 0 \text{ then } p \text{ else } n - 1
 \end{aligned}$$

Dies ist ein sogenannter Ringzähler, der „modulo  $p + 1$ “ zählt.

Wir erkennen, daß unsere Datentyp-Spezifikationsprache momentan noch nicht mächtig genug ist; es wäre wünschenswert, die Klasse der Modelle stärker einschränken zu können. Ein unproblematischer Weg zu diesem Ziel ist durch *Gleichungen* gegeben. Wir könnten etwa das Zähler-Beispiel durch folgende Gleichungen erweitern:

#### equations

$$\begin{aligned}
 \text{dec}(\text{inc}(c)) &= c; \\
 \text{inc}(\text{dec}(\text{inc}(c))) &= \text{inc}(c)
 \end{aligned}$$

und dann verlangen, daß nur solche Algebren als Modelle in Frage kommen, in denen diese beiden Gleichungen gelten. Die Gültigkeit von Gleichungen läßt sich sinnvoll wie folgt definieren:

**Definition 4.3** Sei  $\Sigma$  ein Operationsalphabet,  $\mathcal{A}$  eine  $\Sigma$ -Algebra,  $X$  eine Variablenmenge,  $t_1, t_2 \in T_\Sigma(X)$ . Die Gleichung

$$t_1 = t_2$$

ist *gültig in  $\mathcal{A}$*  genau dann, wenn für alle Variablenbelegungen  $f : X \rightarrow A$  gilt

$$\hat{f}(t_1) = \hat{f}(t_2)$$

wobei  $\hat{f}$  der eindeutige Homomorphismus entsprechend Satz 2.35 ist. Eine Menge  $E$  von Gleichungen ist *gültig in  $\mathcal{A}$*  genau dann, wenn jede einzelne Gleichung  $e \in E$  in  $\mathcal{A}$  gültig ist.

Mit dieser Definition scheidet die oben vorgestellte Algebra  $\mathcal{B}$  aus, da in ihr beide Gleichungen nicht gelten. Die Algebren  $\mathcal{A}$  (unbeschränkter Zähler) und  $\mathcal{C}$  (Ringzähler) sind jedoch gültige Modelle für die Spezifikation mit den beiden Gleichungen.

## 4.4 Listen

Am Beispiel der Listen, die wir schon von den *Scheme*-Programmen kennen, wollen wir einen abstrakten Datentyp vorführen, der Mengen unterschiedlicher „Sorte“ braucht, von denen eine die Trägermenge des Datentyps ist:

### Beispiel 4.4

```
datatype List(A);
sorts A, List;
constructors
  empty:  → List;
  cons : A × List → List;
operations
  head : List → A;
  tail : List → List;
  empty?: List → Boolean;
  cat: List × List → List;
  len: List → ℕ;
```

**equations**

```

head(cons(a,l)) = a;

tail(cons(a,l)) = l;

empty?(empty)    = True;
empty?(cons(a,l)) = False;

cat(empty,v)     = v;
cat(cons(a,l),l') = cons(a,cat(l,l'));

len(empty)       = 0;
len(cons(a,l))  = len(l) + 1

```

**end**

Wir erkennen zunächst, daß wir nunmehr einen erweiterten Begriff der *Stelligkeit* von Operationssymbolen brauchen. `cons` und `cat` sind beides zweistellige Operationen und sind doch von grundsätzlich anderem Typ. Bei heterogenen Algebren auf der Basis einer Menge  $S$  von Sorten definiert man ein Paar  $(w, s)$  mit  $w \in S^*$  und  $s \in S$  als Stelligkeit von Operationssymbolen. `cons` ist somit ein  $(A \text{ List}, \text{List})$ -stellige Operationssymbol und `cat` ein  $(\text{List List}, \text{List})$ -stelliges. Wir wollen dies hier nicht formal ausführen; wer sich für die Details interessiert, kann in [?] nachschauen.

Die definierenden Gleichungen für `empty?`, `cat` und `len` erfüllen die Bedingungen für die strukturelle Rekursion, so daß wir sicher sein können, daß dadurch Operationen auf  $\text{List}(A)$  eindeutig definiert werden. Die Gleichungen für `head` und `tail` dagegen sind keine vollständigen strukturell-rekursiven Definitionen; in der Tat sind diese Operationen beide partiell. Wir erlauben auch die Spezifikation partieller Operationen durch strukturell-rekursive Gleichungsmengen.

In *Scheme*-Programmen verwendet man z.T. andere Bezeichnungen für die hier beschriebenen Operationen; so schreibt man `car` statt `head`, `cdr` statt `tail` etc. Die hier abweichend gewählte Notation soll helfen, den abstrakten Datentyp „Liste“ von seiner Implementierung in *Scheme* zu unterscheiden.

Wir führen noch die folgenden Bezeichnungen für verschiedene Arten von Operationen bei abstrakten Datentypen ein:

**Definition 4.5 (Operationen in abstrakten Datentypen)**

Konstruktoren	bauen eine Datenstruktur auf. Alle Elemente des abstrakten Datentyps lassen sich durch einen variablenfreien Term beschreiben, der nur Konstruktoren enthält.
Selektoren	zerlegen eine Datenstruktur in ihre Bestandteile und sind somit Inverse zu den Konstruktoren. Sie sind in der Regel partielle Operationen.
Observatoren	liefern Informationen über Elemente des abstrakten Datentyps. Man geht davon aus, daß der innere Aufbau der Elemente eines abstrakten Datentyps unbekannt ist und daß alles, was wir über diese Elemente in Erfahrung bringen können, durch Operationen geliefert wird, deren Werte nicht im abstrakten Datentyp liegen. Manche Selektoren kann man auch als Observatoren betrachten, z.B. <code>head</code> in Beispiel 4.4.
Transformatoren	formen Elemente eines abstrakten Datentyps um. Dies ist eine Klasse von Operationen mit Ergebnis im abstrakten Datentyp, die weder Konstruktoren noch Selektoren sind.

Im Beispiel 4.4 sind `head` und `tail` Selektoren, `empty?`, `head` und `len` Observatoren und `cat` ein Transformator.

Strukturell-rekursive Gleichungssysteme wie in 4.4 kann man, wie wir schon gesehen haben, als *Rechenregeln* verstehen und kann, so lange einem nichts besseres einfällt, Programme schreiben, die genau so funktionieren. Mathematisch haben aber Gleichungssysteme die Eigenschaft, daß man darauf *Kalküle* aufbauen kann, die es ermöglichen, Beweise über gleichungsspezifizierte Objekte zu führen. Den Begriff des Kalküls werden wir im nächsten Abschnitt betrachten. An dieser Stelle wollen wir uns auf ein intuitives Verständnis des „Rechnen in einem Gleichungssystem“ verlassen, wie wir es aus der Schulmathematik mitbringen.

**Beispiel 4.6** Als Beispiel soll gezeigt werden, daß im Datentyp  $\text{List}(A)$  der Term  $\text{cons}(\text{head}(l), \text{tail}(l))$  für alle  $l \neq \text{empty}$  gleichbedeutend zu  $l$  ist. Das bedeutet, daß die durch diese beiden Terme bezeichneten Datenelemente in allen Implementierungen von  $\text{List}(A)$  gleich sind.

Das Erzeugungsprinzip lehrt zunächst, daß stets entweder  $l = \text{empty}$  oder  $l = \text{cons}(a, l')$  für bestimmte  $a, l'$  gilt.  $l \neq \text{empty}$  ist hier vorausgesetzt, also gilt  $l = \text{cons}(a, l')$ . Daraus folgt

$$\begin{aligned} \text{cons}(\text{head}(l), \text{tail}(l)) &= \text{cons}(\text{head}(\text{cons}(a, l')), \text{tail}(\text{cons}(a, l'))) \\ &= \text{cons}(a, \text{tail}(\text{cons}(a, l'))) \text{ nach Gleichung 1} \\ &= \text{cons}(a, l') \text{ nach Gleichung 2} \\ &= l. \end{aligned}$$

Unter Voraussetzung eines Gleichheitsprädikats auf dem Basistyp  $A$  können wir die Gleichheit in  $\text{List}(A)$  durch eine (doppelt) strukturell-rekursive Operation beschreiben:

#### operations

`equal?: List × List → Boolean;`

#### equations

`equal?(empty, empty) = True;`

`equal?(empty, cons(a, l)) = False;`

`equal?(cons(a, l), empty) = False;`

`equal?(cons(a, l), cons(b, m)) = (a=b) and equal?(l, m)`

`equal?` ist das durch das Erzeugungsprinzip induzierte „natürliche“ Gleichheitsprädikat auf  $\text{List}(A)$ . Vielfach verzichtet man darauf, es eigens zu spezifizieren, sondern versteht die Gleichheit von Elementen abstrakter Datentypen standardmäßig auf diese Weise.

Für Suchprobleme in Listen kann es durchaus interessant sein, zu wissen, ob die Liste sortiert ist. Wir beschreiben zunächst sortierte bzw. sortierbare Listen als *Erweiterung* der Listen aus Beispiel 4.4, wobei wir gleichzeitig eine Operation hinzunehmen, die nachprüft, ob ein Element in einer Liste vorkommt:

**Beispiel 4.7** Der Datentyp „Sortierbare Liste“ wird als *Erweiterung* des Datentyps „Liste“ beschrieben. In der nachfolgenden Spezifikation wird dies durch die Zeile „`uses List(A)`“ ausgedrückt. Diese ist so zu verstehen, als ob der Text von 4.4 hier jeweils hinter die entsprechenden Schlüsselwörter einsortiert wäre. Die Zeile mit „`assumes`“ kann als eine Einschränkung an die möglichen Parameter  $A$  verstanden werden, die wir hier akzeptieren. Es sind nur solche Typen  $A$  erlaubt, die über ein Prädikat  $\leq$  verfügen.

**datatype** Sortable\_list(A);

**uses** List(A);

**assumes**  $\leq : A \times A \rightarrow \text{Boolean};$   
 (\*  $\leq$  soll die Gesetze einer Halbordnung befolgen \*)

**operations**

insert:  $A \times \text{List} \rightarrow \text{List};$   
 delete:  $A \times \text{List} \rightarrow \text{List};$   
 sort:  $\text{List} \rightarrow \text{List};$   
 member?:  $A \times \text{List} \rightarrow \text{Boolean};$   
 count:  $A \times \text{List} \rightarrow \mathbb{N};$   
 sorted?:  $\text{List} \rightarrow \text{Boolean};$

**equations**

insert(a,empty) = cons(a,empty);  
 insert(a,cons(b,l)) = if a $\leq$ b then cons(a,cons(b,l))  
                           else cons(b,insert(a,l))  
                           **endif;**

delete(a,empty) = empty;  
 delete(a,cons(b,l)) = if a=b then l  
                           else cons(b,delete(a,l))  
                           **endif;**

(\* Kommt ein Element mehrfach in einer Liste vor, so löscht  
 delete nur das erste Vorkommen \*)

sort(empty) = empty;  
 sort(cons(a,l)) = insert(a,sort(l))

member?(a,empty) = False;  
 member?(a,cons(b,l)) = (a=b) or member?(a,l);

count(a,empty) = 0;  
 count(a,cons(b,l)) = if a=b then 1+count(a,l)  
                           else count(a,l)  
                           **endif;**

```

sorted?(empty)      = True;
sorted?(cons(a,l)) = if empty?(l) then True
                    else a ≤ head(l)
                    and sorted?(l)
                    endif;
end

```

Die Sortiermethode, die wir hier bei der Operation `sort` angewendet haben, heißt „Sortieren durch Einfügen“. Diese elementare Sortiermethode wird wohl von den meisten Menschen z.B. beim Sortieren eines Kartenstapels verwendet. Die hier zusätzlich aufgenommene Operation `count` brauchen wir in Spezifikation 4.8.

Wer einmal versucht hat, Operationen wie `insert` in einer imperativen Programmiersprache wie z.B. PASCAL aufzuschreiben, wird an dieser Stelle bemerken, daß diese deutlich weniger verständlich sind, und dann erkennen, wie vorteilhaft eine abstrakte Beschreibung demgegenüber sein kann.

Als Beispiel, wie man Eigenschaften eines rekursiven Algorithmus über abstrakten Datentypen durch Induktion beweisen kann, führen wir das *Mischen* zweier sortierter Listen in eine sortierte Ausgabeliste vor:

**Spezifikation 4.8 (Mischen)** Zu zwei sortierten Listen  $A, B$  ist die Mischung  $\text{merge}(A, B)$  zu einer sortierten Ausgabefolge zu ermitteln.

**Deklarationen: Typen:**  $S$  eine Menge mit einer Halbordnung  $\leq$ ,  $\text{List}(S)$  die Menge der endlichen Listen über  $S$ .

**Funktionen:**  $\text{merge} : \text{List}(S) \times \text{List}(S) \rightarrow \text{List}(S)$ .

**Eingabe:**  $A, B \in \text{List}(S)$ .

**Vorbedingung:**  $\text{sorted}(A) \wedge \text{sorted}(B)$ .

**Ausgabe:**  $\text{merge}(A, B) \stackrel{\text{def}}{=} C \in \text{List}(S)$ .

**Nachbedingung:**

1.  $(\forall s \in S) \text{count}(s, A) + \text{count}(s, B) = \text{count}(s, C)$
2.  $\text{sorted}(C)$

Auch an diesem Beispiel möchten wir nochmals zeigen, wie bequem mit einem funktionalen Programmierstil gearbeitet werden kann:

**Beispiel 4.9** In unserer Datentyp-Spezifikationsprache kann die Operation `merge` wie folgt beschrieben werden:

```

operations merge: List × List → List;
equations
  merge(A, empty) = A;
  merge(empty, B) = B;
  merge(cons(a1, A), cons(b1, B)) =
    if a1 ≤ b1 then cons(a1, merge(A, cons(b1, B)))
    else cons(b1, merge(cons(a1, A), B))
    endif

```

Wir möchten nun zeigen, daß eine Operation `merge` mit diesen Eigenschaften die Nachbedingung aus 4.8 erfüllt:

**Beweis:** Wir führen eine Induktion über  $l \stackrel{\text{def}}{=} \text{len}(A) + \text{len}(B)$  durch. Für  $l = 0$  ist die Aussage trivial. Gelte die Behauptung bereits für ein gewisses  $l$  und sei  $\text{len}(A) + \text{len}(B) = l + 1$ . Für  $A = \text{empty}$  oder  $B = \text{empty}$  ist die Aussage wieder trivial. Für  $A = \text{cons}(a_1, A')$  und  $B = \text{cons}(b_1, B')$  gilt zunächst `sorted?(A')` und `sorted?(B')` nach Definition von `sorted?`. Somit ist die Vorbedingung für den rekursiven Aufruf von `merge` jeweils gegeben. Wir unterscheiden dann die beiden Fälle in der dritten Zeile der Definition von `merge`:

1.  $a_1 \leq b_1$ : Wegen `sorted?(A)` gilt  $a_1 \leq \text{head}(A')$  und damit  $a_1 \leq c_1 \stackrel{\text{def}}{=} \text{head}(\text{merge}(A', \text{cons}(b_1, B')))$ , da nach Definition von `merge` nur  $c_1 = \text{head}(A')$  oder  $c_1 = b_1$  in Frage kommt. Anwendung der Induktionsvoraussetzung liefert `sorted?(merge(A', B))`, woraus

$$\text{sorted?}(\text{cons}(a_1, \text{merge}(A', B)))$$

also `sorted?(C)` folgt. Ebenso ergibt sich aus der Induktionsvoraussetzung

$$\text{count}(s, A') + \text{count}(s, B) = \text{count}(s, \text{merge}(A', B))$$

für alle  $s \in S$ , daß

$$\text{count}(a_1, \text{merge}(A, B)) = \text{count}(a_1, \text{cons}(a_1, \text{merge}(A', B))) .$$

Hieraus ergibt sich Nachbedingung 1, da die  $s \neq a_1$  nicht betroffen sind.

2.  $a_1 > b_1$ : Dieser Fall läßt sich auf gleiche Art behandeln.

□

Wie man sieht, ist die Korrektheit für die funktionale, strukturell-rekursive Definition leicht einzusehen.

## 4.5 Bäume

Wir haben Bäume schon kurz im Zusammenhang mit Termdarstellungen kennengelernt. Sie stellen oft die übersichtlichste Art dar, komplexe Strukturen zu präsentieren. In der Tat sind für manche Leute die Begriffe „Term“ und „Baum“ geradezu synonym. Terme sind in gewissem Sinne jedoch „ordentlicher“ aufgebaut als ganz allgemeine Bäume: Bei den Termen ist mit jedem Symbol eine feste Zahl von möglichen Teiltermen verbunden, während bei den Bäumen im allgemeinen an jedem Symbol eine beliebige Zahl von Teilbäumen möglich ist.

Für die maschinelle Behandlung bieten Bäume gegenüber einer textlichen Darstellung als Zeichenkette unübersehbare Vorteile, da es leichter ist, Teilstrukturen zu erkennen und getrennt zu bearbeiten. Bevor wir Binärbäume betrachten, wollen wir zunächst Bäume im allgemeinen definieren

**Definition 4.10 (Bäume, [?])** Sei  $T$  eine Menge („Typ“); dann ist die Menge  $B(T)$  der *Bäume über  $T$*  gegeben durch

1.  $\text{emptytree} \in B(T)$  (der *leere Baum*)
2. Ist  $I$  eine endliche Menge („Indexmenge“) und  $\alpha : I \rightarrow B(T)$  eine Abbildung, so ist  $t = \langle w, \alpha \rangle \in B(T)$ , wobei  $w \in T$  *Wurzel* von  $t$  genannt wird. Die  $t_i \in \alpha(I)$  heißen (direkte) *Nachfolger von  $w$*  oder (direkte) *Teilbäume von  $t$* .

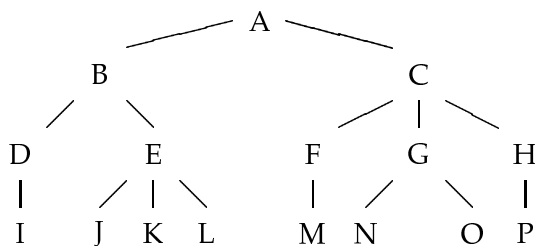
Die Gesamtheit der nach Bedingung 2 in  $t$  vorkommenden Elemente  $w \in T$  nennen wir die *Knoten* des Baums. Ein Baum heißt *geordneter Baum*, wenn für alle Knoten die Indexmenge  $I$  geordnet ist (z.B.  $I = \{1, \dots, n\}$ ). Knoten, die keine Nachfolger haben (leere Indexmenge), heißen *Blätter*. Knoten, die keine Blätter sind, heißen *innere Knoten*.

Wenn wir von Bäumen sprechen, so meinen wir in aller Regel geordnete Bäume. Bezüglich der Knoten eines Baumes ist auch die folgende Terminologie üblich: Die Nachfolger eines Knotens heißen auch *Söhne*; dementsprechend heißt der Knoten selbst *Vater*. Knoten, die Nachfolger desselben anderen Knotens sind, heißen sinngemäß *Brüder*. Bei geordneten Bäumen ergibt es dann auch einen Sinn, vom *ersten* bzw. *letzten* Sohn zu sprechen oder vom *nächsten* bzw. *vorangehenden* Bruder.

Für Bäume sind verschiedene Notationen üblich; einige davon stellen wir hier vor:

#### Beispiel 4.11 (Darstellungen von Bäumen)

##### 1. Graphen:



##### 2. Klammerstrukturen: Wir haben bereits auf dem Zusammenhang zwischen Termen und Bäumen hingewiesen. Der oben vorgestellte Beispielbaum läßt sich durch folgenden Term charakterisieren:

$$A(B(D(I), E(J, K, L)), C(F(M), G(N, O), H(P)))$$

Dabei wurde die Wurzel jeweils wie ein Operationssymbol vor die Klammern mit den direkten Nachfolgern geschrieben. Wie in der Alltagsmathematik haben wir Kommas zur Trennung der Argumente verwendet. Bei Bäumen kann man in diesem Zusammenhang auf die Klammern nicht verzichten, da die Knoten keine feste „Stelligkeit“

wie die Operationssymbole in einer Algebra haben. Von *Scheme* her kennen wir schon eine andere Notation für Bäume als Klammerstrukturen; dabei wird jeweils ein Knoten zusammen mit seinen direkten Nachfolgern in Klammern eingeschlossen, wozu man dann keine Kommas braucht:

$$(A (B (D I)(E J K L)) (C (F M) (G N O) (H P)))$$

### 3. Einrückungsstrukturen:

```

A
  B
    D
      I
    E
      J
      K
      L
  C
    F
      M
    G
      N
      O
  H
    P

```

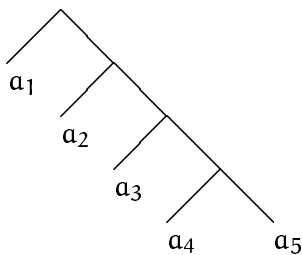
Man mache sich klar, daß der Informationsgehalt bei allen diesen unterschiedlichen Darstellungen der gleiche ist.

**Definition 4.12** Die Zahl der direkten Nachfolger eines Knotens in einem Baum heißt *Grad* des Knotens. Der maximale Grad der Knoten eines Baums heißt Grad des Baums.

Von besonderer Bedeutung sind die sogenannten Binärbäume, die man im wesentlichen als geordnete Bäume vom Grad 2 betrachten kann. Wir wollen 4.15 als Definition von Binärbäumen betrachten.

**Beispiel 4.13 (Rechenbäume)** Eine verbreitete Anwendung von Binärbäumen ist die Darstellung arithmetischer oder logischer Terme, wie sie in Beispiel 2.33 bereits vorgeführt wurde. Solche Bäume nennt man auch Rechenbäume, weil sich an Ihnen der Rechenprozeß zur Auswertung des Terms sehr gut einsehen läßt: An den Blättern des Rechenbaums beginnt die Rechnung mit den dort vorhandenen Konstanten oder mit den Werten der dort vorhandenen Variablen. Man läuft dann hoch im Baum, wobei die Operationen in den Knoten auf die zuvor ermittelten Werte der unmittelbaren Teilbäume angewendet werden und dann diese Knoten durch Blätter mit den ermittelten Werten ersetzt werden. Man kann dabei auch sehr schön erkennen, welche Rechnungen unabhängig voneinander („parallel“) erfolgen können, weil sie im Rechenbaum nicht in einer Vorgänger-Nachfolger-Beziehung stehen.

**Beispiel 4.14 (Entartete Bäume)** Auch die zuvor betrachteten Elemente des Datentyps  $\text{List}(A)$  lassen sich als Binärbäume betrachten, indem z.B. die Liste  $(a_1, a_2, a_3, a_4, a_5)$  dargestellt wird durch den Baum



Solch einen entarteten Baum nennt man auch einen *Kamm*, der natürlich auch in der anderen Richtung verlaufen kann. Noch extremer wird die Entartung, wenn wir die Elemente  $a_1, \dots, a_5$  perlenschnurartig ohne Verzweigungen aufreihen. Die dann entstehende Struktur ist zwar formal ein Baum vom Grad 1, wird jedoch auch noch als (entarteter) Binärbaum angesehen.

Wir präsentieren jetzt einen abstrakten Datentyp „Binärbaum“ mit minimalen Fähigkeiten:

**Definition 4.15 (Binärbäume)** Binäre Bäume über  $S$  sind durch folgenden abstrakten Datentyp gegeben:

```
datatype Tree(S);
```

```
sorts Tree, S;
```

```
constructors
```

```
emptytree:  $\rightarrow$  Tree;
```

```
maketree: Tree  $\times$  S  $\times$  Tree  $\rightarrow$  Tree;
```

```
operations
```

```
left: Tree  $\rightarrow$  Tree;
```

```
right: Tree  $\rightarrow$  Tree;
```

```
elem: Tree  $\rightarrow$  S;
```

```
depth: Tree  $\rightarrow$   $\mathbb{N}$ ;
```

```
nodecount: Tree  $\rightarrow$   $\mathbb{N}$ ;
```

```
equations
```

```
left(maketree(L,s,R)) = L;
```

```
right(maketree(L,s,R)) = R;
```

```
elem(maketree(L,s,R)) = s
```

```
depth(emptytree) = 0;
```

```
depth(maketree(L,s,R)) = 1 + max(depth(L),depth(R))
```

```
nodecount(emptytree) = 0;
```

```
nodecount(maketree(L,s,R)) = 1 + nodecount(L)  
+ nodecount(R);
```

```
end
```

Mit dieser Definition haben wir uns von der Notwendigkeit befreit, Blätter als eigenes Konzept in die Konstruktion der Binärbäume einzubringen: Blätter sind einfach Knoten der Form

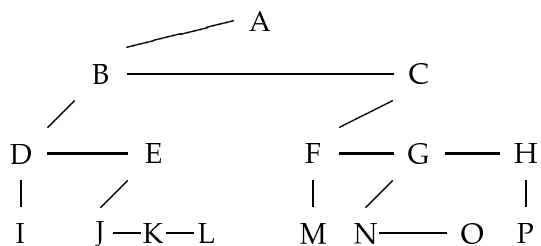
$$\text{maketree}(\text{emptytree}, s, \text{emptytree}).$$

Es ist klar, daß die Komplexität von Algorithmen, die auf Binärbäumen arbeiten, von der Baumtiefe `depth` und der Knotenanzahl `nodecount` als Aufwandsparemtern abhängen. Dabei wird der Aufwand für Algorithmen, die im wesentlichen nur einen Pfad (d.h. eine durchgehende Folge von Nachfolgern) von der Wurzel zu einem Blatt besuchen, durch die Baum-

tiefe bestimmt sein, während die Knotenanzahl bestimmend wirkt, wenn fast alle Pfade besucht werden müssen (vgl. Bsp. 4.19).

Eine interessante und für *Scheme*-Implementierung genutzte Tatsache ist, daß wir jeden beliebigen Baum durch einen Binärbaum repräsentieren können. Im Grunde ist dies auch nicht verwunderlich, da Bäume letzten Endes zweidimensionale Strukturen sind. Eine Möglichkeit der Darstellung eines beliebigen Baums durch einen Binärbaum besteht darin, im linken Teilbaum eines jeden Knotens den ersten (am weitesten links stehenden) Sohn zu vermerken und im rechten Teilbaum den jeweils nächsten Bruder. Das folgende Beispiel zeigt den Baum aus Beispiel 4.11 nach dieser Umformung.

**Beispiel 4.16** Um eine möglichst große Ähnlichkeit zu Beispiel 4.11 zu erzielen, haben wir den linken Teilbaum jeweils durch eine schräge und den rechten Teilbaum durch eine gerade Linie vermerkt:



Die Ökonomie der Datendarstellung (nur zweistellige Knoten) muß allerdings durch eine vergrößerte Komplexität von Algorithmen erkauft werden, die auf Bäumen operieren. Im ursprünglichen Baum ist der Zugriff von jedem Knoten aus zu jedem Sohn direkt möglich; in dem transformierten Binärbaum muß man, um zum  $i$ -ten Sohn zu gelangen, zunächst zum linken Teilbaum übergehen und dann  $i - 1$  mal zum rechten Teilbaum. So kommen wir etwa in dem Baum aus 4.11 von der Wurzel A zum Blatt O in drei Schritten (C, G, O), während wir in dem oben aufgeführten Binärbaum 6 Schritte brauchen (B, C, F, G, N, O).

Die hier vorgeführte Umformung läßt sich wie folgt formal beschreiben:

**Definition 4.17 (Umformung geordneter Bäume in Binärbäume)** Sei  $S$  eine Menge,  $\mathcal{T}(S)$  die Menge der endlichen Folgen geordneter Bäume über

S. Wir schreiben einen nicht-leeren geordneten Baum als  $\langle w, T \rangle$  mit  $T \in \mathcal{T}(S)$  und verwenden für Folgen den abstrakten Datentyp aus 4.4:

Die Umformung ist dann beschrieben durch eine Abbildung

$$B : \mathcal{T}(S) \longrightarrow \text{Tree}(S)$$

mit

$$B(\text{empty}) \stackrel{\text{def}}{=} \text{emptytree}$$

$$B(\text{cons}(t_1, r)) \stackrel{\text{def}}{=} \begin{cases} B(r) & \text{falls } t_1 = \text{emptytree} \\ \text{maketree}(B(T), w, B(r)) & \text{falls } t_1 = \langle w, T \rangle \end{cases}$$

Im ersten Schritt der Umformung wird dann ein Baum als eine einelementige Folge von Bäumen betrachtet.

Binärbäume werden häufig für Sortier- und Suchaufgaben eingesetzt. Wir erweitern daher die Datentyp-Definition wie folgt:

**Definition 4.18 (Suchbäume)** Ist  $S$  eine geordnete Menge, so ist ein *Suchbaum* über  $S$  ein Binärbaum, so daß in jedem Knoten die Elemente im linken Teilbaum kleiner und die im rechten Teilbaum größer als das Knotenelement sind. In der folgenden Datentypdefinition garantiert eine Operation `insert` das Einfügen eines Elements in den richtigen Teilbaum.

```
datatype Searchtree(S);
```

```
uses Tree(S);
```

```
assumes <: S × S → Boolean;
```

(\* < soll die Gesetze einer Ordnungsrelation erfüllen \*)

```
operations
```

```
  search: S × Tree → Boolean;
```

```
  insert: S × Tree → Tree;
```

```
equations
```

```
  search(s, emptytree) = False;
```

```
  search(s, maketree(L, t, R)) = if s=t then True
```

```
                                elseif s<t then search(s, L)
```

```

else search(s,R)
endif;

insert(s,emptytree) = maketree(emptytree,s,emptytree);
insert(s,maketree(L,t,R)) =
  if s=t then maketree(L,s,R)
  elsif s<t then maketree(insert(s,L),t,R)
  else maketree(L,t,insert(s,R))
  endif

```

Die Operation `search` ist offensichtlich der binären Suche nachgebildet, die in der Tat eigentlich eine Suche in Binärbäumen ist. `insert` stellt aus einer unsortierten Eingabefolge einen Suchbaum her. Offensichtlich ist die Anzahl der rekursiven Aufrufe von `search` durch die Tiefe des Suchbaums beschränkt. Diese kann jedoch stark schwanken, je nachdem, wie der Suchbaum entstanden ist:

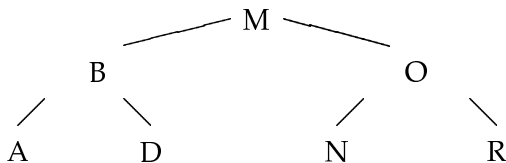
**Beispiel 4.19** Wir betrachten Suchbäume über Buchstaben, wobei die Relationen  $<$ ,  $>$  durch die alphabetische Reihenfolge gegeben sind. Die Operationsfolge

```

b := emptytree;
b := insert(M,b);
b := insert(B,b);
b := insert(O,b);
b := insert(A,b);
b := insert(D,b);
b := insert(N,b);
b := insert(R,b)

```

erzeugt den Suchbaum



während ein Einfügen derselben Elemente in alphabetischer Reihenfolge einen entarteten Baum erzeugt, bei dem alle linken Teilbäume leer sind.

In einem solchen Baum verkommt die binäre Suche zu einer sequentiellen Suche.

Die Komplexität von `search` kann daher im ungünstigsten Fall  $O(n)$  sein, wobei  $n$  die Anzahl der Knoten ist. Im günstigsten Fall ist die Komplexität  $O(\log_2(n))$ , denn ein Baum der Tiefe  $n$  hat  $2^n - 1$  Knoten. Um eine minimale Suchzeit zu garantieren, muß der Baum allerdings *vollständig ausgeglichen* sein, das heißt, die Elemente müssen möglichst gleichmäßig auf linke und rechte Teilbäume verteilt sein. Eine einfache Datenstruktur wie die hier vorgestellte und die einfache `insert`-Operation sind dazu nicht ausreichend. Dies soll hier jedoch nicht betrachtet werden. Hier ging es nur darum, einige Grundlagen der Spezifikation und Verwendung von abstrakten Datentypen vorzustellen.

## 4.6 Aufgaben

**Aufgabe 4.1** Formulieren Sie den Algorithmus „`Mische(A, B)`“ aus 4.8 als endrekursive Prozedur.

## Literaturverzeichnis

- [Bau68] BAUMANN, RICHARD: *Algol-Manual der Alcor-Gruppe*. R. Oldenbourg, München/Wien, 3. Auflage, 1968.
- [Bau89] BAUER, F. L.: *100 Jahre Peano-Zahlen*. Informatik-Spektrum, 12:340–341, 1989.
- [End72] ENDERTON, H. B.: *A Mathematical Introduction to Logic*. Academic Press, 1972.
- [GKP89] GRAHAM, R. L., D. E. KNUTH und O. PATASHNIK: *Concrete Mathematics*. Addison-Wesley, 1989.
- [Hal69] HALMOS, P: *Naive Mengenlehre*. Vandenhoeck & Ruprecht, Göttingen, 1969.
- [Kla83] KLAEREN, HERBERT: *Algebraische Spezifikation — Eine Einführung*. Springer Verlag, Berlin-Heidelberg-New York, 1983.
- [Knu73] KNUTH, D. E.: *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973. Sorting and Searching.
- [Mes71] MESCHKOWSKI, H.: *Einführung in die moderne Mathematik*, Band 75/75a der Reihe *Hochschultaschenbücher*. BI, 1971.
- [MHR80] METROPOLIS, N., J. HOWLETT, and GIAN-CARLO ROTA (editors): *A History of Computing in the Twentieth Century*. Academic Press, 1980.
- [Wex81] WEXELBLAT, RICHARD L. (editor): *History of Programming Languages*, New York, 1981. Academic Press.
- [Wir85] WIRTH, NIKLAUS: *Programming in Modula-2*. Springer, 3rd edition, 1985.