

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 des Kurses 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 leicht anhand eines Java-Buches 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 Einsendaufgaben 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.

Papierfassung

Der Kurs besteht aus sieben Textheften für die einzelnen Kurseinheiten sowie einem Begleitheft, das für den gesamten Kurs relevant ist. Das Begleitheft enthält Vorwort und Anhang des Kurses.

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 IV an der FernUniversität. Hauptarbeitsgebiete: Geo-Datenbanksysteme, Architektur 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 Betriebswirtschaftslehre (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://www.informatik.fernuni-hagen.de/import/pi4/index.html>

Gliederung in Kurse und Kurseinheiten

<p><b style="color: #C85130;">Datenstrukturen I / Datenstrukturen</p> <p>Kurseinheit 1</p> <ul style="list-style-type: none"> 1 Einführung 2 Programmiersprachliche Konzepte für Datenstrukturen 	
<p>Kurseinheit 2</p> <ul style="list-style-type: none"> 3 Grundlegende Datentypen 	
<p>Kurseinheit 3</p> <ul style="list-style-type: none"> 4 Datentypen zur Darstellung von Mengen (*) 	
<p>Kurseinheit 4</p> <ul style="list-style-type: none"> 5 Sortieralgorithmen 6 Graphen 	
<p><b style="color: #C85130;">Datenstrukturen I</p> <p>Kurseinheit 5</p> <ul style="list-style-type: none"> 7 Graph-Algorithmen <ul style="list-style-type: none"> 7.1 Bestimmung kürzester Wege 9 Externes Suchen und Sortieren <ul style="list-style-type: none"> 9.1 Externes Suchen: B-Bäume 	<p><b style="color: #C85130;">Datenstrukturen II / Datenstrukturen</p> <p>Kurseinheit 5</p> <ul style="list-style-type: none"> 7 Graph-Algorithmen
	<p>Kurseinheit 6</p> <ul style="list-style-type: none"> 8 Geometrische Algorithmen
	<p>Kurseinheit 7</p> <ul style="list-style-type: none"> 9 Externes Suchen und Sortieren

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

Inhalt

1	Einführung	1
1.1	Algorithmen und ihre Analyse	2
1.2	Datenstrukturen, Algebren, Abstrakte Datentypen	22
1.3	Grundbegriffe	32
1.4	Weitere Aufgaben	35
1.5	Literaturhinweise	36
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
	Aufzählungstypen	58
	Unterbereichstypen	59
	Sets	60
2.4	Literaturhinweise	61
3	Grundlegende Datentypen	63
3.1	Sequenzen (Folgen, Listen)	63
3.1.1	Modelle	64
	(a) Listen mit first, rest, append, concat	64
	(b) Listen mit expliziten Positionen	65
3.1.2	Implementierungen	68
	(a) Doppelt verkettete Liste	68
	(b) Einfach verkettete Liste	73
	(c) Sequentielle Darstellung im Array	77
	(d) Einfach oder doppelt verkettete Liste im Array	78
3.2	Stacks	82
3.3	Queues	89
3.4	Abbildungen	91
3.5	Binäre Bäume	92
	Implementierungen	99
	(a) mit Zeigern	99
	(b) Array - Einbettung	100

3.6	(Allgemeine) Bäume	101
	Implementierungen	104
	(a) über Arrays	104
	(b) über Binärbäume	104
3.7	Weitere Aufgaben	105
3.8	Literaturhinweise	107
4	Datentypen zur Darstellung von Mengen	109
4.1	Mengen mit Durchschnitt, Vereinigung, Differenz	109
	Implementierungen	110
	(a) Bitvektor	110
	(b) Ungeordnete Liste	111
	(c) Geordnete Liste	111
4.2	Dictionaries: Mengen mit INSERT, DELETE, MEMBER	113
	4.2.1 Einfache Implementierungen	114
	4.2.2 Hashing	115
	Analyse des “idealen” geschlossenen Hashing (*)	120
	Kollisionsstrategien	126
	(a) Lineares Sondieren (Verallgemeinerung)	126
	(b) Quadratisches Sondieren	126
	(c) Doppel-Hashing	127
	Hashfunktionen	128
	(a) Divisionsmethode	128
	(b) Mittel-Quadrat-Methode	128
	4.2.3 Binäre Suchbäume	129
	Durchschnittsanalyse für binäre Suchbäume (*)	136
	4.2.4 AVL-Bäume	141
	Updates	141
	Rebalancieren	142
4.3	Priority Queues: Mengen mit INSERT, DELETETEMIN	152
	Implementierung	153
4.4	Partitionen von Mengen mit MERGE, FIND (*)	156
	Implementierungen	157
	(a) Implementierung mit Arrays	157
	(b) Implementierung mit Bäumen	160
	Letzte Verbesserung: Pfadkompression	162
4.5	Weitere Aufgaben	163
4.6	Literaturhinweise	166

5	Sortieralgorithmen	167
5.1	Einfache Sortierverfahren: Direktes Auswählen und Einfügen	168
5.2	Divide-and-Conquer-Methoden: Mergesort und Quicksort	171
	Durchschnittsanalyse für Quicksort	179
5.3	Verfeinertes Auswählen und Einfügen: Heapsort und Baumsortieren	182
	Standard-Heapsort	182
	Analyse von Heapsort	184
	Bottom-Up-Heapsort	186
5.4	Untere Schranke für allgemeine Sortierverfahren	188
5.5	Sortieren durch Fachverteilen: Bucketsort und Radixsort	192
5.6	Weitere Aufgaben	195
5.7	Literaturhinweise	196
6	Graphen	199
6.1	Gerichtete Graphen	200
6.2	(Speicher-) Darstellungen von Graphen	202
	(a) Adjazenzmatrix	202
	(b) Adjazenzlisten	204
6.3	Graphdurchlauf	205
6.4	Literaturhinweise	209
7	Graph-Algorithmen	211
7.1	Bestimmung kürzester Wege von einem Knoten zu allen anderen	211
	Implementierungen des Algorithmus Dijkstra	216
	(a) mit einer Adjazenzmatrix	216
	(b) mit Adjazenzlisten und als Heap dargestellter Priority Queue	217
7.2	Bestimmung kürzester Wege zwischen allen Knoten im Graphen	218
	Implementierung des Algorithmus von Floyd	220
	(a) mit der Kostenmatrix-Darstellung	220
	(b) mit Adjazenzlisten	221
7.3	Transitive Hülle	223
7.4	Starke Komponenten	223
7.5	Ungerichtete Graphen	227
7.6	Minimaler Spannbaum (Algorithmus von Kruskal)	228
7.7	Weitere Aufgaben	232
7.8	Literaturhinweise	234

8 Geometrische Algorithmen	237
8.1 Plane-Sweep-Algorithmen für orthogonale Objekte in der Ebene	242
8.1.1 Das Segmentschnitt-Problem	242
8.1.2 Das Rechteckschnitt-Problem	247
Das Punkteinschluß-Problem und seine Plane-Sweep-Reduktion	248
Der Segment-Baum	250
Komplexität der Lösungen	252
8.1.3 Das Maßproblem	254
Plane-Sweep-Reduktion	254
Ein modifizierter Segment-Baum	256
Komplexität der Lösung des Maßproblems	257
8.2 Divide-and-Conquer-Algorithmen für orthogonale Objekte	258
8.2.1 Das Segmentschnitt-Problem	259
8.2.2 Das Maßproblem	265
8.2.3 Das Konturproblem	272
8.3 Suchen auf Mengen orthogonaler Objekte	278
Der Range-Baum	279
Der Intervall-Baum	280
Baumhierarchien	284
8.4 Plane-Sweep-Algorithmen für beliebig orientierte Objekte	287
8.5 Weitere Aufgaben	290
8.6 Literaturhinweise	293
9 Externes Suchen und Sortieren	297
9.1 Externes Suchen: B-Bäume	298
Einfügen und Löschen	302
Overflow	303
Underflow	304
9.2 Externes Sortieren	308
Anfangsläufe fester Länge - direktes Mischen	311
Anfangsläufe variabler Länge - natürliches Mischen	312
Vielweg-Mischen	314
9.3 Weitere Aufgaben	315
9.4 Literaturhinweise	317
Mathematische Grundlagen	A-1
Lösungen zu den Selbsttestaufgaben	A-9
Literatur	A-49
Index	A-57

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.

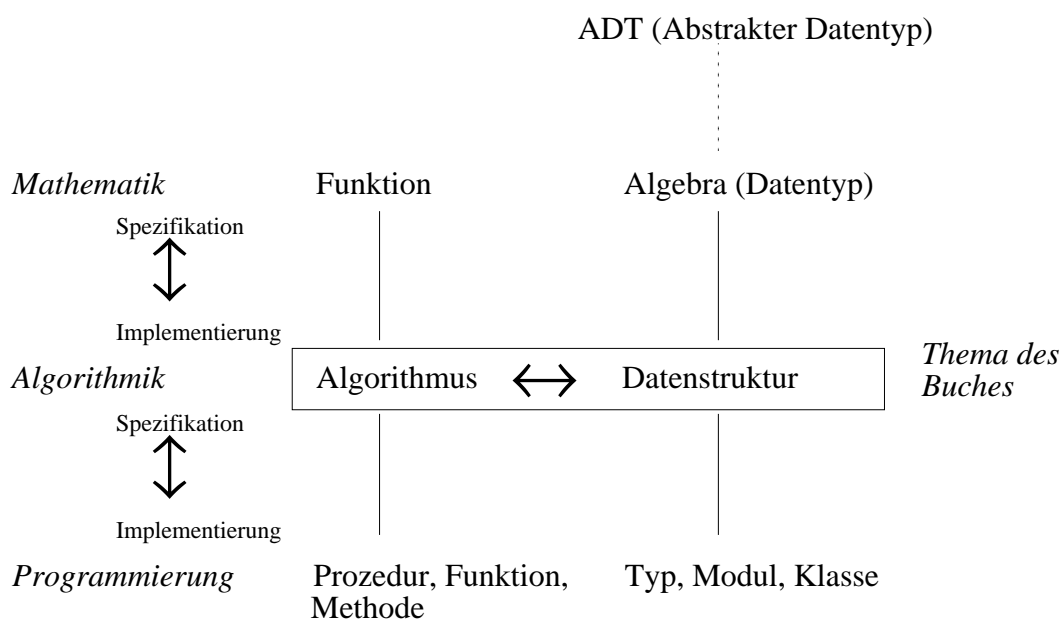


Abbildung 1.1: Abstraktionsebenen von Algorithmen und Datenstrukturen

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 Algorith-

mus 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:

$$\begin{aligned} \text{contains: } & F(\mathbb{Z}) \times \mathbb{Z} \rightarrow BOOL \\ \text{contains}(S, c) &= \begin{cases} true & \text{falls } c \in S \\ false & \text{sonst} \end{cases} \end{aligned}$$

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.

<i>Elementaroperationen</i>		<i>nicht elementare Operationen</i>	
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, \dots , 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.

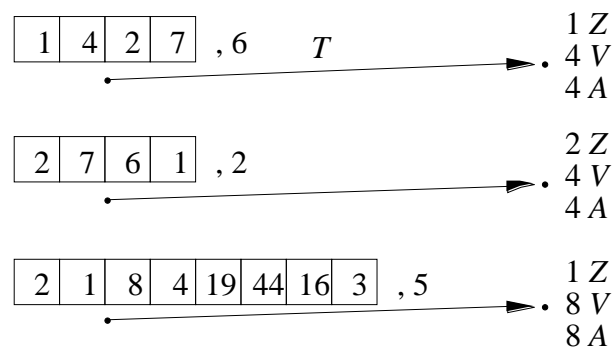


Abbildung 1.2: Anzahl von Elementaroperationen

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.) \square

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:

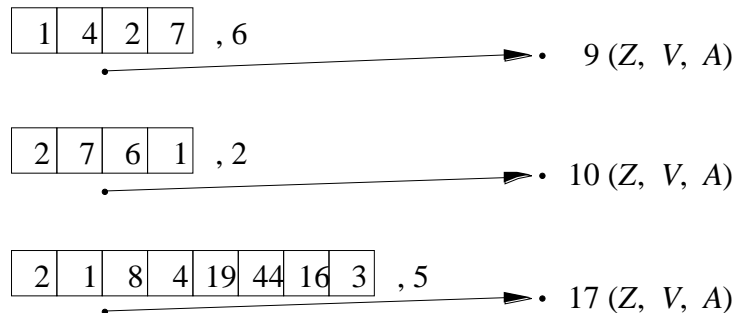


Abbildung 1.3: Erster Abstraktionsschritt

□

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.

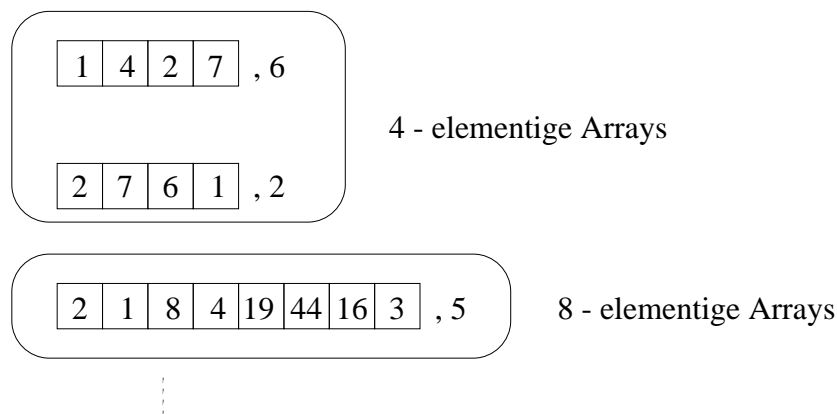


Abbildung 1.4: Zweiter Abstraktionsschritt

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}

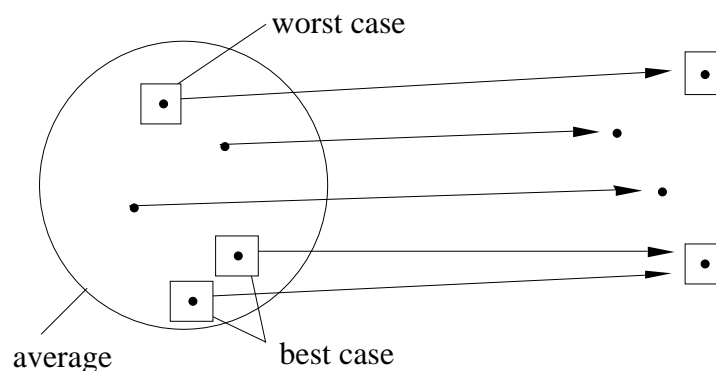


Abbildung 1.5: Dritter Abstraktionsschritt

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

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

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. \square

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 $f: \mathbb{N} \rightarrow \mathbb{R}^+$, $g: \mathbb{N} \rightarrow \mathbb{R}^+$.

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

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

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

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 $f, g: \mathbb{N} \rightarrow \mathbb{N}$ 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:

$$\begin{aligned} T_1(n) &= n + 3 = O(n) && \text{da } n + 3 \leq 2n \quad \forall n \geq 3 \\ T_2(n) &= 3n + 7 = O(n) \\ T_3(n) &= 1000n = O(n) \\ T_4(n) &= 695n^2 + 397n + 6148 = O(n^2) \end{aligned}$$

□

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

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

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:

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

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 “vergrößernde” 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. Algorithmus 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 $O(n) + O(n^2) + O(n^2) = O(n^2)$. \square

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 $O(1)$. \square

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 . \square

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 “eingesortiert” in gewisse Klassen:

	<i>Sprechweise</i>	<i>Typische Algorithmen</i>
$O(1)$	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 können wir jetzt in O-Notation ausdrücken.

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

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.

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

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

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

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

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

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  $m := (low + high) \text{ div } 2$  ;
      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.

```

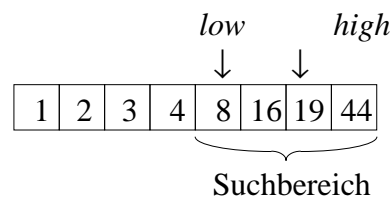


Abbildung 1.6: Algorithmus $contains_3$

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:

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

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

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

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

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

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 $O(n)$.

Selbsttestaufgabe 1.3: Seien $T_1(n)$ und $T_2(n)$ die Laufzeiten zweier Programmstücke P_1 und P_2 . Sei ferner $T_1(n) = O(f(n))$ und $T_2(n) = O(g(n))$. 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 $(f(n)/g(n))_{n \in \mathbb{N}}$ 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 *contains₂*) 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, \dots, 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 $\Omega(n^2)$:

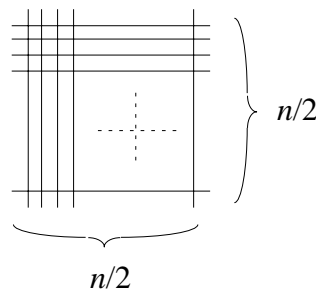


Abbildung 1.7: Schnittpunkte in der Ebene

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. \square

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:

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

Programm₁ 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:

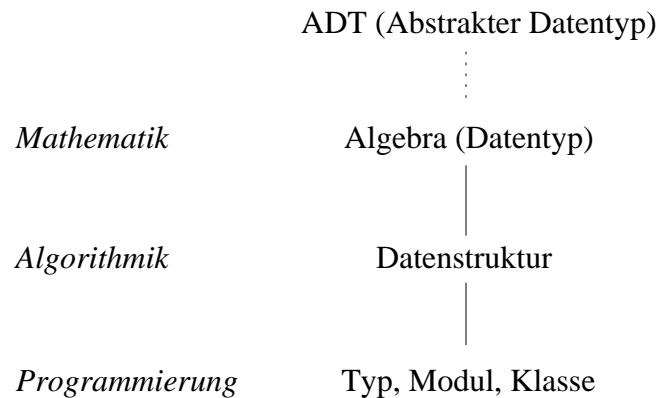


Abbildung 1.8: Abstraktionsebenen von Datenstrukturen

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. \square

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 Bei-

spiele: 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	<i>intset, int, bool</i>		
ops	<i>empty:</i>		\rightarrow <i>intset</i>
	<i>insert:</i>	<i>intset</i> \times <i>int</i>	\rightarrow <i>intset</i>
	<i>delete:</i>	<i>intset</i> \times <i>int</i>	\rightarrow <i>intset</i>
	<i>contains:</i>	<i>intset</i> \times <i>int</i>	\rightarrow <i>bool</i>
	<i>isempty:</i>	<i>intset</i>	\rightarrow <i>bool</i>

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	<i>intset</i>		
sorts	<i>intset, int, bool</i>		
ops	<i>empty:</i>		\rightarrow <i>intset</i>
	<i>insert:</i>	<i>intset</i> \times <i>int</i>	\rightarrow <i>intset</i>
	<i>delete:</i>	<i>intset</i> \times <i>int</i>	\rightarrow <i>intset</i>
	<i>contains:</i>	<i>intset</i> \times <i>int</i>	\rightarrow <i>bool</i>
	<i>isempty:</i>	<i>intset</i>	\rightarrow <i>bool</i>

```

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) =  $\begin{cases} \text{true} & \text{falls } i \in M \\ \text{false} & \text{sonst} \end{cases}$ 
      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:           → intset
      insert:      intset × int → intset
      delete:     intset × int → intset
      contains:  intset × int → bool
      isempty:   intset       → 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 ≠ 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

$$\textit{insert}(\textit{insert}(x, i), i) = \textit{insert}(x, i)$$

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.

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, Entwurfswerkzeuge 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.

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* = $\{0, 1, 2, \dots, p\}$

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

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;

```

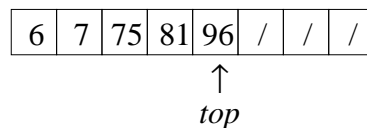


Abbildung 1.9: Beispiel-Ausprägung des Arrays *s*

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:

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

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!

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 ($top = 0$).

-
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.

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

<i>empty</i>	$O(1)$	
<i>insert</i>	$O(n)$	($O(\log n)$ für die Suche und $O(n)$ für das Verschieben)
<i>delete</i>	$O(n)$	(ebenso)
<i>contains</i>	$O(\log n)$	
<i>isempty</i>	$O(1)$	
<i>Platzbedarf</i>	$O(n)$	

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 dargestellt³ 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 verschiedenen 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, Schnittstellendefinitionen 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. Pro-

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

gramme, 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.

Sichtbarkeitsbereich	Programmiersprache			Bedeutung
	Modula-2	C/C++	Java	
für Benutzer sichtbar	Definitionsmodul	Header-Datei	javadoc-Schnittstellenbeschreibung	entspricht der Signatur einer Algebra
für Benutzer verborgen	Implementationsmodul	Implementierung 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.

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

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 <i>elem</i> .
void	Delete (int elem)	Löscht das Element <i>elem</i> aus der Menge.
void	Insert (int elem)	Fügt das Element <i>elem</i> 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

```
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++;
    }
}

```

(*shiftright* 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 Algorithmusbegriffs untersucht (z.B. Turingmaschinen, RAMs, partiell rekursive Funktionen, ...) und über die Churchsche These mit dem intuitiven Algorithmusbegriff 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 Ressourcen verbrauchen‐.

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

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

Eine Relation ‐Rechteckschnitt‐ kann definiert werden:

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

oder

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

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

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 $\Sigma_{w,s}$ in einer Familie von Mengen $\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$, 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

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

ausgedrückt durch

$$\begin{array}{ll} \text{insert} & \in \Sigma_{\text{intset int, intset}} \\ \text{empty} & \in \Sigma_{\epsilon, \text{intset}} \end{array}$$

\square

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

$$f_\sigma : A_{s_1} \times A_{s_2} \times \dots \times A_{s_n} \rightarrow A_s$$

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

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

Selbsttestaufgabe 1.7: 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 1.8: Gegeben sei eine Folge ganzer Zahlen s_1, \dots, s_n , deren Werte alle aus einem relativ kleinen Bereich $[1..N]$ stammen ($n \gg N$). 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 1.9: 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 1.10](#) damit lösbar ist. Definieren Sie für jeden Befehl seine Wirkung auf Programmzähler und Speicherzellen.

Aufgabe 1.10: Implementieren Sie den Algorithmus contains_2 auf einer RAM mit dem in [Aufgabe 1.9](#) entwickelten Befehlssatz. Wieviele RAM-Instruktionen führt dieses RAM-Programm im besten Fall, im schlimmsten Fall und im Durchschnitt aus?

Aufgabe 1.11: 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 1.12: Gegeben sei eine Zahlenfolge $S = s_1, \dots, s_n$, von der bekannt ist, daß sie eine Permutation der Folge $1, \dots, n$ 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 $1, \dots, n$ gleich wahrscheinlich sind? (Hier ist das exakte Ergebnis gefragt, das heißt, $O(n)$ ist keine richtige Antwort.)

Aufgabe 1.13: Gegeben seien Programme P_1, P_2, P_3 und P_4 , die auf einem Rechner R Laufzeiten

$$\begin{aligned} T_1(n) &= a_1 n \\ T_2(n) &= a_2 n \log n \\ T_3(n) &= a_3 n^3 \end{aligned}$$

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

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 1.14: Sei n die Anzahl verschiedener Seminare, die in einem Semester stattfinden. Die Seminare seien durchnummeriert. 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 2002]. Weitere gute Darstellungen finden sich in [Mehlhorn 1984a-c], [Horowitz, Sahni und Anderson-Freed 1993], [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 Darstel-

lung 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 [1998].

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 Ehrlich *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.

Literatur

- Adelson-Velskii, G.M., und Y.M. Landis [1962]. An Algorithm for the Organization of Information. *Soviet Math. Dokl.* 3, 1259-1263.
- 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.
- Baeza-Yates, R.A. [1995]. Fringe Analysis Revisited. *ACM Computing Surveys* 1995, 109-119.
- Banachowski, L., A. Kreczmar und W. Rytter [1991]. Analysis of Algorithms and Data Structures. Addison-Wesley Publishing Co., Reading, Massachusetts.
- Barve, R.D., E.F. Grove und J.S. Vitter [1997]. Simple Randomized Mergesort on Parallel Disks. *Parallel Computing* 23, 601-631.
- Bauer, F. L. und H. Wössner [1984]. Algorithmische Sprache und Programmentwicklung. 2.Aufl., Springer-Verlag, Berlin.
- Bayer, R., und E.M. McCreight [1972]. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 173-189.
- Bayer, R., und K. Unterauer [1977]. Prefix-B-Trees. *ACM Transactions on Database Systems* 2, 11-26.
- BenOr, M. [1983]. Lower Bounds for Algebraic Computation Trees. Proceedings of the 15th Annual ACM Symposium on Theory of Computing, Boston, Massachusetts, 80-86.
- Bentley, J.L. [1977]. Solutions to Klee's Rectangle Problems. Carnegie-Mellon University, Dept. of Computer Science, Manuskript.
- Bentley, J.L. [1979]. Decomposable Searching Problems. *Information Processing Letters* 8, 244-251.
- Bentley, J.L., und D. Wood [1980]. An Optimal Worst-Case Algorithm for Reporting Intersections of Rectangles. *IEEE Transactions on Computers* C-29, 571-577.
- Bentley, J.L., und T. Ottmann [1979]. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers* C-28, 643-647.
- Booth, A.D., und A.J.T. Colin [1960]. On the Efficiency of a New Method of Dictionary Construction. *Information and Control* 3, 327-334.

- Carlsson, S. [1987]. A Variant of Heapsort With Almost Optimal Number of Comparisons. *Information Processing Letters* 24, 247-250.
- Chazelle, B.M. [1986]. Reporting and Counting Segment Intersections. *Journal of Computer and System Sciences*, 156-182.
- Chazelle, B.M., und H. Edelsbrunner [1988]. An Optimal Algorithm for Intersecting Line Segments in the Plane. Proceedings of the 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, 590-600.
- Comer, D. [1979]. The Ubiquitous B-Tree. *ACM Computing Surveys* 11, 121-137.
- Culberson, J. [1985]. The Effect of Updates in Binary Search Trees. In: Proceedings of the 17th Annual ACM Symposium on Theory of Computing, Providence, Rhode Island, 205-212.
- Culik, K., T. Ottmann und D. Wood [1981]. Dense Multiway Trees. *ACM Transactions on Database Systems* 6, 486-512.
- Deo, N. und C. Pang [1984]. Shortest-Path Algorithms: Taxonomy and Annotation. *Networks* 14, 275-323.
- Dijkstra, E.W. [1959]. A Note on Two Problems in Connexion With Graphs. *Numerische Mathematik* 1, 269-271.
- Dijkstra, E.W. [1982]. Smoothsort, an Alternative for Sorting in Situ. *Science of Computer Programming* 1, 223-233. Siehe auch: Errata, *Science of Computer Programming* 2 (1985), 85.
- Doberkat, E.E. [1984]. An Average Case Analysis of Floyd's Algorithm to Construct Heaps. *Information and Control* 61, 114-131.
- Dvorak, S., und B. Durian [1988]. Unstable Linear Time $O(1)$ Space Merging. *The Computer Journal* 31, 279-283.
- Edelsbrunner, H. [1980]. Dynamic Data Structures for Orthogonal Intersection Queries. Technische Universität Graz, Institute für Informationsverarbeitung, Graz, Österreich, Report F 59.
- Edelsbrunner, H. [1983]. A New Approach to Rectangle Intersections. *International Journal of Computer Mathematics* 13, 209-229.
- Edelsbrunner, H. [1987]. Algorithms in Combinatorial Geometry. Springer-Verlag, Berlin.
- Edelsbrunner, H., und H.A. Maurer [1981]. On the Intersection of Orthogonal Objects. *Information Processing Letters* 13, 177-181.
- 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.
- Euler, L. [1736]. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Petropolitanae* 8, 128-140.

- Flanagan, D. [2005]. Java in a Nutshell, Fifth Edition. O'Reilly & Associates.
- Floyd, R.W. [1962]. Algorithm 97: Shortest Paths. *Communications of the ACM* 5, 345.
- Floyd, R.W. [1964]. Algorithm 245, Treesort 3. *Communications of the ACM* 7, 701.
- Ford, L.R., und S.M. Johnson [1959]. A Tournament Problem. *American Mathematical Monthly* 66, 387-389.
- Fredman, M.L., und R.E. Tarjan [1987]. Fibonacci Heaps and Their Use in Network Optimization. *Journal of the ACM* 34, 596-615.
- Gabow, H.N., Z. Galil, T.H. Spencer und R.E. Tarjan [1986]. Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs. *Combinatorica* 6, 109-122.
- Galil, Z., und G.F. Italiano [1991]. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Computing Surveys* 23, 319-344.
- Ghosh, S., und M. Senko [1969]. File Organization: On the Selection of Random Access Index Points for Sequential Files. *Journal of the ACM* 16, 569-579.
- Gibbons, A. [1985]. Algorithmic Graph Theory. Cambridge University Press, Cambridge.
- 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.
- Gütting, R.H. [1984a]. An Optimal Contour Algorithm for Iso-Oriented Rectangles. *Journal of Algorithms* 5, 303-326.
- Gütting, R.H. [1984b]. Optimal Divide-and-Conquer to Compute Measure and Contour for a Set of Iso-Rectangles. *Acta Informatica* 21, 271-291.
- Gütting, R.H. [1985]. Divide-and-Conquer in Planar Geometry. *International Journal of Computer Mathematics* 18, 247-263.
- Gütting, R.H., und D. Wood [1984]. Finding Rectangle Intersections by Divide-and-Conquer. *IEEE Transactions on Computers* C-33, 671-675.
- Gütting, R.H., und W. Schilling [1987]. A Practical Divide-and-Conquer Algorithm for the Rectangle Intersection Problem. *Information Sciences* 42, 95-112.
- Harary, F. [1994]. Graph Theory. Addison-Wesley Publishing Co., Reading, Massachusetts.
- Hart, P.E., N.J. Nilsson und B. Raphael [1968]. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on System Science and Cybernetics SSC-4*, 100-107.
- Hibbard, T.N. [1962]. Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting. *Journal of the ACM* 9, 13-28.

- Hoare, C.A.R. [1962]. Quicksort. *The Computer Journal* 5, 10-15.
- Hopcroft, J.E., und J.D. Ullman [1973]. Set Merging Algorithms. *SIAM Journal on Computing* 2, 294-303.
- 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 [1993]. Fundamentals of Data Structures in C. Computer Science Press, New York.
- Huang, B.C. und M.A. Langston [1988]. Practical In-Place Merging. *Communications of the ACM* 31, 348-352.
- Johnson, D.B. [1977]. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM* 24, 1-13.
- Jungnickel, D. [1994]. Graphen, Netzwerke und Algorithmen. Spektrum Akademischer Verlag, Heidelberg.
- Jungnickel, D. [1998]. Graphs, Networks, and Algorithms. Springer-Verlag, Berlin.
- Kemp, R. [1989]. Pers. Mitteilung an I. Wegener, zitiert in [Wegener 1990b].
- Khuller, S. und B. Raghavachari [1996]. Graph and Network Algorithms. *ACM Computing Surveys* 28, 43-45.
- Klaeren, H.A. [1983]. Algebraische Spezifikation. Eine Einführung. Springer-Verlag, Berlin.
- Klein, R. [1997]. Algorithmische Geometrie. Addison-Wesley-Longman, Bonn.
- Knuth, D.E. [1997]. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. 3rd Ed., Addison-Wesley Publishing Co., Reading, Massachusetts.
- Knuth, D.E. [1998]. The Art of Computer Programming, Vol. 3: Sorting and Searching. 2nd Ed., Addison-Wesley Publishing Co., Reading, Massachusetts.
- Kronrod, M.A. [1969]. An Optimal Ordering Algorithm Without a Field of Operation. *Dokladi Akademia Nauk SSSR* 186, 1256-1258.
- Krüger, G. [2005]. Handbuch der Java-Programmierung. 4. Aufl., Addison-Wesley, München.
- Kruskal, J.B. [1956]. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society* 71, 48-50.
- Küspert, K. [1983]. Storage Utilization in B*-Trees With a Generalized Overflow Technique. *Acta Informatica* 19, 35-55.
- Laszlo, M. [1996]. Computational Geometry and Computer Graphics in C++. Prentice Hall, Englewood Cliffs, NJ.
- Lee, D.T. [1995]. Computational Geometry. In: A. Tucker (Ed.), Handbook of Computer Science and Engineering. CRC Press, Boca Raton, FL.
- Lee, D.T. [1996]. Computational Geometry. *ACM Computing Surveys* 28, 27-31.

- Lipski, W., und F.P. Preparata [1980]. Finding the Contour of a Union of Iso-Oriented Rectangles. *Journal of Algorithms* 1, 235-246.
- Lockemann, P.C., und J.W. Schmidt (Hrsg.) [1987]. Datenbank-Handbuch. Springer-Verlag, Berlin.
- Loeckx, J., H.D. Ehrich und M. Wolf [1996]. Specification of Abstract Data Types. Wiley-Teubner Publishers, Chichester, Stuttgart.
- Lum, V.Y., P.S.T. Yuen und M. Dodd [1971]. Key-To-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files. *Communications of the ACM* 14, 228-239.
- Manber, U. [1989]. Introduction to Algorithms. A Creative Approach. Addison-Wesley Publishing Co., Reading, Massachusetts.
- Maurer, W.D., und T. Lewis [1975]. Hash Table Methods. *ACM Computing Surveys* 7, 5-20.
- McCreight, E.M. [1980]. Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles. Xerox Palo Alto Research Center, Palo Alto, California, Report PARC-CSL-80-9.
- McCreight, E.M. [1985]. Priority Search Trees. *SIAM Journal on Computing* 14, 257-276.
- 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.
- Meyer, B. [1990]. Lessons from the Design of the Eiffel Libraries. *Communications of the ACM* 33, 68-88.
- Moffat, A., und T. Takaoka [1987]. An All Pairs Shortest Path Algorithm With Expected Time $O(n^2 \log n)$. *SIAM Journal on Computing* 16, 1023-1031.
- Morris, R. [1968]. Scatter Storage Techniques. *Communications of the ACM* 11, 35-44.
- Nagl, M. [1999]. Die Software-Programmiersprache ADA 95. Entwicklung großer Systeme in ADA. Vieweg-Verlag, Wiesbaden.
- Nakamura, T., und T. Mizzogushi [1978]. An Analysis of Storage Utilization Factor in Block Split Data Structuring Scheme. Proceedings of the 4th Intl. Conference on Very Large Data Bases, 489-495.
- Nievergelt, J., und E.M. Reingold [1973]. Binary Search Trees of Bounded Balance. *SIAM Journal on Computing* 2, 33-43.
- Nievergelt, J., und F.P. Preparata [1982]. Plane-Sweep Algorithms for Intersecting Geometric Figures. *Communications of the ACM* 25, 739-747.

- Nievergelt, J., und K.H. Hinrichs [1993]. Algorithms and Data Structures: With Applications to Graphics and Geometry. Prentice-Hall, Englewood Cliffs, N.J.
- Nilsson, N.J. [1982]. Principles of Artificial Intelligence. Springer-Verlag, Berlin.
- O'Rourke, J. [1998]. Computational Geometry in C. 2nd Ed., Cambridge University Press, Cambridge, UK.
- Ottmann, T., und P. Widmayer [1982]. On the Placement of Line Segments into a Skeleton Structure. Universität Karlsruhe, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Report 114.
- Ottmann, T., und P. Widmayer [2002]. Algorithmen und Datenstrukturen. 4. Aufl., Spektrum Akademischer Verlag, Heidelberg.
- Ottmann, T., und P. Widmayer [1995]. Programmierung mit PASCAL. 6. Aufl., Teubner-Verlag, Stuttgart.
- Papadimitriou, C.H., und K Steiglitz [1998]. Combinatorial Optimization: Networks and Complexity. Dover Publications.
- Pardo, L.T. [1977]. Stable Sorting and Merging With Optimal Space and Time Bounds. *SIAM Journal on Computing* 6, 351-372.
- Peterson, W.W. [1957]. Addressing for Random Access Storage. *IBM Journal of Research and Development* 1, 130-146.
- Preparata, F.P., und M.I. Shamos [1985]. Computational Geometry. An Introduction. Springer-Verlag, Berlin.
- Prim, R.C. [1957]. Shortest Connection Networks and Some Generalizations. *Bell System Technical Journal* 36, 1389-1401.
- Radke, C.E. [1970]. The Use of Quadratic Residue Search. *Communications of the ACM* 13, 103-105.
- Sack, J. und J. Urrutia (Hrsg.) [1996]. Handbook on Computational Geometry. Elsevier Publishers, Amsterdam.
- Salzberg, B. [1989]. Merging Sorted Runs Using Large Main Memory. *Acta Informatica* 27, 195-215.
- Salzberg, B., A. Tsukerman, J. Gray, M. Stewart, S. Uren und B. Vaughan [1990]. Fast-Sort: A Distributed Single-Input Single-Output External Sort. Proceedings of the ACM SIGMOD Intl. Conference on Management of Data, Atlantic City, NJ, 94-101.
- Schiedermeier, R. [2005]. Programmieren mit Java. Eine methodische Einführung. Pearson Studium, München.
- Schmitt, A. [1983]. On the Number of Relational Operators Necessary to Compute Certain Functions of Real Variables. *Acta Informatica* 19, 297-304.
- Sedgewick, R. [1978]. Quicksort. Garland Publishing Co., New York.
- 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.
- Severance, D., und R. Duhne [1976]. A Practitioner's Guide to Addressing Algorithms. *Communications of the ACM* 19, 314-326.
- Shamos, M.I. [1975]. *Problems in Computational Geometry*. Unveröffentlichtes Manuskript.
- Shamos, M.I. [1978]. *Computational Geometry*. Ph. D. Thesis, Dept. of Computer Science, Yale University.
- Sharir, M. [1981]. A Strong-Connectivity Algorithm and Its Application in Data Flow Analysis. *Computers and Mathematics with Applications* 7, 67-72.
- Shell, D.L. [1959]. A High-Speed Sorting Procedure. *Communications of the ACM* 2, 30-32.
- Shell, D.L. [1971]. Optimizing the Polyphase Sort. *Communications of the ACM* 14, 713-719.
- Sheperdson, J.C., und H.E. Sturgis [1963]. Computability of Recursive Functions. *Journal of the ACM* 10, 217-255.
- Six, H.W., und D. Wood [1982]. Counting and Reporting Intersections of d -Ranges. *IEEE Transactions on Computers* C-31, 181-187.
- Six, H.W., und L. Wegner [1981]. EXQUISIT: Applying Quicksort to External Files. Proceedings of the 19th Annual Allerton Conference on Communication, Control, and Computing, 348-354.
- Spira, P.M. [1973]. A New Algorithm for Finding All Shortest Paths in a Graph of Positive Arcs in Average Time $O(n^2 \log^2 n)$. *SIAM Journal on Computing* 2, 28-32.
- Standish, T.A. [1998]. *Data Structures in Java*. Addison-Wesley Publishing Co., Reading, Massachusetts.
- Steinhaus, H. [1958]. *One Hundred Problems in Elementary Mathematics*. Problem 52. Pergamon Press, London.
- Tarjan, R.E. [1975]. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM* 22, 215-225.
- van Leeuwen, J., und D. Wood [1981]. The Measure Problem for Rectangular Ranges in d -Space. *Journal of Algorithms* 2, 282-300.
- Vitter, J.S. [2001]. External Memory Algorithms and Data Structures: Dealing With Massive Data. *ACM Computing Surveys* 33, 209-271.
- Wagner, R.E. [1973]. Indexing Design Considerations. *IBM Systems Journal* 12, 351-367.
- Warshall, S. [1962]. A Theorem on Boolean Matrices. *Journal of the ACM* 9, 11-12.

- Wedekind, H. [1974]. On the Selection of Access Paths in a Data Base System. In: J.W. Klimbie und K.L. Koffeman (eds.), *Data Base Management*. North-Holland Publishing Co., Amsterdam.
- Wegener, I. [1990a]. Bottom-Up-Heapsort, a New Variant of Heapsort Beating on Average Quicksort (If n Is Not Very Small). *Proceedings of the 15th International Conference on Mathematical Foundations of Computer Science, Banská Bystrica, Czechoslovakia*, 516-522.
- Wegener, I. [1990b]. Bekannte Sortierverfahren und eine Heapsort-Variante, die Quicksort schlägt. *Informatik-Spektrum 13*, 321-330.
- Wegner, L. [1985]. Quicksort for Equal Keys. *IEEE Transactions on Computers C-34*, 362-366.
- Weiss, M.A. [1998]. *Data Structures and Problem Solving Using Java*. Addison-Wesley Publishing Co., Reading, Massachusetts.
- West, D.B. [2001]. *Introduction to Graph Theory*. 2nd Ed., Prentice Hall, Englewood Cliffs, NJ.
- Williams, J.W.J. [1964]. Algorithm 232: Heapsort. *Communications of the ACM 7*, 347-348.
- Wilson, R.J. [1996]. *Introduction to Graph Theory*. 4th Ed., Prentice Hall, Englewood Cliffs, NJ.
- Windley, P.F. [1960]. Trees, Forests, and Rearranging. *The Computer Journal 3*, 84-88.
- Wirth, N. [1991]. *Programmieren in Modula-2*. 2.Aufl., Springer-Verlag, Berlin.
- Wirth, N. [2000]. *Algorithmen und Datenstrukturen. Pascal-Version*. 5. Aufl., Teubner-Verlag, Stuttgart.
- Wirth, N. [1996]. *Algorithmen und Datenstrukturen mit Modula-2*. 5. Aufl., Teubner-Verlag, Stuttgart.
- Wood, D. [1993]. *Data Structures, Algorithms, and Performance*. Addison-Wesley Publishing Co., Reading, Massachusetts.
- Yao, A.C. [1975]. An $O(|E| \log \log |V|)$ Algorithm for Finding Minimum Spanning Trees. *Information Processing Letters 4*, 21-23.
- Yao, A.C. [1985]. On Random 2-3 Trees. *Acta Informatica 9*, 159-170.

Index

Symbole

Ω -Notation 20

Numerisch

2-3-Baum 308

A

A*-Algorithmus 235
Abbildung 91
Abkömmling 95
abstrakter Datentyp 1, 2, 34, 37
Abstraktionsebene 1
Ada 61
addcomp 158
adjazent 200
Adjazenzlisten 204, 221
Adjazenzmatrix 202, 216
ADT 34
Aggregation 39
Akkumulator 7
aktives Segment 287
Aktivierungs-Record 86
Algebra 1, 2, 22
algebraische Spezifikation 37
algebraischer Entscheidungsbaum 197
algorithmische Geometrie 237
Algorithmus 1, 32
Algorithmus von Dijkstra 211, 218
Algorithmus von Floyd 218
Algorithmus von Kruskal 230
all pairs shortest path-Problem 218
allgemeiner Baum 101
allgemeiner Suchbaum 298
allgemeines Sortierverfahren 188
amortisierte Laufzeit 160
Analyse 2

Analyse von Algorithmen 37
ancestor 95
append 64, 82
Äquivalenzrelation 156
Array 39
Assemblersprache 7
atomarer Datentyp 41
Aufzählungstyp 58
Ausgangsgrad 200
average case 10
AVL-Baum 109, 141, 182, 245
Axiom 24, 34

B

Bag 152
balance 305
balancierter Suchbaum 141
Baum 92
Baum beschränkter Balance 166
Baumhierarchie 278, 284
Baumsortieren 182
B-Baum 299
Behälter 115, 192
beliebig orientierte Segmente 287
beliebig orientiertes Objekt 240
best case 10
bester Fall 10
Betriebssystem 52
binäre Suche 17
binärer Suchbaum 109, 129, 278
Bitvektor-Darstellung 110
Blatt 94
Block 297
bol 66
Bottom-Up-Heapsort 186
breadth-first 228
breadth-first-Spannbaum 207
breadth-first-traversal 207
Breitendurchlauf 205, 207

Bruder 95
BubbleSort 171
Bucket 192
bucket 115
BucketSort 193

C

Clever Quicksort 181
CN(i) 251, 257
computational geometry 237
concat 64, 82
contracted segment tree 294

D

DAC-Algorithmus 175
DAG 209
Datenobjekt 27
Datenspeicher 7
Datenstruktur 1, 2, 22, 34
Datentyp 1, 22, 34
Definitionsmodul 28
degenerierter binärer Suchbaum 135
delete 66, 113, 133, 302
deletemin 153, 155
denotationelle Spezifikation 37
depth-first 228
depth-first-Spannbaum 207
depth-first-traversal 206
dequeue 89
Dereferenzierung 50
descendant 95
Dictionary 109, 113
difference 110
directed acyclic graph 209
direktes Auswählen 168
direktes Mischen 311
dispose 52
Distanzproblem 293
Divide-and-Conquer 171, 241, 277
Divide-and-Conquer-Algorithmus 258
Divisionsmethode 128

domain 91
Doppel-Hashing 127
Doppelrotation 143
Duplikat 63
dynamisch 278

E

eindimensionales Punkteinschluß-
Mengenproblem 264
einfacher Pfad 201
Eingangsgrad 200
Einheitskosten 7
Elementaroperation 6, 7
EMergeSort 310
empty 64, 109
enqueue 89
Entscheidungsbaum 189
enumerate 109, 110
eol 66
Ersetzungs-Auswahl 312
Erwartungswert 2
Euler-Konstante 5
Expansion eines Graphen 205
exponentiell 15
extern 167
externe Datenstruktur 297
externer Algorithmus 297
externes Verfahren 167
Exzentrizität 233

F

Feld 42
Fibonacci-Zahlen 150, 7
FIFO 89
find 66, 157, 158, 161
findx 180
first 64, 82
Fragmentintervall 280
freier Baum 227
freier Punkt 292
front 65, 89

Funktion 1, 3, 23

G

Garbage Collection 52
Geburtstagsparadoxon 117
gegabelter Pfad 251, 257
gerichteter azyklischer Graph 209
gerichteter Graph 199, 200
geschlossenes Hashing 116, 118
Gesetz 24
getrennte Darstellung 259
Gewicht eines Baumes 165
gewichtsbalancierter Baum 166
Gleichverteilung 10
Grad eines Baumes 102
Grad eines Knotens 102, 200
Graph 199

H

Halbrechteck 295
Haltepunkt 288
harmonische Zahl 125, 5
Hashfunktion 115, 128
Hashing 109
Haufen 153
Heap 153
Heapsort 153, 171, 182
heterogene Algebra 22, 34
Heuristikfunktion 235
Höhe eines Baumes 95

I

ideales Hashing 120
in situ 167, 197
index sequential access method 317
Indextyp 42
Infix-Notation 99
InitialRuns 309, 311

innerer Knoten 95
inorder 97
Inorder-Durchlauf 182
insert 66, 110, 113, 132, 153, 154, 302
InsertionSort 168, 171
Instruktion 7
intern 167
internes Verfahren 167
intersection 110
Intervall-Baum 278, 280
Intervallschnitt-Suche 282
inverse Adjazenzliste 204
inzident 200
ISAM-Technik 317
isempty 64
isomorph 25

K

kanonisch bedeckte Knoten 251
Kante 93, 200
Kaskaden-Mergesort 316
key 98
key-Komponente 167, 182
Klammerstruktur 84, 93
Knoten 93, 200
Knotenliste 250
Knotenmarkierung 129
Kollision 116
Komplexität der Eingabe 6, 21
Komplexität des Problems 20
Komplexitätsklasse 9, 20
Komponente 157, 227
Königsberger Brückenproblem 209, 234
konstant 15
Kontur 266, 272
Konturproblem 272
Konturzyklus 272
konvexe Hülle 240, 293
Korrektheit 5
Kostenmaß 7, 297
Kostenmatrix 220

L

Länge eines Pfades 95
last 65
Lauf 309
Laufzeit 2
Laufzeitsystem 52
leere Liste 64, 65
left 98
LIFO 84
linear 15
lineares Sondieren 119, 126
links-vollständiger partiell geordneter Baum 154
linsect 259, 263
Liste 64
Liste im Array 78
Listenkopf 107
Listenschwanz 107
logarithmisch 15
logarithmisches Kostenmaß 7

M

maketree 98
mapping 91
markierte Adjazenzmatrix 203
markierter Graph 202
Maschinenmodell 37
Maschinensprache 7
Maßproblem 254
Mehrphasen-Mischsortieren 317
mehrsortige Algebra 22, 34
member 113, 114, 131
member-Problem 239
Menge 109
Mengenoperation 60
Merge 310
merge 157, 158, 159, 161, 304, 305
MergeSort 172
Mergesort 171, 308
minimaler Spannbaum 228
Mittel-Quadrat-Methode 128
Modell 25, 34

modifizierter Segment-Baum 256, 257
Modul 1, 2
monomorph 25, 34
Multimenge 152
Multiset 152

N

Nachbar 304
Nachfahr 95
Nachfolger 58
natürliches Mischen 311
nearest-neighbour-Problem 241
next 65
nil 49

O

offene Adressierung, 119
offenes Hashing 116, 117
offset 48
O-Notation 11, 12, 15, 19, 21
Operandenstack 84
Operation 22, 33
Operationssymbol 23, 33
Operatorenstack 84
optimal 20
Ordnung 63, 110
orthogonal 240
Overflow 302, 303, 304
overflow 116

P

partiell geordneter Baum 153
Partition 156
partition 158
PASCAL 61
Permutation 167, 190
persistent 297
Pfad 95, 200
Pfadkompression 162
Phase 308

Plane-Sweep 241, 277
Plattenspeicher 297
Platzbedarf 2, 6
Pointer 49
Polygonschnitt-Algorithmus 295
polymorph 25, 34
Polynom 106
polynomiell 15
polyphase merging 317
pop 82
Postfix-Notation 99
postorder 97
pqueue 153
Präfix-Notation 99
pred 58
preorder 97
previous 65
Primärkollision 127
Priorität 152
Prioritätssuchbaum 295
Priority Queue 109, 152, 217
priority queue 290
Probe 120
Problem 32
Programmspeicher 7
Programmzähler 7
Prozedur 1, 2
Prozedurinkarnation 86
Punkteinschluß-Problem 247, 248
Punkteinschlußsuche 281
push 82

Q

quadratisch 15
Quadratisches Sondieren 126
Queue 89
QuickSort 172, 175
Quicksort 171, 176, 180, 308, 317

R

Radixsort 194

Radixsortieren 194
RAM 7, 32, 37
Random-Access-Maschine 7
range 91
Range-Baum 278, 279
Range-Intervall-Baum 284
Range-Range-Binär-Baum 286
rank 315
rappend 89
Raster 280
rationaler Entscheidungsbaum 197
real RAM 7, 37
Rebalancieren 142
Rebalancieroperation 141
Rechteckeinschluß-Problem 247
Rechteckschnitt-Problem 247, 283
Record 39, 47, 61
Register 7
Registermaschine 37
rehashing 119
Reheap 186
Reihung 42
Rekursionsgleichung 15, 173, 174, 5
rekursive Invariante 260, 268
rekursive Struktur 97
Repräsentation 39
rest 64, 82
retrieve 66
RI-Baum 284
right 98
Ring 105
Rotation 142
RRB-Baum 286

S

SB-Baum 285
schlimmster Fall 10
Schlüssel 115
Schlüsseltransformation 115
Schlüsselvergleichs-Sortieralg. 191
Schlüsselvergleichs-Verfahren 188
Schlüsselwert 167
schrittweise Verfeinerung 34

Segment-Baum 250, 256, 278
 Segment-Binär-Baum 285
 SegmentIntersectionDAC 262
 SegmentIntersectionPS 244
 Segment-Intervall-Baum 284
 Segmentschnitt-Problem 242, 287
 Seite 297
 Seitenfolge 308
 Seitenzugriff 297
 Sekundärkollision 127
 SelectionSort 168
 Selektion 42, 48
 Selektor 48
 Semantik 23
 semidynamisch 278
 separate chaining 118
 Sequenz 63
 Set 60
 set 110
 Shellsort 195
 SI-Baum 284
 Signatur 23, 33, 34
 single source shortest path-Problem 211
 Sorte 23, 33
 Sortieralgorithmus 167
 Sortierproblem 167, 188
 Spannbaum 228
 spannender Wald 207
 Speicherplatzausnutzung 297, 317
 Speicherstruktur 297
 Speicherzelle 7
 Spezifikation 1
 Spezifikation als abstrakter Datentyp 24
 Spezifikation als Algebra 23
 split 303
 SS-Baum 285
 STAB (p) 280
 stabil 168
 Stack 82
 stack 82
 Stackebene 84
 Standard-Heapsort 182
 Stapel 82
 stark verbunden 201

stark verbundene Komponente 201
 starke Komponente 201, 223
 Station 288
 statisch 278
 Stirling'sche Formel 191
 Streifenmenge 266
 stripes 270
 StripesDAC 270
 StrongComponents 223
 Strukturinvariante 141
 SUB (p) 279
 succ 58
 Suchen 109
 Suchproblem 239
 Sweep-Event-Struktur 244
 Sweepline 242
 Sweepline-Status-Struktur 243

T

tail 107
 Teilbaum 94, 95
 Teilgraph 201
 Teilheap 182
 Tiefe eines Knotens 95
 Tiefendurchlauf 205, 206
 tile tree 294
 top 82
 Trägermenge 23
 transitive Hülle 223
 tree 98
 Turingmaschine 7, 32
 Türme von Hanoi 87
 Typ 1, 39
 Typkonstruktor 68
 Typsystem 40

U

Überlauf 116
 unabhängig 127
 Underflow 302, 304
 underflow 304

ungerichteter Graph 199, 227
uniformes Hashing 120
union 110
universale Algebra 22
Universum 110
Unterbereichstyp 59
untere Schranke 20

V

Vater 94
verbunden 227
Vielweg-Mischen 314, 317
Vielweg-Suchbaum 298, 299
vollständiger binärer Baum 96, 100
von Neumann, John 196
Vorfahr 95
Vorgänger 58
Voronoi-Diagramm 293

W

Wahrscheinlichkeitsraum 2
Wald 102
Warshalls Algorithmus 223
Warteschlange 90, 152
worst case 10
Wörterbuch 113
Wurzel 94, 205
Wurzelgraph 205

Z

Zeiger 49
Zeigerstruktur 97
Zentrum 233
Zickzack-Paradigma 295
Zufallsvariable 2
Zugriffscharakteristika 297
zusammenhängend 240
zyklische Liste 105
Zyklus 201, 227

