Prof. Dr. Ralf Hartmut Güting, Dr. Stefan Dieker

Kurs 01662

Datenstrukturen II

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Vorwort

Liebe Fernstudentin, lieber Fernstudent,

wir freuen uns, daß Sie am Kurs 1663 “Datenstrukturen” bzw. 1661 “Datenstrukturen I” oder 1662 “Datenstrukturen II” teilnehmen und wünschen Ihnen viel Spaß und Erfolg beim Durcharbeiten des Kursmaterials.

Thema und Schwerpunkte

Effiziente Algorithmen und Datenstrukturen bilden ein zentrales Thema der Informatik. Wer programmiert, sollte zu den wichtigsten Problembereichen grundlegende Lösungsverfahren kennen; er sollte auch in der Lage sein, neue Algorithmen zu entwerfen, ggf. als Kombination bekannter Verfahren, und ihre Kosten in Bezug auf Laufzeit und Speicherplatz zu analysieren. Datenstrukturen organisieren Information so, daß effiziente Algorithmen möglich werden. Dieser Kurs möchte entsprechende Kenntnisse und Fähigkeiten vermitteln.

Im Vergleich zu anderen Darstellungen zu Algorithmen und Datenstrukturen setzt dieser Kurs folgende Akzente:

- Es wurde versucht, in relativ kompakter Form alle wichtigen Themenbereiche des Gebietes abzudecken. Die meisten Bücher sind wesentlich umfangreicher oder behandeln wichtige Themenbereiche (Graphalgorithmen, geometrische Algorithmen) nicht.
- Die kompakte Darstellung wurde zum Teil erreicht durch Konzentration auf die Darstellung der wesentlichen Ideen. In diesem Kurs wird die Darstellung von Algorithmen mit dem richtigen Abstraktionsgrad besonders betont. Die Idee eines Algorithmus kann auch in seitenlangen Programmen versteckt sein; dies sollte vermieden werden. Selbstverständlich geht die Beschreibung von Algorithmen immer Hand in Hand mit ihrer Analyse.
- *Datentyp* als Spezifikation und Anwendungssicht einer Datenstruktur und *Datenstruktur* als Implementierung eines Datentyps werden klar voneinander unterschieden. Es wird gezeigt, daß es zu einem Datentyp durchaus verschiedene Implementierungen geben kann. Die Spezifikation von Datentypen mit einer recht praxisnahen Methode wird an etlichen Beispielen vorgeführt.
- Der Kurs setzt einen deutlichen Schwerpunkt im Bereich der algorithmischen Geometrie.

Das Kapitel zur algorithmischen Geometrie ist zweifellos etwas anspruchsvoller als der übrige Text. Es wird von Studenten gelegentlich als etwas schwierig, oft aber auch als

hochinteressant empfunden. Wir finden, daß die Beschäftigung mit diesem Kapitel sich aus mehreren Gründen besonders lohnt:

- Der Blick wird geweitet; man erkennt z.B., daß "schlichtes" Suchen und Sortieren nur der eindimensionale Spezialfall eines allgemeineren Problems ist, oder daß Bäume auch ganz anders konstruiert werden können als einfache binäre Suchbäume, daß man sie schachteln kann usw.
- Der Umgang mit *algorithmischen Paradigmen* wird anhand von *Plane-Sweep* und *Divide-and-Conquer* eingeübt; man sieht, daß man mit verschiedenen Techniken zu optimalen Lösungen für das gleiche Problem kommen kann.
- Der Entwurf von Algorithmen auf hohem Abstraktionsniveau zusammen mit systematischer Problemreduktion wird eingeübt.
- Schließlich begreift man, daß all die Algorithmen und Datenstrukturen der vorhergehenden Kapitel als *Werkzeuge* zur Konstruktion neuer Algorithmen eingesetzt werden können.

Aufbau des Kurses

Der Kurs ist modular aufgebaut, um den Einsatz in verschiedenen Studiengängen zu ermöglichen. Kurs 1663 "Datenstrukturen" hat 7 Kurseinheiten. Kurs 1661 "Datenstrukturen I" besteht aus den ersten 4 Kurseinheiten von 1663 (reduziert um einige Abschnitte in Kurseinheit 3) sowie einer eigenen Kurseinheit 5, die noch zwei wichtige Abschnitte aus dem Rest des Kurses 1663 enthält. Kurs 1662 "Datenstrukturen II" besteht aus den letzten 3 Kurseinheiten des Kurses 1663. Kurs 1661 wird im Bachelor-Studiengang und in der Lehrerausbildung eingesetzt. Kurs 1662 setzt den ersten Kurs 1661 voraus und dient z.B. dem Übergang vom Bachelor zum Diplomstudiengang. Er kann auch als Teil eines Moduls in den Masterstudiengängen verwendet werden.

Im Anhang der ersten Kurseinheit gibt es ein kurzes Kapitel "Mathematische Grundlagen", in dem die benötigten mathematischen Grundkenntnisse "importiert" werden. Im Text finden sich gelegentlich Verweise auf einen Abschnitt der mathematischen Grundlagen; Sie sollten dann vor dem Weiterlesen zunächst in Ruhe diesen Abschnitt durcharbeiten.

Die aktuelle Fassung des Kurses basiert auf dem im Teubner-Verlag erschienenen Buch:

> Ralf Hartmut Güting und Stefan Dieker
> Datenstrukturen und Algorithmen
> 3. Auflage, Teubner-Verlag, Stuttgart 2004
> Reihe Leitfäden der Informatik
> ISBN 3-519-22121-7

Voraussetzungen

Kursteilnehmer sollten grundlegende Kenntnisse der Programmierung besitzen (wie sie etwa in den Kursen 1612 “Konzepte imperativer Programmierung” oder 1613 “Einführung in die imperative Programmierung” vermittelt werden) und dementsprechend eine Programmiersprache wie z.B. PASCAL, C oder Java beherrschen. In der aktuellen Fassung des Kurses sind konkrete Programme in Java formuliert. Es werden aber nur Grundkenntnisse in Java benötigt, die in den meisten Studiengängen bzw. Studienplänen parallel zur Bearbeitung dieses Kurses erworben werden (etwa anhand der Kurse 1616 “Einführung in die objektorientierte Programmierung I” oder 1618 “Einführung in die objektorientierte Programmierung”) und die man sich andernfalls anhand eines Java-Buches selbst aneignen kann. Meist werden Algorithmen allerdings auf einer höheren Abstraktionsebene als der programmiersprachlichen formuliert. Programme, die als Lösung von Aufgaben zu erstellen sind, sind in Java zu schreiben.

Für die Analyse von Algorithmen sind Grundkenntnisse der Wahrscheinlichkeitsrechnung vorteilhaft; im wesentlichen werden die benötigten Kenntnisse allerdings auch im Kurs vermittelt. In diesem Zusammenhang können wir sehr das im Literaturverzeichnis erwähnte Buch von Graham, Knuth und Patashnik empfehlen. Das ist ein ganz ausgezeichnetes Buch über mathematische Grundlagen der Informatik, die bei der Analyse von Algorithmen eine Rolle spielen.

Selbsttestaufgaben

In den Text eingestreut sind zum Selbsttest gedachte Aufgaben, deren Lösungen im Anhang zu finden sind. Wie Sie es von anderen Kursen der FernUniversität bereits gewohnt sind, sollten Sie Selbsttestaufgaben unbedingt bearbeiten, also nicht einfach die Lösung nachsehen. Für das Verständnis des Stoffes ist der eigene kreative Umgang mit den gestellten Problemen von entscheidender Bedeutung. Durch bloßes Lesen werden Sie den Stoff nicht wirklich beherrschen. Selbstverständlich sollten Sie auch am Übungsbetrieb teilnehmen, d.h. die Einsendeaufgaben bearbeiten.

Weitere Aufgaben

Am Ende fast jeden Kapitels finden Sie weitere Aufgaben. Diese Aufgaben wurden einmal für die oben erwähnte Buchversion gesammelt – in Lehrbüchern ist es üblich, den Dozenten auch Aufgabenkataloge anzubieten. Sie wurden in den Kurs aufgenommen, um Ihnen, falls Sie übermäßigen Lerneifer an den Tag legen, weiteres Übungsmaterial zur Verfügung zu stellen. Aus Sicht des Kurses sind sie aber reiner Luxus; Sie können

bedenkenlos diese Aufgaben völlig ignorieren, und wir könnten sie auch weglassen. In jedem Fall wollen wir keine Klagen darüber hören, daß zu diesen Aufgaben keine Lösungen angeboten werden! Schließlich gibt es schon genügend viele Selbsttestaufgaben.

Literatur zum Kurs

Hinweise auf relevante Literatur finden sich am Ende jedes Kapitels. Insbesondere werden andere gute Bücher zu Datenstrukturen in Abschnitt 1.5 genannt.

Digitale Fassung

Eine digitale Fassung dieses Kurses wird im Internet, d.h. in der virtuellen Universität, angeboten. Die digitale Fassung besteht aus

- dem Kurstext, in Form von PDF-Dateien (Portable Document Format, lesbar mit Acrobat Reader, bzw. entsprechenden Browser Plug-Ins, auf allen Plattformen).
- Aufgabenblättern zu den einzelnen Kurseinheiten sowie – zu gegebener Zeit – Lösungen dazu.
- Animationen und teilweise Experimentierumgebungen in Form von Java-Applets zu ausgewählten Algorithmen und Datenstrukturen des Kurses.

Die digitale Fassung bietet Ihnen über den Papierkurs hinaus folgenden Nutzen:

- Querverweise im Kurstext sind aktive Links, ebenso Inhaltsverzeichnisse und Indexe. Sie können Acrobat Reader auch im Text nach Begriffen suchen lassen. Der Kurstext ist weiterhin ein bißchen mit Farbe “aufgepeppt”.
- Die Beschäftigung mit den Animationen sollte die Funktion der Algorithmen oder Datenstrukturen leichter verständlich machen.

Über die Kursautoren

Prof. Dr. Ralf Hartmut Güting, geb. 1955. Studium der Informatik an der Universität Dortmund. 1980 Diplom. 1981/82 einjähriger Aufenthalt an der McMaster University, Hamilton, Kanada, Forschung über algorithmische Geometrie. 1983 Promotion über algorithmische Geometrie an der Universität Dortmund. 1985 einjähriger Aufenthalt am IBM Almaden Research Center, San Jose, USA, Forschung im Bereich Büroinformationssysteme, Nicht-Standard-Datenbanken. Ab 1984 Hochschulassistent, ab 1987 Professor an der Universität Dortmund. Seit November 1989 Professor für Praktische Informatik an der FernUniversität. Hauptarbeitsgebiete: Geo-Datenbanksysteme, Archi-

tektur von Datenbanksystemen (insbesondere Erweiterbarkeit und Modularität), raumzeitliche Datenbanken und Behandlung von Graphen (etwa Verkehrsnetzen) in Datenbanken.

Dr. Stefan Dieker, geb. 1968. Studium der Angewandten Informatik mit den Nebenfächern Elektrotechnik und Betriebswitschaftslehre (1989-1996). Softwareentwickler für betriebswirtschaftliche Standardsoftware (1996). Wissenschaftlicher Mitarbeiter an der Fernuniversität Hagen (1996-2001). Forschungsgebiet: Architektur und Implementierung erweiterbarer Datenbanksysteme. Promotion 2001 bei Ralf Hartmut Güting. Seit 2001 Anwendungsentwickler in der Industrie.

Weitere Informationen zu Autoren und Kursbetreuern finden Sie auch auf den Webseiten des Lehrgebiets “Datenbanksysteme für neue Anwendungen”:

http://dna.fernuni-hagen.de/

Gliederung in Kurse und Kurseinheiten

Datenstrukturen I / Datenstrukturen Kurseinheit 1 1 Einführung 2 Programmiersprachliche Konzepte für Datenstrukturen	
Kurseinheit 2 3 Grundlegende Datentypen	
Kurseinheit 3 4 Datentypen zur Darstellung von Mengen (*)	
Kurseinheit 4 5 Sortieralgorithmen 6 Graphen	
Datenstrukturen I Kurseinheit 5 7 Weitere Themen 7.1 Bestimmung kürzester Wege 7.2 Externes Suchen: B-Bäume	**Datenstrukturen II / Datenstrukturen** Kurseinheit 5 7 Graph-Algorithmen
	Kurseinheit 6 8 Geometrische Algorithmen
	Kurseinheit 7 8 Geometrische Algorithmen 9 Externes Suchen und Sortieren

(*) Kapitel 4 ist für Kurs Datenstrukturen I um einige Abschnitte reduziert.

Inhalt des Kurses (vorläufig[1])

1 Einführung

1.1 Algorithmen und ihre Analyse
1.2 Datenstrukturen, Algebren, Abstrakte Datentypen
1.3 Grundbegriffe

2 Programmiersprachliche Konzepte für Datenstrukturen

2.1 Datentypen in Java
2.2 Dynamische Datenstrukturen
2.3 Weitere Konzepte zur Konstruktion von Datentypen

3 Grundlegende Datentypen

3.1 Sequenzen (Folgen, Listen)
3.2 Stacks
3.3 Queues
3.4 Abbildungen
3.5 Binäre Bäume
3.6 (Allgemeine) Bäume

4 Datentypen zur Darstellung von Mengen

4.1 Mengen mit Durchschnitt, Vereinigung, Differenz
4.2 Dictionaries: Mengen mit INSERT, DELETE, MEMBER
4.2.1 Einfache Implementierungen
4.2.2 Hashing
4.2.3 Binäre Suchbäume
4.2.4 AVL-Bäume
4.3 Priority Queues: Mengen mit INSERT, DELETEMIN
4.4 Partitionen von Mengen mit MERGE, FIND

5 Sortieralgorithmen

5.1 Einfache Sortierverfahren: Direktes Auswählen und Einfügen
5.2 Divide-and-Conquer-Methoden: Mergesort und Quicksort
5.3 Verfeinertes Auswählen und Einfügen: Heapsort und Baumsortieren
5.4 Untere Schranke für allgemeine Sortierverfahren
5.5 Sortieren durch Fachverteilen: Bucketsort und Radixsort

1. Änderungen am Kurstext sind prinzipiell noch möglich. Mit der letzten Kurseinheit erhalten Sie ein aktualisiertes Inhaltsverzeichnis.

6 Graphen

6.1 Gerichtete Graphen
6.2 (Speicher-) Darstellungen von Graphen
6.3 Graphdurchlauf

7 Graph-Algorithmen

7.1 Bestimmung kürzester Wege von einem Knoten zu allen anderen
7.2 Bestimmung kürzester Wege zwischen allen Knoten im Graphen
7.3 Transitive Hülle
7.4 Starke Komponenten
7.5 Ungerichtete Graphen
7.6 Minimaler Spannbaum (Algorithmus von Kruskal)

8 Geometrische Algorithmen

8.1 Plane-Sweep-Algorithmen für orthogonale Objekte in der Ebene
8.1.1 Das Segmentschnitt-Problem
8.1.2 Das Rechteckschnitt-Problem
8.1.3 Das Maßproblem
8.2 Divide-and-Conquer-Algorithmen für orthogonale Objekte
8.2.1 Das Segmentschnitt-Problem
8.2.2 Das Maßproblem
8.2.3 Das Konturproblem
8.3 Suchen auf Mengen orthogonaler Objekte
8.4 Plane-Sweep-Algorithmen für beliebig orientierte Objekte

9 Externes Suchen und Sortieren

9.1 Externes Suchen: B-Bäume
9.2 Externes Sortieren

Inhalt der Kurseinheit 1

1 Einführung **1**

1.1 Algorithmen und ihre Analyse 2
1.2 Datenstrukturen, Algebren, Abstrakte Datentypen 22
1.3 Grundbegriffe 33
1.4 Weitere Aufgaben 36
1.5 Literaturhinweise 37

2 Programmiersprachliche Konzepte für Datenstrukturen **39**

2.1 Datentypen in Java 40
2.1.1 Basisdatentypen 41
2.1.2 Arrays 42
2.1.3 Klassen 45
2.2 Dynamische Datenstrukturen 49
2.2.1 Programmiersprachenunabhängig: Zeigertypen 49
2.2.2 Zeiger in Java: Referenztypen 53
2.3 Weitere Konzepte zur Konstruktion von Datentypen 57
2.4 Literaturhinweise 61

Lösungen zu den Selbsttestaufgaben **A-1**

Literatur **A-7**

Index **A-9**

Mathematische Grundlagen **A-13**

Lehrziele

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- wissen, was die Begriffe *abstrakter Datentyp*, *Datentyp, Algebra*, *Typ*, *Funktion*, *Prozedur*, *Modul* , *Klasse* bedeuten;
- verschiedene Abstraktionsebenen bei der Spezifikation benennen können;
- Kriterien für die Qualität eines Algorithmus kennen;
- die Abstraktionsschritte zur Analyse eines Algorithmus erklären können;
- mit der O-Notation umgehen können;
- erklären können, wie sich Spezifikation und Implementierung einer Algebra in verschiedenen Programmiersprachen darstellen lassen;
- verstehen, was der Unterschied zwischen einer allgemeinen Datenstruktur und einem Datentyp in einer Programmiersprache ist;
- die wesentlichen Konzepte zur Konstruktion von Datenstrukturen in höheren Programmiersprachen kennen;
- beschreiben können, welche dieser Konzepte sich in Java wiederfinden;
- eine Vorstellung von der Repräsentation solcher Datentypen im Speicher haben;
- die verschiedenen Datentypen sinnvoll einsetzen können;
- den grundlegenden Unterschied zwischen statischen und dynamischen Datenstrukturen verstanden haben.

1 Einführung

Algorithmen und Datenstrukturen sind Thema dieses Kurses. Algorithmen arbeiten auf Datenstrukturen und Datenstrukturen enthalten Algorithmen als Komponenten; insofern sind beide untrennbar miteinander verknüpft. In der Einleitung wollen wir diese Begriffe etwas beleuchten und sie einordnen in eine "Umgebung" eng damit zusammenhängender Konzepte wie *Funktion, Prozedur, Abstrakter Datentyp, Datentyp, Algebra, Typ* (in einer Programmiersprache), *Klasse* und *Modul*.

Wie für viele fundamentale Begriffe der Informatik gibt es auch für diese beiden, also für Algorithmen und Datenstrukturen, nicht eine einzige, scharfe, allgemein akzeptierte Definition. Vielmehr werden sie in der Praxis in allerlei Bedeutungsschattierungen verwendet; wenn man Lehrbücher ansieht, findet man durchaus unterschiedliche "Definitionen". Das Diagramm in Abbildung 1.1 und spätere Bemerkungen dazu geben also die persönliche Sicht der Autoren wieder.

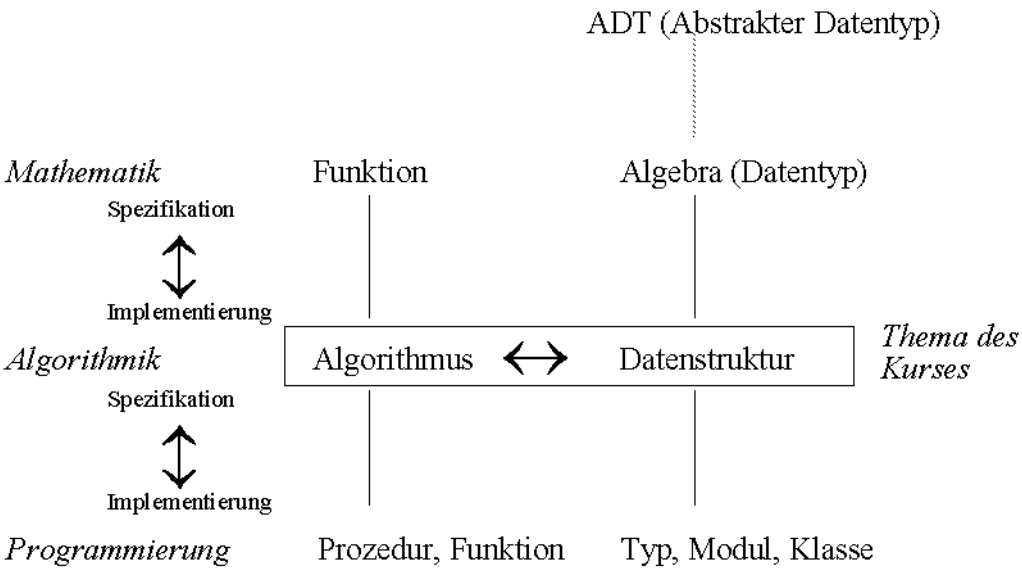

Abbildung 1.1: Abstraktionsebenen von Algorithmen und Datenstrukturen

Das Diagramm läßt sich zunächst zerlegen in einen linken und einen rechten Teil; der linke Teil hat mit Algorithmen, der rechte mit Datenstrukturen zu tun. Weiterhin gibt es drei *Abstraktionsebenen*. Die abstrakteste Ebene ist die der Mathematik bzw. der formalen Spezifikation von Algorithmen oder Datenstrukturen. Ein Algorithmus realisiert eine *Funktion*, die entsprechend eine Spezifikation eines Algorithmus darstellt. Ein

Algorithmus stellt seinerseits eine Spezifikation einer zu realisierenden *Prozedur* (oder Funktion oder Methode im Sinne einer Programmiersprache) dar. Gewöhnlich werden Algorithmen, sofern sie einigermaßen komplex und damit interessant sind, *nicht* als Programm in einer Programmiersprache angegeben, sondern auf einer höheren Ebene, die der Kommunikation zwischen Menschen angemessen ist. Eine Ausformulierung als Programm ist natürlich *eine* Möglichkeit, einen Algorithmus zu beschreiben. Mit anderen Worten, ein Programm stellt einen Algorithmus dar, eine Beschreibung eines Algorithmus ist aber gewöhnlich kein Programm. In diesem einführenden Kapitel ist das zentrale Thema im Zusammenhang mit Algorithmen ihre Analyse, die Ermittlung von Laufzeit und Speicherplatzbedarf.

Auf der Seite der Datenstrukturen finden sich auf der Ebene der Spezifikation die Begriffe des *abstrakten Datentyps* und der *Algebra*, die einen "konkreten" Datentyp darstellt. Für uns ist eine *Datenstruktur* eine *Implementierung einer Algebra oder eines ADT auf algorithmischer Ebene*. Eine Datenstruktur kann selbst wieder in einer Programmiersprache implementiert werden; auf der programmiersprachlichen Ebene sind eng verwandte Begriffe die des (Daten-) *Typs*, der *Klasse* oder des *Moduls*.

In den folgenden Abschnitten der Einleitung werden wir auf dem obigen Diagramm ein wenig "umherwandern", um die Begriffe näher zu erklären und an Beispielen zu illustrieren. Abschnitt 1.1 behandelt den linken Teil des Diagramms, also Algorithmen und ihre Analyse. Abschnitt 1.2 ist dem rechten Teil, also Datenstrukturen und ihrer Spezifikation und Implementierung gewidmet. Abschnitt 1.3 faßt die Grundbegriffe zusammen und definiert sie zum Teil präziser.

Noch ein kleiner Hinweis vorweg: In diesem einleitenden Kapitel wollen wir einen Überblick geben und dabei die wesentlichen schon erwähnten Begriffe klären und die allgemeine Vorgehensweise bei der Analyse von Algorithmen durchsprechen. Verzweifeln Sie nicht, wenn Ihnen in diesem Kapitel noch nicht alles restlos klar wird, insbesondere für die Analyse von Algorithmen. Der ganze Rest des Kurses wird dieses Thema vertiefen und die Analyse einüben; es genügt, wenn Sie am Ende des Kurses die Methodik beherrschen.

1.1 Algorithmen und ihre Analyse

Wir betrachten zunächst die verschiedenen Abstraktionsebenen für Algorithmen anhand eines Beispiels:

Beispiel 1.1: Gegeben sei eine Menge S von ganzen Zahlen. Stelle fest, ob eine bestimmte Zahl c enthalten ist.

Eine Spezifikation als *Funktion* auf der Ebene der Mathematik könnte z.B. so aussehen:

Sei $\mathbb{Z}$ die Menge der ganzen Zahlen und bezeichne $F(\mathbb{Z})$ die Menge aller *endlichen* Teilmengen von $\mathbb{Z}$ (analog zur Potenzmenge $P(\mathbb{Z})$, der Menge aller Teilmengen von $\mathbb{Z}$). Sei $BOOL = \{true, false\}$. Wir definieren:

$$contains\colon F(\mathbb{Z}) \times \mathbb{Z} \to BOOL$$

$$contains(S,c) = \begin{cases} true & \text{falls } c \in S \\ false & \text{sonst} \end{cases}$$

Auf der algorithmischen Ebene müssen wir eine Repräsentation für die Objektmenge wählen. Der Einfachheit halber benutzen wir hier einen Array.

```
algorithm contains (S, c)
{Eingaben sind S, ein Integer-Array der Länge n, und c, ein Integer-Wert. Ausgabe
   ist true, falls c ∈ S, sonst false.}
var b : bool;
b := false;
for i := 1 to n do
   if S[i] = c then b := true  end if
end for;
return b.
```

Auf der programmiersprachlichen Ebene müssen wir uns offensichtlich für eine bestimmte Sprache entscheiden. Wir wählen Java.

```
public boolean contains (int[] s, int c)
{
   boolean b = false;
   for (int i = 0; i < s.length; i++)
      if (s[i] == c) b = true;
   return b;
}
```

□

(Das kleine Kästchen am rechten Rand bezeichnet das Ende eines Beispiels, einer Definition, eines Beweises oder dergleichen – falls Sie so etwas noch nicht gesehen haben.)

Es geht uns darum, die verschiedenen Abstraktionsebenen klar voneinander abzugrenzen und insbesondere die Unterschiede in der Beschreibung von Algorithmen und von Programmen zu erklären:

Auf der *Ebene der Mathematik* wird präzise beschrieben, *was* berechnet wird; es bleibt offen, *wie* es berechnet wird. Die Spezifikation einer Funktion kann durch viele verschiedene Algorithmen implementiert werden.

Das Ziel einer Beschreibung *auf algorithmischer Ebene* besteht darin, einem anderen *Menschen* mitzuteilen, *wie* etwas berechnet wird. Man schreibt also nicht für einen Compiler; die Details einer speziellen Programmiersprache sind irrelevant. Es ist wesentlich, daß die Beschreibung auf dieser Ebene einen Abstraktionsgrad besitzt, der der Kommunikation zwischen Menschen angemessen ist. Teile eines Algorithmus, die dem Leser sowieso klar sind, sollten weggelassen bzw. knapp umgangssprachlich skizziert werden. Dabei muß derjenige, der den Algorithmus beschreibt, allerdings wissen, welche Grundlagen für das Verständnis des Lesers vorhanden sind. Im Rahmen dieses Kurses wird diese Voraussetzung dadurch erfüllt, daß Autoren und Leser darin übereinstimmen, daß der Text von vorne nach hinten gelesen wird. Am Ende des Kurses können in einem Algorithmus deshalb z.B. solche Anweisungen stehen:

> sortiere die Menge S nach x-Koordinate
> berechne C als Menge der starken Komponenten des Graphen G
> stelle S als AVL-Baum dar

Der obige Algorithmus ist eigentlich etwas zu einfach, um diesen Aspekt zu illustrieren. Man könnte ihn durchaus auch so beschreiben:

> durchlaufe S, um festzustellen, ob c vorhanden ist

Für einen komplexeren Algorithmus hätte allerdings das entsprechende Programm nicht gut an diese Stelle gepaßt!

Neben dem richtigen Abstraktionsgrad ist ein zweiter wichtiger Aspekt der Beschreibung von Algorithmen die Unabhängigkeit von einer speziellen Programmiersprache. Dies erlaubt eine gewisse Freiheit: Man kann syntaktische Konstrukte nach Geschmack wählen, solange ihre Bedeutung für den Leser klar ist. Man ist auch nicht an Eigentümlichkeiten einer speziellen Sprache gebunden und muß sich nicht sklavisch an eine Syntax halten, die ein bestimmter Compiler akzeptiert. Mit anderen Worten: Man kann sich auf das Wesentliche konzentrieren.

Konkret haben wir oben einige Notationen für Kontrollstrukturen verwendet, die Sie bisher vielleicht nicht gesehen haben:

> **if** <Bedingung> **then** <Anweisungen> **end if**
> **if** <Bedingung> **then** <Anweisungen> **else** <Anweisungen> **end if**
> **for** <Schleifen-Kontrolle> **do** <Anweisungen> **end for**

Analog gibt es z.B.

> **while** <Bedingung> **do** <Anweisungen> **end while**

Wir werden in Algorithmen meist diesen Stil verwenden. Es kommt aber nicht besonders darauf an; z.B. findet sich in [Bauer und Wössner 1984] in Beschreibungen von Algorithmen ein anderer Klammerungsstil, bei dem Schlüsselwörter umgedreht werden (if - fi, do - od):

if <Bedingung> **then** <Anweisungen> **fi**
if <Bedingung> **then** <Anweisungen> **else** <Anweisungen> **fi**
for <Schleifen-Kontrolle> **do** <Anweisungen> **od**

Ebenso ist auch eine an die Sprache C oder Java angelehnte Notation möglich:

if (<Bedingung>) { <Anweisungen> }
if (<Bedingung>) { <Anweisungen> } **else** { <Anweisungen> }
for (<Schleifen-Kontrolle>) { <Anweisungen> }
while (<Bedingung>) { <Anweisungen> }

Wichtig ist vor allem, daß der Leser die Bedeutung der Konstrukte versteht. Natürlich ist es sinnvoll, nicht innerhalb eines Algorithmus verschiedene Stile zu mischen.

Die Unabhängigkeit von einer speziellen Programmiersprache bedeutet andererseits, daß man keine Techniken und Tricks in der Notation benutzen sollte, die nur für diese Sprache gültig sind. Schließlich sollte der Algorithmus in jeder universellen Programmiersprache implementiert werden können. Das obige Java-Programm illustriert dies. In Java ist es erlaubt, in einer Methode Array-Parameter unbestimmter Größe zu verwenden; dabei wird angenommen, daß ein solcher Parameter einen Indexbereich hat, der mit 0 beginnt. Die obere Indexgrenze kann man über das Attribut *length* des Arrays erfragen. Ist es nun für die Beschreibung des Algorithmus wesentlich, dies zu erklären? Natürlich nicht.

In diesem Kurs werden Algorithmen daher im allgemeinen auf der gerade beschriebenen algorithmischen Ebene formuliert; nur selten – meist in den ersten Kapiteln, die sich noch recht nahe an der Programmierung bewegen – werden auch Programme dazu angegeben. In diesen Fällen verwenden wir die Programmiersprache Java.

Welche Kriterien gibt es nun, um die Qualität eines Algorithmus zu beurteilen? Zwingend notwendig ist zunächst die *Korrektheit*. Ein Algorithmus, der eine gegebene Problemstellung nicht realisiert, ist völlig unnütz. Wünschenswert sind darüber hinaus folgende Eigenschaften:

- Er sollte *einfach zu verstehen* sein. Dies erhöht die Chance, daß der Algorithmus tatsächlich korrekt ist; es erleichtert die Implementierung und spätere Änderungen.
- Eine *einfache Implementierbarkeit* ist ebenfalls anzustreben. Vor allem, wenn abzusehen ist, daß ein Programm nur sehr selten laufen wird, sollte man bei mehre-

ren möglichen Algorithmen denjenigen wählen, der schnell implementiert werden kann, da hier die zeitbestimmende Komponente das Schreiben und Debuggen ist.
- Laufzeit und Platzbedarf sollten so gering wie möglich sein. Diese beiden Punkte interessieren uns im Rahmen der *Analyse von Algorithmen*, die wir im folgenden besprechen.

Zwischen den einzelnen Kriterien gibt es oft einen "trade-off", das heißt, man kann eine Eigenschaft nur erreichen, wenn man in Kauf nimmt, daß dabei eine andere schlechter erfüllt wird. So könnte z.B. ein sehr effizienter Algorithmus nur schwer verständlich sein.

Bei der Analyse ergibt sich zuerst die Frage, wie denn Laufzeit oder Speicherplatz eines Algorithmus gemessen werden können. Betrachten wir zunächst die Laufzeit. Die Rechenzeit eines Programms, also eines implementierten Algorithmus, könnte man etwa in Millisekunden messen. Diese Größe ist aber abhängig von vielen Parametern wie dem verwendeten Rechner, Compiler, Betriebssystem, Programmiertricks, usw. Außerdem ist sie ja nur für Programme meßbar, nicht aber für Algorithmen. Um das Ziel zu erreichen, tatsächlich die Laufzeiteigenschaften eines Algorithmus zu beschreiben, geht man so vor:

- Für eine gegebene Eingabe werden im Prinzip die durchgeführten Elementaroperationen gezählt.
- Das Verhalten des Algorithmus kann dann durch eine Funktion angegeben werden, die die Anzahl der durchgeführten Elementaroperationen in Abhängigkeit von der "Komplexität" der Eingabe darstellt (diese ist im allgemeinen gegeben durch die Kardinalität der Eingabemengen).

Aus praktischer Sicht sind Elementaroperationen Primitive, die üblicherweise von Programmiersprachen angeboten werden und die in eine feste, kurze Folge von Maschineninstruktionen abgebildet werden. Einige Beispiele für elementare und nicht elementare Konstrukte sind in Tabelle 1.1 angegeben.

Elementaroperationen		*nicht elementare Operationen*	
Zuweisung	$x := 1$	Schleife	**while** ...
Vergleich	$x \leq y$		**for** ...
Arithmetische Operation	$x + y$		**repeat** ...
Arrayzugriff	$s[i]$	Prozeduraufruf (insbes. Rekursion)	
...			

Tabelle 1.1: Elementare und nicht elementare Operationen

Um die Komplexität von Algorithmen formal zu studieren, führt man mathematische Maschinenmodelle ein, die geeignete Abstraktionen realer Rechner darstellen, z.B. *Turingmaschinen* oder *Random-Access-Maschinen* (*RAM*). Eine RAM besitzt einen Programmspeicher und einen Datenspeicher. Der Programmspeicher kann eine Folge von Befehlen aus einem festgelegten kleinen Satz von Instruktionen aufnehmen. Der Datenspeicher ist eine unendliche Folge von Speicherzellen (oder *Registern*) $r_0, r_1, r_2, \ldots$, die jeweils eine natürliche Zahl aufnehmen können. Register r_0 spielt die Rolle eines *Akkumulators*, das heißt, es stellt in arithmetischen Operationen, Vergleichen, usw. implizit einen Operanden dar. Weiterhin gibt es einen Programmzähler, der zu Anfang auf den ersten Befehl, später auf den gerade auszuführenden Befehl im Programmspeicher zeigt. Der Instruktionssatz einer RAM enthält Speicher- und Ladebefehle für den Akkumulator, arithmetische Operationen, Vergleiche und Sprungbefehle; für alle diese Befehle ist ihre Wirkung auf den Datenspeicher und den Befehlszähler präzise definiert. Wie man sieht, entspricht der RAM-Instruktionssatz in etwa einem minimalen Ausschnitt der Maschinen- oder Assemblersprache eines realen Rechners.

Bei einer solchen formalen Betrachtung entspricht eine Elementaroperation gerade einer RAM-Instruktion. Man kann nun jeder Instruktion ein *Kostenmaß* zuordnen; die Laufzeit eines RAM-Programms ist dann die Summe der Kosten der ausgeführten Instruktionen. Gewöhnlich werden *Einheitskosten* angenommen, das heißt, jede Instruktion hat Kostenmaß 1. Dies ist realistisch, falls die Anwendung nicht mit sehr großen Zahlen umgeht, die nicht mehr in ein Speicherwort eines realen Rechners passen würden (eine RAM-Speicherzelle kann ja beliebig große Zahlen aufnehmen). Bei Programmen mit sehr großen Zahlen ist ein *logarithmisches Kostenmaß* realistischer, da die Darstellung einer Zahl n etwa $\log n$ Bits benötigt. Die Kosten für einen Ladebefehl (Register $\rightarrow$ Akkumulator) sind dann z.B. $\log n$, die Kosten für arithmetische Operationen müssen entsprechend gewählt werden.

Eine Modifikation des RAM-Modells ist die *real RAM*, bei der angenommen wird, daß jede Speicherzelle eine reelle Zahl in voller Genauigkeit darstellen kann und daß Operationen auf reellen Zahlen, z.B. auch Wurzelziehen, trigonometrische Funktionen, usw. angeboten werden und Kostenmaß 1 haben. Dieses Modell abstrahiert von Problemen, die durch die Darstellung reeller Zahlen in realen Rechnern entstehen, z.B. Rundungsfehler oder die Notwendigkeit, sehr große Zahlendarstellungen zu benutzen, um Rundungsfehler zu vermeiden. Die real RAM wird oft als Grundlage für die Analyse geometrischer Algorithmen (Kapitel 8) benutzt.

Derartige Modelle bilden also die formale Grundlage für die Analyse von Algorithmen und präzisieren den Begriff der Elementaroperation. Nach der oben beschriebenen

Vorgehensweise müßte man nun eine Funktion T (= Time, Laufzeit) angeben, die jeder möglichen Eingabe die Anzahl durchgeführter Elementaroperationen zuordnet.

Beispiel 1.2: In den folgenden Skizzen werden verschiedene mögliche Eingaben für den Algorithmus aus Beispiel 1.1 betrachtet:

- eine vierelementige Menge und eine Zahl, die darin nicht vorkommt,
- eine vierelementige Menge und eine Zahl, die darin vorkommt,
- eine achtelementige Menge und eine Zahl, die darin nicht vorkommt.

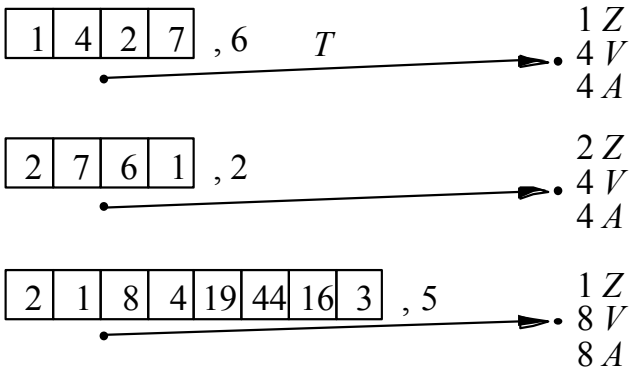

Abbildung 1.2: Anzahl von Elementaroperationen

Die einzigen Elementaroperationen, die im Algorithmus auftreten, sind Zuweisungen (Z), Arrayzugriffe (A) und Vergleiche (V). Die Inkrementierung und der Vergleich der Schleifenvariablen wird hier außer acht gelassen. (Um dies zu berücksichtigen, müßten wir die Implementierung des Schleifenkonstrukts durch den Compiler kennen.) □

Eine so präzise Bestimmung der Funktion T wird im allgemeinen *nicht* durchgeführt, denn

- es ist uninteressant (zu detailliert, man kann sich das Ergebnis nicht merken), und
- eine so detaillierte Analyse ist gewöhnlich mathematisch nicht handhabbar.

Bei der formalen Betrachtungsweise müßte man die Anzahlen der RAM-Operationen zuordnen; das ist aber nur für RAM-Programme, nicht für auf höherer Ebene formulierte Algorithmen machbar. Man macht deshalb eine Reihe von *Abstraktionsschritten*, um zu einer einfacheren Beschreibung zu kommen und um auf der Ebene der algorithmischen Beschreibung analysieren zu können:

1. Abstraktionsschritt. Die Art der Elementaroperationen wird nicht mehr unterschieden. Das heißt, man konzentriert sich auf die Beobachtung "dominanter" Operationen, die die

Laufzeit im wesentlichen bestimmen, oder “wirft alle in einen Topf”, nimmt also an, daß alle gleich lange dauern.

Beispiel 1.3: Mit den gleichen Eingaben wie im vorigen Beispiel ergibt sich:

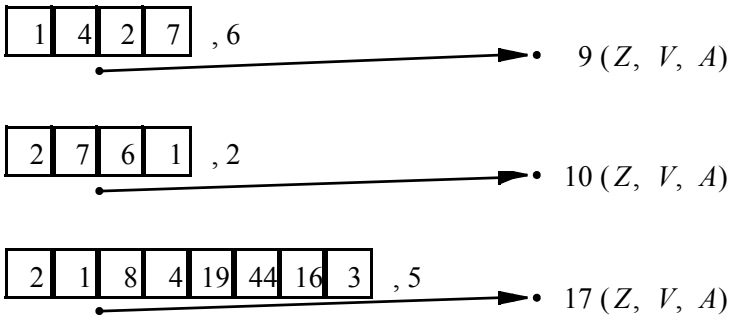

Abbildung 1.3: Erster Abstraktionsschritt

□

2. Abstraktionsschritt. Die Menge aller Eingaben wird aufgeteilt in “Komplexitätsklassen”. Weitere Untersuchungen werden nicht mehr für jede mögliche Eingabe, sondern nur noch für die möglichen Komplexitätsklassen durchgeführt. Im einfachsten Fall wird die Komplexitätsklasse durch die Größe der Eingabe bestimmt. Manchmal spielen aber weitere Parameter eine Rolle; dies wird unten genauer diskutiert (s. Beispiel 1.16).

Beispiel 1.4: Für unseren einfachen Algorithmus wird die Laufzeit offensichtlich durch die Größe des Arrays bestimmt.

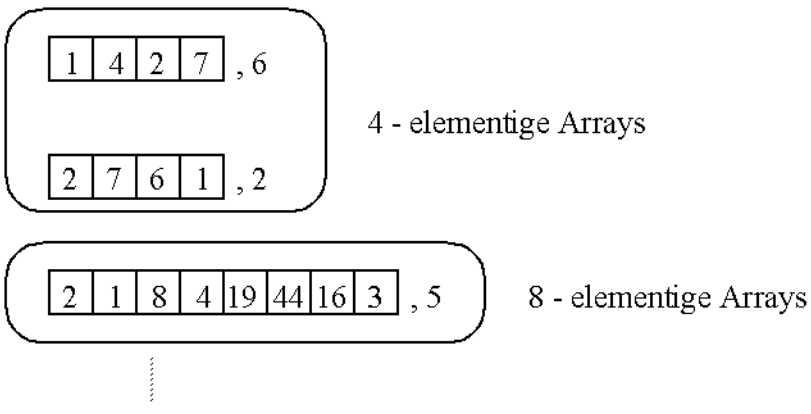

Abbildung 1.4: Zweiter Abstraktionsschritt

Wir betrachten also ab jetzt n-elementige Arrays. □

Üblicherweise wird die Laufzeit $T(n)$ eines Algorithmus bei einer Eingabe der Größe n dann als Funktion von n angegeben.

3. Abstraktionsschritt. Innerhalb einer Komplexitätsklasse wird abstrahiert von den vielen möglichen Eingaben durch

(a) Betrachtung von Spezialfällen
- der beste Fall (*best case*) T_{best}
- der schlimmste Fall (*worst case*) T_{worst}

(b) Betrachtung des
- Durchschnittsverhaltens (*average case*) T_{avg}

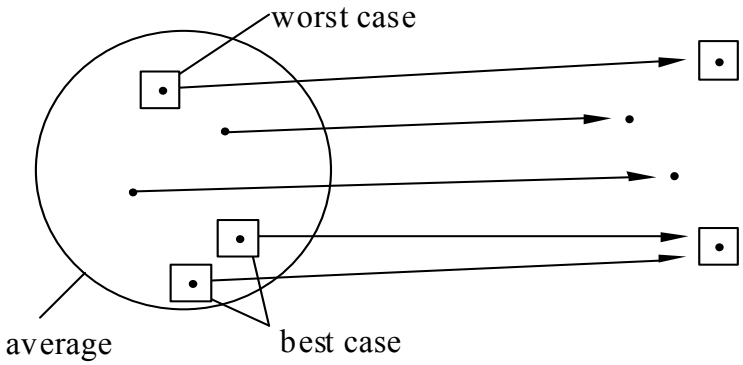

Abbildung 1.5: Dritter Abstraktionsschritt

Abbildung 1.5 illustriert dies: Innerhalb der Menge aller Eingaben dieser Komplexitätsklasse gibt es eine Klasse von Eingaben, für die sich die geringste Laufzeit (Anzahl von Elementaroperationen) ergibt, ebenso eine oder mehrere Eingaben, die die höchste Laufzeit benötigen (worst case). Beim Durchschnittsverhalten betrachtet man *alle* Eingaben. Dabei ist es aber fraglich, ob alle Eingaben bei der Anwendung des Algorithmus mit gleicher Häufigkeit auftreten. Man käme z.B. zu einem zumindest aus praktischer Sicht völlig falschen Ergebnis, wenn ein Algorithmus für viele Eingaben eine hohe Laufzeit hat, er aber tatsächlich nur für die Eingaben benutzt wird, bei denen die Laufzeit gering ist. Deshalb kann man nicht einfach den Durchschnitt über die Laufzeiten aller Eingaben bilden, sondern muß ein *gewichtetes Mittel* bilden, bei dem die Häufigkeiten oder Wahrscheinlichkeiten der Eingaben berücksichtigt werden. Problematisch daran ist, daß man entsprechende Annahmen über die Anwendung machen muß. Der einfachste Fall ist natürlich die Gleichverteilung; dies ist aber nicht immer realistisch.

Beispiel 1.5: Wir betrachten die drei Arten der Analyse für unser Beispiel.

(a) Der beste Fall: Das gesuchte Element ist *nicht* im Array vorhanden

$$T_{\text{best}}(n) = n + 1 \quad (n \text{ Vergleiche} + 1 \text{ Zuweisung})$$

Der schlimmste Fall: Das gesuchte Element ist im Array vorhanden

$$T_{\text{worst}}(n) = n + 2 \quad (n \text{ Vergleiche} + 2 \text{ Zuweisungen})$$

(b) Durchschnittsverhalten: Welche zusätzlichen Annahmen sind realistisch?

- Alle Array-Werte sind verschieden (da der Array eine Menge darstellt).
- Die Gleichverteilungsannahme besagt, daß die Array-Elemente und der Suchwert zufällig aus dem gesamten Integer-Bereich gewählt sein können. Dann ist es sehr unwahrscheinlich, für nicht sehr großes n, daß c vorkommt. Wir nehmen hier einfach an, wir wissen von der Anwendung, daß mit 50% Wahrscheinlichkeit c in S vorkommt. Dann ist

$$\begin{aligned} T_{avg}(n) &= \frac{T_{best} + T_{worst}}{2} \\ &= \frac{1}{2} \cdot (n+1) + \frac{1}{2} \cdot (n+2) \\ &= n + \frac{3}{2} \end{aligned}$$

T_{best} und T_{worst} sind hier zufällig die einzigen überhaupt möglichen Fälle; nach der Annahme sollen sie mit gleicher Wahrscheinlichkeit vorkommen. Übrigens wird die genaue Berechnung, ob $n + 1$ oder $n + 2$ Operationen benötigt werden, durch den nächsten Abstraktionsschritt überflüssig. □

Die Durchschnittsanalyse ist im allgemeinen mathematisch wesentlich schwieriger zu behandeln als die Betrachtung des worst-case-Verhaltens. Deshalb beschränkt man sich häufig auf die worst-case-Analyse. Der beste Fall ist nur selten interessant.

4. Abstraktionsschritt. Durch Weglassen von multiplikativen und additiven Konstanten wird nur noch das *Wachstum* einer Laufzeitfunktion $T(n)$ betrachtet. Dies geschieht mit Hilfe der *O-Notation*:

Definition 1.6: (O-Notation) Seien $f : \mathbb{N} \to \mathbb{R}^+$, $g : \mathbb{N} \to \mathbb{R}^+$.

$$f = \mathrm{O}(g) :\Leftrightarrow \exists\, n_0 \in \mathbb{N}, c \in \mathbb{R}, c > 0 \colon \forall\, n \geq n_0 \; f(n) \leq c \cdot g(n)$$

Das bedeutet intuitiv: f wächst höchstens so schnell wie g. Die Schreibweise $f = \mathrm{O}(g)$ hat sich eingebürgert für die präzisere Schreibweise $f \in \mathrm{O}(g)$, wobei $\mathrm{O}(g)$ eine wie folgt definierte Funktionenmenge ist:

$$O(g) = \{f: \mathbb{N} \to \mathbb{R}^+ \mid \exists\, n_0 \in \mathbb{N}, c \in \mathbb{R}, c > 0: \forall\, n \geq n_0\ f(n) \leq c \cdot g(n)\}$$

Das hat als Konsequenz, daß man eine "Gleichung" $f = O(g)$ nur von links nach rechts lesen kann. Eine Aussage $O(g) = f$ ist sinnlos. Bei der Analyse von Algorithmen sind gewöhnlich $f, g: \mathbb{N} \to \mathbb{N}$ definiert, da das Argument die Größe der Eingabe und der Funktionswert die Anzahl durchgeführter Elementaroperationen ist. Unter anderem wegen Durchschnittsanalysen kann rechts auch $\mathbb{R}^+$ stehen.

Beispiel 1.7: Es gilt:

$T_1(n) = n + 3 = O(n)$ da $n + 3 \leq 2n \quad \forall n \geq 3$
$T_2(n) = 3n + 7 = O(n)$
$T_3(n) = 1000n = O(n)$
$T_4(n) = 695n^2 + 397n + 6148 = O(n^2)$

□

Um derartige Aussagen zu überprüfen, muß man nicht unbedingt Konstanten suchen, die die Definition erfüllen. Die Funktionen, mit denen man umgeht, sind praktisch immer monoton wachsend und überall von 0 verschieden. Dann kann man den Quotienten der beiden Funktionen bilden. Die Definition besagt nun, daß für alle n ab irgendeinem n_0 gilt $f(n)/g(n) \leq c$. Man kann daher die beiden Funktionen "vergleichen", indem man versucht, den Grenzwert

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

zu bilden. Falls dieser Grenzwert existiert, so gilt $f = O(g)$. Falls der Grenzwert 0 ist, so gilt natürlich auch $f = O(g)$ und g wächst sogar echt schneller als f; dafür werden wir im folgenden noch eine spezielle Notation einführen. Wenn aber $f(n)/g(n)$ über alle Grenzen wächst, dann gilt nicht $f = O(g)$.

Beispiel 1.8:

$$\lim_{n\to\infty} \frac{T_4(n)}{n^2} = \lim_{n\to\infty} \frac{695n^2 + 397n + 6148}{n^2} = 695$$

Also gilt $T_4(n) = O(n^2)$. □

Selbsttestaufgabe 1.1: Gilt $3\sqrt{n} + 5 = O(n)$? □

Selbsttestaufgabe 1.2: Gilt $\log(n) = O(n)$? □

Man sollte sich klarmachen, daß die O-Notation eine "vergröbernde" Betrachtung von Funktionen liefert, die zwei wesentliche Aspekte hat:

- Sie eliminiert Konstanten: $O(n) = O(n/2) = O(17n) = O(6n + 5)$. Für alle diese Ausdrücke schreibt man $O(n)$.
- Sie bildet obere Schranken: $O(1) = O(n) = O(n^2) = O(2^n)$. (Hier ist es wesentlich, daß die Gleichungsfolge von links nach rechts gelesen wird! Die "mathematisch korrekte" Schreibweise wäre $O(1) \subset O(n) \subset O(n^2) \subset O(2^n)$.) Es ist also nicht verkehrt, zu sagen: $3n = O(n^2)$.

Aufgrund der Bildung oberer Schranken erleichtert die O-Notation insbesondere die worst-case-Analyse von Algorithmen, bei der man ja eine obere Schranke für die Laufzeit ermitteln will. Wir betrachten die Analyse einiger grundlegender Kontrollstrukturen und zeigen dabei zugleich einige "Rechenregeln" für die O-Notation.

Im folgenden seien S_1 und S_2 Anweisungen (oder Programmteile) mit Laufzeiten $T_1(n) = O(f(n))$ und $T_2(n) = O(g(n))$. Wir nehmen an, daß $f(n)$ und $g(n)$ von 0 verschieden sind, also z.B. $O(f(n))$ ist mindestens $O(1)$.

Die Laufzeit einer *Elementaroperation* ist $O(1)$. Eine *Folge von c Elementaroperationen* (c eine Konstante) hat Laufzeit $c \cdot O(1) = O(1)$.

Beispiel 1.9: Die Anweisungsfolge

```
x := 15;
y := x;
if x ≤ z then a := 1; else a := 0 end if
```

hat Laufzeit $O(1)$. □

Eine *Sequenz* $S_1; S_2$ hat Laufzeit

$$T(n) = T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(f(n) + g(n))$$

Gewöhnlich ist eine der beiden Funktionen dominant, das heißt, $f = O(g)$ oder $g = O(f)$. Dann gilt:

$$T(n) = \begin{cases} O(f(n)) & \text{falls } g = O(f) \\ O(g(n)) & \text{falls } f = O(g) \end{cases}$$

Die Laufzeit einer *Folge von Anweisungen* kann man daher gewöhnlich mit der Laufzeit der aufwendigsten Operation in der Folge abschätzen.Wenn mehrere Anweisungen mit dieser maximalen Laufzeit $f(n)$ auftreten, spielt das keine Rolle, da ja gilt:

$$O(f(n)) + O(f(n)) = O(f(n))$$

Beispiel 1.10: Gegeben seien zwei Algorithmen $alg_1(U)$ und $alg_2(U)$. Das Argument U ist eine Datenstruktur, die eine zu verarbeitende Mengen von Objekten darstellt. Algo-

rithmus alg_1, angewandt auf eine Menge U mit n Elementen hat Laufzeit $O(n)$, Algorithmus alg_2 hat Laufzeit $O(n^2)$. Das Programmstück

```
alg1(U);
alg2(U);
alg2(U);
```

hat für eine n-elementige Menge U die Laufzeit $O(n) + O(n^2) + O(n^2) = O(n^2)$. □

Bei einer *Schleife* kann jeder Schleifendurchlauf eine andere Laufzeit haben. Dann muß man alle diese Laufzeiten aufsummieren. Oft ist die Laufzeit aber bei jedem Durchlauf gleich, z.B. $O(1)$. Dann kann man multiplizieren. Sei also $T_0(n) = O(g(n))$ die Laufzeit für einen Schleifendurchlauf. Zwei Fälle sind zu unterscheiden:

(a) Die Anzahl der Schleifendurchläufe hängt nicht von n ab, ist also eine Konstante c. Dann ist die Laufzeit für die Schleife insgesamt

$$\begin{aligned} T(n) &= O(1) + c \cdot O(g(n)) \\ &= O(g(n)), \text{ falls } c > 0. \end{aligned}$$

Der $O(1)$-Beitrag entsteht, weil mindestens einmal die Schleifenbedingung ausgewertet werden muß.

(b) Die Anzahl der Schleifendurchläufe ist $O(f(n))$:

$$T(n) = O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

Beispiel 1.11: Die Anweisungsfolge

```
const k = 70;
for i := 1 to n do
   for j := 1 to k do
      s := s + i * j
   end for
end for
```

hat Laufzeit $O(n)$. Denn die Laufzeit der inneren Schleife hängt nicht von n ab, ist also konstant oder $O(1)$. □

Bei einer *bedingten Anweisung* **if** B **then** S_1 **else** S_2 ist die Laufzeit durch den Ausdruck

$$O(1) + O(f(n)) + O(g(n))$$

gegeben, den man dann vereinfachen kann. Gewöhnlich erhält man dabei als Ergebnis die Laufzeit der dominanten Operation, also $O(f(n))$ oder $O(g(n))$. Wenn die beiden

Laufzeitfunktionen nicht vergleichbar sind, kann man mit der Summe weiterrechnen; diese ist sicher eine obere Schranke.

Beispiel 1.12: Die Anweisungsfolge

if $a > b$
then $alg_1(U)$
else if $a > c$ **then** $x := 0$ **else** $alg_2(U)$ **end if**
end if

hat für eine n-elementige Menge U die Laufzeit $O(n^2)$. In den verschiedenen Zweigen der Fallunterscheidung treten die Laufzeiten $O(n)$, $O(1)$ und $O(n^2)$ auf; da wir den schlimmsten Fall betrachten, ist die Laufzeit die von alg_2. □

Nicht-rekursive *Prozeduren*, *Methoden* oder *Subalgorithmen* kann man für sich analysieren und ihre Laufzeit bei Aufrufen entsprechend einsetzen. Bei rekursiven Algorithmen hingegen wird die Laufzeit durch eine *Rekursionsgleichung* beschrieben, die man lösen muß. Dafür werden wir später noch genügend Beispiele kennenlernen. Überhaupt ist die Analyse von Algorithmen ein zentrales Thema, das uns durch den ganzen Kurs begleiten wird. Hier sollten nur einige Grundtechniken eingeführt werden.

Mit der O-Notation werden Laufzeitfunktionen "eingeordnet" in gewisse Klassen:

	Sprechweise	*Typische Algorithmen*
$O(1)$	konstant	
$O(\log n)$	logarithmisch	Suchen auf einer Menge
$O(n)$	linear	Bearbeiten jedes Elementes einer Menge
$O(n \log n)$		Gute Sortierverfahren, z.B. Heapsort
$O(n \log^2 n)$		
...		
$O(n^2)$	quadratisch	primitive Sortierverfahren
$O(n^k)$, $k \geq 2$	polynomiell	
...		
$O(2^n)$	exponentiell	Backtracking-Algorithmen

Tabelle 1.2: Klassen der O-Notation

In Tabelle 1.2 wie auch allgemein in diesem Kurs bezeichnet log (ohne Angabe der Basis) den Logarithmus zur Basis 2. Innerhalb von O-Notation spielt das aber keine Rolle, da Logarithmen zu verschiedenen Basen sich nur durch einen konstanten Faktor unterscheiden, aufgrund der Beziehung

$$\log_b x = \log_b a \cdot \log_a x$$

Deshalb kann die Basis bei Verwendung in O-Notation allgemein weggelassen werden.

Die Laufzeiten für unser Beispiel 1.1 können wir jetzt in O-Notation ausdrücken.

$$\left.\begin{array}{lcl} T_{best}(n) & = & O(n) \\ T_{worst}(n) & = & O(n) \\ T_{avg}(n) & = & O(n) \end{array}\right\} \quad \text{“lineare Laufzeit”}$$

Nachdem wir nun in der Lage sind, Algorithmen zu analysieren, können wir versuchen, den Algorithmus *contains* zu verbessern. Zunächst einmal ist es offenbar ungeschickt, daß die Schleife nicht abgebrochen wird, sobald das gesuchte Element gefunden wird.

```
algorithm contains_2 (S, c)
i := 1;
while S[i] ≠ c and i ≤ n do i := i + 1 end while;   {Abbruch, sobald c gefunden}
if i ≤ n then return true
         else return false
end if.
```

Die Analyse im besten und schlimmsten Fall ist offensichtlich.

$$\begin{array}{ll} T_{best}(n) & = \mathrm{O}(1) \\ T_{worst}(n) & = \mathrm{O}(n) \end{array}$$

Wie steht es mit dem Durchschnittsverhalten?

Fall 1: c kommt unter den n Elementen in S vor. Wir nehmen an, mit gleicher Wahrscheinlichkeit auf Platz 1, Platz 2, ..., Platz n. Also ist

$$\begin{aligned} T_1(n) &= \frac{1}{n}\cdot 1 + \frac{1}{n}\cdot 2 + \ldots + \frac{1}{n}\cdot n \\ &= \frac{1}{n}\cdot\sum_{i=1}^{n} i = \frac{1}{n}\cdot\frac{n\cdot(n+1)}{2} = \frac{n+1}{2} \end{aligned}$$

Das heißt, falls c vorhanden ist, findet man es im Durchschnitt in der Mitte. (Überraschung!)

Fall 2: c kommt nicht vor.

$$T_2(n) = n$$

Da beide Fälle nach der obigen Annahme jeweils mit einer Wahrscheinlichkeit von 0.5 vorkommen, gilt

$$T_{avg}(n) = \frac{1}{2} \cdot T_1(n) + \frac{1}{2} \cdot T_2(n) = \frac{1}{2} \cdot \frac{n+1}{2} + \frac{n}{2} = \frac{3}{4}n + \frac{1}{4} = \mathrm{O}(n)$$

Also haben wir im Durchschnitt und im worst case asymptotisch (größenordnungsmäßig) keine Verbesserung erzielt: Der Algorithmus hat noch immer lineare Laufzeit.

Wir nehmen eine weitere Modifikation vor: Die Elemente im Array seien aufsteigend geordnet. Dann kann *binäre Suche* benutzt werden:

algorithm *contains*$_3$ (*low*, *high*, *c*)
{Eingaben: *low*, *high* – unterer und oberer Index des zu durchsuchenden Array-Bereichs (siehe Abbildung 1.6); *c* – der zu suchende Integer-Wert. Ausgabe ist *true*, falls *c* im Bereich *S*[*low*] .. *S*[*high*] vorkommt, sonst *false*.}

if *low* > *high* **then return** *false*
else *m* := (*low* + *high*) **div** 2 ;
  if *S*[*m*] = *c* **then return** *true*
  else
    if *S*[*m*] < *c* **then return** *contains* (*m*+1, *high*, *c*)
    else return *contains* (*low*, *m*-1, *c*) **end if**
  end if
end if.

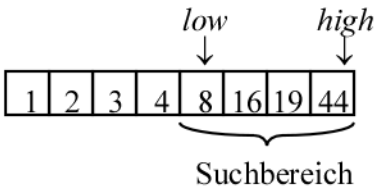

Abbildung 1.6: Algorithmus contains$_3$

Dieser rekursive Algorithmus wird zu Anfang mit *contains* (1, *n*, *c*) aufgerufen; *S* wird diesmal als globale Variable aufgefaßt. Wir analysieren das Verhalten im schlimmsten Fall. Wie oben erwähnt, führt ein rekursiver Algorithmus zu Rekursionsgleichungen für die Laufzeit: Sei $T(m)$ die worst-case-Laufzeit von *contains*, angesetzt auf einen Teil-Array mit *m* Elementen. Es gilt:

$$\begin{aligned} T(0) &= a \\ T(m) &= b + T(m/2) \end{aligned}$$

Hier sind a und b Konstanten: a beschreibt die Laufzeit (eine obere Schranke für die Anzahl von Elementaroperationen), falls kein rekursiver Aufruf erfolgt; b beschreibt im anderen Fall die Laufzeit bis zum rekursiven Aufruf. Einsetzen liefert:

$$\begin{aligned} T(m) &= b + T(m/2) \\ &= b + b + T(m/4) \\ &= b + b + b + T(m/8) \\ &\dots \\ &= \underbrace{b + b + \dots + b}_{\log m \text{ mal}} + a \\ &= b \cdot \log_2 m + a \\ &= \mathrm{O}(\log m) \end{aligned}$$

Der Logarithmus zur Basis 2 einer Zahl drückt ja gerade aus, wie oft man diese Zahl halbieren kann, bis man 1 erhält. Also sind nun

$$\begin{aligned} T_{worst}(n) &= \mathrm{O}(\log n) \text{ und} \\ T_{best}(n) &= \mathrm{O}(1) \end{aligned}$$

Im Durchschnitt wird man gelegentlich das gesuchte Element "etwas eher" finden, das spielt aber asymptotisch keine Rolle, deshalb ist auch

$$T_{avg}(n) = \mathrm{O}(\log n)$$

Dieses Suchverfahren ist also deutlich besser als die bisherigen. Unter der Annahme, daß ein Schritt eine Millisekunde benötigt, ergeben sich beispielsweise folgende Werte:

Anzahl Schritte/Laufzeit	$n = 1000$		$n = 1000000$	
$contains_2$	1000	1 s	1000000	ca. 17 min
$contains_3$	10	0.01 s	20	0.02 s

Tabelle 1.3: Laufzeitvergleich

Der *Platzbedarf* ist bei all diesen Verfahren proportional zur Größe des Array, also $\mathrm{O}(n)$.

Selbsttestaufgabe 1.3: Seien $T_1(n)$ und $T_2(n)$ die Laufzeiten zweier Programmstücke P_1 und P_2. Sei ferner $T_1(n) = \mathrm{O}(f(n))$ und $T_2(n) = \mathrm{O}(g(n))$. Beweisen Sie folgende Eigenschaften der O-Notation:

- *Additionsregel:* $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$
- *Multiplikationsregel:* $T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$

□

Mit der O-Notation $f = O(g)$ kann man auf einfache Art ausdrücken, daß eine Funktion f höchstens so schnell wächst wie eine andere Funktion g. Es gibt noch weitere Notationen, um das Wachstum von Funktionen zu vergleichen:

Definition 1.13: (allgemeine O-Notation)

(i) $f = \Omega(g)$ ("f wächst mindestens so schnell wie g"), falls $g = O(f)$.

(ii) $f = \Theta(g)$ ("f und g wachsen größenordnungsmäßig gleich schnell"), falls $f = O(g)$ und $g = O(f)$.

(iii) $f = o(g)$ ("f wächst langsamer als g"), wenn die Folge $(f(n)/g(n))_{n \in N}$ eine Nullfolge ist.

(iv) $f = \omega(g)$ ("f wächst schneller als g"), falls $g = o(f)$.

Auch hier stehen die Symbole (wie bei Definition 1.6) formal für Funktionenmengen und das Gleichheitszeichen für die Elementbeziehung; die Gleichungen dürfen also nur von links nach rechts gelesen werden. Damit hat man praktisch so etwas wie die üblichen Vergleichsoperationen, um Funktionen größenordnungsmäßig zu vergleichen:

$f = O(g)$	"$f \leq g$"
$f = o(g)$	"$f < g$"
$f = \Theta(g)$	"$f = g$"
$f = \omega(g)$	"$f > g$"
$f = \Omega(g)$	"$f \geq g$"

Tabelle 1.4: Allgemeine O-Notation

Mit diesen Notationen kann man auf einfache Art Funktionsklassen voneinander abgrenzen. Wir geben ohne Beweis einige grundlegende Beziehungen an:

(i) Seien p und p' Polynome vom Grad d bzw. d', wobei die Koeffizienten von n^d bzw. $n^{d'}$ positiv sind. Dann gilt:
- $p = \Theta(p') \Leftrightarrow d = d'$
- $p = o(p') \Leftrightarrow d < d'$
- $p = \omega(p') \Leftrightarrow d > d'$

(ii) $\forall k > 0, \forall \varepsilon > 0 : \log^k n = o(n^\varepsilon)$

(iii) $\forall k > 0 : n^k = o(2^n)$

(iv) $2^{n/2} = o(2^n)$

Diese Beziehungen erlauben uns den einfachen Vergleich von Polynomen, logarithmischen und Exponentialfunktionen.

Selbsttestaufgabe 1.4: Gilt $\log n = O(\sqrt{n})$? □

Beispiel 1.14: Die Laufzeit der einfachen Suchverfahren in diesem Abschnitt (*contains* und $contains_2$) ist $O(n)$ im worst case, aber auch $\Omega(n)$ und daher auch $\Theta(n)$. □

Da die Ω-Notation eine untere Schranke für das Wachstum einer Funktion beschreibt, wird sie oft benutzt, um eine untere Schranke für die Laufzeit *aller* Algorithmen zur Lösung eines Problems anzugeben und damit die *Komplexität des Problems* zu charakterisieren.

Beispiel 1.15: Das Problem, aus einer (ungeordneten) Liste von Zahlen das Minimum zu bestimmen, hat Komplexität $\Omega(n)$.

Beweis: Jeder Algorithmus zur Lösung dieses Problems muß mindestens jede der Zahlen lesen und braucht dazu $\Omega(n)$ Operationen. □

In einem späteren Kapitel werden wir sehen, daß jeder Algorithmus zum Sortieren einer (beliebigen) Menge von n Zahlen $\Omega(n \log n)$ Operationen im worst case braucht.

Ein Algorithmus heißt (asymptotisch) *optimal*, wenn die obere Schranke für seine Laufzeit mit der unteren Schranke für die Komplexität des Problems zusammenfällt. Zum Beispiel ist ein Sortieralgorithmus mit Laufzeit $O(n \log n)$ optimal.

Selbsttestaufgabe 1.5: Eine Zahlenfolge $s_1, \ldots, s_n$ sei in einem Array der Länge n dargestellt. Geben Sie rekursive Algorithmen an (ähnlich der binären Suche)

(a) mit Rekursionstiefe $O(n)$
(b) mit Rekursionstiefe $O(\log n)$

die die Summe dieser Zahlen berechnen. Betrachten Sie dabei jeweils das worst-case-Verhalten dieser Algorithmen. □

Wir hatten oben die vielen möglichen Eingaben eines Algorithmus aufgeteilt in gewisse "Komplexitätsklassen" (was nichts mit der gerade erwähnten Komplexität eines Problems zu tun hat). Diese Komplexitätsklassen sind gewöhnlich durch die Größe der Eingabemenge gegeben. Es gibt aber Probleme, bei denen man weitere Kriterien heranziehen möchte:

Beispiel 1.16: Gegeben eine Menge von n horizontalen und vertikalen Liniensegmenten in der Ebene, bestimme alle Schnittpunkte (bzw. alle Paare sich schneidender Segmente).

Wenn man nur die Größe der Eingabe heranzieht, hat dieses Problem Komplexität $\Omega(n^2)$:

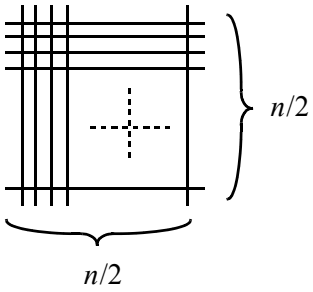

Abbildung 1.7: Schnittpunkte in der Ebene

Die Segmente könnten so angeordnet sein, daß es $n^2/4$ Schnittpunkte gibt. In diesem Fall braucht jeder Algorithmus $\Omega(n^2)$ Operationen. Damit ist der triviale Algorithmus, der sämtliche Paare von Segmenten in $\Theta(n^2)$ Zeit betrachtet, optimal. □

Man benutzt deshalb als Maß für die "Komplexität" der Eingabe nicht nur die Größe der Eingabemenge, sondern auch die Anzahl vorhandener Schnittpunkte k. Das bedeutet, daß es für die Laufzeitanalyse nun zwei Parameter n und k gibt. Wir werden in einem späteren Kapitel Algorithmen kennenlernen, die dieses Problem in $O(n \log n + k)$ Zeit lösen. Das ist optimal.

Wir vergleichen Algorithmen auf der Basis ihrer Zeitkomplexität in O-Notation, das heißt, unter Vernachlässigung von Konstanten. Diese Beurteilung ist aus praktischer Sicht mit etwas Vorsicht zu genießen. Zum Beispiel könnten implementierte Algorithmen, also Programme, folgende Laufzeiten haben:

$$\left.\begin{array}{ll} \text{Programm}_1: & 1000n^2 \text{ ms} \\ \text{Programm}_2: & 5n^3 \text{ ms} \end{array}\right\} \begin{array}{l} \text{für bestimmten Compiler} \\ \text{und bestimmte Maschine} \end{array}$$

Programm_1 ist schneller ab $n = 200$.

Ein Algorithmus mit $O(n^2)$ wird besser als einer mit $O(n^3)$ ab irgendeinem n ("asymptotisch"). Für "kleine" Eingaben kann ein asymptotisch schlechterer Algorithmus der bessere sein! Im Extremfall, wenn die von der O-Notation "verschwiegenen" Konstanten zu groß werden, gibt es in allen praktischen Fällen nur "kleine" Eingaben, selbst wenn $n =$ 1 000 000 ist.

Für die Verarbeitung "großer" Eingabemengen eignen sich praktisch nur Algorithmen mit einer Komplexität von $O(n)$ oder $O(n \log n)$. Exponentielle Algorithmen ($O(2^n)$) kann man nur auf sehr kleine Eingaben anwenden (sagen wir $n < 20$).

1.2 Datenstrukturen, Algebren, Abstrakte Datentypen

Wir wenden uns nun dem rechten Teil des Diagramms aus Abbildung 1.1 zu:

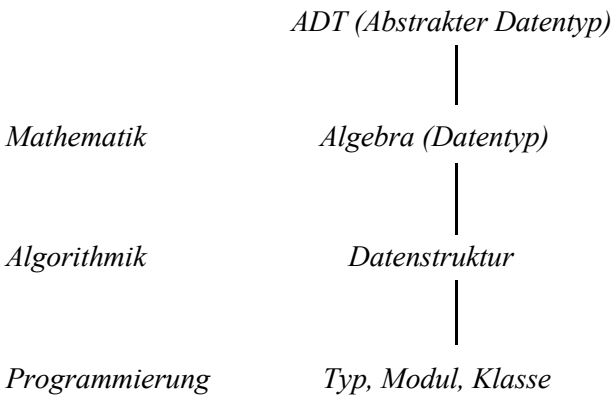

Abbildung 1.8: Abstraktionsebenen von Datenstrukturen

Bisher standen Algorithmen und ihre Effizienz im Vordergrund. Für das binäre Suchen war es aber wesentlich, daß die Elemente im Array aufsteigend sortiert waren. Das heißt, die Methode des Suchens muß bereits beim Einfügen eines Elementes beachtet werden. Wir ändern jetzt den Blickwinkel und betrachten eine Datenstruktur zusammen mit den darauf auszuführenden Operationen als Einheit, stellen also die Datenstruktur in den Vordergrund. Wie in Abschnitt 1.1 studieren wir die verschiedenen Abstraktionsebenen anhand eines Beispiels.

Beispiel 1.17: Verwalte eine Menge ganzer Zahlen, so daß Zahlen eingefügt und gelöscht werden können und der Test auf Enthaltensein durchgeführt werden kann. □

Wir betrachten zunächst die Ebene der Mathematik bzw. Spezifikation. Die abstrakteste Sicht einer Datenstruktur ist offenbar die, daß es eine Klasse von Objekten gibt (die möglichen "Ausprägungen" oder "Werte" der Datenstruktur), auf die gewisse Operationen anwendbar sind. Bei genauerem Hinsehen wird man etwas verallgemeinern: Offensichtlich können mehrere Klassen von Objekten eine Rolle spielen. In unserem Beispiel kommen etwa Mengen ganzer Zahlen, aber auch ganze Zahlen selbst als Objektklassen vor. Die Operationen erzeugen dann aus gegebenen Objekten in diesen Klassen neue Objekte, die ebenfalls zu einer der Objektklassen gehören.

Ein solches System, bestehend aus einer oder mehreren Objektklassen mit dazugehörigen Operationen, bezeichnet man als *Datentyp*. In der Mathematik ist es seit langem als *Algebra* bekannt. Wenn nur eine Objektklasse vorkommt, spricht man von einer *universalen Algebra*, sonst von einer *mehrsortigen* oder *heterogenen Algebra*. Jeder kennt Beispiele: Die natürlichen Zahlen etwa zusammen mit den Operationen Addition und Multiplikation bilden eine (universale) Algebra, Vektorräume mit Vektoren und reellen Zahlen als Objektklassen und Operationen wie Vektoraddition usw. eine mehrsortige Algebra.

Um einen Datentyp oder eine Algebra (wir verwenden die Begriffe im folgenden synonym) zu beschreiben, muß man festlegen, wie die Objektmengen und Operationen heißen, wieviele und was für Objekte die Operationen als Argumente benötigen und welche Art von Objekt sie als Ergebnis liefern. Dies ist ein rein syntaktischer Aspekt, er wird durch eine *Signatur* festgelegt, die man für unser Beispiel etwa so aufschreiben kann:

sorts	*intset*, *int*, *bool*		
ops	*empty*:		$\rightarrow$ *intset*
	insert:	*intset* $\times$ *int*	$\rightarrow$ *intset*
	delete:	*intset* $\times$ *int*	$\rightarrow$ *intset*
	contains:	*intset* $\times$ *int*	$\rightarrow$ *bool*
	isempty:	*intset*	$\rightarrow$ *bool*

Es gibt also drei Objektmengen, die *intset*, *int* und *bool* heißen. Diese Namen der Objektmengen heißen *Sorten.* Weiter kann man z.B. die Operation *contains* auf ein Objekt der Art *intset* und ein Objekt der Art *int* anwenden und erhält als Ergebnis ein Objekt der Art *bool*. Die Operation *empty* braucht kein Argument; sie liefert stets das gleiche Objekt der Art *intset*, stellt also eine *Konstante* dar.

Man beachte, daß die Signatur weiter nichts über die *Semantik*, also die Bedeutung all dieser Bezeichnungen aussagt. Wir haben natürlich eine Vorstellung davon, was z.B. das Wort *bool* bedeutet; die Signatur läßt das aber völlig offen.

Man muß also zusätzlich die Semantik festlegen. Dazu ist im Prinzip jeder Sorte eine *Trägermenge* zuzuordnen und jedem Operationssymbol eine *Funktion* mit entsprechenden Argument- und Wertebereichen. Es gibt nun zwei Vorgehensweisen. Die erste, *Spezifikation als Algebra*, gibt Trägermengen und Funktionen direkt an, unter Verwendung der in der Mathematik üblichen Notationen. Für unser Beispiel sieht das so aus:

algebra *intset*
sorts *intset*, *int*, *bool*
ops

empty:		$\rightarrow$ *intset*
insert:	*intset* $\times$ *int*	$\rightarrow$ *intset*
delete:	*intset* $\times$ *int*	$\rightarrow$ *intset*
contains:	*intset* $\times$ *int*	$\rightarrow$ *bool*
isempty:	*intset*	$\rightarrow$ *bool*

sets *intset* $= F(\mathbb{Z}) = \{M \subset \mathbb{Z} \mid M \text{ endlich}\}$
functions

empty	$= \emptyset$
insert (M, i)	$= M \cup \{i\}$
delete (M, i)	$= M \setminus \{i\}$
contains (M, i)	$= \begin{cases} true & \text{falls } i \in M \\ false & \text{sonst} \end{cases}$
isempty (M)	$= (M = \emptyset)$

end *intset*.

Diese Art der Spezifikation ist relativ einfach und anschaulich; bei etwas mathematischer Vorbildung sind solche Spezifikationen leicht zu lesen und (nicht ganz so leicht) zu schreiben. Ein Nachteil liegt darin, daß man unter Umständen gezwungen ist, Aspekte der Datenstruktur festzulegen, die man gar nicht festlegen wollte.

Die zweite Vorgehensweise, *Spezifikation als abstrakter Datentyp*, versucht, dies zu vermeiden. Die Idee ist, Trägermengen und Operationen nicht explizit anzugeben, sondern sie nur anhand interessierender Aspekte der Wirkungsweise der Operationen, *Gesetze* oder *Axiome* genannt, zu charakterisieren. Das führt für unser Beispiel zu folgender Spezifikation:

adt *intset*
sorts *intset*, *int*, *bool*
ops

empty:		$\rightarrow$ *intset*
insert:	*intset* $\times$ *int*	$\rightarrow$ *intset*
delete:	*intset* $\times$ *int*	$\rightarrow$ *intset*
contains:	*intset* $\times$ *int*	$\rightarrow$ *bool*
isempty:	*intset*	$\rightarrow$ *bool*

axs

isempty $(empty)$	$= true$	
isempty $(insert\ (x, i))$	$= false$	
insert $(insert\ (x, i), i)$	$= insert\ (x, i)$	
contains $(insert\ (x, i), i)$	$= true$	
contains $(insert\ (x, j), i)$	$= contains\ (x, i)$	$(i \neq j)$
...		

end *intset*.

Die Gesetze sind, zumindest im einfachsten Fall, *Gleichungen über Ausdrücken*, die mit Hilfe der Operationssymbole entsprechend der Signatur gebildet werden. Variablen, die in den Ausdrücken vorkommen, sind implizit allquantifiziert. Das Gesetz

$$insert\ (insert\ (x, i), i) \quad = insert\ (x, i)$$

sagt also aus, daß für alle x aus (der Trägermenge von) *intset*, für alle i aus *int*, das Objekt, das durch *insert* (*insert* (x, i), i) beschrieben ist, das gleiche ist wie das Objekt *insert* (x, i). Intuitiv heißt das, daß mehrfaches Einfügen eines Elementes i die Menge x nicht verändert. – Streng genommen müßten oben *true* und *false* noch als 0-stellige Operationen, also Konstanten, des Ergebnistyps *bool* eingeführt werden.

Selbsttestaufgabe 1.6: Gegeben sei die Signatur einer Algebra für einen Zähler, den man zurücksetzen, inkrementieren oder dekrementieren kann:

algebra *counter*			
sorts	*counter*		
ops	*reset*:		→ *counter*
	increment:	*counter*	→ *counter*
	decrement:	*counter*	→ *counter*

Geben sie die Funktionen zu den einzelnen Operationen an, wenn die Trägermenge der Sorte *counter*

(a) die Menge der natürlichen Zahlen: **sets** *counter* = $\mathbb{N}$
(b) die Menge der ganzen Zahlen: **sets** *counter* = $\mathbb{Z}$
(c) ein endlicher Bereich: **sets** *counter* = $\{0, 1, 2, ..., p\}$

ist. Formulieren Sie außerdem die Axiome für den entsprechenden abstrakten Datentyp. □

Eine derartige Spezifikation als abstrakter Datentyp legt eine Algebra im allgemeinen nur unvollständig fest, möglicherweise gerade so unvollständig, wie man es beabsichtigt. Das heißt, es kann mehrere oder viele Algebren mit echt unterschiedlichen Trägermengen geben, die alle die Gesetze erfüllen. Eine Algebra mit gleicher Signatur, die die Gesetze erfüllt, heißt *Modell* für den Datentyp. Wenn es also mehrere Modelle gibt, nennt man den Datentyp *polymorph*. Es ist aber auch möglich, daß die Gesetze eine Algebra bis auf Umbenennung eindeutig festlegen (das heißt, alle Modelle sind *isomorph*). In diesem Fall heißt der Datentyp *monomorph*.

Ein Vorteil dieser Spezifikationsmethode liegt darin, daß man einen Datentyp gerade so weit festlegen kann, wie es erwünscht ist, daß man insbesondere keine Details festlegt, die für die Implementierung gar nicht wesentlich sind, und daß man polymorphe Datentypen spezifizieren kann. Ein weiterer Vorteil ist, daß die Sprache, in der Gesetze formuliert werden, sehr präzise formal beschrieben werden kann; dies erlaubt es, Entwurfs-

werkzeuge zu konstruieren, die etwa die Korrektheit einer Spezifikation prüfen oder auch einen Prototyp erzeugen. Dies ist bei der Angabe einer Algebra mit allgemeiner mathematischer Notation nicht möglich.

Aus praktischer Sicht birgt die Spezifikation mit abstrakten Datentypen aber auch einige Probleme:

- Bei komplexen Anwendungen wird die Anzahl der Gesetze sehr groß.
- Es ist nicht leicht, anhand der Gesetze die intuitive Bedeutung des Datentyps zu erkennen; oft kann man die Spezifikation nur verstehen, wenn man schon weiß, was für eine Struktur gemeint ist.
- Es ist schwer, eine Datenstruktur anhand von Gesetzen zu charakterisieren. Insbesondere ist es schwierig, dabei zu überprüfen, ob die Menge der Gesetze vollständig und widerspruchsfrei ist.

Als Konsequenz ergibt sich, daß diese Spezifikationsmethode wohl nur nach einer speziellen Ausbildung einsetzbar ist; selbst dann entstehen vermutlich noch Schwierigkeiten bei der Spezifikation komplexer Datenstrukturen.

Da es in diesem Kurs nicht um algebraische Spezifikation an sich geht, sondern um Algorithmen und Datenstrukturen, werden wir uns auf die einfachere Technik der direkten Angabe einer Algebra beschränken. Dies ist nichts anderes als mathematische Modellierung einer Datenstruktur. Die Technik bewährt sich nach der Erfahrung der Autoren auch bei komplexeren Problemen.

Im übrigen sollte man festhalten, daß von ganz zentraler Bedeutung die Charakterisierung einer Datenstruktur anhand ihrer Signatur ist. Darüber hinausgehende Spezifikation, sei es als Algebra oder als abstrakter Datentyp, ist sicher wünschenswert, wird aber in der Praxis oft unterbleiben. Dort wird man meist die darzustellenden Objekte und die darauf ausführbaren Operationen nur verbal, also informal, charakterisieren.

Das obige Algebra-Beispiel (*intset*) ist sehr einfach in den verwendeten mathematischen Notationen. In den folgenden Kapiteln werden wir verschiedene grundlegende Datentypen als Algebren spezifizieren. Dabei werden auch komplexere mathematische Beschreibungen vorkommen. Die folgende Selbsttestaufgabe bietet schon in dieser Kurseinheit ein etwas anspruchsvolleres Beispiel (auch als Grundlage für Übungsaufgaben).

Selbsttestaufgabe 1.7: Sicherlich kennen Sie das klassische 15-Puzzle, auch Schiebepuzzle genannt. Auf einem umrahmten Feld von 4x4 Feldern sind 15, mit den Zahlen 1-15 durchnummerierte Steine horizontal und vertikal gegeneinander verschiebbar montiert. Ein Feld bleibt leer. So kann jeweils ein benachbarter Stein auf das

freie 16-te Feld geschoben werden. Das Ziel des Spiels besteht darin, folgende Konstellation zu erreichen:

1	**2**	**3**	**4**
5	**6**	**7**	**8**
9	**10**	**11**	**12**
13	**14**	**15**	

Definieren Sie eine Algebra für das 15-Puzzle. Insbesondere geht es um die Datentypen (Sorten) *board* und *tile* sowie die Operationen *init*, um ein *board* mit einer beliebigen Ausgangskonfiguration zu erzeugen, und *move*, um einen Zug durchzuführen (Bewegen eines gegebenen *tile* auf dem *board*). Der Operator *solved* prüft, ob ein *board* die Gewinnsituation aufweist. Der Operator *pos* gibt die *position* eines gegebenen *tile* auf einem *board* zurück. Die Operationen dürfen nur regelkonforme Spielsituationen oder explizite "Fehlerwerte" erzeugen.

(a) Geben Sie die verwendeten Sorten und Operationen mit deren Signaturen an.

(b) Ordnen Sie den Sorten geeignete Trägermengen zu. Verwenden Sie Mengen und Tupel in den Definitionen.

(c) Legen Sie die Semantik der Operationen durch Angabe von Funktionen fest.

□

Wir betrachten nun die Darstellung unserer Beispiel-Datenstruktur auf der *algorithmischen Ebene*. Dort muß zunächst eine Darstellung für Objekte der Art *intset* festgelegt werden. Das kann z.B. so geschehen:

```
var top: 0..n;
var s  : array [1..n] of integer;
```

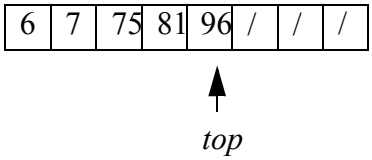

Abbildung 1.9: Beispiel-Ausprägung des Arrays *s*

Wir legen fest, daß Elemente im Array aufsteigend geordnet sein sollen und daß keine Duplikate vorkommen dürfen; dies muß durch die Operationen sichergestellt werden. Die Darstellung führt auch zu der Einschränkung, daß ein *intset* nicht mehr als n Elemente enthalten kann.

Vor der Beschreibung der Algorithmen ist folgendes zu bedenken: In der algebraischen Spezifikation werden Objekte vom Typ *intset* spezifiziert; alle Operationen haben solche Objekte als Argumente und evtl. als Ergebnisse. In der Programmierung weiß man häufig, daß man zur Lösung des gegebenen Problems tatsächlich nur *ein* solches Objekt braucht, oder daß klar ist, auf welches Objekt sich die Operationen beziehen.[1] Als Konsequenz daraus fallen gegenüber der allgemeinen Spezifikation die Parameter weg, die das Objekt bezeichnen, und manche Operationen werden zu Prozeduren anstatt Funktionen, liefern also kein Ergebnis. Wir formulieren nun die Algorithmen auf einer etwas höheren Abstraktionsebene als in Abschnitt 1.1.

algorithm *empty*
$top := 0$.

algorithm *insert* (x)
bestimme den Index j des ersten Elementes $s[j] \geq x$;
if $s[j] \neq x$ **then**
  schiebe alle Elemente ab $s[j]$ um eine Position nach rechts;
  füge Element x auf Position j ein
end if.

algorithm *delete* (x)
bestimme den Index j von Element x. $j = 0$ bedeutet dabei: x nicht gefunden;[2]
if $j > 0$ **then** schiebe alle Elemente ab $s[j + 1]$ um eine Position nach links
end if.

Was soll man tun, wenn das zu löschende Element nicht gefunden wird? Gibt das eine Fehlermeldung? – Wir sehen in der Algebra-Spezifikation nach:

$$delete\ (M, i) = M \setminus \{i\}$$

Also tritt kein Fehler auf, es geschieht einfach nichts in diesem Fall. Man beachte, daß auch der Benutzer dieser Datenstruktur diese Frage anhand der Spezifikation klären kann; er muß sich nicht in den Programmcode vertiefen!

1. Man spricht dann auch von einem *Datenobjekt* anstelle eines Datentyps.
2. Diese Festlegung wird später bei der Implementierung geändert. Wir zeigen dies trotzdem als Beispiel dafür, daß Programmentwicklung nicht immer streng top-down verläuft, sondern daß gelegentlich Entwurfsentscheidungen höherer Ebenen zurückgenommen werden müssen.

algorithm *contains* (*x*)
führe binäre Suche im Bereich 1..*top* durch, wie in Abschnitt 1.1 beschrieben;
if *x* gefunden **then return** *true* **else return** *false* **end if**.

algorithm *isempty*
return (*top* = 0).

Wir können schon auf der algorithmischen Ebene das Verhalten dieser Datenstruktur analysieren, also vor bzw. ohne Implementierung! Im schlimmsten Fall entstehen folgende Kosten:

empty	O(1)	
insert	O(*n*)	(O(log *n*) für die Suche und O(*n*) für das Verschieben)
delete	O(*n*)	(ebenso)
contains	O(log *n*)	
isempty	O(1)	
Platzbedarf	O(*n*)	

Schließlich kommen wir zur Ebene der *Programmierung*. Manche Sprachen stellen Konstrukte zur Verfügung, die die Implementierung von ADTs (bzw. Algebren) unterstützen, z.B. Klassen (SIMULA, SMALLTALK, C++, Java), Module (Modula-2), Packages (ADA, Java), … Wir implementieren im folgenden unsere Datenstruktur in Java.

Zum ADT auf der Ebene der Spezifikation korrespondiert auf der Implementierungsebene die *Klasse*. Vereinfacht dargestellt[3] besteht eine Klassendefinition aus der Angabe aller Komponenten und der Implementierung aller Methoden der Klasse. Zudem wird zu jeder Komponente und Methode definiert, aus welchem Sichtbarkeitsbereich man auf sie zugreifen bzw. sie aufrufen darf. Die Deklaration einer Komponente oder Methode als *private* hat zur Folge, daß sie nur innerhalb von Methoden der eigenen Klasse verwendet werden können. Im Gegensatz dazu erlaubt die *public*-Deklaration den Zugriff von beliebiger Stelle.

Der *Implementierer* einer Klasse muß alle Einzelheiten der Klassendefinition kennen. Der *Benutzer* einer Klasse hingegen ist lediglich an ihrer Schnittstelle, d.h. an allen als public deklarierten Komponenten und Methoden, interessiert. Daß es sinnvoll ist, zwei verschieden detaillierte Sichten auf Implementierungen bereitzustellen, ist seit langem bekannt. Die verschiedenen Programmiersprachen verwenden dazu unterschiedliche Strategien. In Modula-2 ist der Programmierer gezwungen, die Schnittstelle in Form eines *Definitionsmoduls* anzugeben. Der Compiler prüft dann, ob das zugehörige *Implementationsmodul* zur Schnittstelle paßt. In C und C++ ist es üblich, Schnittstellen-

3. Dieser Kurs ist kein Java-Kurs. Grundlegende Java-Kenntnisse setzen wir voraus. Weitergehende Informationen entnehmen Sie bitte der entsprechenden Fachliteratur.

definitionen in *Header-Dateien* anzugeben. Im Unterschied zu Modula-2 macht der Compiler jedoch keine Vorgaben bezüglich der Benennung und der Struktur der verwendeten Dateien.

Java sieht keinen Mechanismus zur expliziten Trennung von Schnittstelle und Implementierung vor. Java-Klassendefinitionen enthalten stets komplette Implementierungen. Programme, die solche Klassen verwenden wollen, importieren nicht nur ein Definitionsmodul oder lesen eine Header-Datei ein, sondern importieren die komplette Klasse. In die Java-Entwicklungsumgebung ist jedoch das Werkzeug *javadoc* integriert, das aus einer Klassendefinition die Schnittstellenbeschreibung extrahiert und als HTML-Datei in übersichtlicher Formatierung zur Verfügung stellt. Klassen können darüber hinaus zu *Paketen* (*packages*) mit einer übergreifenden, einheitlichen Schnittstellenbeschreibung zusammengefaßt werden. Tabelle 1.5 stellt den Zusammenhang zwischen den verschiedenen Strategien und Begriffen dar.

Sichtbarkeits-bereich	Programmiersprache			Bedeutung
	Modula-2	C/C++	Java	
für Benutzer sichtbar	**Definitions-modul**	**Header-Datei**	**javadoc-Schnittstellen-beschreibung**	entspricht der Signatur einer Algebra
für Benutzer verborgen	**Implemen-tationsmodul**	**Implemen-tierung in Datei(en)**	**Klasse/Paket**	entspricht den Trägermengen und Funktionen einer Algebra

Tabelle 1.5: Sichtbarkeitsbereiche in Programmiersprachen

Die Implementierung einer Klasse kann geändert oder ausgetauscht werden, ohne daß der Benutzer (das heißt, ein Programmierer, der diese Klasse verwenden will) es merkt bzw. ohne daß das umgebende Programmsystem geändert werden muß, sofern die Schnittstelle sich nicht ändert. Eine von javadoc generierte Schnittstellenbeschreibung für unser Beispiel könnte dann so aussehen, wie in Abbildung 1.10 gezeigt.

Gewöhnlich wird in den Kommentaren noch genauer beschrieben, was die einzelnen Methoden leisten; das haben wir ja in diesem Fall bereits durch die Algebra spezifiziert. Es ist wesentlich, dem Benutzer hier die durch die Implementierung gegebene Einschränkung mitzuteilen, da ihm nur die Schnittstelle (und, wie wir annehmen, unsere Algebra-Spezifikation) bekannt gemacht wird.

Class IntSet

```
java.lang.Object
  |
  +--IntSet
```

---

public class **IntSet**
extends java.lang.Object

Diese Klasse implementiert eine Integer-Menge.

Einschränkung: Bei der aktuellen Implementierung kann die Menge maximal 100 Elemente enthalten.

Constructor Summary	
	`IntSet()` Initialisiert die leere Menge.

Method Summary	
boolean	`Contains(int elem)` Prüft das Vorhandensein des Elementes *elem*.
void	`Delete(int elem)` Löscht das Element *elem* aus der Menge.
void	`Insert(int elem)` Fügt das Element *elem* in die Menge ein.
boolean	`IsEmpty()` Prüft, ob die Menge leer ist.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

IntSet

`public IntSet()`

Initialisiert die leere Menge. Ersetzt die *empty*-Operation der Algebra.

Method Detail

Insert

`public void Insert(int elem)`

Fügt das Element *elem* in die Menge ein. Falls *elem* bereits in der Menge enthalten ist, geschieht nichts.

Delete

`public void Delete(int elem)`

Löscht das Element *elem* aus der Menge. Falls *elem* nicht in der Menge enthalten ist, geschieht nichts.

Contains

`public boolean Contains(int elem)`

Prüft das Vorhandensein des Elementes *elem*. Liefert *true* zurück, falls *elem* in der Menge enthalten ist, sonst *false*.

IsEmpty

`public boolean IsEmpty()`

Prüft, ob die Menge leer ist. Falls ja, wird *true* zurückgeliefert, sonst *false*.

Seite 1 Seite 2

Abbildung 1.10: Javadoc-Schnittstellenbeschreibung für unsere IntSet-Algebra

In der Klassendefinition sind zusätzliche Hilfsmethoden *find, shiftright, shiftleft* enthalten, die zwar Vereinfachungen für die Implementierung darstellen, nach außen aber nicht sichtbar sind.

```
public class IntSet
{
  static int maxelem = 100;
  int s[] = new int[maxelem];
  private int top = 0; /* Erster freier Index */

  private void shiftright(int j)
  /* Schiebt die Elemente ab Position j um ein Feld nach rechts, wenn möglich.
```

```
   Erhöht top. */
{
   if (top == maxelem) <Fehlerbehandlung>
   else
   {
      for (int i = top; i > j; i--)
         s[i] = s[i-1];
      top++;
   }
}

(shiftleft ähnlich)

private int find(int x, int low, int high)
/* Bestimme den Index j des ersten Elementes s[j] ≥ x im Bereich low bis
   high - 1. Falls x größer als alle gespeicherten Elemente ist, wird high
   zurückgegeben. */
{
   if (low > high-1) return high;
   else
   {
      int m = (low + high-1) / 2;
      if (s[m] == x) return m;
      if (s[m] > x) return find(x, low, m);
      return find(x, m+1, high);
   }
}

public IntSet(){}; /* Konstruktor */

public void Insert(int elem)
{
   if (top == maxelem) <Überlaufbehandlung>
   else
   {
      int j = find(elem, 0, top);
      if (j == top) {s[j] = elem; top++;}
      else
         if (s[j] != elem) {
            shiftright(j);
            s[j] = elem;
         }
   }
}
```

```
    public void Delete(int elem)
    {
      int j = find(elem, 0, top);
      if (j < top && s[j] == elem) shiftleft(j);
    }

    public boolean Contains(int elem)
    {
      int j = find(elem, 0, top);
      return (j < top && s[j] == elem);
    }

    public boolean IsEmpty()
    {
      return (top == 0);
    }

}
```

1.3 Grundbegriffe

In diesem Abschnitt sollen die bisher diskutierten Begriffe noch einmal zusammengefaßt bzw. etwas präziser definiert werden.

Ein *Algorithmus* ist ein Verfahren zur Lösung eines Problems. Ein *Problem* besteht jeweils in der Zuordnung eines Ergebnisses zu jedem Element aus einer Klasse von Probleminstanzen; insofern realisiert ein Algorithmus eine Funktion. Die Beschreibung eines Algorithmus besteht aus einzelnen Schritten und Vorschriften, die die Ausführung dieser Schritte kontrollieren. Jeder Schritt muß

- klar und eindeutig beschrieben sein und
- mit endlichem Aufwand in endlicher Zeit ausführbar sein.

In Büchern zu Algorithmen und Datenstrukturen wird gewöhnlich zusätzlich verlangt, daß ein Algorithmus für jede Eingabe (Probleminstanz) *terminiert*. Aus Anwendungssicht ist das sicher vernünftig. Andererseits werden in der theoretischen Informatik verschiedene Formalisierungen des Algorithmenbegriffs untersucht (z.B. Turingmaschinen, RAMs, partiell rekursive Funktionen, …) und über die Churchsche These mit dem intuitiven Algorithmenbegriff gleichgesetzt. All diese Formalisierungen sind aber zu nicht terminierenden Berechnungen in der Lage.

Algorithmische Beschreibungen dienen der Kommunikation zwischen Menschen; insofern kann Uneinigkeit entstehen, ob ein Schritt genügend klar beschrieben ist. Algorithmische Beschreibungen enthalten auch häufig abstrakte Spezifikationen einzelner Schritte; es ist dann im folgenden noch zu zeigen, daß solche Schritte endlich ausführbar sind.

Die endliche Ausführbarkeit ist der wesentliche Unterschied zur Spezifikation einer Funktion auf der Ebene der Mathematik. Die "auszuführenden Schritte" in der Definition einer Funktion sind ebenfalls präzise beschrieben, aber sie dürfen "unendliche Resourcen verbrauchen".

Beispiel 1.18: Ein Rechteck $r = (x_l, x_r, y_b, y_t)$ ist definiert durch die Punktmenge

$$r = \{(x, y) \in \mathrm{R}^2 \mid x_l \leq x \leq x_r \wedge y_b \leq y \leq y_t\}$$

Eine Relation "Rechteckschnitt" kann definiert werden:

$$r_1 \text{ schneidet } r_2 :\Leftrightarrow r_1 \cap r_2 \neq \varnothing$$

oder

$$r_1 \text{ schneidet } r_2 :\Leftrightarrow \exists\, (x, y) \in \mathrm{R}^2 : (x, y) \in r_1 \wedge (x, y) \in r_2$$

Diese Definitionen arbeiten mit unendlichen Punktmengen. Ein Algorithmus muß endliche Repräsentationen solcher Mengen in endlich vielen Schritten verarbeiten. □

Um eine Algebra formal zu beschreiben, benötigt man zunächst die Definition einer Signatur. Eine *Signatur* ist ein Paar (S, Σ), wobei S eine Menge ist, deren Elemente *Sorten* heißen, und Σ eine Menge von *Operationen* (oder *Operationssymbolen*). Σ ist die Vereinigung der Mengen $\sum_{w,s}$ in einer Familie von Mengen $\{\sum_{w,s} \mid w \in S^*, s \in S\}$, die jeweils mit der Funktionalität der in ihnen enthaltenen Operationssymbole indiziert sind. Es bezeichnet nämlich S^* die Menge aller Folgen beliebiger Länge von Elementen aus S. Die leere Folge heißt ε und liegt ebenfalls in S^*.

Beispiel 1.19: Formal würden die Spezifikationen

$$\begin{array}{lll} \mathit{insert}: & \mathit{intset} \times \mathit{int} & \rightarrow \mathit{intset} \\ \mathit{empty}: & & \rightarrow \mathit{intset} \end{array}$$

ausgedrückt durch

$$\begin{array}{ll} \mathit{insert} & \in \sum_{\mathit{intset\ int},\ \mathit{intset}} \\ \mathit{empty} & \in \sum_{\varepsilon,\ \mathit{intset}} \end{array}$$

□

Eine (*mehrsortige = heterogene*) *Algebra* ist ein System von Mengen und Operationen auf diesen Mengen. Sie ist gegeben durch eine Signatur (S, Σ), eine Trägermenge A_s für jedes $s \in S$ und eine Funktion

$$f_\sigma : A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n} \to A_s$$

für jedes $\sigma \in \Sigma_{s_1 \ldots s_n, s}$

Ein *abstrakter Datentyp* (ADT) besteht aus einer Signatur (S, Σ) sowie Gleichungen ("Axiomen"), die das Verhalten der Operationen beschreiben.

Selbsttestaufgabe 1.8: Für die ganzen Zahlen seien die Operationen

0 (Null),
succ (Nachfolger), *pred* (Vorgänger)
+, -, *

vorgesehen. Geben Sie hierzu einen abstrakten Datentyp an. □

Eine Algebra ist ein *Modell* für einen abstrakten Datentyp, falls sie die gleiche Signatur besitzt und ihre Operationen die Gesetze des ADT erfüllen. Ein *Datentyp* ist eine Algebra mit einer ausgezeichneten Sorte, die dem Typ den Namen gibt. Häufig bestimmt ein abstrakter Datentyp einen (konkreten) Datentyp, also eine Algebra, eindeutig. In diesem Fall heißt der ADT *monomorph*, sonst *polymorph*.

Unter einer *Datenstruktur* verstehen wir die *Implementierung eines Datentyps auf algorithmischer Ebene*. Das heißt, für die Objekte der Trägermengen der Algebra wird eine Repräsentation festgelegt und die Operationen werden durch Algorithmen realisiert.

Die Beschreibung einer Datenstruktur kann andere Datentypen benutzen, für die zugehörige Datenstrukturen bereits existieren oder noch zu entwerfen sind ("schrittweise Verfeinerung"). So entsteht eine Hierarchie von Datentypen bzw. Datenstrukturen. Ein Datentyp ist vollständig implementiert, wenn alle benutzten Datentypen implementiert sind. Letztlich müssen sich alle Implementierungen auf die elementaren Typen und Typkonstruktoren (Arrays, Records, ...) einer Programmiersprache abstützen.

Die *Implementierung einer Datenstruktur* in einer Programmiersprache, das heißt, die komplette Ausformulierung mit programmiersprachlichen Mitteln, läßt sich in manchen Sprachen zu einer Einheit zusammenfassen, etwa zu einer *Klasse* in Java.

1.4 Weitere Aufgaben

Aufgabe 1.9: Gegeben sei eine Folge ganzer Zahlen $s_1, \ldots, s_n$, deren Werte alle aus einem relativ kleinen Bereich $[1..N]$ stammen ($n >> N$). Es ist zu ermitteln, welche Zahl in der Folge am häufigsten vorkommt (bei mehreren maximal häufigen Werten kann ein beliebiger davon ausgegeben werden).

Lösen Sie dieses Problem auf den drei Abstraktionsebenen, das heißt, geben Sie Funktion, Algorithmus und Programm dazu an.

Aufgabe 1.10: Entwerfen Sie einen kleinen Instruktionssatz für die Random-Access-Maschine. Ein Befehl kann als Paar (b, i) dargestellt werden, wobei b der Befehlsname ist und i eine natürliche Zahl, die als Adresse einer Speicherzelle aufgefaßt wird (ggf. kann der Befehlsname indirekte Adressierung mitausdrücken). Der Befehlssatz sollte so gewählt werden, daß die folgende Aufgabe 1.11 damit lösbar ist. Definieren Sie für jeden Befehl seine Wirkung auf Programmzähler und Speicherzellen.

Aufgabe 1.11: Implementieren Sie den Algorithmus $contains_2$ auf einer RAM mit dem in Aufgabe 1.10 entwickelten Befehlssatz. Wieviele RAM-Instruktionen führt dieses RAM-Programm im besten Fall, im schlimmsten Fall und im Durchschnitt aus?

Aufgabe 1.12: Beweisen Sie die Behauptungen aus Abschnitt 1.1

(a) $\forall\, k > 0$: $n^k = o(2^n)$
(b) $2^{n/2} = o(2^n)$

Aufgabe 1.13: Gegeben sei eine Zahlenfolge $S = s_1, \ldots, s_n$, von der bekannt ist, daß sie eine Permutation der Folge $1, \ldots, n$ darstellt. Es soll festgestellt werden, ob in der Folge S die Zahlen 1, 2 und 3 gerade in dieser Reihenfolge stehen. Ein Algorithmus dazu geht so vor: Die Folge wird durchlaufen. Dabei wird jedes Element überprüft, ob es eine der Zahlen 1, 2 oder 3 ist. Sobald entschieden werden kann, ob diese drei Zahlen in der richtigen Reihenfolge stehen, wird der Durchlauf abgebrochen.

Wie weit wird die Folge von diesem Algorithmus im Durchschnitt durchlaufen unter der Annahme, daß alle Permutationen der Folge $1, \ldots, n$ gleich wahrscheinlich sind? (Hier ist das exakte Ergebnis gefragt, das heißt, $O(n)$ ist keine richtige Antwort.)

Aufgabe 1.14: Gegeben seien Programme P_1, P_2, P_3 und P_4, die auf einem Rechner R Laufzeiten

$$T_1(n) = a_1\, n$$
$$T_2(n) = a_2\, n \log n$$
$$T_3(n) = a_3\, n^3$$

$$T_4(n) = a_4\, 2^n$$

haben sollen, wobei die a_i Konstanten sind. Bezeichne für jedes Programm m_i die Größe der Eingabe, die innerhalb einer fest vorgegebenen Zeit T verarbeitet werden kann. Wie ändern sich die m_i, wenn der Rechner R durch einen 10-mal schnelleren Rechner R' ersetzt wird?

(Diese Aufgabe illustriert den Zusammenhang zwischen algorithmischer Komplexität und Technologiefortschritt in Bezug auf die Größe lösbarer Probleme.)

Aufgabe 1.15: Sei n die Anzahl verschiedener Seminare, die in einem Semester stattfinden. Die Seminare seien durchnumeriert. Zu jedem Seminar können sich maximal m Studenten anmelden. Vorausgesetzt sei, daß die Teilnehmer alle verschiedene Nachnamen haben. Um die Anmeldungen zu Seminaren verwalten zu können, soll ein Datentyp "Seminare" entwickelt werden, der folgende Operationen bereitstellt:

- "Ein Student meldet sich zu einem Seminar an."
- "Ist ein gegebener Student in einem bestimmten Seminar eingeschrieben?"
- "Wieviele Teilnehmer haben sich zu einem gegebenen Seminar angemeldet?"

(a) Spezifizieren Sie eine Algebra für diesen Datentyp.
(b) Implementieren Sie die Spezifikation, indem Sie die Anmeldungen zu Seminaren als zweidimensionalen Array darstellen und für die Operationen entsprechende Algorithmen formulieren.
(c) Implementieren Sie die in (b) erarbeitete Lösung in Java.

1.5 Literaturhinweise

Zu Algorithmen und Datenstrukturen gibt es eine Fülle guter Bücher, von denen nur einige erwähnt werden können. Hervorheben wollen wir das Buch von Aho, Hopcroft und Ullman [1983], das den Aufbau und die Darstellung in diesem Kurs besonders beeinflußt hat. Wichtige "Klassiker" sind [Knuth 1998], [Aho *et al.* 1974] und Wirth [2000, 1996] (die ersten Auflagen von Knuth und Wirth sind 1973 bzw. 1975 erschienen). Ein hervorragendes deutsches Buch ist [Ottmann und Widmayer 2012]. Auch [Cormen et al. 2010] ist sehr zu empfehlen.

Eine gute Einführung in Algorithmen und Datenstrukturen auf Basis der Sprache Java bietet [Saake und Sattler 2010]. Weitere gute Darstellungen finden sich in [Mehlhorn 1984a-c], [Horowitz, Sahni und Anderson-Freed 2007], [Sedgewick 2002a, 2002b] und [Wood 1993]. Manber [1989] betont den kreativen Prozess bei der Entwicklung von Algorithmen, beschreibt also nicht nur das Endergebnis. Gonnet und Baeza-Yates [1991]

stellen eine große Fülle von Algorithmen und Datenstrukturen jeweils knapp dar, bieten also so etwas wie einen "Katalog". Die Analyse von Algorithmen wird besonders betont bei Baase und Van Gelder [2000] und Banachowski *et al.* [1991]. Nievergelt und Hinrichs [1993] bieten eine originelle Darstellung mit vielen Querverbindungen und Themen, die man sonst in Büchern zu Algorithmen und Datenstrukturen nicht findet, u.a. zu Computergraphik, geometrischen Algorithmen und externen Datenstrukturen.

Einige Bücher zu Datenstrukturen haben Versionen in einer ganzen Reihe von Programmiersprachen, etwa in PASCAL, C, C++ oder Java, so z.B. Sedgewick [2002a, 2002b], Standish [1998] oder Weiss [2009].

Die bei uns für die Ausformulierung konkreter Programme verwendete Sprache Java ist z.B. in [Flanagan 2005] beschrieben.

Eine ausgezeichnete Darstellung mathematischer Grundlagen und Techniken für die Analyse von Algorithmen bietet das Buch von Graham, Knuth und Patashnik [1994]. Wir empfehlen es besonders als Begleitlektüre. Unser Material zu mathematischen Grundlagen im Anhang kann man dort, natürlich wesentlich vertieft, wiederfinden.

Eine gründliche Einführung in die Theorie und Praxis der Analyse von Algorithmen mit einer Darstellung möglicher Maschinenmodelle wie der RAM findet sich bei [Aho *et al.* 1974]. "Registermaschinen" werden auch bei Albert und Ottmann [1990] diskutiert; das Konzept stammt aus einer Arbeit von Sheperdson und Sturgis [1963]. Die "real RAM" wird bei Preparata und Shamos [1985] beschrieben.

Abstrakte Datentypen und algebraische Spezifikation werden in den Büchern von Ehrich *et al.* [1989] und Klaeren [1983] eingehend behandelt. Die von uns verwendete Spezifikationsmethode (direkte Beschreibung einer Algebra mit allgemeiner mathematischer Notation) wird dort als "exemplarische applikative Spezifikation" bzw. als "denotationelle Spezifikation" bezeichnet und in einführenden Kapiteln kurz erwähnt; die Bücher konzentrieren sich dann auf die formale Behandlung abstrakter Datentypen. Auch Loeckx *et al.* [1996] bieten eine umfassende Darstellung des Gebietes; der Zusammenhang zwischen Signatur, mehrsortiger Algebra und abstraktem Datentyp wird dort in Kapitel 2 beschrieben. Eine gute Einführung zu diesem Thema findet sich auch bei Bauer und Wössner [1984]. Ein Buch zu Datenstrukturen, in dem algebraische Spezifikation für einige grundlegende Datentypen durchgeführt wird, ist [Horowitz *et al.* 1993]. Auch Sedgewick [2002a, 2002b] betont abstrakte Datentypen und zeigt die Verbindung zu objekt-orientierter Programmierung, also Klassen in C++. Wood [1993] arbeitet systematisch mit abstrakten Datentypen, wobei jeweils die Signatur angegeben und die Semantik der Operationen möglichst präzise umgangssprachlich beschrieben wird.

2 Programmiersprachliche Konzepte für Datenstrukturen

Im ersten Kapitel haben wir gesagt, daß wir unter einer Datenstruktur die Implementierung eines Datentyps auf algorithmischer Ebene verstehen wollen. Die Implementierung stützt sich letztendlich ab auf Primitive, die von einer Programmiersprache zur Verfügung gestellt werden. Diese Primitive sind wiederum Datentypen, eben die *Typen* der Programmiersprache. Man beachte die etwas andere Bedeutung dieses Typ-Begriffs im Vergleich zum allgemeinen Begriff des Datentyps aus Kapitel 1. Dort geht man von der Anwendungssicht aus, um einen Typ, also eine Objektmenge mit zugehörigen Operationen festzulegen, und bemüht sich dann um eine effiziente Implementierung. Es kann durchaus verschiedene Implementierungen geben. Beim Entwurf der Typen einer Programmiersprache überlegt man, was einigermaßen einfach und effizient zu realisieren ist und andererseits interessante Objektstrukturen und Operationen zur Verfügung stellt. Die Implementierung solcher Typen ist durch den Compiler festgelegt, also bekannt. Die Effizienzüberlegungen haben z.B. zur Folge, daß viele Compiler für jeden Wert eines Typs eine Repräsentation in einem zusammenhängenden Speicherblock festlegen, also jeden Wert auf eine Bytefolge abbilden. Im Gegensatz dazu liegt beim allgemeinen Begriff des Datentyps für dessen Werte eine Repräsentation zunächst überhaupt nicht fest; sie muß auch keineswegs in einem zusammenhängenden Speicherbereich erfolgen.

Die Rolle des *Arrays* ist nicht ganz klar. Er wird in der Literatur bisweilen als eigenständiger, aus Anwendungssicht interessanter Datentyp angesehen (z.B. in [Horowitz *et al.* 1993]). Wir verstehen den Array als programmiersprachliches Werkzeug mit *fester* Implementierung und besprechen ihn deshalb in diesem Kapitel. Ein entsprechender Datentyp *Abbildung* wird in Kapitel 3 eingeführt; für diesen ist der Array eine von mehreren Implementierungsmöglichkeiten.

Die wesentlichen Werkzeuge, die man zur Konstruktion von Datenstrukturen braucht, sind folgende:

- die Möglichkeit, variabel viele Objekte gleichen Typs aneinanderzureihen und in O(1) Zeit auf Objekte einer Reihung zuzugreifen (*Arrays*),
- die Möglichkeit, einige Objekte verschiedenenen Typs zu einem neuen Objekt zusammenzufassen und daraus die Komponentenobjekte zurückzuerhalten (*Records*), zur Implementation der *Aggregation* bzw. der Tupelbildung aus der Mathematik, und
- die Möglichkeit, Objekte zur Laufzeit des Programms zu erzeugen und zu referenzieren.

Diese Möglichkeiten werden von allen gängigen imperativen Programmiersprachen (wie PASCAL, Modula-2, C, C++, Java, Ada, …) innerhalb ihres *Typsystems* angeboten. Dieses enthält grundsätzlich folgende Konzepte zur Bildung von Typen:

- atomare Typen
- Typkonstruktoren / strukturierte Typen
- Zeigertypen / Referenztypen

Wir besprechen diese Konzepte kurz in den folgenden Abschnitten, da sie die Grundlage für die Implementierung der uns interessierenden Anwendungs-Datentypen bilden.

In objektorientierten Sprachen wie C++ und Java ist darüber hinaus die *Vererbung* ein wichtiger Bestandteil des Typsystems. So bedeutend die Vererbung als wesentliches Strukturierungskonzept beim Entwurf und der robusten Implementierung großer Softwaresysteme auch ist, spielt sie bei der bewußt isolierten Betrachtung algorithmischer Probleme, wie wir sie in diesem Kurs vornehmen werden, jedoch kaum eine Rolle. Deshalb werden wir von ihr in den Algorithmen und Programmbeispielen auch nur wenig Gebrauch machen.

In diesem Kurs verwenden wir Java als Programmiersprache für Implementierungsbeispiele. Deshalb beschreiben wir in Abschnitt 2.1 zunächst die Möglichkeiten zur Konstruktion von Datentypen in Java. Die Implementierung dynamischer Datenstrukturen ist Gegenstand von Abschnitt 2.2. Auf weitere interessante Konzepte, wie sie von anderen imperativen und objektorientierten Sprachen angeboten werden, gehen wir in Abschnitt 2.3 kurz ein. Begleitend erläutern wir in diesem Kapitel die programmiersprachenunabhängige Notation der jeweiligen Konzepte in den Algorithmen dieses Kurses.

2.1 Datentypen in Java

Java unterscheidet drei verschiedene Kategorien von Datentypen: *Basisdatentypen* (*primitive data types*), *Array-Typen* und *Klassen*. Array-Typen und Klassen werden auch unter dem Oberbegriff *Referenztypen* zusammengefaßt (die Begründung liefert Abschnitt 2.2). Die *Instanzen* von Basisdatentypen nennt man *Werte*, Instanzen von Array-Typen heißen *Arrays,* und Instanzen von Klassen werden *Objekte* genannt. Tabelle 2.1 faßt die Beziehungen zwischen diesen Begriffen noch einmal zusammen.

In den folgenden Abschnitten behandeln wir Basisdatentypen, Array-Typen und Klassen etwas genauer.

Typen		Instanzen
Basisdatentypen		Werte
Referenztypen	Array-Typen	Arrays
	Klassen	Objekte

Tabelle 2.1: Datentypen und Instanzen in Java

2.1.1 Basisdatentypen

Jede übliche Programmiersprache gibt als *atomare* Datentypen die Standard-Typen wie *integer*, *real*, *boolean*, *char* usw. mit entsprechenden Operationen vor, also z.B.

$integer \times integer$	$\rightarrow integer$	+, -, *, **div**, **mod**
$real \times real$	$\rightarrow real$	+, -, *, /
$boolean \times boolean$	$\rightarrow boolean$	**and**, **or**
$boolean$	$\rightarrow boolean$	**not**
$integer \times integer$	$\rightarrow boolean$	$=, \neq, <, \leq, >, \geq$
$real \times real$	$\rightarrow boolean$	$=, \neq, <, \leq, >, \geq$
$char \times char$	$\rightarrow boolean$	$=, \neq, <, \leq, >, \geq$

Im letzten Fall (Vergleichsoperationen auf *char*) wird eine Ordnung auf der Menge der Zeichen unterstellt, da andernfalls nur Gleichheit und Ungleichheit festgelegt werden kann. Diese Ordnung kann z.B. gemäß dem ASCII-Alphabet definiert sein. Der Vergleichsoperator "=" ist grundsätzlich auf allen atomaren Typen definiert.

Ein Wert eines atomaren Typs ist gewöhnlich in *einem* Speicherwort, Byte oder Bit repräsentiert, bisweilen auch in einer kleinen, festen Zahl von Wörtern. Typen wie *integer*, *real*, *boolean* realisieren jeweils die entsprechende aus der Mathematik bekannte Algebra (Z, R, B) bis auf Einschränkungen bzgl. der Größe des Wertebereichs und der Genauigkeit der Darstellung. *Wie* Werte im Rechner repräsentiert und Operationen darauf ausgeführt werden, fällt in die Gebiete Rechnerarchitektur und Compilerbau.

Java stellt die atomaren Standard-Typen in Form der *Basisdatentypen* zur Verfügung. Sie sind in Tabelle 2.2 aufgeführt.

Wie man sieht, existieren verschiedene integer- und real-Typen in Java, die sich durch die Länge ihrer Darstellung im Speicher und somit in bezug auf den darstellbaren Wertebereich und die darstellbare Genauigkeit voneinander unterscheiden. Es ist Aufgabe des Programmierers, bei der Umsetzung eines Algorithmus in ein Java-Programm einen geeigneten Typ zu wählen.

Standard-Typ	Java-Basisdatentyp
integer	int (32 Bit) long (64 Bit) byte (8 Bit) short (16 Bit)
real	float (32 Bit) double (64 Bit)
boolean	boolean
char	char

Tabelle 2.2: Standard-Typen vs. Java-Basisdatentypen

Alle zuvor vorgestellten Operationen auf Standard-Typen gibt es auch für Java-Basisdatentypen. Lediglich die Notation weicht teilweise leicht ab. So wird der Gleichheitsoperator durch zwei hintereinanderfolgende Gleichheitszeichen notiert (`==`), der Modulo-Operator durch das Prozentzeichen (`%`). Generell ist Java der Programmiersprache C++ sehr ähnlich.

2.1.2 Arrays

Arrays sind Felder (oder *Reihungen*) von Werten zu einem festen Grundtyp. Auf die einzelnen Elemente dieser Felder kann über einen Index zugegriffen werden.

In Java definiert die Anweisung

T[] *V* = **new** *T*[*n*]

eine Array-Variable *V*. Der Array besteht aus *n* Elementen vom Grundtyp *T*. Der Grundtyp ist ein beliebiger Typ, *n* ein ganzzahliger Wert.

```
char[] zeile = new char[80];
char[][] seite = new char[25][80];
```

Die wesentliche angebotene Operation ist die *Selektion*, die bei Angabe eines Indexwertes eine Variable des Grundtyps zurückliefert. In Java hat das erste Element eines Arrays immer die Position 0. Beim Zugriff auf Elemente eines *n*-elementigen Arrays *V* mittels

V[*i*]

muß für *i* deshalb stets gelten: $0 \leq i < n$.

Mit der Deklaration

int *i*, *j*;

bezeichnet

zeile[*i*] eine Variable vom Typ *char*,
seite[*j*] eine Variable vom Typ *char*[] und
seite[*j*][*i*] eine Variable vom Typ *char*.

Solche Variablen sind wie gewöhnliche Variablen in Zuweisungen, Vergleichen, arithmetischen Operationen usw. verwendbar, z.B.

zeile[0] = 'a'; *zeile*[79] = '0'; **if** (*zeile*[*i*] == 'x') ...

Der Wertebereich eines Array-Typs ist das *homogene* kartesische Produkt des Wertebereichs des Grundtyps

$$W(T) = \underbrace{W(T_0) \times W(T_0) \times \ldots \times W(T_0)}_{n\text{-mal, } n \text{ ist die Kardinalität des Indextyps}}$$

Arrays werden im Speicher durch das Aneinanderreihen der Repräsentationen des Grundtyps dargestellt. Auf diese Weise entsteht für ein Array des Typs T zum Grundtyp T_0, dessen erstes Element an der Adresse a_0 abgelegt wird, die in Abbildung 2.1 gezeigte Speicherrepräsentation, wobei $R(T_0)$ die Repräsentation eines Elementes des Grunddatentyps ist und $R(T)$ die des gesamten Arrays.

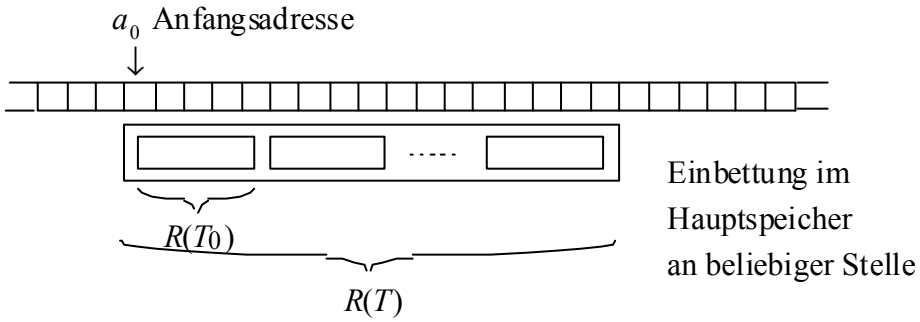

Abbildung 2.1: Speicherrepräsentation von Arrays

Daraus ergibt sich, daß die Adresse der i-ten Komponente

$a_0 + (i - 1) * size\ (T_0)$

ist, wenn $size\ (T_0)$ die Größe eines Objekts des Typs T_0 ist. Es folgt, daß der Zugriff auf jede Komponente eines Arrays beliebiger Größe in O(1) Zeit möglich ist; falls die Kom-

ponente ein atomares Objekt ist, kann sie daher in konstanter Zeit gelesen oder geschrieben werden — eine entscheidende Eigenschaft für den Entwurf effizienter Algorithmen.

In Abschnitt 2.2 werden wir sehen, daß Java im Gegensatz zu vielen anderen Programmiersprachen nur die Basisdatentypen als zusammenhängenden Block im Hauptspeicher repräsentiert. Arrays und Objekte hingegen werden ohne Einflußmöglichkeit des Programmierers immer durch einen Zeiger dargestellt, der auf den eigentlichen Wert verweist. Insofern sind Arrays oder Klassen nicht als atomare Objekte repräsentiert. Im Vorgriff auf Abschnitt 2.2 sei hier jedoch schon einmal darauf hingewiesen, daß dennoch auch in Java alle Array-Zugriffe in O(1) Zeit erfolgen, da auch im Falle von Nicht-Basisdatentypen als Array-Komponenten nur genau *eine* Indirektion bei einem Zugriff verfolgt werden muß.

Wir haben schon in der Einleitung gesehen, daß sich Arrays als Werkzeug zur Darstellung von Mengen von Objekten einsetzen lassen. Mit Arrays kann man auch direkt ein- oder mehrdimensionale lineare Anordnungen (*Felder*) von Objekten der "realen" oder einer gedachten Welt darstellen, beispielsweise Vektoren oder Matrizen der Mathematik oder etwa das folgende Labyrinth (Abbildung 2.2):

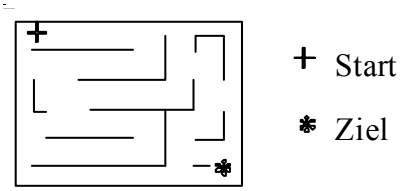

Abbildung 2.2: Labyrinth

```
final int maxh = ..., maxv = ...;
boolean Labyrinth [][] = new boolean [maxh][maxv];

boolean wand (int i, int j)
{
   return Labyrinth [i][j];
}
```

Eine der Verarbeitung von Arrays "angepaßte" Kontrollstruktur in imperativen Programmiersprachen ist die *for*-Schleife. Damit kann man z.B. den Anteil der durch Wände belegten Fläche im Labyrinth berechnen (Ergebnis in Prozent):

```
int f = 0;
int flaeche;
```

```
for(int i = 0; i < hrange; i++)
   for (int j = 0; j < vrange; j++)
      if (wand (i, j)) f = f + 1;
flaeche = (f * 100) / (hrange * vrange);
```

Auch in Algorithmen verwenden wir Arrays. Da es jedoch häufig natürlicher ist, den Arrayindex bei einer anderen Zahl als 0 beginnen zu lassen, verwenden wir in Algorithmen folgende Notation zur Definition eines Arraytyps T:

type T = **array** [$min..max$] **of** T_0

Bei min und max, $min < max$, handelt es sich um Konstanten vom Typ *integer*. Beim Zugriff auf Elemente eines Arrays V vom Typ T mittels

$V[i]$

muß für i stets gelten: $min \leq i \leq max$. Die Größe des Arrays, also die Anzahl von Elementen des Grundtyps T_0, ergibt sich zu $max - min + 1$.

In Java ist es nicht möglich, den Indexbereich mit einer anderen Zahl als 0 zu beginnen. Wenn ein Algorithmus Arrays mit anderen Indexbereichen spezifiziert, muß der Programmierer bei der Array-Definition und beim Zugriff auf Elemente des Arrays in Java eine entsprechende Transformation der Indexwerte vornehmen. Immerhin führt Java — im Gegensatz zu C oder C++ — eine Bereichsüberprüfung beim Zugriff auf Array-Elemente durch.

2.1.3 Klassen

Mit dem Begriff der *Klasse* verbinden sich in Java drei wesentliche Konzepte:

1. *Aggregation*: Eine Klasse definiert einen neuen Datentyp, der mehrere Einzelobjekte unterschiedlichen Typs zu einem "großen" Objekt zusammenfaßt. Die Einzelobjekte werden auch *Attribute* genannt.

2. *Kapselung*: Zusätzlich zu den Attributen gibt eine Klassendefinition an, welche *Methoden* auf Instanzen der Klasse angewandt werden können. Die *Implementierung* der Methoden enthält dann typischerweise Zugriffe auf die Klassenattribute. Der Benutzer einer Klasse muß nur die Schnittstellen ihrer Methoden kennen; die Implementierung der Methoden und — bei konsequenter Kapselung — die Attribute der Klasse sind für den Benutzer unwichtig. Kapselung ist ein wichtiges Werkzeug, um Programme zur Lösung komplexer Probleme überschaubar zu halten. Außerdem ermöglicht die Kapselung, lokale Änderungen von Klas-

senimplementierungen, ohne dadurch eine Reimplementierung der Umgebung nötig zu machen, da die Schnittstellen unverändert bleiben.

3. *Vererbung*: Eine Klasse kann ihre Eigenschaften (Attribute und Methoden) an andere Klassen vererben, die den ursprünglichen Eigenschaften neue Attribute und Methoden hinzufügen. Im vorliegenden Kurs werden wir die Vererbung trotz ihres anerkannten Nutzens bei der Strukturierung großer Programme nur selten verwenden, da wir relativ isoliert algorithmische Probleme und ihre Lösungen behandeln, nicht deren Einbettung in große Softwaresysteme. Unsere Algorithmen sind programmiersprachenunabhängig formuliert und verzichten deshalb ohnehin auf Verwendung von Konzepten, die spezielle Eigenschaften einer Programmiersprache voraussetzen.

Eine Klassendefinition hat folgende Grundstruktur:

Modifikatorenliste `class` *Klassenname*
{
  Attributdeklarationen
  Konstruktordeklarationen
  Methodendeklarationen
}

Die Modifikatorenliste enthält Schlüsselwörter wie *public*, *final* oder *static*, die die standardmäßige Verwendbarkeit einer Klasse modifizieren. Im Rahmen dieses Kurses gehen wir nicht weiter darauf ein. Die *Attributdeklarationen* geben die Komponenten der Klasse an. Klassenkomponenten werden auch *Attribute*, *Member-Variablen* oder *Members* genannt. Die Attributdeklarationen bestimmen also, welche Unterobjekte zu einer neuen Klasse aggregiert werden.

Konstruktordeklarationen legen fest, wie neue Instanzen einer Klasse erzeugt werden können. *Methodendeklarationen* enthalten die Schnittstellen der Methoden einer Klasse. Methoden sind zunächst nichts anderes als Funktionen oder Prozeduren, wie man sie aus vielen, auch nicht objekt-orientierten, Programmiersprachen kennt. Man ruft sie mit bestimmten Parametern auf, deren Typ in der Schnittstelle der Funktion festgelegt wird, und sie liefern ein Ergebnis, dessen Typ ebenfalls aus der Schnittstellendefinition bekannt ist. Das besondere bei Methoden ist nun, daß ihnen als impliziter Parameter, der nicht in der Schnittstelle auftaucht, auch die Klasseninstanz übergeben wird, auf die sie im Methodenaufruf angewendet wird. Dadurch ist es möglich, innerhalb einer Methode die Attribute der Klasseninstanz zu verwenden. Der implizite Parameter ist dabei nicht der unsichtbaren Verwendung durch das Java-Laufzeitsystem vorbehalten, sondern kann vom Programmierer über den automatisch vergebenen Namen *this* wie jeder andere Parameter verwendet werden.

Konstruktoren und Methoden ermöglichen uns die direkte Umsetzung der Signatur einer Algebra oder eines abstrakten Datentyps in eine Java-Klasse. Die "konstruierenden" Operationen der Signatur (in unseren Beispielen oft die parameterlose Operation *empty*) werden als Konstruktoren implementiert, alle übrigen Operationen werden zu Methoden der Klasse. Die Parameter der Methoden entsprechen den Parametern der Operatoren. Die Methodenparameter enthalten allerdings nicht den "Hauptparameter" der Operationen, da dieser als impliziter Parameter automatisch übergeben wird. Ein Beispiel haben wir in Kapitel 1 in Abbildung 1.10 mit der Java-Schnittstelle zur *intset*-Algebra bereits kennengelernt.

Während Konstruktoren und Methoden als Bestandteile von Typdefinitionen nur in objektorientierten Sprachen zu finden sind, ist die Aggregation von Komponenten eine Fähigkeit jeder höheren Programmiersprache. In C heißt eine Aggregation beispielsweise *struct*, während sie in vielen anderen Sprachen, darunter auch Pascal und Modula-2, *Record* genannt wird. Unter diesem Namen werden wir sie auch in den programmiersprachenunabhängig formulierten Algorithmen in diesem Kurs verwenden. Im folgenden gehen wir näher auf Aufbau und Eigenschaften von Records ein.

Records konstruieren aus einer gegebenen Menge verschiedener Datentypen einen neuen Datentyp. Ein Record-Typ T wird mit

```
type T = record   s_1: T_1;
                  ...
                  s_n : T_n
         end
```

definiert, wobei T_1, ..., T_n beliebige Typen (die Typen der Komponenten) sind und s_1, ..., s_n hiermit definierte Namen, die für den Zugriff auf die Komponenten benutzt werden.

```
type  complex  = record  re : real;     {Realteil der komplexen Zahl}
                         im : real      {Imaginärteil der komplexen Zahl}
                 end
type  Punkt    = record  x : real; y : real end;
type  Datum    = record  Tag: [1..31];
                         Monat : [1..12];
                         Jahr : [0..2099]
                 end
type  Person   = record  Nachname, Vorname: string;
                         Alter: integer;
                         GebDatum: Datum;
                         Geschlecht : (männlich,weiblich)
                 end
```

(mit **type** *string* = **array** [1..*maxlength*] **of** *char*)

Wie bei Arrays ist auch bei Records die wesentliche Operation die *Selektion* von Komponenten. Mit den obigen Definitionen und der Deklaration

var *p* : *Person*;

liefert

p.Nachname	eine Variable vom Typ *string*,
p.Nachname [*i*]	eine Variable vom Typ *char*,
p.GebDatum	eine Variable vom Typ *Datum* und
p.GebDatum.Tag	eine Variable vom Typ [1..31].

Diese Variablen können wie gewöhnliche Variablen in Zuweisungen, Vergleichen, Ausdrücken usw. benutzt werden.

Der Wertebereich eines Record-Typs ist das heterogene kartesische Produkt der Wertebereiche seiner Grundtypen. Als Wertebereich für einen wie oben definierten Record-Typ ergibt sich damit:

$$W(T) = W(T_1) \times W(T_2) \times ... \times W(T_n)$$

Wiederum erfolgt die Repräsentation durch Aneinanderreihen der Repräsentationen von Werten der Grundtypen. Diese sind aber nun im allgemeinen unterschiedlich groß:

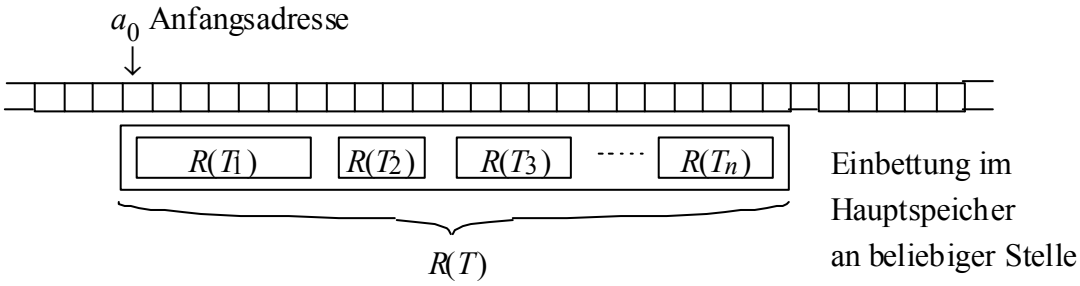

Abbildung 2.3: Speicherrepräsentation von Records

Der Compiler der verwendeten Programmiersprache kennt die Größen der Repräsentationen der Grundtypen und kann daher beim Übersetzen der Typdeklaration für jeden Selektor s_i (= Komponentennamen) ein *offset* berechnen, mit Hilfe dessen man aus der Anfangsadresse des Records die Anfangsadresse der selektierten Komponente berechnen kann:

$$offset\,(s_i) = \sum_{j=1}^{i-1} size\,(T_j)$$

Für Record r vom Typ T mit der Anfangsadresse a_0 ist daher die Adresse von $r.s_i$

$a_0 + offset\,(s_i)$.

Wie bei Arrays ist also der Zugriff auf Komponenten in konstanter Zeit möglich. Da die Größe von Records fest ist (ein Record also nicht mit der Größe des betrachteten algorithmischen Problems wächst), ist dies allerdings nicht ganz so spannend wie bei Arrays.

2.2 Dynamische Datenstrukturen

Alle mit den bisherigen Mitteln konstruierbaren Datenstrukturen sind *statisch*, das heißt, während der Laufzeit eines Programms kann sich zwar ihr Wert, nicht aber ihre *Größe oder Struktur* ändern. Das klassische Mittel zur Implementierung *dynamischer Datenstrukturen*, die während eines Programmablaufs beliebig wachsen oder schrumpfen und ihre Struktur in gewissen Grenzen verändern können, sind *Zeigertypen*. In Abschnitt 2.2.1 wird das allgemeine Prinzip der Verwendung von Zeigertypen zur Konstruktion und Manipulation dynamischer Datenstrukturen vorgestellt. Abschnitt 2.2.2 beschreibt dann die Umsetzung des allgemeinen Konzepts mit Hilfe von Javas Referenztypen.

2.2.1 Programmiersprachenunabhängig: Zeigertypen

Ein Zeiger (*Pointer*) ist ein Verweis auf eine Objektrepräsentation im Hauptspeicher, de facto nichts anderes als eine Adresse. Die Einführung von Zeigertypen und Zeigervariablen erlaubt aber ein gewisses Maß an Kontrolle des korrekten Umgangs mit solchen Adressen durch den Compiler. Ein Zeiger-Typ auf ein Objekt vom Typ T_0 wird definiert als[1]:

type $T = \uparrow T_0$

Der Wertebereich von T ist dynamisch: es ist die Menge aller Adressen von Objekten des Grundtyps T_0, die im bisherigen Verlauf des Programms dynamisch erzeugt worden sind. Zusätzlich gibt es einen speziellen Wert *nil* (in Java *null*), der Element jedes Zeigertyps

1. Wir benutzen hier die PASCAL-Notation, in Modula-2 wäre **pointer to** T_0 zu schreiben, in C und C++ lautete die Anweisung **typedef** T_0 * T.

ist. Eine Zeigervariable hat den Wert *nil*, wenn sie "auf nichts zeigt". Mit der obigen Definition deklariert

var *p* : *T*;

eine Zeigervariable, die auf Objekte vom Typ T_0 zeigen kann.

Die Anweisung *new* (*p*) erzeugt eine neue, unbenannte ("anonyme") Variable vom Typ T_0, das heißt, sie stellt im Hauptspeicher irgendwo Speicherplatz zur Aufnahme eines Wertes vom Typ T_0 bereit und *weist die Adresse dieses Speicherplatzes der Zeigervariablen p zu*.

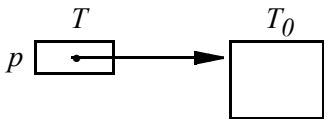

Abbildung 2.4: Zeiger *p*

Werte von Zeigertypen (also Adressen) werden in einem Speicherwort repräsentiert. Operationen auf Zeigervariablen sind Zuweisung, Vergleich und *Dereferenzierung*:

```
var p, q : T;
p := nil;   q := p;
if p = q then ...
```

Mit Dereferenzierung bezeichnet man die Operation, die ein Objekt vom Typ Zeiger auf das Objekt abbildet, auf das es zeigt. Wie in PASCAL und Modula-2 wird in unseren Algorithmen als Dereferenzierungsoperator ↑ benutzt. Mit den obigen Deklarationen bezeichnet also $p\uparrow$ das Objekt, auf das *p* zeigt, ist also eine Variable vom Typ T_0.

```
type Person' = record   Vorname : string;
                        Vater   : ↑ Person';
                        Mutter  : ↑ Person'
               end;
```

Nachdem zu diesem Typ zwei Variablen deklariert worden sind und ihnen zunächst der Wert *nil* zugewiesen wird, ergibt sich das in Abbildung 2.5 gezeigte Speicherabbild:

```
var p, q : ↑ Person';
p := nil;   q := nil;
```

Abbildung 2.5: Leere Zeiger *p* und *q*

Nun muß zunächst Speicherplatz für die Objekte vom Typ *Person'* bereitgestellt werden, und diesen Objekten müssen Werte zugewiesen werden:

new (*p*); *p*↑.*Vorname* := "Anna"; *p*↑.*Vater* := *nil*; *p*↑.*Mutter* := *nil*;

Danach erhalten wir folgende Situation:

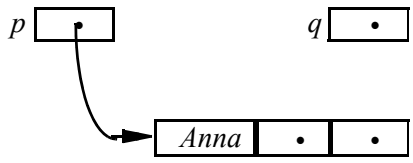

Abbildung 2.6: Zeiger *p* verweist auf *Person' mit p*↑.*Vorname* = "Anna"

Ebenso wird ein weiteres Objekt des Typs *Person'* erzeugt und *q* zugeordnet, sowie *q*↑ und *p*↑ miteinander verknüpft:

new (*q*); *q*↑.*Vorname* := "Otto"; *q*↑.*Vater* := *nil*; *q*↑.*Mutter* := *nil*;
p↑.*Vater* := *q*;

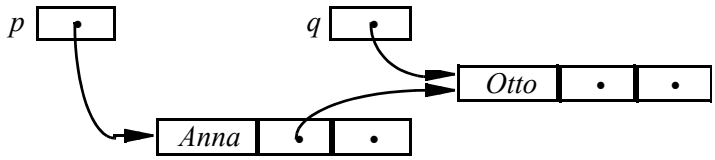

Abbildung 2.7: Zeiger *q* und *p*↑.*Vater* verweisen auf "Otto"

Diese Struktur kann nun zur Laufzeit des Programms durch Erzeugen neuer Objekte und Anhängen dieser Objekte an die bis dahin existierende Struktur beliebig erweitert werden:

new (*q*); *q*↑.*Vorname* := "*Erna*"; *q*↑.*Vater* := *nil*; *q*↑.*Mutter* := *nil*;
p↑.*Mutter* := *q*; *q* := *nil*;

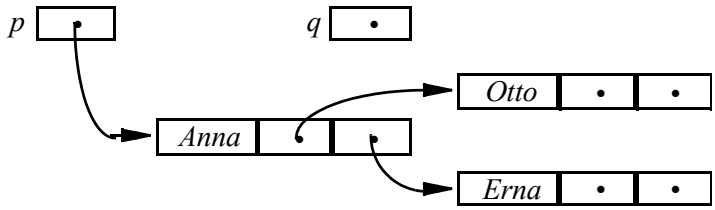

Abbildung 2.8: Zeiger *q* ist wieder leer, *p*↑.*Mutter* verweist auf "Erna"

Was geschieht, wenn jetzt "*p* := *nil* " ausgeführt wird?

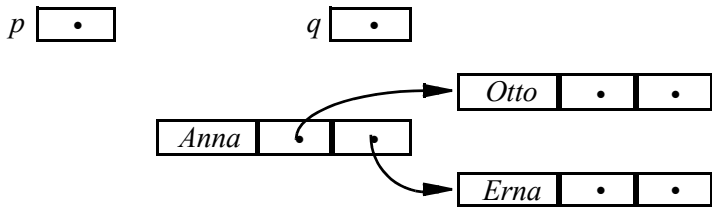

Abbildung 2.9: Zeiger *p* und *q* sind leer, "Anna", "Erna" und "Otto" sind unerreichbar

Anna, Otto und Erna belegen immer noch Speicherplatz, obwohl für das Programm keine Möglichkeit mehr besteht, sie je wieder zu erreichen!

Dieses Problem tritt immer dann auf, wenn Einträge in einer durch Zeiger verbundenen Struktur gelöscht werden. Es ist wünschenswert, daß das Betriebs- oder Laufzeitsystem solche Situationen automatisch erkennt und den nicht mehr belegten Speicher dem Programm wieder zur Verfügung stellt. Dieser Vorgang wird als "Garbage Collection" bezeichnet. Java ist eine der wenigen Programmiersprachen von Praxisrelevanz, in der Garbage Collection stattfindet— im Gegensatz zu Sprachen wie PASCAL, Modula-2, C oder C++. Statt dessen muß in solchen Sprachen dem Laufzeitsystem explizit mitgeteilt werden, daß eine Variable nicht mehr benötigt wird. Es reicht also nicht, nur die Zeiger auf nicht mehr benötigte Objekte zu löschen. Die Freigabe von nicht mehr benötigten Variablen geschieht beispielsweise in PASCAL und Modula-2 mit der *dispose*-Anweisung. Das Laufzeitsystem (bzw. Betriebssystem) kann dann von Zeit zu Zeit den Speicher reorganisieren, indem es mehrere solcher kleinen unbenutzten Speicherbereiche zu größeren Blöcken zusammenfaßt, die es dem Programm bei Bedarf wieder zur Verfügung stellen kann.

So ergibt sich nach der Freigabe der nicht mehr benötigten Felder mit

dispose (*p*↑.*Vater*); *dispose* (*p*↑.*Mutter*);
(anstelle von *p* := *nil*;)

diese Situation im Speicher:

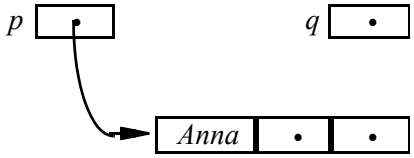

Abbildung 2.10: "Otto" und "Erna" sind wieder freigegeben

Der nicht benötigte Speicher steht dem System also wieder für neue Variablen zur Verfügung.

2.2.2 Zeiger in Java: Referenztypen

Häufig liest und hört man, daß Java eine Sprache ohne Zeiger sei. Ebenso häufig haben Java-Neulinge große Schwierigkeiten, die Ergebnisse von Variablenvergleichen und Zuweisungen und die Auswirkungen von Methodenaufrufen auf die übergebenen Parameter zu verstehen. Diese Schwierigkeiten können zu einem großen Teil vermieden werden, indem man sich klarmacht, daß es in Java sehr wohl Zeiger gibt! Allerdings hat der Programmierer keinen direkten Einfluß darauf, wann Zeiger verwendet werden, muß sich als Ausgleich jedoch auch nicht um die Garbage Collection kümmern.

In Java gibt es genau drei verschiedene Kategorien von Daten: Werte, Arrays und Objekte.[2] Um welche Kategorie es sich bei Variablen, Parametern, Attributen und Ergebnissen von Methodenaufrufen handelt, wird allein durch ihren Typ bestimmt. Ohne Einflußmöglichkeit des Programmierers werden in Java Werte immer direkt durch einen zusammenhängenden Bereich im Hauptspeicher repräsentiert, während Objekte und Arrays immer durch Zeiger realisiert werden. Klassen und Array-Typen werden deshalb auch als *Referenztypen* bezeichnet.

Die Konsequenzen der unterschiedlichen Handhabung von Werten einerseits und Objekten und Arrays andererseits wollen wir am Beispiel des Basisdatentyps *int* und der Klasse *Int* verdeutlichen, die uns als "Klassenhülle" um ein einziges Attribut vom Typ *int* dient und wie folgt definiert sei:

```
public class Int
{
   public int value;
   public Int(int i) {value = i;}
}
```

In unserem Beispiel verwenden wir zwei Variablen *i* und *I*:

```
int i = 5;
Int I = new Int(5);
```

Die Variable *i* repräsentiert einen Wert vom Typ *int*, initialisiert mit 5. Die Variable *I* referenziert ein Objekt vom Typ *Int*, dessen *value*-Attribut durch die Initialisierung den Wert 5 hat. Nun definieren wir zwei Variablen *j* und *J*, die wir mit *i* bzw. *I* initialisieren:

2. Zur Erinnerung: Werte sind Instanzen von Basistypen (*int*, *float*, …), Objekte sind Instanzen von Klassen.

```
int j = i;
Int J = I;
```

Abbildung 2.11 illustriert diese Ausgangssituation.

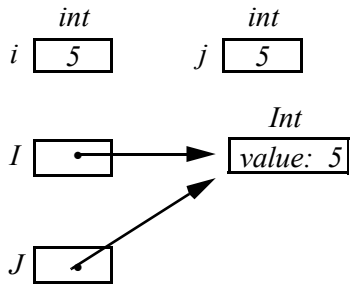

Abbildung 2.11: Variablen *i* und *j*, Zeiger *I* und *J*

Nun weisen wir *j* und *J.value* neue Werte zu:

```
j = 6;
J.value = 6;
```

Wie Abbildung 2.12 verdeutlicht, hat *i* nach wie vor den Wert 5, *I.value* jedoch den neuen Wert 6.

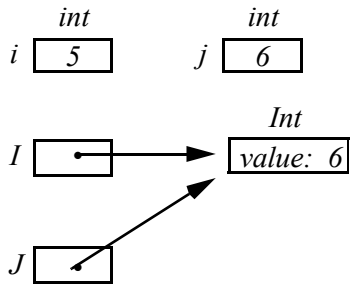

Abbildung 2.12: $i \neq j$, $I.value = J.value$

Schließlich betrachten wir noch, wie sich eine neue Zuweisung an *J* auswirkt:

```
J = new Int(7);
```

Das Ergebnis sehen wir in Abbildung 2.13. Nun referenzieren *I* und *J* zwei voneinander unabhängige Objekte vom Typ *Int*.

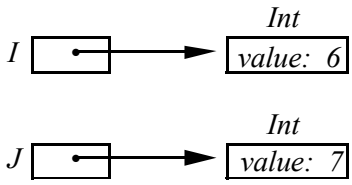

Abbildung 2.13: $I \neq J$

Mit der Thematik der Bedeutung von Zuweisungen eng verwandt ist die Frage nach der Wirkung von *Methodenaufrufen* auf die als Parameter übergebenen Werte und Objekte. Von imperativen Programmiersprachen kennen wir die beiden Strategien *call by value* und *call by reference*.[3] Aus so manchem Buch über Java lernt man, daß Werte *by value* übergeben werden, Objekte *by reference*. Die erste Aussage ist sicher richtig, die zweite aber nur die halbe Wahrheit. Betrachten wir dazu folgende Methoden:

```
public static void Double1(Int arg1)
{
   arg1.value = arg1.value + arg1.value;
}

public static void Double2(Int arg2)
{
   arg2 = new Int(arg2.value + arg2.value);
}
```

Beide Methoden berechnen das Doppelte des als Parameter übergebenen *Int*-Objektes und aktualisieren den Parameter entsprechend. Die Methode *Double1* setzt dazu das *value*-Attribut des Argumentes auf den errechneten Wert, *Double2* initialisiert ein neues *Int*-Objekt mit dem Ergebnis und weist es dem Argument zu. Nun wenden wir die Methoden an:

```
Int I = new Int(5);
Int J = new Int(5);
Double1(I);
Double2(J);
```

3. Zur Erinnerung: *Call by value* bedeutet, daß beim Aufruf einer Prozedur (Funktion, Methode) zunächst eine Kopie des übergebenen Parameters erzeugt wird und Parameterzugriffe im Rumpf der Prozedur sich auf diese Kopie beziehen. Deshalb ändern sich durch einen Prozeduraufruf die per call by value übergebenen Variablen nicht. Im Gegensatz dazu arbeiten bei *call by reference* Parameterzugriffe innerhalb des Prozedurrumpfes direkt auf den übergebenen Variablen. Änderungen der Parameter innerhalb der Prozedur werden nach außen wirksam.

Welche Werte weisen *I.value* und *J.value* jetzt auf? Ausgehend von der Information, daß Objektparameter per *call by reference* behandelt werden, sollten beide den Wert 10 enthalten. Tatsächlich gilt dies aber nur für *I*, während *J* unverändert geblieben ist. Die Erklärung liegt darin, daß in Java Parameterübergabe *immer* mittels *call by value* stattfindet. Objekte werden allerdings durch einen *Zeiger* auf die eigentliche Objektdarstellung repräsentiert. Es sind also *Zeiger*, die an die Methoden *Double1* und *Double2* im obigen Beispiel als Parameter übergeben wurden. Innerhalb dieser Methoden sind die Argumente namens *arg1* und *arg2 Kopien* der übergebenen Zeiger. Ein Komponentenzugriff über *arg1* verändert deshalb das sowohl von *I* als auch von *arg1* referenzierte Objekt dauerhaft, während eine direkte Zuweisung an die Zeigerkopie *arg2* nicht nach außen weitergegeben wird.

Neben Objekten werden auch *Arrays* in Java mit Hilfe eines Zeigers repräsentiert. Unsere bisherigen Feststellungen zu Zuweisungen und Parameterübergaben in bezug auf Objekte lassen sich deshalb direkt auf Arrays übertragen.

Konsequenzen hat die Strategie, Objekte und Arrays mittels Zeigern zu repräsentieren, auch auf das Layout von Arrays und Klasseninstanzen im Hauptspeicher. Betrachten wir folgendes Codefragment:

```
public class Foo
{
   public int a;
   public Int B;
   public int c;
   public Int D;
   public Foo(int arg1, int arg2, int arg3, int arg4) {
      a = arg1; B = new Int(arg2); c = arg3; D = new Int(arg4);
   }
}

Foo f = new Foo(1, 2, 3, 4);
```

Die Speicherdarstellung von *f* zeigt Abbildung 2.14.

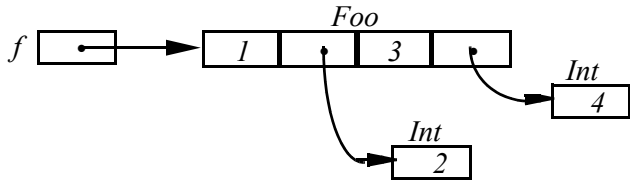

Abbildung 2.14: Klasse *Foo* mit Standard- und Klassenattributen

Ein Java-Programmierer hat keine Möglichkeit, eine Hauptspeicherrepräsentation von Klassen oder auch Arrays zu erzwingen, die alle Bestandteile eines komplexen Datentyps in einen zusammenhängenden Speicherbereich einbettet.

Abbildung 2.15 zeigt die Hauptspeicherdarstellung eines Arrays *A*, der aus 3 Elementen des Typs *Foo* besteht. Auch die Einbettung von Arrays kann in Java nicht vom Programmierer beeinflußt werden.

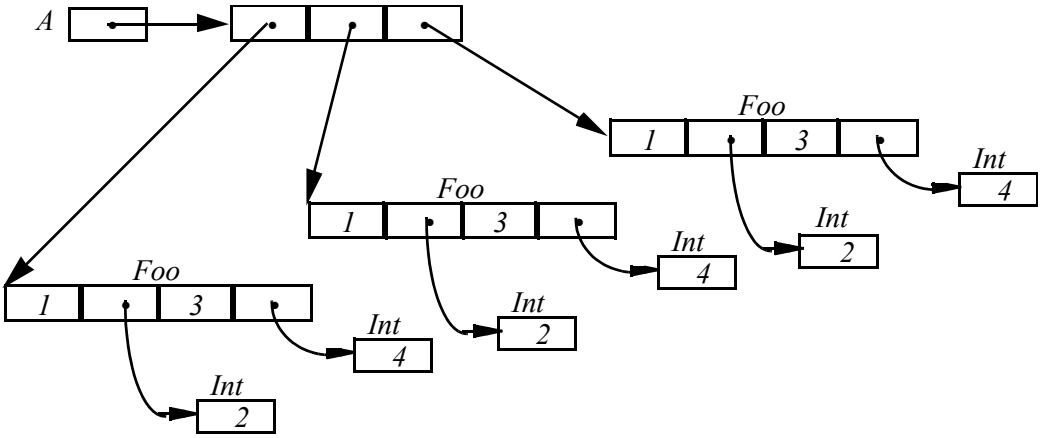

Abbildung 2.15: Array *A* mit drei *Foo*-Elementen

Komplexe Datentypen werden in Java also nicht in einem zusammenhängenden Bereich des Hauptspeichers dargestellt. Dennoch können die in diesem Kurs vorgestellten grundlegenden Algorithmen und Datenstrukturen, die ausnahmslos entwickelt wurden, als noch niemand an eine Programmiersprache wie Java dachte, auch in Java implementiert werden, ohne völlig neue Laufzeitanalysen anstellen zu müssen. Der Grund liegt darin, daß die Eigenschaft von Arrays und Records, Komponentenzugriff in konstanter Zeit zu ermöglichen, auch in Java erhalten bleibt. Manche systemnahe Anwendungen, wie beispielsweise das Speichermanagement eines Datenbanksystems, sind allerdings kaum in Java implementierbar, da hier die effiziente Manipulation des Hauptspeicherinhalts mit Hilfe von Zeigerarithmetik, blockweisem Kopieren von Speicherbereichen usw. unabdingbar ist.

2.3 Weitere Konzepte zur Konstruktion von Datentypen

In diesem Abschnitt gehen wir kurz auf einige weniger geläufige Konzepte zur Konstruktion von Datentypen ein. Bis vor relativ kurzer Zeit gehörte keines von ihnen zum Sprachumfang von Java; ab Java Version 5.0 hat sich dies geändert.

Aufzählungstypen

Aufzählungstypen werden vom Programmierer definiert und verwendet, um Werte aus einer festen, kleinen Menge von diskreten Werten zu repräsentieren. Ein neuer Aufzählungstyp wird in der Form

type $T = (c_1, c_2, ..., c_n)$

definiert, wobei die c_i die Elemente der zu spezifizierenden Menge sind.

```
type Wochentag  = (Mo, Di, Mi, Do, Fr, Sa, So);
type Figur      = (Koenig, Dame, Turm, Laeufer, Springer, Bauer);
var  f, g   : Figur;
     t      : Wochentag;
f := Koenig; g := Bauer;
if f = g ...
```

Als Operationen auf Aufzählungstypen werden etwa in PASCAL die Bildung des Nachfolgers und des Vorgängers angeboten, genannt *succ* (successor = Nachfolger) bzw. *pred* (predecessor = Vorgänger). Es gilt z.B. *succ*(*Koenig*) = *Dame*.

Der Compiler bildet den Wertebereich gewöhnlich auf einen Anfangsabschnitt der positiven ganzen Zahlen ab, also *Koenig* → 1, *Dame* → 2, ... In älteren Programmiersprachen mußte der Programmierer solche "Codierungen" selbst vornehmen. Der Vorteil gegenüber der direkten Repräsentation durch ganze Zahlen liegt im wesentlichen in der erhöhten Lesbarkeit der Programme und der besseren Fehlerkontrolle durch den Compiler.

Ab der Sprachversion 5.0 gibt es in Java Aufzählungstypen, die ähnlich wie in C/C++ notiert werden. Die Syntax in Java ist:

enum $T\ \{c_1, c_2, ..., c_n\};$

Die obigen Beispiele könnte man in Java so formulieren:

```
public enum Wochentag {Mo, Di, Mi, Do, Fr, Sa, So};
public enum Figur {Koenig, Dame, Turm, Laeufer, Springer, Bauer};
public final Figur f, g;
public final Wochentag t;
f = Figur.Koenig; g = Figur.Bauer;
if ( f == g ) ...
```

Tatsächlich ist eine *enum*-Definition eine besondere Art von Klassendefinition. Enum-Klassen besitzen einige vordefinierte Methoden, z.B. liefert

static *T*[] *values*()

einen Array, der alle Werte des Aufzählungstyps *T* enthält. Auf diese Art kann man in einer Schleife über alle Elemente des Aufzählungstyps iterieren:

```
for ( Figur f: Figur.values() ){
   System.out.println(f);
}
```

Darüber hinaus kann man in Enum-Klassen eigene Felder, Methoden und Konstruktoren definieren. Details finden sich in aktuellen Java-Büchern.

Unterbereichstypen

Soll die Verwendung eines Standardtyps auf einen bestimmten Wertebereich eingeschränkt werden, so wird ein Unterbereichstyp verwendet. Ein solcher Typ wird mit

```
type T = [min..max]
```

definiert; gültige Werte zu diesem Typ sind dann alle Werte x des betreffenden Standardtyps mit $min \leq x \leq max$.

```
type Jahr = [1900..2099];          (Unterbereich von integer)
type Buchstabe = ["A".."Z"];       (Unterbereich von char)
var j : Jahr;
j := 1910;        ist eine korrekte Zuweisung
j := 3001;        Dieser Fehler wird vom Compiler festgestellt
j := 3 * k + 7    Diese Zuweisung wird vom Laufzeitsystem überwacht
```

Unterbereichstypen werden im Speicher wie der Grundtyp repräsentiert, eventuell kann dabei eine Transformation des definierten Bereichs auf den Anfangsabschnitt erfolgen, um Speicherplatz zu sparen. So könnte der Typ

```
type BigNumber = [16894272..16894280]
```

auf den ganzzahligen Bereich 0..8 abgebildet werden.

Unterbereichstypen werden von Java nicht unterstützt. Unterbereichstypen schränken einen Standard-Typ auf einen kleineren Wertebereich ein. In der Java-Implementierung wird man meist anstelle des Unterbereichstyps den zum Standard-Typ korrespondierenden Basistyp verwenden, im Falle von Zuweisungen an Variablen dieses Typs die Bereichsüberprüfung “von Hand” vornehmen und gegebenenfalls eine angemessene Fehlerbehandlung durchführen. Sollte ein Unterbereichstyp häufiger verwendet werden, ist es sinnvoll, eine entsprechende Klasse zu definieren und die Bereichsüberprüfung in ihren Zugriffsmethoden zu implementieren.

Sets

Einige Programmiersprachen erlauben die Definition von Typen, deren Werte Mengen von Werten eines Grundtyps sind:

type *T* = **set of** T_0
↑ Grundtyp

Aus Gründen der Implementierung darf der Grundtyp dabei meist nur eine kleine Kardinalität besitzen und er muß atomar sein. Es eignen sich also nur Aufzählungs- und Unterbereichstypen. Insofern ist der *set* - Konstruktor kein allgemeiner Konstruktor wie *array* und *record*.

```
type KursNummer  = (1157, 1611, 1575, 1662, 1653, 1654, 1719, 5107...);
type Auswahl     = set of KursNummer;
var Grundstudium, Hauptstudium : Auswahl;
Grundstudium := {1662, 1611, 1653, 1575, 1654};
```

Set - Typen stellen die Operationen Durchschnitt, Vereinigung, Differenz und Elementtest zur Verfügung:

∩	*
∪	+
\	−
∈	in

} in PASCAL und Modula-2

Damit kann man etwa formulieren:

```
var n : KursNummer
if n in Grundstudium then ...
if not (Grundstudium * Hauptstudium) = {} then <Fehler>...
```

Dabei bezeichnet {} die leere Menge.

Der Wertebereich eines Set-Typs ist die Potenzmenge des Grundtyps, also

$$W(T) = P(W(T_0))$$

wobei P für den Potenzmengenoperator steht. Entsprechend ist die Kardinalität von T

$$card\,(T) = 2^{card\,(T_0)}$$

Der Compiler repräsentiert eine solche Menge gewöhnlich in einem Speicherwort (denkbar ist auch eine kurze Folge von Speicherworten). Jeder Bitposition des Speicherwortes

wird ein Element des Grundtyps zugeordnet. Sei $W(T_0) = \{c_1, ..., c_n\}$ und A eine Menge vom Typ T, dann gilt für die Repräsentation von A:

$c_i \in A \Leftrightarrow$ Bit i der Repräsentation von A hat den Wert 1.

Die Mengenoperationen können damit wie folgt implementiert werden:

$A \cap B$	entspricht bitweisem logischem UND auf den Repräsentationen von A und B, also $R(A)$ AND $R(B)$
$A \cup B$	entspricht $R(A)$ OR $R(B)$
$A \setminus B$	entspricht $R(A)$ AND (NOT $R(B)$)
$x_i \in A$	entspricht einer *shift*-Operation + Vorzeichentest

Aufgrund der gewählten Repräsentationen können diese Operationen also sehr schnell ausgeführt werden (aus algorithmischer Sicht in O(1) Zeit, was klar ist, da die Mengen nur O(1) groß werden können).

In Java gibt es ab Version 5.0 derartige Mengenimplementierungen auf der Basis von Aufzählungstypen; sie finden sich in der Klasse *java.util.EnumSet*. So könnte man z.B. definieren:

```
EnumSet<Wochentag> Wochenende = EnumSet.of(Wochentag.Sa, Wochentag.So);
```

Nähere Einzelheiten findet man wieder in entsprechenden Java-Büchern.

2.4 Literaturhinweise

Die klassische Quelle für die in diesem Kapitel beschriebenen Primitive für die Definition von Datentypen ist die Sprache PASCAL bzw. das Buch von Wirth [2000] (von dem es auch eine Modula-2-Version gibt [Wirth 1996]). Die Konzepte sind aber auch in jedem einführenden Buch zu PASCAL oder Modula-2 beschrieben (z.B. [Ottmann und Widmayer 2011], [Wirth 1991]). Ähnliche, noch etwas verfeinerte Konzepte zur Typbildung finden sich in der Sprache Ada (siehe z.B. [Nagl 2003]).

Zur Programmiersprache Java existiert eine Vielzahl von Büchern. Ein bewährtes, umfassendes Nachschlagewerk zur Programmierung in Java ist das Buch von Flanagan [2005]. Weitere Java-Bücher sind [Schiedermeier 2010] und [Krüger und Stark 2009].

Eine weitere wichtige Informationsquelle sind die Java-Seiten im WWW-Angebot der Firma Sun, zu erreichen unter `http://java.sun.com`.

Lösungen zu den Selbsttestaufgaben

Aufgabe 1.1

Ja, denn es gilt $\lim_{n\to\infty} \frac{3\sqrt{n}+5}{n} = \lim_{n\to\infty} \frac{3}{\sqrt{n}} + \frac{5}{n} = 0$.

Aufgabe 1.2

Hier ist zu berechnen $\lim_{n\to\infty} \frac{\log n}{n} = \frac{\infty}{\infty}$.

In diesem Fall kann man für reelle Funktionen bekanntlich (?) den Grenzwert des Quotienten der Ableitungen berechnen. Wir berechnen für $x \in \mathbb{R}$:

$$\lim_{x\to\infty} \frac{\ln x}{x} = \lim_{x\to\infty} \frac{\frac{d}{dx}\ln x}{\frac{d}{dx}x} = \lim_{x\to\infty} \frac{1/x}{1} = 0$$

Das Ergebnis kann man auf natürliche Zahlen übertragen und es gilt ln n = O(n) und ebenso log n = O(n) (die Logarithmen zu verschiedenen Basen unterscheiden sich ja nur durch einen konstanten Faktor, wie schon in Kapitel 1 erwähnt).

Aufgabe 1.3

Additionsregel:

Um zu zeigen, daß

$$T_1(n) + T_2(n) = O(\max(f(n), g(n))),$$

suchen wir $n_0 \in$ N, $c \in$ R mit $c > 0$, so daß für alle $n \geq n_0$

$$T_1(n)+T_2(n) \leq c \cdot \max(f(n), g(n)).$$

Aufgrund der Annahmen aus der Aufgabenstellung haben wir

$$n_1 \in \text{N}, c_1 \in \text{R}, \text{ so daß } \forall\, n \geq n_1 \colon T_1(n) \leq c_1 \cdot f(n),$$
$$n_2 \in \text{N}, c_2 \in \text{R}, \text{ so daß } \forall\, n \geq n_2 \colon T_2(n) \leq c_2 \cdot g(n). \qquad (*)$$

Wir setzen $n_0 = \max(n_1, n_2)$, dann gilt für alle $n \geq n_0$:

$$T_1(n)+T_2(n) \leq c_1 \cdot f(n) + c_2 \cdot g(n) \leq (c_1 + c_2) \cdot \max(f(n), g(n)).$$

Multiplikationsregel:

Man kann sinnvollerweise annehmen, daß $f(n) \geq 0$ und $g(n) \geq 0$. Wegen (*) (s.o.) gilt für $n \geq \max(n_1, n_2)$

$$T_1(n) \cdot T_2(n) \leq c_1 f(n) \cdot c_2 g(n) \leq (c_1 \cdot c_2) \cdot f(n) \cdot g(n).$$

Aufgabe 1.4

Natürlich, das ist eine einfache Anwendung von Beziehung (ii), die sagt, daß Logarithmen langsamer wachsen als beliebige Potenzen von n, also auch $n^{1/2}$.

Aufgabe 1.5

(a) Ein rekursiver Algorithmus zum Aufsummieren der Elemente einer Liste, der O(n) Rekursionstiefe hat, ist:

```
algorithm sum1(nums, i, j)
{Eingabe: Ein Feld von zu summierenden Zahlen nums und zwei Indizes i, j darin
   mit i ≤ j. Ausgabe: die Summe der Zahlen im Bereich i bis j.}
if i = j  then return nums[i]
else
   return sum1(nums, i, j-1) + nums[j]
end if.
```

(b) Um eine Rekursionstiefe von $O(\log n)$ zu erhalten, werden jeweils die Teilsummen von zwei gleichgroßen Teilfeldern berechnet:

```
algorithm sum2(nums, i, j)
{Eingabe: Ein Feld von zu summierenden Zahlen nums und zwei Indizes i, j darin
   mit i ≤ j. Ausgabe: die Summe der Zahlen im Bereich i bis j.}
if i = j  then return nums[i]
else
   return sum2(nums, i, i+⌊(j-i)/2⌋) + sum2(nums, i+⌊(j-i)/2⌋+1, j)
end if.
```

Zu beachten ist, daß in beiden Fällen der Algorithmus mit den aktuellen Parametern 1 für i und n für j aufgerufen werden muß.

Aufgabe 1.6

functions

(a)

$reset = 0$

$inc(n) = n + 1$

$$dec(n) = \begin{cases} 0 & \text{falls } n = 0 \\ n-1 & \text{sonst} \end{cases}$$

(b)

$reset = 0$

$inc(n) = n + 1$

$dec(n) = n - 1$

(c)

$reset = 0$

$$inc(n) = \begin{cases} n+1 & \text{falls } n < p \\ 0 & \text{falls } n=p \end{cases}$$

$$dec(n) = \begin{cases} n-1 & \text{falls } n > 0 \\ p & \text{falls } n=0 \end{cases}$$

axioms

$dec(inc(n)) = n$

$inc(dec(n)) = n$

Beachten Sie bitte, daß die angegebenen Axiome nur für die natürlichen Zahlen ohne Null gelten, da beim Versuch, Null zu dekrementieren eine “Bereichsunterschreitung” auftreten würde.

Aufgabe 1.7

(a)

algebra *puzzle15*

sorts *tile, position, board, bool*

ops

init:	$tile^{16}$	$\rightarrow board$
move:	$board \times tile$	$\rightarrow board$
solved:	*board*	$\rightarrow bool$
pos:	$board \times tile$	$\rightarrow position$

(b)

sets

$$tile = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, \text{blank}\}$$
$$position = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, \text{error}\}$$

$$\begin{aligned} board = \{ & C \subset (position \setminus \{\text{error}\}) \times tile \mid |C| = 16 \\ & \wedge \forall c = (p,t) \in C : \\ & \quad \forall c' = (p',t') \in C : \\ & \quad\quad c' \neq c \Rightarrow p' \neq p \wedge t' \neq t \\ \} & \cup \{\text{error}\} \end{aligned}$$

Ein *tile* (“Kachel”) bezeichnet dabei einen Stein des Puzzles, dargestellt durch seine Beschriftung bzw. “blank”. Eine *position* beschreibt einen der 16 möglichen Plätze innerhalb des Spielfeldes, nummeriert von links oben nach rechts unten, oder eine Fehlerposition. Das Spielfeld selbst ist eine Menge von Paaren (*position*, *tile*) mit einigen Nebenbedingungen: es müssen 16 verschiedene Paare sein und zwei verschiedene Paare müssen sich sowohl in ihrer Position als auch ihrem Stein unterscheiden. Schließlich gibt es den Wert “error” für ein Spielfeld, den eine Operation im Fehlerfall zurückliefern kann. Die Sorte *bool* wird als bekannt angenommen.

(c)

functions

$$init(t_1, t_2, \ldots, t_{16}) = \begin{cases} \{(1,t_1),(2,t_2),\ldots,(16,t_{16})\}, \text{ falls } |\{t_1,t_2,\ldots,t_{16}\}| = 16 \\ \text{error, sonst} \end{cases}$$

$$move(b,t) = \begin{cases} \text{error, falls } t = \text{blank} \vee b = \text{error} \\ \\ \{(p_{tile}, \text{blank}), (p_{blank}, t)\} \\ \quad \cup \{(p',t') \mid (p',t') \in b \wedge p' \notin \{p_{tile}, p_{blank}\}\}, \\ \quad \text{falls} \\ \quad \{(p_{blank}, \text{blank}), (p_{tile}, t)\} \subset b \wedge \\ \quad\quad (\ (p_{tile} \in \{(p_{blank} - 4), (p_{blank} + 4)\} \vee \\ \quad\quad\ \ (p_{tile} = (p_{blank} - 1) \wedge (p_{tile} \bmod 4) \neq 0) \vee \\ \quad\quad\ \ (p_{tile} = (p_{blank} + 1) \wedge (p_{tile} \bmod 4) \neq 1) \ ) \\ \\ \text{error, sonst} \end{cases}$$

Ein Zug (“move”) liefert ein an zwei Positionen verändertes Spielfeld zurück, nämlich an den Positionen p_{tile} des als Argument angegebenen Steins t und p_{blank} des freien Feldes. Alle anderen Paare werden unverändert vom Argument b ins Ergebnis übernommen. Die Nebenbedingungen überprüfen, daß die Position von t ein Nachbar des freien Feldes ist.

$$solved(b) = \begin{cases} \text{true, falls } b = \{(1,1),(2,2),(3,3),\ldots,(15,15),(16,\text{blank})\} \\ \text{false, sonst} \end{cases}$$

$$pos(b,t) = \begin{cases} \text{error, falls } b = \text{error} \\ p, \text{ mit } (p,t) \in b, \text{ sonst} \end{cases}$$

end *puzzle15*.

Aufgabe 1.8

adt *int*

sorts	*int*		
ops	0:		$\rightarrow int$
	succ:	*int*	$\rightarrow int$
	pred:	*int*	$\rightarrow int$
	+:	$int \times int$	$\rightarrow int$
	-:	$int \times int$	$\rightarrow int$
	*:	$int \times int$	$\rightarrow int$
axs			
	$pred(succ(x))$		$= x$
	$succ(pred(x))$		$= x$
	$x + 0$		$= x$
	$x + succ(y)$		$= succ(x + y)$
	$x + pred(y)$		$= pred(x + y)$
	x - 0		$= x$
	x - $succ(y)$		$= pred(x - y)$
	x - $pred(y)$		$= succ(x - y)$
	x * 0		$= 0$
	x * $succ(y)$		$= (x * y) + x$
	x * $pred(y)$		$= (x * y) - x$

end *int*.

Selbstverständlich kann man noch viele weitere Axiome angeben. Wir haben uns hier, da im Text kein Vollständigkeitskriterium für Axiomenmengen eingeführt wurde, auf solche beschränkt, die elementar erscheinen.

Literatur

Aho, A.V., J.E. Hopcroft und J.D. Ullman [1974]. The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Co., Reading, Massachusetts.

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

Albert, J., und T. Ottmann [1990]. Automaten, Sprachen und Maschinen für Anwender. Spektrum Akademischer Verlag, Heidelberg.

Baase, S., und A. Van Gelder [2000]. Computer Algorithms. Introduction to Design and Analysis. 3rd Ed., Addison-Wesley Publishing Co., Reading, Massachusetts.

Banachowski, L., A. Kreczmar und W. Rytter [1991]. Analysis of Algorithms and Data Structures. Addison-Wesley Publishing Co., Reading, Massachusetts.

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

Cormen, T.H., C.E. Leiserson, R.L. Rivest, und C. Stein [2010]. Algorithmen - Eine Einführung. 3. Aufl., Oldenbourg-Verlag, München.

Ehrich, H.D., M. Gogolla und U.W. Lipeck [1989]. Algebraische Spezifikation abstrakter Datentypen. Eine Einführung in die Theorie. Teubner-Verlag, Stuttgart.

Enbody, R.J., und H.C. Du [1988]. Dynamic Hashing Schemes. *ACM Computing Surveys 20*, 85-113.

Flanagan, D. [2005]. Java in a Nutshell, Fifth Edition. O'Reilly & Associates.

Gonnet, G.H., und R. Baeza-Yates [1991]. Handbook of Algorithms and Data Structures. In Pascal and C. 2nd Ed., Addison-Wesley Publishing Co., Reading, Massachusetts.

Graham, R.L., D.E. Knuth und O. Patashnik [1994]. Concrete Mathematics. A Foundation for Computer Science. 2nd Ed., Addison-Wesley Publishing Co., Reading, Massachusetts.

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.

Klaeren, H.A. [1983]. Algebraische Spezifikation. Eine Einführung. Springer-Verlag, Berlin.

Knuth, D.E. [1998]. The Art of Computer Programming, Vol. 3: Sorting and Searching. 2nd Ed., Addison-Wesley Publishing Co., Reading, Massachusetts.

Krüger, G. und T. Stark [2009]. Handbuch der Java-Programmierung: Standard Edition Version 6. 6. Aufl, Addison-Wesley Longman Verlag, München.

Loeckx, J., H.D. Ehrich und M. Wolf [1996]. Specification of Abstract Data Types. Wiley-Teubner Publishers, Chichester, Stuttgart.

Manber, U. [1989]. Introduction to Algorithms. A Creative Approach. Addison-Wesley Publishing Co., Reading, Massachusetts.

Mehlhorn, K. [1984a]. Data Structures and Algorithms 1: Sorting and Searching. Springer-Verlag, Berlin.

Mehlhorn, K. [1984b]. Data Structures and Algorithms 2: Graph-Algorithms and NP-Completeness. Springer-Verlag, Berlin.

Mehlhorn, K. [1984c]. Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry. Springer-Verlag, Berlin.

Nagl, M. [2003]. Softwaretechnik mit Ada 95: Entwicklung großer Systeme. 2. Aufl., Vieweg+Teubner Verlag, Wiesbaden.

Nievergelt, J., und K.H. Hinrichs [1993]. Algorithms and Data Structures: With Applications to Graphics and Geometry. Prentice-Hall, Englewood Cliffs, N.J.

Ottmann, T., und P. Widmayer [2011]. Programmierung mit PASCAL. 8. Aufl., Vieweg+Teubner-Verlag, Wiesbaden.

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

Preparata, F.P., und M.I. Shamos [1985]. Computational Geometry. An Introduction. Springer-Verlag, Berlin.

Saake, G., und K.U. Sattler [2010]. Algorithmen und Datenstrukturen. Eine Einführung mit Java. 4. Aufl., dpunkt.verlag, Heidelberg.

Schiedermeier, R. [2010]. Programmieren mit Java. Eine methodische Einführung. 2. Aufl., Pearson Studium, München.

Sedgewick, R. [2002a]. Algorithmen. 2. Aufl., Addison-Wesley Longman Verlag, Pearson Studium, München.

Sedgewick, R. [2002b]. Algorithmen in C++. Teil 1-4. 3. Aufl., Addison-Wesley Longman Verlag, Pearson Studium, München.

Sheperdson, J.C., und H.E. Sturgis [1963]. Computability of Recursive Functions. *Journal of the ACM 10*, 217-255.

Standish, T.A. [1998]. Data Structures in Java. Addison-Wesley Publishing Co., Reading, Massachusetts.

Weiss, M.A. [2009]. Data Structures and Problem Solving Using Java. 4. Aufl., Addison-Wesley Publishing Co., Reading, Massachusetts.

Wirth, N. [1991]. Programmieren in Modula-2. 2.Aufl., Springer-Verlag, Berlin.

Wirth, N. [1996]. Algorithmen und Datenstrukturen mit Modula-2. 5. Aufl., Teubner-Verlag, Stuttgart.

Wirth, N. [2000]. Algorithmen und Datenstrukturen. Pascal-Version. 5. Aufl., Teubner-Verlag, Stuttgart.

Wood, D. [1993]. Data Structures, Algorithms, and Performance. Addison-Wesley Publishing Co., Reading, Massachusetts.

Index

Symbole

Ω-Notation 20

A

abstrakter Datentyp 1, 2, 35, 38
Abstraktionsebene 1
Ada 61
ADT 35
Aggregation 39
Akkumulator 7
Algebra 1, 2, 23
algebraische Spezifikation 38
Algorithmus 1, 33
Analyse 2
Analyse von Algorithmen 38
Array 39
Assemblersprache 7
atomarer Datentyp 41
Aufzählungstyp 59
average case 10
Axiom 24, 35

B

best case 10
bester Fall 10
Betriebssystem 52
binäre Suche 17

D

Datenobjekt 28
Datenspeicher 7
Datenstruktur 1, 2, 22, 35
Datentyp 1, 23, 35
Definitionsmodul 29
denotationelle Spezifikation 38
Dereferenzierung 50
dispose 52

E

Einheitskosten 7
Elementaroperation 6, 7
Erwartungswert 14
Euler-Konstante 17
exponentiell 15

F

Feld 42
Fibonacci-Zahlen 19
Funktion 1, 3, 23

G

Garbage Collection 52
Gesetz 24
Gleichverteilung 10

H

harmonische Zahl 17
heterogene Algebra 23, 35

I

Indextyp 42
Instruktion 7
isomorph 25

K

Komplexität der Eingabe 6, 21

Komplexität des Problems 20
Komplexitätsklasse 9, 20
konstant 15
Korrektheit 5
Kostenmaß 7

L

Laufzeit 2
Laufzeitsystem 52
linear 15
logarithmisch 15
logarithmisches Kostenmaß 7

M

Maschinenmodell 38
Maschinensprache 7
mehrsortige Algebra 23, 35
Mengenoperation 61
Modell 25, 35
Modul 1, 2
monomorph 25, 35

N

Nachfolger 58
nil 49

O

offset 48
O-Notation 11, 12, 15, 19, 21
Operation 22, 34
Operationssymbol 23, 34
optimal 20

P

PASCAL 61
Platzbedarf 2, 6
Pointer 49
polymorph 25, 35
polynomiell 15
pred 58
Problem 33
Programmspeicher 7
Programmzähler 7
Prozedur 1, 2

Q

quadratisch 15

R

RAM 7, 33, 38
Random-Access-Maschine 7
real RAM 7, 38
Record 39, 47, 61
Register 7
Registermaschine 38
Reihung 42
Rekursionsgleichung 15, 17
Repräsentation 39

S

schlimmster Fall 10
schrittweise Verfeinerung 35
Selektion 42, 48
Selektor 48
Semantik 23
Set 60
Signatur 23, 34, 35
Sorte 23, 34
Speicherzelle 7
Spezifikation 1
Spezifikation als abstrakter Datentyp 24
Spezifikation als Algebra 23
succ 58

T

Trägermenge 23
Turingmaschine 7, 33
Typ 1, 39
Typsystem 40

U

universale Algebra 23
Unterbereichstyp 59
untere Schranke 20

V

Vorgänger 58

W

Wahrscheinlichkeitsraum 14
worst case 10

Z

Zeiger 49
Zufallsvariable 14

Mathematische Grundlagen

I Einige Summenformeln

(1) $$\sum_{i=1}^{n} i = \frac{n\,(n+1)}{2}$$

(2) $$\sum_{i=1}^{n} i^2 = \frac{n\,(n+1)(2n+1)}{6}$$

(3) $$\sum_{i=1}^{n} i^3 = \frac{n^2\,(n+1)^2}{4}$$

(4) $$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1} = \frac{1-x^{n+1}}{1-x} \qquad x \neq 1$$

$$(\text{für } |x| < 1,\ n \to \infty : \frac{1}{1-x})$$

Beweis:

Sei $S_n := \sum_{i=0}^{n} x^i$.

Die Technik bei diesem Beweis ist die Ausnutzung der Assoziativität:

$$S_{n+1} = (a_1 + \ldots + a_n) + a_{n+1}$$
$$= a_1 + (a_2 + \ldots + a_{n+1})$$

$$
\begin{aligned}
S_{n+1} = x^{n+1} + S_n = x^0 + \sum_{i=1}^{n+1} x^i \\
&= 1 + \sum_{i=0}^{n} x^{i+1} \\
&= 1 + x \cdot \sum_{i=0}^{n} x^i \\
&= 1 + x \cdot S_n
\end{aligned}
$$

$$
\begin{aligned}
& S_n + x^{n+1} = 1 + x \cdot S_n \\
\Leftrightarrow \quad & x^{n+1} - 1 = S_n \,(x-1) \\
\Leftrightarrow \quad & S_n = \frac{x^{n+1}-1}{x-1} \qquad x \neq 1
\end{aligned}
$$

□

II Einige Grundlagen der Wahrscheinlichkeitsrechnung und Kombinatorik

(1) Ein *Wahrscheinlichkeitsraum* ist eine Menge (von "Ereignissen") Ω, wobei jedes Ereignis $\omega \in \Omega$ mit Wahrscheinlichkeit $P(\omega)$ auftritt und

$$\sum_{\omega \in \Omega} P(\omega) = 1$$

(2) Eine *Zufallsvariable* X ist eine Abbildung X: $\Omega \to D$ (mit D = N oder D = R); sie ordnet also jedem Ereignis in Ω einen Zahlenwert zu.

(3) Der *Erwartungswert* (=Durchschnitt, Mittelwert) einer Zufallsvariablen X ist

$$\sum_{x \in X(\Omega)} x \cdot P(X = x) \qquad \text{(Abkürzung } \sum x \cdot P(x)\text{)}$$

Beispiel: Der "durchschnittliche" Wert (also der Erwartungswert) beim Würfeln mit zwei Würfeln ist

$$2 \cdot P(2) + 3 \cdot P(3) + \ldots + 12 \cdot P(12) =$$

$$2 \cdot \frac{1}{36} + 3 \cdot \frac{2}{36} + \ldots + 12 \cdot \frac{1}{36}$$

(1,1) (1,2) (2,1) (6,6)

□

(4) Die Anzahl der möglichen Anordnungen der Elemente einer n-elementigen Menge ist $n!$ (Fakultät von n)

Beweis: Beim Aufbau der Folge hat man bei der Auswahl des ersten Elementes n Möglichkeiten, beim zweiten (n-1), usw., beim n-ten nur noch eine Möglichkeit, also $n \cdot (n\text{-}1) \cdot \ldots \cdot 1 = n!$ □

(5) Notation: $n^{\underline{k}} := n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$

(6) $$\binom{n}{k} := \frac{n^{\underline{k}}}{k!} = \frac{n!}{k!(n-k)!}$$

(7) Die Anzahl der Möglichkeiten, eine k-elementige Teilmenge aus einer n-elementigen Menge auszuwählen, ist

$$\binom{n}{k}$$

Beweis: Die Anzahl der Möglichkeiten, eine k-elementige *Folge* auszuwählen, ist $n \cdot (n\text{-}1) \cdot \ldots \cdot (n\text{-}k\text{+}1) = n^{\underline{k}}$. Diese Folge ist eine von $k!$ äquivalenten Folgen, die alle die gleiche k-elementige Menge darstellen. Also gibt es $n^{\underline{k}} / k!$ verschiedene k-Teilmengen. □

III Umgang mit Binomialkoeffizienten

(1) $$\binom{n}{k} = \binom{n}{n-k}$$

(2) $$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

(3) $$\binom{n}{k} = \frac{n}{k} \cdot \binom{n-1}{k-1} \Leftrightarrow k \cdot \binom{n}{k} = n \cdot \binom{n-1}{k-1}$$

(4) $$\sum_{k=0}^{m} \binom{n+k}{k} = \binom{n+m+1}{m}$$

(5) $$\sum_{m=0}^{n} \binom{m}{k} = \binom{0}{k} + \binom{1}{k} + \binom{2}{k} + \cdots + \binom{n}{k} = \binom{n+1}{k+1}$$

$$\binom{n}{k} = 0 \text{ falls } n < k$$

Anwendung: $$\binom{0}{1} + \binom{1}{1} + \binom{2}{1} + \cdots + \binom{n}{1} = \binom{n+1}{2}$$

also: $1 + 2 + \cdots + n = \frac{(n+1) \cdot n}{2}$ (Gauß'sche Formel)

(6) $$\sum_{k=0}^{n} \binom{r}{k} \cdot \binom{s}{n-k} = \binom{r+s}{n}$$

"Die Anzahl der Möglichkeiten, von r Männern und s Frauen n Personen auszuwählen (rechte Seite) ist gerade die Anzahl der Möglichkeiten, k von den r Männern und $(n–k)$ von den s Frauen auszuwählen, summiert über alle k."

(7) $$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot x^k \cdot y^{n-k}$$

Beispiel:

$$
\begin{aligned}
(x+y)^3 &= \sum_{k=0}^{3} \binom{3}{k} \cdot x^k y^{3-k} \\
&= \binom{3}{0} \cdot x^0 y^3 + \binom{3}{1} \cdot x^1 y^2 + \binom{3}{2} \cdot x^2 y^1 + \binom{3}{3} \cdot x^3 y^0 \\
&= y^3 + 3xy^2 + 3x^2 y + x^3
\end{aligned}
$$

□

IV Harmonische Zahlen

(1) $H_n := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$, n ≥ 0, heißt die n-te *harmonische Zahl*

(2) $\frac{\lfloor \log n \rfloor + 1}{2} < H_n \le \lfloor \log n \rfloor + 1$

(3) $\ln n < H_n < \ln n + 1 \qquad n > 1$

(4) $\lim_{n\to\infty} (H_n - \ln n) = \gamma$, mit γ= 0.5772156649..(Euler-Konstante)

(5) $H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{\varepsilon_n}{120n^4} \qquad 0 < \varepsilon_n < 1$

V Umwandlung von Rekursion in eine Summe

Gegeben: Rekursionsgleichung der Form

$$a_n T_n = b_n T_{n-1} + c_n \qquad n \ge 1$$

Multipliziere beide Seiten mit schlau gewähltem *Summierungsfaktor* s_n

$$s_n a_n T_n = s_n b_n T_{n-1} + s_n c_n$$

Wähle s_n so, daß

$$s_n b_n = s_{n-1} a_{n-1}$$

Dann gilt nämlich

$$\underbrace{s_n a_n T_n}_{=:U_n} = s_n b_n T_{n-1} + s_n c_n = \underbrace{s_{n-1} a_{n-1} T_{n-1}}_{=:U_{n-1}} + s_n c_n, U_n = U_{n-1} + s_n c_n$$

also

$$U_n = U_{n-1} + s_n c_n$$
$$U_n = s_0 a_0 T_0 + \sum_{i=1}^{n} s_i c_i = s_1 b_1 T_0 + \sum_{i=1}^{n} s_i c_i$$

und

$$T_n = \frac{1}{s_n a_n}(s_1 b_1 T_0 + \sum_{i=1}^{n} s_i c_i)$$

Wie findet man s_n? Kein Problem:

$$s_n = \frac{a_{n-1}}{b_n} s_{n-1} = \frac{a_{n-1} a_{n-2}}{b_n b_{n-1}} s_{n-2} = \cdots$$

Also

$$s_n = \frac{a_{n-1} a_{n-2} \cdots a_1}{b_n b_{n-1} \cdots b_2} s_1 \qquad n \geq 2$$

s_1 kann beliebig $\neq 0$ gewählt werden

$$T_1 = \frac{s_1 b_1 T_0 + s_1 c_1}{s_1 a_1} = \frac{b_1 T_0 + c_1}{a_1}$$ da s_1 durch Kürzen herausfällt.

VI Fibonacci-Zahlen

(1) Die Fibonacci-Zahlen sind so definiert:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \qquad n > 1$$

(2) Es gibt eine geschlossene Form. Sei $\Phi = \frac{1+\sqrt{5}}{2}$ und $\hat{\Phi} = \frac{1-\sqrt{5}}{2}$.

Dann gilt: $F_n = \frac{1}{\sqrt{5}}(\Phi^n - \hat{\Phi}^n)$

Beweis: Übung. Man benutze die Tatsache, daß $\Phi^2 = \Phi + 1$ und $\hat{\Phi}^2 = \hat{\Phi} + 1$.

(3) Da $\Phi \approx 1.61803$ und $\hat{\Phi} \approx -0.61803$ wird der Term $\hat{\Phi}^n$ für große n verschwindend klein. Deshalb gilt

$$F_n \approx \frac{1}{\sqrt{5}}\Phi^n$$

bzw.

$$F_n = \left\lfloor \frac{\Phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor = \frac{\Phi^n}{\sqrt{5}} \text{ gerundet zur nächsten ganzen Zahl,}$$

da $\left|\frac{\hat{\Phi}^n}{\sqrt{5}}\right| < \frac{1}{2} \qquad \forall\, n \geq 0.$