

LEHRBUCH

Ralf Hartmut Güting  
Stefan Dieker

# Datenstrukturen und Algorithmen

4. Auflage



Springer Vieweg

---

# Datenstrukturen und Algorithmen

---

Ralf Hartmut Güting · Stefan Dieker

# Datenstrukturen und Algorithmen

4., erweiterte und überarbeitete Auflage



Springer Vieweg

Ralf Hartmut Güting  
Fakultät für Mathematik und Informatik  
Fernuniversität in Hagen  
Hagen, Deutschland

Stefan Dieker  
Hamminkeln, Deutschland

ISBN 978-3-658-04675-0      ISBN 978-3-658-04676-7 (eBook)  
<https://doi.org/10.1007/978-3-658-04676-7>

Die Deutsche Nationalbibliothek verzeichnetet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Springer Vieweg  
© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 1992, 2003, 2004, 2018  
Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags.  
Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature

Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

*Für Edith, Nils David und Helge Jonathan*

*R.H.G.*

*Für Sabine, Tobias und Johannes*

*S.D.*

## **Vorwort zur ersten Auflage**

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. Dieses Buch möchte entsprechende Kenntnisse und Fähigkeiten vermitteln; es wendet sich vornehmlich an Studierende der Informatik im Grundstudium. Vorausgesetzt werden lediglich Grundkenntnisse der Programmierung, wie sie etwa durch Umgang mit einer Sprache wie PASCAL gegeben sind. Zum Verständnis werden zwar keine tiefergehenden mathematischen Vorkenntnisse, aber doch die Bereitschaft zum Umgang mit einfacher mathematischer Notation benötigt. Gelegentlich kommen bei der Analyse von Algorithmen auch etwas anspruchsvollere Berechnungen vor. Diese werden sorgfältig erklärt, und die benötigten Techniken werden im Rahmen des Buches eingeübt.

Grundlage für das Buch waren Vorlesungen zu Datenstrukturen und zu geometrischen Algorithmen, die ich an der Universität Dortmund gehalten habe; diese wurden später zu einem Kurs "Datenstrukturen" für die Fernuniversität Hagen ausgearbeitet und dort bereits einige Jahre eingesetzt. Der Stoffumfang des Buches umfaßt den einer einsemestrigen vierstündigen Vorlesung, die in Dortmund und Hagen jeweils im 3. Semester angeboten wird.

Es gibt zahlreiche Bücher zu Algorithmen und Datenstrukturen. Dieses Buch setzt folgende Akzente:

- Es wurde versucht, in relativ kompakter Form alle wichtigen Themenbereiche des Gebietes abzudecken. Dies erlaubt es, das Buch komplett als Vorlesungsgrundlage zu benutzen und darüber den Stoffumfang einer entsprechenden Prüfung zu definieren. Die meisten anderen Bücher sind wesentlich umfangreicher oder behandeln wichtige Themenbereiche (Graphalgorithmen, geometrische Algorithmen) nicht. Im ersten Fall hat ein Dozent die Möglichkeit, aber auch die Bürde, in erheblichem Maße auszuwählen.
- Die kompakte Darstellung wurde zum Teil erreicht durch Konzentration auf die Darstellung der wesentlichen Ideen. In diesem Buch 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.
- Die Forschung zu Algorithmen und Datenstrukturen ist in den letzten 15 Jahren nicht stehengeblieben. Vor allem die Behandlung geometrischer Objekte, also das Teilgebiet der *algorithmischen Geometrie*, hat in dieser Zeit einen stürmischen Aufschwung genommen. Dies wurde berücksichtigt, indem einerseits neue Ergebnisse zu klassischen Themen (etwa Sortieralgorithmen) mit aufgenommen wurden. Darüber hinaus setzt das Buch 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. Ich bin der Meinung, 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* eingebütt; 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 eingebütt.
- Schließlich begreift man, daß all die Algorithmen und Datenstrukturen der vorhergehenden Kapitel als *Werkzeuge* zur Konstruktion neuer Algorithmen eingesetzt werden können.

Man kann das gesamte Gebiet auf verschiedene Art gliedern, etwa nach Problembe reichen und damit Algorithmen, nach Datentypen, oder nach Datenstrukturen. Dieses Buch führt in den Kapiteln 3 und 4 zunächst grundlegende Datentypen ein; danach "kippt" die Gliederung mit der Behandlung von Graphalgorithmen, Sortieralgorithmen und geometrischen Algorithmen auf die algorithmische Seite. Das letzte Kapitel behandelt *externes* Suchen und Sortieren. Ich fand es angemessener, B-Bäume und externes Mischsortieren getrennt zu behandeln, also nicht zusammen mit internen Baumstrukturen und Sortieren, da hier ein ganz anderes Kostenmaß zugrunde liegt.

Dem Text folgt ein kleiner Anhang, in dem die benötigten mathematischen Grundkenntnisse “importiert” werden. Fast jedes Kapitel schließt mit Aufgaben einfachen bis mittleren Schwierigkeitsgrades und Literaturhinweisen ab. Die Literaturhinweise beziehen sich oft auch auf Material, das den Stoff des Kapitels ergänzt oder weiterführt.

Wie schon erwähnt, ist das Buch vor allem als Grundlage oder Begleitmaterial für eine Vorlesung im Grundstudium gedacht. Darüber hinaus kann der Stoff des Kapitels über geometrische Algorithmen als Kern einer Spezialvorlesung im Hauptstudium benutzt werden.

Am Zustandekommen dieses Buches waren etliche Mitarbeiter, Studenten, Freunde und Kollegen beteiligt, denen ich herzlich danken möchte. Bernd Meyer und Gerd Westerman haben mitgeholfen, das Dortmunder Skript zu einem Kurs für die Fernuniversität umzuarbeiten; sie haben bei der Formulierung von Aufgaben auch eigene Entwürfe eingebracht. Auch Jürgen Freitag und Yan Liu haben Aufgaben mitentworfen. Für das Erfassen des Vorlesungstextes mit Zeichnungen, Setzen von Formeln usw. danke ich besonders Arno Aßmann. Er hat auch mit viel Mühe den Kurstext in das neue Buchformat umgesetzt und bei all diesen Arbeiten großes Geschick und Verständnis bewiesen. Viele Studenten der Universität Dortmund und vor allem der Fernuniversität haben Korrekturen und Verbesserungsvorschläge zu früheren Versionen des Textes eingebracht. Wegen ihres besonderen Einsatzes dabei möchte ich Michael Kohler, Ingrid Nagel und Matthias Wolpers namentlich erwähnen. Teile der Buchversion wurden von Arno Aßmann, Martin Erwig, Bernd Meyer, Markus Schneider und Gerd Westerman gelesen und kommentiert; ich danke ihnen für die oft sehr detaillierten Korrekturen und Verbesserungsvorschläge.

Für die Unterstützung des Buchprojekts und ihre Ermutigung danke ich herzlich den Herausgebern der Reihe “Leitfäden und Monographien der Informatik”, Hans-Jürgen Appelrath und Volker Claus. Auch sie haben das Manuskript gelesen und sehr nützliche Kommentare und Verbesserungsvorschläge eingebracht. Dem Teubner-Verlag danke ich für die reibungslose Zusammenarbeit.

Trotz aller Unterstützung gehen verbleibende Fehler und Schwächen der Darstellung natürlich zu meinen Lasten. Ich bitte Sie, die Leser des Buches, mich darauf aufmerksam zu machen.

Hagen, im Februar 1992

Ralf Hartmut Güting

## Vorwort zur zweiten Auflage

Seit dem Erscheinen der ersten Auflage hat sich das Buch als Basis- oder Sekundärliteratur zur Vorlesung *Datenstrukturen und Algorithmen* im Grundstudium der Informatik an vielen Hochschulen bewährt. In Form des gleichnamigen Kurses ist es auch an der Fernuniversität Hagen alljährlich erfolgreich im Einsatz. Zahlreiche konstruktive Kommentare engagierter Studierender sind seitdem in Details des Kurstextes eingeflossen.

Das “gereifte” Kurs-Manuskript bildet die Grundlage für die vorliegende zweite, überarbeitete Auflage. Die Algorithmen und die Datenstrukturen der algorithmischen Ebene sind dieselben geblieben. Natürlich gibt es auch weiterhin immer wieder neue Ergebnisse zu Algorithmen und Datenstrukturen; bei beschränktem Umfang halten wir es aber nicht für gerechtfertigt, Material der hier zu legenden Grundlagen zu verdrängen. Wir haben jedoch die Präsentation dieser Konzepte und, als augenfälligste Veränderung, die Implementierungsbeispiele der Entwicklung der letzten Jahre angepasst. Dazu wurden im wesentlichen folgende Erneuerungen vorgenommen:

- Modula-2 als Programmiersprache für Implementierungsbeispiele wurde von Java abgelöst. 1992 war Modula-2 die Sprache der Wahl in der Informatikausbildung und wurde, wie das sehr ähnliche Pascal, auch in der industriellen Praxis häufig verwendet. Im Zuge der Entwicklung objektorientierter Programmiersprachen ist heute Java an diese Stelle getreten.
- Eine wichtige Rolle in der alten wie der neuen Version des Buches spielt die Betonung des Unterschiedes zwischen *Programmen*, die in einer konkreten Programmiersprache formuliert sind, und *Algorithmen*, die von konkreten Programmiersprachen abstrahieren und neben programmiersprachlichen Konstrukten auch mathematische Notationen und natürlichsprachliche Formulierungen enthalten. In der neuen Auflage wird der Unterschied zwischen Algorithmen und Programmen noch deutlicher herausgestellt.
- Die Notation von Algorithmen wurde an heutige Standards angepaßt.

Gemäß seinem Ursprung als Kurs der FernUniversität ist dieses Buch hervorragend zum Selbststudium geeignet. Die zweite Auflage unterstützt diesen Anspruch dadurch, daß zum Selbsttest gedachte Aufgaben in den Text eingestreut sind, deren Lösungen im Anhang angegeben sind. Wir empfehlen dem Leser, die Selbsttestaufgaben möglichst zu bearbeiten, also nicht einfach die Lösung nachzusehen. Für ein eingehendes Verständnis des Stoffes ist der eigene kreative Umgang mit den gestellten Problemen von entscheidender Bedeutung.

Am Ende fast jeden Kapitels finden sich weitere Aufgaben, die z.B. von Dozenten in der Lehre genutzt werden können. Natürlich kann sich auch der eifrige Leser daran versuchen! Zu diesen Aufgaben sind keine Lösungen angegeben.

Wir wünschen Ihnen viel Freude und Erfolg bei der Arbeit an dem nicht immer einfachen, aber hochinteressanten und vielfältigen Thema *Datenstrukturen und Algorithmen*.

Hagen, im Februar 2003

Ralf Hartmut Güting

Stefan Dieker

## Vorwort zur dritten Auflage

Wir haben uns über die positiven Reaktionen vieler Leser auf die zweite Auflage des Buches sehr gefreut. Vor allem die Umstellung der Programmbeispiele auf Java hat großen Anklang gefunden. Dementsprechend war die zweite Auflage schnell verkauft.

In der nun vorliegenden dritten Auflage wurden eine Reihe von Druckfehlern und wenige kleine technische Fehler korrigiert. Wir danken allen Lesern, die uns auf solche Fehler hingewiesen haben.

Hagen, im November 2004

Ralf Hartmut Güting

Stefan Dieker

## Vorwort zur vierten Auflage

Seit dem Erscheinen der dritten Auflage wurde das Buch immer wieder unverändert nachgedruckt. Gleichzeitig wurde der Buchtext als Kurs an der Fernuniversität Hagen regelmäßig in der Lehre eingesetzt. Er hat dabei zahlreiche Ergänzungen, Verbesserungen und Korrekturen erfahren. Ich freue mich, dass der verbesserte Text nun Eingang in die neue Auflage findet.

Selbstverständlich ist in der Zwischenzeit die Umstellung auf die aktuelle Rechtschreibung erfolgt. Als besonderes inhaltliches Highlight möchte ich einen neuen Abschnitt zur Kürzesten-Wege-Suche auf sehr großen Straßennetzen erwähnen. In diesem Bereich hat es in den letzten fünfzehn Jahren dramatische Fortschritte gegeben. Wir

besprechen nun im Text die Technik der *Kontraktionshierarchien*. Dabei wird der Graph, der das Straßennetz darstellt, vorab um Kanten erweitert. Auf so erweiterten Graphen lassen sich kürzeste Wege von *A* nach *B* z. B. auf europaweiten Netzen in Sekundenbruchteilen ermitteln, um viele Größenordnungen schneller, als dies mit den klassischen Algorithmen von Dijkstra bzw. A\* möglich war. Dieses äußerst spannende Thema wird meines Wissens noch in keinem anderen Buch zu Datenstrukturen und Algorithmen behandelt.

Das Buch besitzt damit weiterhin einige originelle Schwerpunkte, nämlich vertiefte Behandlung geometrischer Algorithmen, algebraische Spezifikation von Datentypen und nun auch das neue Thema der Kürzesten-Wege-Suche auf Basis von Kontraktionshierarchien.

Ich bedanke mich herzlich bei vielen Studierenden, die durch Anmerkungen zum Kurs- text zu Verbesserungen beigetragen haben. Ich danke auch meinen Mitarbeitern und Mitarbeiterinnen Dirk Ansorge, Markus Spiekermann, Thomas Behr, Simone Jandt, Christian Düntgen, Fabio Valdés und Holger Hennings für ihre Hilfe bei der Kursbetreuung und dem Erstellen von Übungsaufgaben; gelegentlich sind Aspekte davon in den Kurstext eingeflossen. Schließlich gilt mein Dank Sara Betkas für ihre Unterstützung beim Umstellen des Textes auf die neue Rechtschreibung.

Hagen, im Januar 2018

Ralf Hartmut Güting

# Inhalt

<b>1 Einführung</b>	<b>1</b>
1.1 Algorithmen und ihre Analyse	2
1.2 Datenstrukturen, Algebren, Abstrakte Datentypen	22
1.3 Grundbegriffe	33
1.4 Weitere Aufgaben	36
1.5 Literaturhinweise	37
<b>2 Programmiersprachliche Konzepte für Datenstrukturen</b>	<b>39</b>
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
<b>3 Grundlegende Datentypen</b>	<b>63</b>
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	78
(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
<b>4 Datentypen zur Darstellung von Mengen</b>	<b>109</b>
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, DELETEMIN	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
<b>5 Sortieralgorithmen</b>	<b>169</b>
5.1 Einfache Sortierverfahren: Direktes Auswählen und Einfügen	170
5.2 Divide-and-Conquer-Methoden: Mergesort und Quicksort	173

	Durchschnittsanalyse für Quicksort	181
5.3	Verfeinertes Auswählen und Einfügen: Heapsort und Baumsortieren	184
	Standard-Heapsort	184
	Analyse von Heapsort	186
	Bottom-Up-Heapsort	188
5.4	Untere Schranke für allgemeine Sortierverfahren	190
5.5	Sortieren durch Fachverteilen: Bucket sort und Radixsort	194
5.6	Weitere Aufgaben	197
5.7	Literaturhinweise	198
<b>6</b>	<b>Graphen</b>	<b>201</b>
6.1	Gerichtete Graphen	202
6.2	(Speicher-) Darstellungen von Graphen	204
	(a) Adjazenzmatrix	204
	(b) Adjazenzlisten	206
6.3	Graphdurchlauf	207
6.4	Literaturhinweise	211
<b>7</b>	<b>Graph-Algorithmen</b>	<b>213</b>
7.1	Bestimmung kürzester Wege von einem Knoten zu allen anderen	213
	Implementierungen des Algorithmus von Dijkstra	218
	(a) mit einer Adjazenzmatrix	218
	(b) mit Adjazenzlisten und als Heap dargestellter Priority Queue	219
7.2	Bestimmung kürzester Wege zwischen allen Knoten im Graphen	220
	Implementierung des Algorithmus von Floyd	222
	(a) mit der Kostenmatrix-Darstellung	222
	(b) mit Adjazenzlisten	223
7.3	Berechnung kürzester Wege mittels Kontraktionshierarchien	225
7.3.1	Berechnung einer Kontraktionshierarchie für einen Graphen	226
7.3.2	Suche eines kürzesten Weges in einer Kontraktionshierarchie	228
	Berechnen und Verschneiden von Nachfolger- und	
	Vorgängermengen	230
	Hub Labeling	232
	Bidirektionale Variante des Algorithmus von Dijkstra	232
	Pfadexpansion	235
7.4	Transitive Hülle	235
7.5	Starke Komponenten	235
7.6	Ungerichtete Graphen	239
7.7	Minimaler Spannbaum (Algorithmus von Kruskal)	240
7.8	Weitere Aufgaben	244
7.9	Literaturhinweise	246

<b>8 Geometrische Algorithmen</b>	<b>249</b>
8.1 Plane-Sweep-Algorithmen für orthogonale Objekte in der Ebene	254
8.1.1 Das Segmentschnitt-Problem	254
8.1.2 Das Rechteckschnitt-Problem	259
Das Punkteinschluss-Problem und seine Plane-Sweep-Reduktion	260
Der Segment-Baum	262
Komplexität der Lösungen	264
8.1.3 Das Maßproblem	266
Plane-Sweep-Reduktion	266
Ein modifizierter Segment-Baum	268
Komplexität der Lösung des Maßproblems	269
8.2 Divide-and-Conquer-Algorithmen für orthogonale Objekte	270
8.2.1 Das Segmentschnitt-Problem	271
8.2.2 Das Maßproblem	277
8.2.3 Das Konturproblem	284
8.3 Suchen auf Mengen orthogonaler Objekte	290
Der Range-Baum	291
Der Intervall-Baum	292
Baumhierarchien	296
8.4 Plane-Sweep-Algorithmen für beliebig orientierte Objekte	299
8.5 Weitere Aufgaben	302
8.6 Literaturhinweise	305
<b>9 Externes Suchen und Sortieren</b>	<b>309</b>
9.1 Externes Suchen: B-Bäume	310
Einfügen und Löschen	314
Overflow	315
Underflow	316
9.2 Externes Sortieren	320
Anfangsläufe fester Länge - direktes Mischen	323
Anfangsläufe variabler Länge - natürliches Mischen	324
Vielweg-Mischen	326
9.3 Weitere Aufgaben	327
9.4 Literaturhinweise	329
<b>Mathematische Grundlagen</b>	<b>331</b>
<b>Lösungen zu den Selbsttestaufgaben</b>	<b>339</b>
<b>Literatur</b>	<b>377</b>
<b>Index</b>	<b>387</b>



# 1 Einführung

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

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

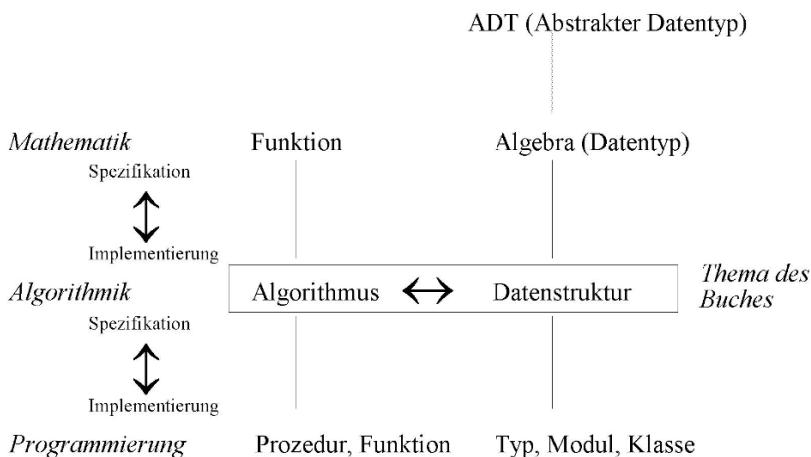


Abbildung 1.1: Abstraktionsebenen von Algorithmen und Datenstrukturen

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

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

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

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

Noch ein kleiner Hinweis vorweg: In diesem einleitenden Kapitel wollen wir einen Überblick geben und dabei die wesentlichen schon erwähnten Begriffe klären und die allgemeine Vorgehensweise bei der Analyse von Algorithmen durchsprechen. Verzweifeln Sie nicht, wenn Ihnen in diesem Kapitel noch nicht alles restlos klar wird, insbesondere für die Analyse von Algorithmen. Der ganze Rest des Buches wird dieses Thema vertiefen und die Analyse einüben; es genügt, wenn Sie am Ende des Buches 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  $BOOL = \{\text{true}, \text{false}\}$ . Wir definieren:

*contains*:  $F(\mathbb{Z}) \times \mathbb{Z} \rightarrow BOOL$

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

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

```
algorithm contains ( $S, c$ )
{Eingaben sind  $S$ , ein Integer-Array der Länge  $n$ , und  $c$ , ein Integer-Wert. Ausgabe
ist true, falls  $c \in S$ , 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 Buches wird diese Voraussetzung dadurch erfüllt, dass Autoren und Leser darin übereinstimmen, dass der Text von vorne nach hinten gelesen wird. Am Ende des Buches 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 Buch 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 mehreren möglichen Algorithmen denjenigen wählen, der schnell implementiert wer-

den 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 <b>while ...</b>
Vergleich $x \leq y$	<b>for ...</b>
Arithmetische Operation $x + y$	<b>repeat ...</b>
Arrayzugriff $s[i]$	Prozedurauftruf (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 → 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 durchgeföhrter 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.

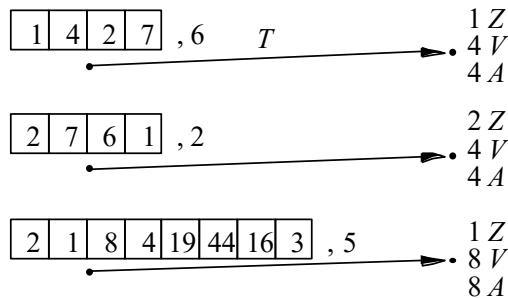


Abbildung 1.2: Anzahl von Elementaroperationen

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:

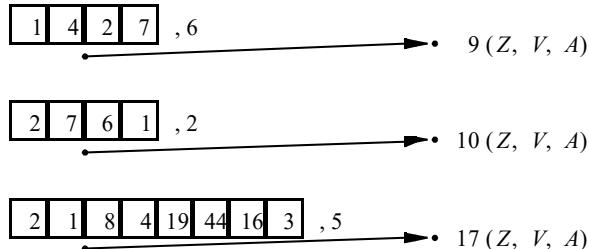


Abbildung 1.3: Erster Abstraktionsschritt

□

*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.

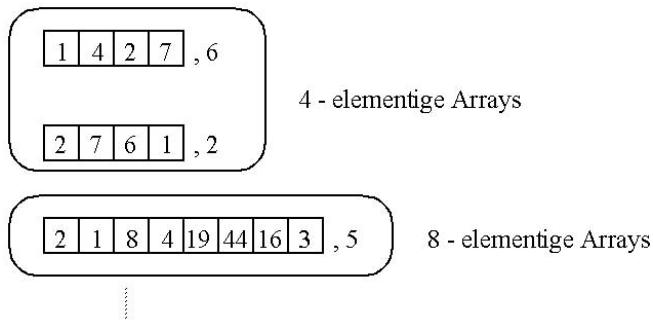


Abbildung 1.4: Zweiter Abstraktionsschritt

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*)  $T_{\text{best}}$
  - der schlimmste Fall (*worst case*)  $T_{\text{worst}}$
- (b) Betrachtung des
  - Durchschnittsverhaltens (*average case*)  $T_{\text{avg}}$

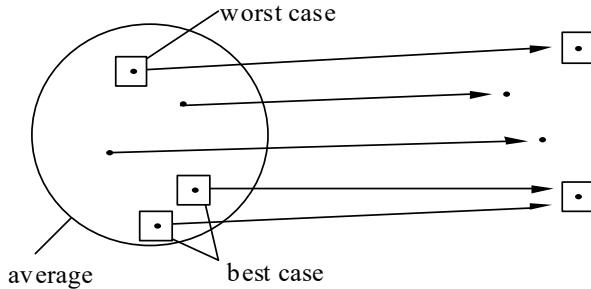


Abbildung 1.5: Dritter Abstraktionsschritt

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

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

$T_{\text{best}}$  und  $T_{\text{worst}}$  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  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ ,  $g: \mathbb{N} \rightarrow \mathbb{R}^+$ .

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

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:

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

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 durchgeföhrter 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 Menge von Objekten darstellt. Algo-

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

```
 $alg_1(U);$ 
 $alg_2(U);$ 
 $alg_2(U);$ 
```

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  $\text{alg}_1(U)$ 
else if  $a > c$  then  $x := 0$  else  $\text{alg}_2(U)$  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  $\text{alg}_2$ .  $\square$

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 das ganze Buch begleiten wird. Hier sollten nur einige Grundtechniken eingeführt werden.

Mit der O-Notation werden Laufzeitfunktionen “eingeordnet” 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 Buch bezeichnet  $\log$  (ohne Angabe der Basis) den Logarithmus zur Basis 2. Innerhalb von O-Notation spielt das aber keine Rolle, da Logarithmen zu verschiedenen Basen sich nur durch einen konstanten Faktor unterscheiden, aufgrund der Beziehung

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

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

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

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

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 contains2(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.

$$\left. \begin{array}{lcl} T_{best}(n) & = & O(1) \\ T_{worst}(n) & = & O(n) \end{array} \right.$$

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

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

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*<sub>3</sub> (*low*, *high*, *c*)

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

```

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

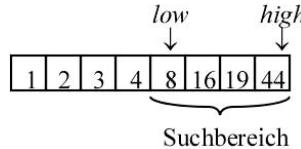


Abbildung 1.6: Algorithmus *contains*<sub>3</sub>

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:

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

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

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

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

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

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:
  - $p = \Theta(p') \Leftrightarrow d = d'$
  - $p = o(p') \Leftrightarrow d < d'$
  - $p = \omega(p') \Leftrightarrow d > d'$

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

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

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

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

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

**Beispiel 1.14:** Die Laufzeit der einfachen Suchverfahren in diesem Abschnitt (*contains* und *contains<sub>2</sub>*) ist  $O(n)$  im worst case, aber auch  $\Omega(n)$  und daher auch  $\Theta(n)$ . □

Da die  $\Omega$ -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  $\Omega(n^2)$ :

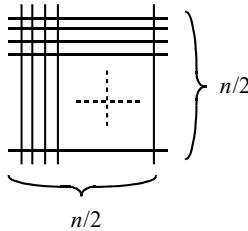


Abbildung 1.7: Schnittpunkte in der Ebene

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:

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

Programm<sub>1</sub> ist schneller ab  $n = 200$ .

Ein Algorithmus mit  $O(n^2)$  wird besser als einer mit  $O(n^3)$  ab irgendeinem  $n$  (“asymptotisch”). Für “kleine” Eingaben kann ein asymptotisch schlechterer Algorithmus der bessere sein! Im Extremfall, wenn die von der O-Notation “verschwiegenen” Konstanten zu

groß werden, gibt es in allen praktischen Fällen nur “kleine” Eingaben, selbst wenn  $n = 1\,000\,000$  ist.

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

## 1.2 Datenstrukturen, Algebren, Abstrakte Datentypen

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

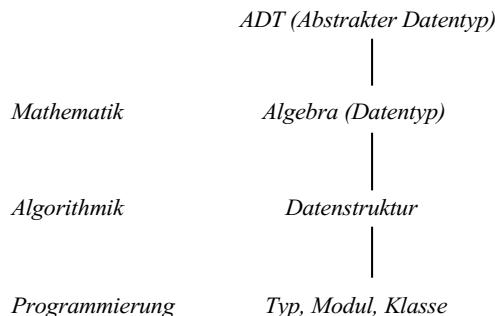


Abbildung 1.8: Abstraktionsebenen von Datenstrukturen

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

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

Wir betrachten zunächst die Ebene der Mathematik bzw. Spezifikation. Die abstrakteste Sicht einer Datenstruktur ist offenbar die, dass es eine Klasse von Objekten gibt (die möglichen “Ausprägungen” oder “Werte” der Datenstruktur), auf die gewisse Operationen anwendbar sind. Bei genauerem Hinsehen wird man etwas verallgemeinern: Offensichtlich können mehrere Klassen von Objekten eine Rolle spielen. In unserem

Beispiel kommen etwa Mengen ganzer Zahlen, aber auch ganze Zahlen selbst als Objektklassen vor. Die Operationen erzeugen dann aus gegebenen Objekten in diesen Klassen neue Objekte, die ebenfalls zu einer der Objektklassen gehören.

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

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

<b>sorts</b>	<i>intset, int, bool</i>	
<b>ops</b>	<i>empty:</i>	$\rightarrow \text{intset}$
	<i>insert:</i> <i>intset</i> $\times$ <i>int</i>	$\rightarrow \text{intset}$
	<i>delete:</i> <i>intset</i> $\times$ <i>int</i>	$\rightarrow \text{intset}$
	<i>contains:</i> <i>intset</i> $\times$ <i>int</i>	$\rightarrow \text{bool}$
	<i>isempty:</i> <i>intset</i>	$\rightarrow \text{bool}$

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

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

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

```

algebra   intset
sorts    intset, int, bool
ops
  empty:           → intset
  insert:          intset × int      → intset
  delete:          intset × int      → intset
  contains:        intset × int      → bool
  isempty:         intset            → bool
sets     intset = F(ℕ) = {M ⊂ ℕ | M endlich}
functions
  empty           = ∅
  insert (M, i)   = M ∪ {i}
  delete (M, i)   = M \ {i}
  contains (M, i) = {true   falls i ∈ M
                     false  sonst
  isempty (M)     = (M = ∅)
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

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

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

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

<b>algebra</b>	<i>counter</i>
<b>sorts</b>	<i>counter</i>
<b>ops</b>	<i>reset</i> : $\rightarrow \text{counter}$
	<i>increment</i> : $\text{counter} \rightarrow \text{counter}$
	<i>decrement</i> : $\text{counter} \rightarrow \text{counter}$

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.  $\square$

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 for-

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

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

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

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

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

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

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

- (a) Geben Sie die verwendeten Sorten und Operationen mit deren Signaturen an.
- (b) Ordnen Sie den Sorten geeignete Trägermengen zu. Verwenden Sie Mengen und Tupel in den Definitionen.
- (c) Legen Sie die Semantik der Operationen durch Angabe von Funktionen fest.

□

Wir betrachten nun die Darstellung unserer Beispiel-Datenstruktur auf der *algorithmmischen 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;
```

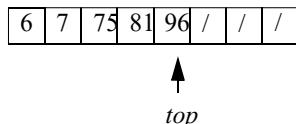


Abbildung 1.9: Beispiel-Ausprägung des Arrays *s*

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.<sup>1</sup> 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;<sup>2</sup>

**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 löschen 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!

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 Programmierung nicht immer streng top-down verläuft, sondern dass gelegentlich Entwurfsentscheidungen höherer Ebenen zurückgenommen werden müssen.

**algorithm** *contains* (*x*)

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

**algorithm** *isempty*

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

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

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

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 Implementierungs-ebene die *Klasse*. Vereinfacht dargestellt<sup>3</sup> besteht eine Klassendefinition aus der Angabe aller Komponenten und der Implementierung aller Methoden der Klasse. Zudem wird zu jeder Komponente und Methode definiert, aus welchem Sichtbarkeitsbereich man auf sie zugreifen bzw. sie aufrufen darf. Die Deklaration einer Komponente oder Methode als *private* hat zur Folge, dass sie nur innerhalb von Methoden der eigenen Klasse verwendet werden können. Im Gegensatz dazu erlaubt die *public*-Deklaration den Zugriff von beliebiger Stelle.

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

---

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

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

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

Sichtbarkeitsbereich	Programmiersprache			Bedeutung
	Modula-2	C/C++	Java	
für Benutzer sichtbar	<b>Definitionsmodul</b>	<b>Header-Datei</b>	<b>javadoc-Schnittstellenbeschreibung</b>	entspricht der Signatur einer Algebra
für Benutzer verborgen	<b>Implementationsmodul</b>	<b>Implementierung in Datei(en)</b>	<b>Klasse/Paket</b>	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.

<p><b>Class IntSet</b></p> <pre>java.lang.Object  +--IntSet</pre> <hr/> <p>public class IntSet extends java.lang.Object</p> <p>Diese Klasse implementiert eine Integer-Menge.</p> <p>Einschränkung: Bei der aktuellen Implementierung kann die Menge maximal 100 Elemente enthalten.</p>	<p><b>Method Detail</b></p> <hr/> <p><b>Insert</b></p> <pre>public void Insert(int elem)</pre> <p>Fügt ein Element in die Menge ein. Ist es bereits in der Menge enthalten, geschieht nichts.</p> <p><b>Parameters:</b> elem - das einzufügende Element</p> <hr/> <p><b>Delete</b></p> <pre>public void Delete(int elem)</pre> <p>Löscht ein Element aus der Menge. Falls es nicht in der Menge enthalten ist, geschieht nichts.</p> <p><b>Parameters:</b> elem - das zu löschende Element</p> <hr/> <p><b>Contains</b></p> <pre>public boolean Contains(int elem)</pre> <p>Prüft das Vorhandensein des Elements elem.</p> <p><b>IsEmpty</b></p> <pre>public boolean IsEmpty()</pre> <p>Prüft, ob die Menge leer ist.</p> <hr/> <p><b>Methods inherited from class java.lang.Object</b></p> <pre>clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait</pre> <hr/> <p><b>Constructor Detail</b></p> <p><b>IntSet</b></p> <pre>public IntSet()</pre> <p>Initialisiert die leere Menge. Ersetzt die <i>empty</i>-Operation der Algebra.</p>
--	--

Abbildung 1.10: Javadoc-Schnittstellenbeschreibung für unsere IntSet-Algebra

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

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

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

```

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

(shiftleft ähnlich)

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

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

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

```

```

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

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

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

```

### 1.3 Grundbegriffe

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

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

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

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

Algorithmische Beschreibungen dienen der Kommunikation zwischen Menschen; insoweit 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  $(S, \Sigma)$ , wobei  $S$  eine Menge ist, deren Elemente *Sorten* heißen, und  $\Sigma$  eine Menge von *Operationen* (oder *Operationssymbolen*).  $\Sigma$  ist die Vereinigung der Mengen  $\sum_{w,s}$  in einer Familie von Mengen  $\{\sum_{w,s} \mid w \in S^*, s \in S\}$ , die jeweils mit der Funktionalität der in ihnen enthaltenen Operationssymbole indiziert sind. Es bezeichnet nämlich  $S^*$  die Menge aller Folgen beliebiger Länge von Elementen aus  $S$ . Die leere Folge heißt  $\varepsilon$  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

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

$\square$

Eine (*mehrsortige = heterogene*) *Algebra* ist ein System von Mengen und Operationen auf diesen Mengen. Sie ist gegeben durch eine Signatur  $(S, \Sigma)$ , 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  $(S, \Sigma)$  sowie Gleichungen (“Axiomen”), die das Verhalten der Operationen beschreiben.

**Selbsttestaufgabe 1.8:** Für die ganzen Zahlen seien die Operationen

$0$  (Null),

$\text{succ}$  (Nachfolger),  $\text{pred}$  (Vorgänger)

$+, -, *$

vorgesehen. Geben Sie hierzu einen abstrakten Datentyp an. □

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

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

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

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

## 1.4 Weitere Aufgaben

**Aufgabe 1.9:** Gegeben sei eine Folge ganzer Zahlen  $s_1, \dots, s_n$ , 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 1.10:** Entwerfen Sie einen kleinen Instruktionssatz für die Random-Access-Maschine. Ein Befehl kann als Paar  $(b, i)$  dargestellt werden, wobei  $b$  der Befehlsname ist und  $i$  eine natürliche Zahl, die als Adresse einer Speicherzelle aufgefasst wird (ggf. kann der Befehlsname indirekte Adressierung mitausdrücken). Der Befehlssatz sollte so gewählt werden, dass die folgende Aufgabe 1.11 damit lösbar ist. Definieren Sie für jeden Befehl seine Wirkung auf Programmzähler und Speicherzellen.

**Aufgabe 1.11:** Implementieren Sie den Algorithmus  $\text{contains}_2$  auf einer RAM mit dem in Aufgabe 1.10 entwickelten Befehlssatz. Wieviele RAM-Instruktionen führt dieses RAM-Programm im besten Fall, im schlimmsten Fall und im Durchschnitt aus?

**Aufgabe 1.12:** Beweisen Sie die Behauptungen aus Abschnitt 1.1

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

**Aufgabe 1.13:** Gegeben sei eine Zahlenfolge  $S = s_1, \dots, s_n$ , 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 1.14:** Gegeben seien Programme  $P_1, P_2, P_3$  und  $P_4$ , die auf einem Rechner  $R$  Laufzeiten

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

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

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

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

**Aufgabe 1.15:** Sei  $n$  die Anzahl verschiedener Seminare, die in einem Semester stattfinden. Die Seminare seien durchnummieriert. 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 Buch besonders beeinflusst hat. Wichtige "Klassiker" sind [Knuth 1998], [Aho et al. 1974] und Wirth [2000, 1996] (die ersten Auflagen von Knuth und Wirth sind 1973 bzw. 1975 erschienen). Ein hervorragendes deutsches Buch ist [Ottmann und Widmayer 2012]. Auch [Cormen et al. 2010] ist sehr zu empfehlen.

Eine gute Einführung in Algorithmen und Datenstrukturen auf Basis der Sprache Java bietet [Saake und Sattler 2013]. Weitere gute Darstellungen finden sich in [Mehlhorn 1984a-c], [Horowitz, Sahni und Anderson-Freed 2007], [Sedgewick 2002a, 2002b] und [Wood 1993]. Manber [1989] betont den kreativen Prozess bei der Entwicklung von Algorithmen, beschreibt also nicht nur das Endergebnis. Gonnet und Baeza-Yates [1991]

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

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

Die bei uns für die Ausformulierung konkreter Programme verwendete Sprache Java ist z. B. in [Evans und Flanagan 2014] beschrieben.

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

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

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



## 2 Programmiersprachliche Konzepte für Datenstrukturen

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

Die Rolle des *Arrays* ist nicht ganz klar. Er wird in der Literatur bisweilen als eigenständiger, aus Anwendungssicht interessanter Datentyp angesehen (z. B. in [Horowitz, Sahni und Anderson-Freed 2007]). Wir verstehen den Array als programmiersprachliches Werkzeug mit *fester* Implementierung und besprechen ihn deshalb in diesem Kapitel. Ein entsprechender Datentyp *Abbildung* wird in Kapitel 3 eingeführt; für diesen ist der Array eine von mehreren Implementierungsmöglichkeiten.

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

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

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

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

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

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

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

## 2.1 Datentypen in Java

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

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

Typen		Instanzen
Basisdatentypen		Werte
Referenztypen	Array-Typen	Arrays
	Klassen	Objekte

Tabelle 2.1: Datentypen und Instanzen in Java

### 2.1.1 Basisdatentypen

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

<i>integer</i> × <i>integer</i>	→ <i>integer</i>	+, -, *, <b>div, mod</b>
<i>real</i> × <i>real</i>	→ <i>real</i>	+, -, *, /
<i>boolean</i> × <i>boolean</i>	→ <i>boolean</i>	<b>and, or</b>
<i>boolean</i>	→ <i>boolean</i>	<b>not</b>
<i>integer</i> × <i>integer</i>	→ <i>boolean</i>	=, ≠, <, ≤, >, ≥
<i>real</i> × <i>real</i>	→ <i>boolean</i>	=, ≠, <, ≤, >, ≥
<i>char</i> × <i>char</i>	→ <i>boolean</i>	=, ≠, <, ≤, >, ≥

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

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

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

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

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

Tabelle 2.2: Standard-Typen vs. Java-Basisdatentypen

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

### 2.1.2 Arrays

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

In Java definiert die Anweisung

`T[] V = new T[n]`

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

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

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

`V[i]`

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

Mit der Deklaration

`int i, j;`

bezeichnet

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

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

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

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

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

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

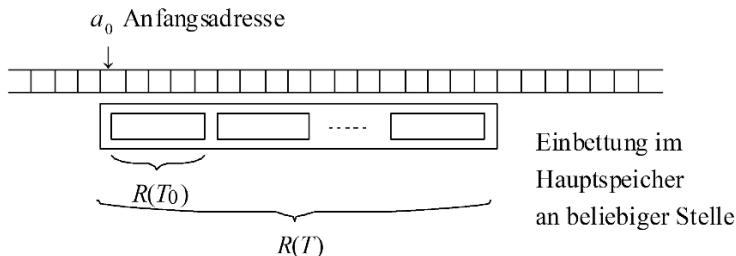


Abbildung 2.1: Speicherrepräsentation von Arrays

Daraus ergibt sich, dass die Adresse der  $i$ -ten Komponente

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

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

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

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

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

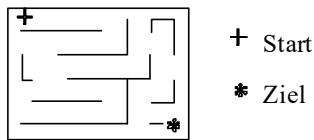


Abbildung 2.2: Labyrinth

```
final int maxh = ..., maxv = ...;  
boolean Labyrinth [][] = new boolean [maxh][maxv];  
  
boolean wand (int i, int j)  
{  
    return Labyrinth [i][j];  
}
```

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

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

```

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

```

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

**type**  $T = \text{array } [\min..\max] \text{ of } T_0$

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

$V[i]$

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

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

### 2.1.3 Klassen

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

1. *Aggregation*: Eine Klasse definiert einen neuen Datentyp, der mehrere Einzelobjekte unterschiedlichen Typs zu einem “großen” Objekt zusammenfasst. Die Einzelobjekte werden auch *Attribute* genannt.
2. *Kapselung*: Zusätzlich zu den Attributen gibt eine Klassendefinition an, welche *Methoden* auf Instanzen der Klasse angewandt werden können. Die *Implementierung* der Methoden enthält dann typischerweise Zugriffe auf die Klassenattribute. Der Benutzer einer Klasse muss nur die Schnittstellen ihrer Methoden kennen; die Implementierung der Methoden und — bei konsequenter Kapselung — die Attribute der Klasse sind für den Benutzer unwichtig. Kapselung ist ein wichtiges Werkzeug, um Programme zur Lösung komplexer Probleme überschaubar zu halten. Außerdem ermöglicht die Kapselung, lokale Änderungen von Klas-

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

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

Eine Klassendefinition hat folgende Grundstruktur:

```
Modifikatorenliste class Klassename
{
    Attributdeklarationen
    Konstruktordeklarationen
    Methodendeklarationen
}
```

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

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

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

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

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

```
type T = record s1: T1;  
          ...  
          sn : Tn  
end
```

definiert, wobei *T<sub>1</sub>*, ..., *T<sub>n</sub>* beliebige Typen (die Typen der Komponenten) sind und *s<sub>1</sub>*, ..., *s<sub>n</sub>* hiermit definierte Namen, die für den Zugriff auf die Komponenten benutzt werden.

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

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

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

```
var p : Person;
```

liefert

- |                                |  |
|--------------------------------|--|
| <i>p.Nachname</i>              | eine Variable vom Typ <i>string</i> ,  |
| <i>p.Nachname</i> [ <i>i</i> ] | eine Variable vom Typ <i>char</i> ,    |
| <i>p.GebDatum</i>              | eine Variable vom Typ <i>Datum</i> und |
| <i>p.GebDatum.Tag</i>          | eine Variable vom Typ [1..31].         |

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

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

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

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

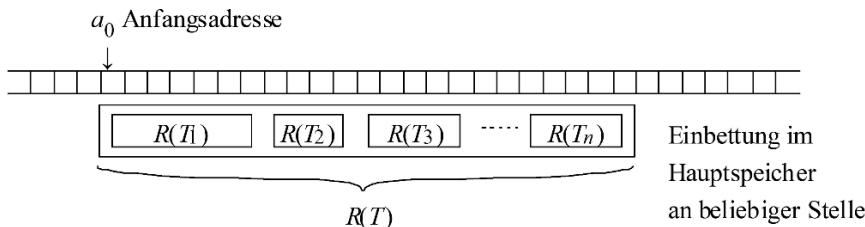


Abbildung 2.3: Speicherrepräsentation von Records

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

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

Für Record  $r$  vom Typ  $T$  mit der Anfangsadresse  $a_0$  ist daher die Adresse von  $r.s_i$   
 $a_0 + \text{offset}(s_i)$ .

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

## 2.2 Dynamische Datenstrukturen

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

### 2.2.1 Programmiersprachenunabhängig: Zeigertypen

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

**type**  $T = \uparrow T_0$

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

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

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

```
var p : T;
```

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

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

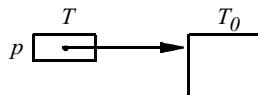


Abbildung 2.4: Zeiger *p*

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

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

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

```
type Person' = record Vorname : string;
          Vater    :  $\uparrow$  Person';
          Mutter   :  $\uparrow$  Person'
        end;
```

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

```
var p, q :  $\uparrow$  Person';
p := nil; q := nil;
```

$p$  [ ] •

$q$  [ ] •

(Bedeutung:  $p = q = \text{nil}$ )

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

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

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

Danach erhalten wir folgende Situation:

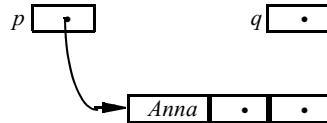


Abbildung 2.6: Zeiger *p* verweist auf *Person'* mit  $p\uparrow.\text{Vorname} = \text{"Anna"}$

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

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

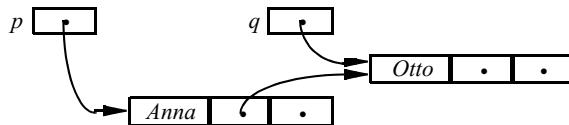


Abbildung 2.7: Zeiger *q* und  $p\uparrow.\text{Vater}$  verweisen auf "Otto"

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

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

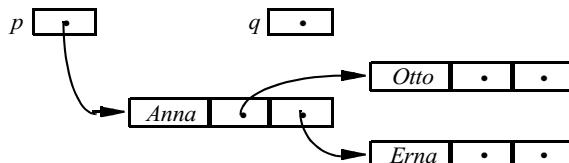


Abbildung 2.8: Zeiger *q* ist wieder leer,  $p\uparrow.\text{Mutter}$  verweist auf "Erna"

Was geschieht, wenn jetzt " $p := \text{nil}$ " ausgeführt wird?

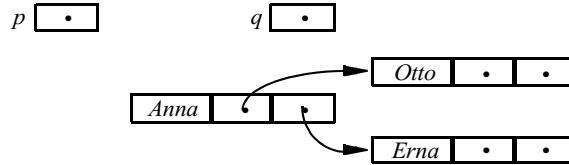


Abbildung 2.9: Zeiger  $p$  und  $q$  sind leer, “Anna”, “Erna” und “Otto” sind unerreichbar

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

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

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

$\text{dispose}(p \uparrow .\text{Vater}); \text{ dispose}(p \uparrow .\text{Mutter});$   
(anstelle von  $p := \text{nil};$ )

diese Situation im Speicher:



Abbildung 2.10: “Otto” und “Erna” sind wieder freigegeben

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

### 2.2.2 Zeiger in Java: Referenztypen

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

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

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

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

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

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

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

---

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

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

Abbildung 2.11 illustriert diese Ausgangssituation.

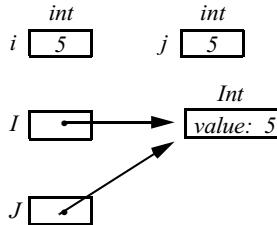


Abbildung 2.11: Variablen *i* und *j*, Zeiger *I* und *J*

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

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

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

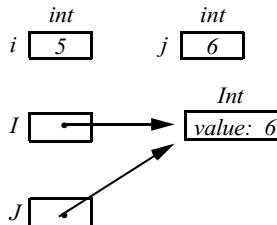
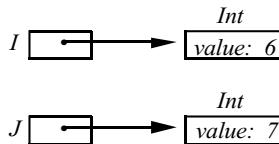


Abbildung 2.12: *i* ≠ *j*, *I.value* = *J.value*

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

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

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

Abbildung 2.13:  $I \neq J$ 

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

```

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

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

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

```

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

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

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

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

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

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

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

Die Speicherdarstellung von *f* zeigt Abbildung 2.14.

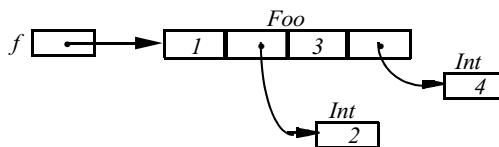


Abbildung 2.14: Klasse *Foo* mit Standard- und Klassenattributten

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

Abbildung 2.15 zeigt die Hauptspeicherdarstellung eines Arrays  $A$ , der aus 3 Elementen des Typs  $\text{Foo}$  besteht. Auch die Einbettung von Arrays kann in Java nicht vom Programmierer beeinflusst werden.

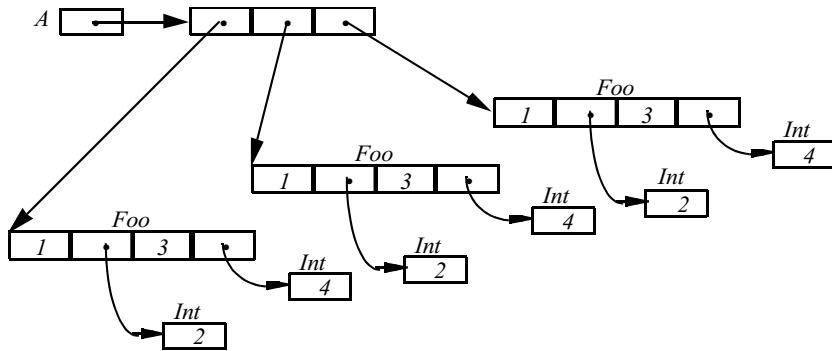


Abbildung 2.15: Array  $A$  mit drei  $\text{Foo}$ -Elementen

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

## 2.3 Weitere Konzepte zur Konstruktion von Datentypen

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

## Aufzählungstypen

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

```
type T = (c1, c2, ..., cn)
```

definiert, wobei die c<sub>i</sub> die Elemente der zu spezifizierenden Menge sind.

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

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

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

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

```
enum T {c1, c2, ..., cn};
```

Die obigen Beispiele könnte man in Java so formulieren:

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

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

```
static T[] values()
```

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

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

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

## Unterbereichstypen

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

**type  $T = [min..max]$**

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

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

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

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

auf den ganzzahligen Bereich 0..8 abgebildet werden.

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

## Sets

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

```
type T = set of T0
      ↑ Grundtyp
```

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

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

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

$$\left. \begin{array}{ll} \cap & * \\ \cup & + \\ \setminus & - \\ \in & \text{in} \end{array} \right\} \text{in PASCAL und Modula-2}$$

Damit kann man etwa formulieren:

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

Dabei bezeichnet {} die leere Menge.

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

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

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

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

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

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

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

Die Mengenoperationen können damit wie folgt implementiert werden:

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

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

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

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

Nähere Einzelheiten findet man wieder in entsprechenden Java-Büchern.

## 2.4 Literaturhinweise

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

Zur Programmiersprache Java existiert eine Vielzahl von Büchern. Ein bewährtes, umfassendes Nachschlagewerk zur Programmierung in Java ist das Buch von Evans und Flanagan [2014]. Weitere Java-Bücher sind [Schiedermeier 2010] und [Krüger und Stark 2009].

Eine weitere wichtige Informationsquelle sind die Java-Seiten im WWW-Angebot der Firma Sun, zu erreichen unter <http://www.oracle.com/technetwork/java/>.



### 3 Grundlegende Datentypen

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

#### 3.1 Sequenzen (Folgen, Listen)

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

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

$$a_1 \dots a_n$$

eine Sequenz mit  $n$  Elementen. Wir schreiben auch

$$\langle a_1, \dots, a_n \rangle,$$

wobei die spitzen Klammern explizit ausdrücken, dass es sich um eine Sequenz handelt. Insbesondere muss man eine einelementige Sequenz  $\langle a \rangle$  so von dem Objekt  $a$  unterscheiden. Weiterhin stellt das Symbol

$\diamond$

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

$\circ$

einen Konkatenationsoperator für Sequenzen, definiert als

$$\langle a_1, \dots, a_n \rangle \circ \langle b_1, \dots, b_m \rangle = \langle a_1, \dots, a_n, b_1, \dots, b_m \rangle.$$

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

### 3.1.1 Modelle

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

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

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

```

algebra list1
sorts list, elem {bool ist im Folgenden implizit immer dabei}
ops   empty      : → list
        first       : list      → elem
        rest        : list      → list
        append      : list × elem → list
        concat      : list × list → list
        isempty     : list      → bool
sets   list       = {<a1, ..., an> | n ≥ 0, ai ∈ elem}
functions
        empty      = ◊
        first (a1 ... an) = {a1           falls n > 0
                                undefiniert sonst
        rest (a1 ... an)  = {a2 ... an    falls n > 0
                                undefiniert sonst
        append (a1 ... an, x) = x a1 ... an
        concat (a1 ... an, b1 ... bm) = a1 ... an ∘ b1 ... bm
        isempty (a1 ... an) = (n = 0)
end list1.

```

### (b) Listen mit expliziten Positionen

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

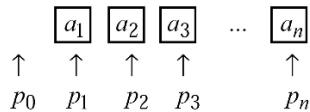


Abbildung 3.1: Zeiger als Positionswerte

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

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

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

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

```
algebra list2
sorts list, elem, pos
ops
    empty      :           → list
    front, last : list     → pos
    next, previous : list × pos → pos ∪ {null}
    bol, eol      : list × pos → bool
    insert       : list × pos × elem → list
    delete       : list × pos     → list
    concat       : list × list   → list
    isempty      : list          → bool
    find         : list × (elem → bool) → pos ∪ {null}
    retrieve     : list × pos   → elem
sets
```

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

$$\begin{aligned} list &= \{(a_1 \dots a_n, p_0 \dots p_n) \mid n \geq 0, a_i \in elem, p_i \in POS, \\ &\quad p_i \neq p_j \text{ falls } i \neq j\} \\ pos &= POS \end{aligned}$$

---

1. genauer: die Position vor dem ersten Listenelement

**functions**

$$\text{empty} = (\emptyset, \langle p_0 \rangle)$$

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

$$\text{front}(l) = p_0$$

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

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

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

$$\text{bol}(l, p) = (p = p_0)$$

$$\text{eol}(l, p) = (p = p_n)$$

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

$$\begin{aligned} \text{insert}(l, p, x) = & (\langle a_1, \dots, a_i, x, a_{i+1}, \dots, a_n \rangle, \\ & \langle p_0, \dots, p_i, p', p_{i+1}, \dots, p_n \rangle) \end{aligned}$$

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

$$\begin{aligned} \text{delete}(l, p) = & (\langle a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n \rangle, \\ & \langle p_0, \dots, p_{i-1}, p_{i+1}, \dots, p_n \rangle) \end{aligned}$$

Für  $\text{concat}$  sei  $\{p_0, \dots, p_n\} \cap \{q_1, \dots, q_m\} = \emptyset$ . Sonst ist  $\text{concat}$  undefiniert.

$$\begin{aligned} \text{concat}((a_1 \dots a_n, p_0 \dots p_n), (b_1 \dots b_m, q_0 \dots q_m)) = & (\langle a_1, \dots, a_n, b_1, \dots, b_m \rangle, \langle p_0, \dots, p_n, q_1, \dots, q_m \rangle) \end{aligned}$$

$$\text{isempty}(l) = (n = 0)$$

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

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

$$\text{retrieve}(l, p) = a_i$$

**end** *list*<sub>2</sub>.

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

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

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

### 3.1.2 Implementierungen

#### (a) Doppelt verkettete Liste

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

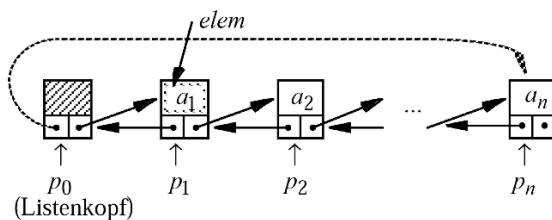


Abbildung 3.2: Doppelt verkettete Liste

Die leere Liste sieht bei dieser Darstellung so aus:



Abbildung 3.3: Leere Liste

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

Die gezeigte Darstellung lässt sich etwa mit folgenden Deklarationen realisieren:

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

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

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

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

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

Die Klasse *List* erweitert die Klasse *Pos* um Methoden, die die Operationen des Datentyps *list*<sub>2</sub> realisieren. Eine *List*-Instanz ist das Kopfelement einer Liste, deren Elemente aus *Pos*-Instanzen bestehen.

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

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

erzeugt.

Die Methode *front* ist schnell programmiert:

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

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

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

- (a) das Element soll in der Mitte oder am Anfang der Liste eingefügt werden, oder
- (b) es soll an das Ende der Liste angehängt werden.

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

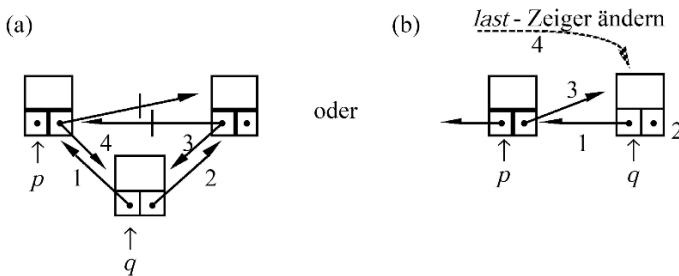


Abbildung 3.4: (a) Einfügen am Anfang oder in der Mitte, (b) Anhängen am Ende

```

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

```

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

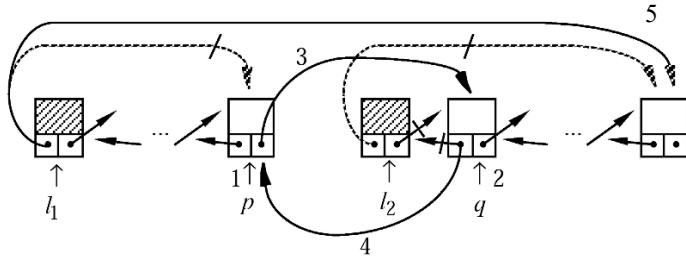


Abbildung 3.5: Verkettung zweier Listen

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

```

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

```

```

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

```

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

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

```

interface ElemTest
{
    boolean check(Elem l);
}

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

```

```

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

```

Als Beispiel betrachten wir einen Elementtyp *Int*:

```

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

```

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

```

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

```

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

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

### (b) Einfach verkettete Liste

Diese Struktur (Abbildung 3.6) unterstützt effizient einen Teil der Operationen des Datentyps *list*<sub>2</sub>; sie ist einfacher, braucht etwas weniger Speicherplatz (für jedes Listenelement nur *einen* Zeiger auf den Nachfolger) und ist daher bei Anwendungen, die mit diesen Operationen auskommen, vorzuziehen.

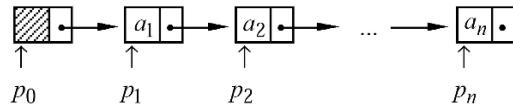


Abbildung 3.6: Einfach verkettete Liste (Prinzip)

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

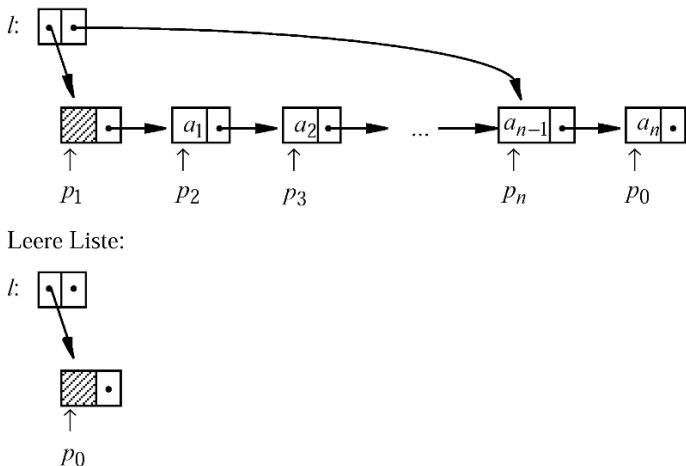


Abbildung 3.7: Einfach verkettete Liste (Implementierung)

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

```

class Pos
{
    Elem value;
    Pos succ;
}

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

```

Der *last*-Zeiger im Record *l* muss auf das vorletzte Element zeigen, da die Operation *last* die Position *p<sub>n</sub>* zurückliefern soll.

Ein Aufruf von *insert* mit *p<sub>0</sub>* wird aufgefasst als Auftrag, ein Element hinter dem Listenkopf einzufügen; *p<sub>0</sub>* selbst wird dazu nicht gebraucht. Ob eine Position *p* gleich *p<sub>0</sub>* ist, lässt sich leicht feststellen:

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

Leider sind die Positionszeiger nicht stabil unter Änderungsoperationen. Beispielsweise zeigt der Zeiger *p<sub>i</sub>* nicht mehr auf *a<sub>i</sub>*, wenn hinter der Position *p<sub>i-1</sub>* ein Element eingefügt wird.

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

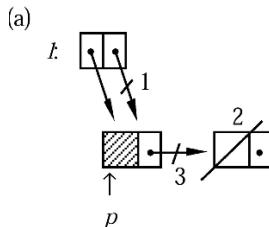


Abbildung 3.8: *p* zeigt auf das letzte Element einer einelementigen Liste.

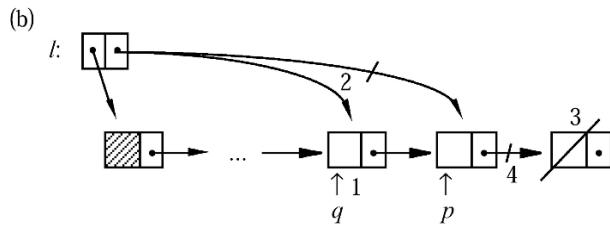


Abbildung 3.9:  $p$  zeigt auf das letzte Element einer mehrelementigen Liste.

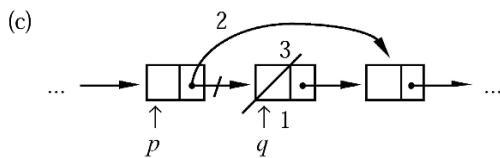


Abbildung 3.10:  $p$  zeigt nicht auf das letzte Element.

```

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

```

```

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

```

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

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

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

**Selbsttestaufgabe 3.2:** Schreiben Sie für die Operation

*swap*:  $list \times pos \rightarrow list$

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

### (c) Sequentielle Darstellung im Array

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

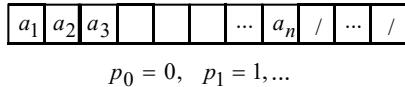


Abbildung 3.11: Listendarstellung im Array

Diese Darstellung hat einige offensichtliche Nachteile:

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

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

### (d) Einfach oder doppelt verkettete Liste im Array

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

**Beispiel 3.1:** Die Liste  $L$



Abbildung 3.12: Beispilliste

könnte im Array so dargestellt sein:

	value	succ
0		
1	44	2
2	81	-1
3		6
4		
5		
6	3	1
7		
8		
9		

$L \rightarrow$

Abbildung 3.13: Beispielliste im Array

□

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

<i>frei</i> →	0		2
	1	a	11
	2		6
	3		10
	4	y	-1
$L \rightarrow$	5		1
	6		3
	7	c	-1
$M \rightarrow$	8		9
	9	x	4
	10		-1
	11	b	7

Abbildung 3.14: Zwei Listen in einem Array

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

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

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

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

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

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

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

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

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

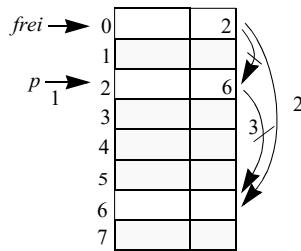


Abbildung 3.15: Freispeicherverwaltung

Ein freigegebenes Element wird an den Anfang der Freiliste gehängt.

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

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

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

(b) die komplette im Array dargestellte Struktur kann *extern gespeichert* werden.

Beim Neustart des Programms kann der Array-Inhalt wieder eingelesen werden; die Zeiger (Array-Indizes) sind danach wiederum gültig. Explizite Zeiger hingegen „überleben“ das Programmende nicht; sie haben beim nächsten Aufruf keine Bedeutung mehr.

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

## 3.2 Stacks

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

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

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

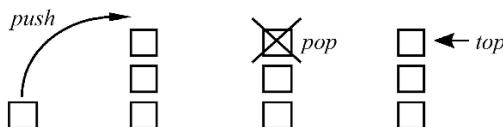


Abbildung 3.16: Stack-Operationen

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

```

adt stack
sorts   stack, elem
ops    empty      :           → stack
          push       : stack × elem → stack
          pop        : stack     → stack
          top        : stack     → elem
          isempty    : stack     → bool
axs    isempty(empty)      = true
          isempty(push(s, e)) = false
          pop(empty)         = error
          pop(push(s, e))   = s
          top(empty)         = error
          top(push(s, e))   = e
end stack.

```

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

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

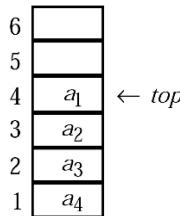


Abbildung 3.17: Stack-Implementierung im Array

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

```

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

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

```

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

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

### Beispiel 3.2: Auswertung arithmetischer Ausdrücke

Es sollen vollständig geklammerte arithmetische Ausdrücke, wie etwa

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

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

$$\begin{array}{c} ( \quad 672 \quad + \\ \cancel{(6 * \quad 112 \quad )} \quad (9 \quad - \\ \cancel{(4 * \quad 28)} \quad ( \quad 3 \quad * \\ \cancel{(12 / 4)} \end{array}$$

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

- öffnende Klammer: ignorieren
- Operand: auf Operandenstack legen

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

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

Operandenstack	Operatorstack
6 <del>/</del> 28	* <del>/</del>
<del>6</del> 1 <del>/</del> 2	<del>*</del>
672 9 <del>/</del> 4 <del>/</del>	+ - <del>/</del>
672 9 <del>/</del> 2 <del>/</del>	+ - <del>*</del>
672 <del>/</del> 6 <del>/</del>	+ <del>/</del>
<del>672</del> <del>/</del> <del>/</del>	<del>/</del>
675	

□

### Beispiel 3.3: Verwaltung geschachtelter Prozeduraufrufe

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

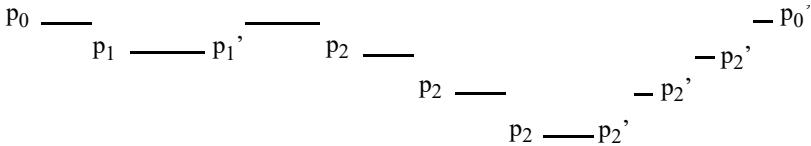
```

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

procedure p1;
begin ... end p1;

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

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



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

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

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

---

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

c. **goto** <erste Sprungmarke>

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

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

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

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

**Beispiel 3.4:** Türme von Hanoi

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

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

Der folgende rekursive Algorithmus erzeugt als Ausgabe eine entsprechende “Zugfolge”:

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

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

```

if  $n = 1$ 
then output (source, “->”, target)
else
    move ( $n - 1$ , source, other, target);
    output (source, “->”, target);
    move ( $n - 1$ , other, target, source)
end if

```

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

1. Wir führen folgende Sprungmarken ein:

```

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

```

2. Der Stack wird so definiert:

```

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

```

**var** *s* : stack (*activation*)

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

```

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

```

```

var return_label : label;
begin
    s := empty;
    1: if  $n = 1$ 
        then output (source, “->”, target)
        else {Ersetze “move ( $n - 1$ , source, other, target)”}
            push (s, ( $n$ , source, target, other, 2)); (*)  

            (n, source, target, other) := ( $n - 1$ , source, other, target); (**)
            goto 1;

```

```

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

```

□

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

### 3.3 Queues

Queues sind ebenfalls spezielle Listen, in denen Elemente nur an einem Ende (“vorne”) entnommen und nur am anderen Ende (“hinten”) angehängt werden. Sie heißen deshalb FIFO (first-in-first-out-Strukturen. Der deutsche Ausdruck ist (Warte-)Schlange. Queues entsprechen unserem Datentyp *list*<sub>1</sub> ohne die Operationen *concat* und *append*; anstelle von *append* gibt es eine Operation *rappend* (*rear - append*), die ein Element ans Ende einer Liste hängt. Die Algebra *list*<sub>1</sub> wäre entsprechend zu erweitern:

<b>ops</b>	<i>rappend</i> : <i>list</i> × <i>elem</i> → <i>list</i>
<b>functions</b>	<i>rappend</i> ( <i>a</i> <sub>1</sub> … <i>a</i> <sub><i>n</i></sub> , <i>x</i> ) = <i>a</i> <sub>1</sub> … <i>a</i> <sub><i>n</i></sub> <i>x</i>

Traditionell gibt es auch hier wieder andere Bezeichnungen:

<i>front</i>	= <i>first</i>
<i>enqueue</i>	= <i>rappend</i>
<i>dequeue</i>	= <i>rest</i>
<i>queue</i>	= <i>list</i>

Somit erhalten wir die Signatur

<b>sorts</b>	<i>queue, elem</i>
<b>ops</b>	
	<i>empty</i> : $\rightarrow \text{queue}$
	<i>front</i> : $\text{queue} \rightarrow \text{elem}$
	<i>enqueue</i> : $\text{queue} \times \text{elem} \rightarrow \text{queue}$
	<i>dequeue</i> : $\text{queue} \rightarrow \text{queue}$
	<i>isempty</i> : $\text{queue} \rightarrow \text{bool}$

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

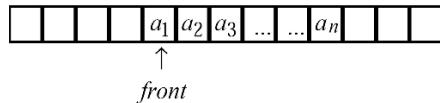


Abbildung 3.18: Queue-Implementierung im Array

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

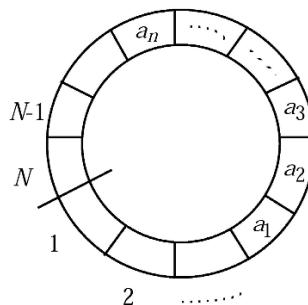


Abbildung 3.19: Zyklische Queue-Implementierung im Array

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

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

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

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

□

## 3.4 Abbildungen

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

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

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

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

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

**algebra mapping**

<b>sorts</b>	mapping, domain, range
<b>ops</b>	$\text{empty} : \rightarrow \text{mapping}$ $\text{assign} : \text{mapping} \times \text{domain} \times \text{range} \rightarrow \text{mapping}$ $\text{apply} : \text{mapping} \times \text{domain} \rightarrow \text{range} \cup \{\perp\}$

**sets**       $\text{mapping} = \text{die Menge aller } S, \text{ für die gilt:}$

- (i)  $S \subseteq \{(d, r) \mid d \in \text{domain}, r \in \text{range} \cup \{\perp\}\}$
- (ii)  $\forall d \in \text{domain} \exists r \in \text{range} \cup \{\perp\} : (d, r) \in S$
- (iii)  $\forall (d_1, r_1) \in S, (d_2, r_2) \in S : d_1 = d_2 \Rightarrow r_1 = r_2$

**functions**

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

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

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

$\text{apply}(m, d) = x$

**end mapping.**

---

Abbildung 3.20: Algebra mapping

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

### 3.5 Binäre Bäume

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

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

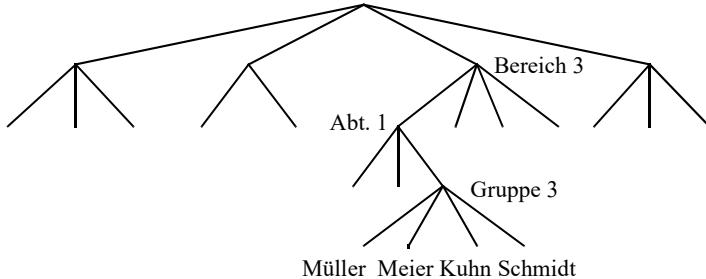


Abbildung 3.21: Baumdarstellung einer Unternehmenshierarchie („Organigramm“)

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

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

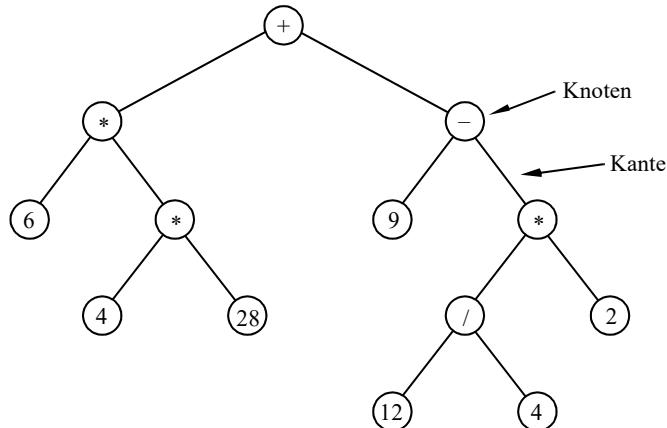


Abbildung 3.22: Operatorbaum

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

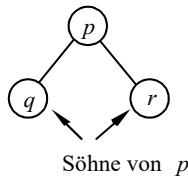


Abbildung 3.23: Vater-Sohn-Beziehung

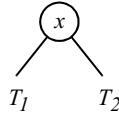
$p$  heißt *Vater* für diese Knoten. Wir betrachten in diesem Abschnitt zunächst den Spezialfall *binärer Bäume*. Das sind Bäume, in denen jeder Knoten *zwei* Söhne hat, die allerdings jeweils fehlen dürfen, so dass jeder Knoten entweder keinen Sohn hat, oder einen linken Sohn, oder einen rechten Sohn,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

**Definition 3.5:** (Binäre Bäume)

- (i) Der leere Baum ist ein binärer Baum.

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

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

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

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

**Beispiel 3.6:** Teilausdruck in graphischer und linearer Notation

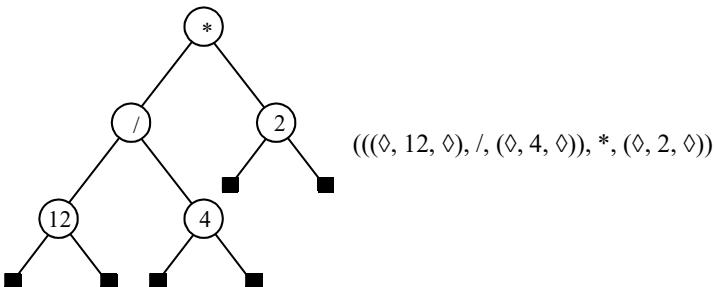


Abbildung 3.25: Graphische vs. lineare Notation eines Baumes

Gewöhnlich lässt man allerdings in der graphischen Darstellung die leeren Bäume weg und zeichnet Blätter als Rechtecke, die anderen Knoten als Kreise.  $\square$

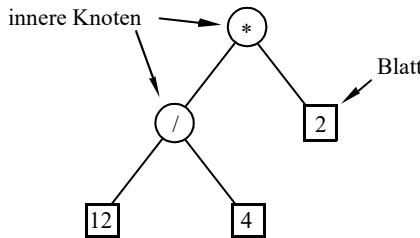


Abbildung 3.26: Innere Knoten und Blätter

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

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

Die folgenden Eigenschaften binärer Bäume lassen sich recht einfach herleiten:

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

**Beweis:** Dieser Fall tritt ein, wenn der Baum zu einer Liste entartet (Abbildung 3.27).  $\square$

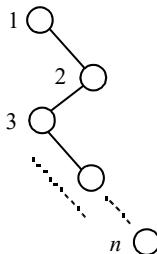


Abbildung 3.27: Zur Liste entarteter Baum

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

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

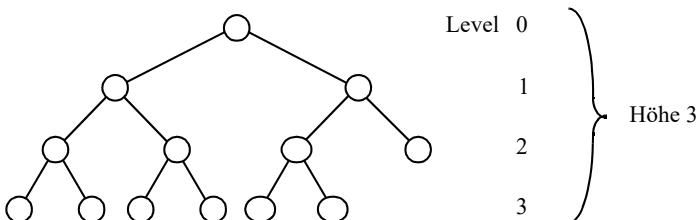


Abbildung 3.28: Vollständiger binärer Baum

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

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

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

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

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

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

Damit lässt sich  $h$  ausrechnen:

$$(2^h - 1) + 1 \leq n < (2^{h+1} - 1) + 1$$

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

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

□

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

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

□

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

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

Listen lassen sich ebenfalls als rekursive Strukturen auffassen:

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

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

**algebra tree**

**sorts**      *tree, elem*

**ops**       $\text{empty} : \rightarrow \text{tree}$   
 $\text{maketree} : \text{tree} \times \text{elem} \times \text{tree} \rightarrow \text{tree}$   
 $\text{key} : \text{tree} \rightarrow \text{elem}$   
 $\text{left}, \text{right} : \text{tree} \rightarrow \text{tree}$   
 $\text{isempty} : \text{tree} \rightarrow \text{bool}$

**sets**       $\text{tree} = \{\Diamond\} \cup \{(l, x, r) \mid x \in \text{elem}, l, r \in \text{tree}\}$   
{man beachte die rekursive Mengendefinition}

**functions**

$\text{empty} = \Diamond$   
 $\text{maketree}(l, x, r) = (l, x, r)$   
Sei  $t = (l, x, r)$ . Sonst sind  $\text{key}, \text{left}, \text{right}$  undefiniert.  
 $\text{key}(t) = x$   
 $\text{left}(t) = l$   
 $\text{right}(t) = r$

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

**end tree.**

Abbildung 3.29: Algebra tree

**ops**      *inorder, preorder, postorder : tree → list*

**functions**

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

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

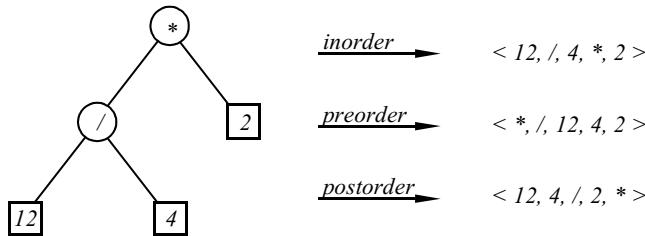


Abbildung 3.30: Baumdurchläufe

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

**Selbsttestaufgabe 3.5:** Schreiben Sie eine Methode für die Operation

*height: tree  $\rightarrow$  integer,*

die die Höhe eines binären Baumes berechnet.  $\square$

## Implementierungen

### (a) mit Zeigern

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

```

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

```

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

```

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

```

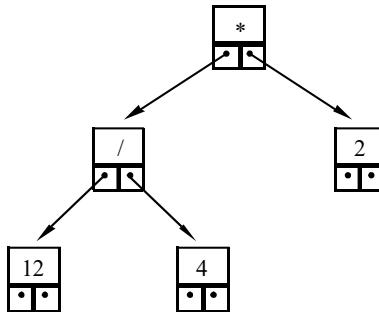


Abbildung 3.31: Binärbaum-Implementierung mit Zeigern

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

```

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

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

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

### (b) Array - Einbettung

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

Man betrachte einen vollständigen binären Baum der Höhe  $h$ .

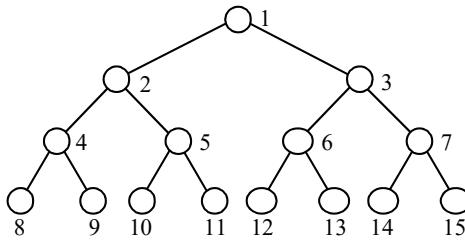


Abbildung 3.32: Vollständiger binärer Baum

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

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

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

Zum Schluss dieses Abschnitts sei noch eine Implementierung der Funktion *inorder* als Methode der Klasse *Node* auf der Basis der Datentypen *tree* und *list*<sub>1</sub> (deren Implementierungen hier gar nicht bekannt sein müssen) gezeigt:

```

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

### 3.6 (Allgemeine) Bäume

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

**Definition 3.12:** (Bäume)

- (i) Ein einzelner Knoten  $x$  ist ein Baum.

Graphische Darstellung:



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

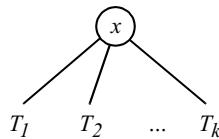


Abbildung 3.33: Graphische Darstellung des  $(k+1)$ -Tupels  $(x, T_1, \dots, T_k)$

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

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

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

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

Wie man in Abbildung 3.34 erkennt, gilt:

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

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

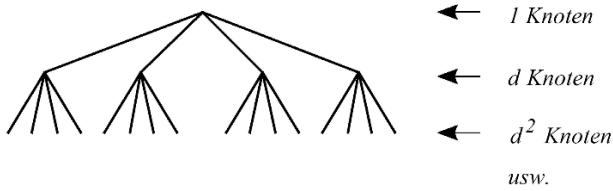


Abbildung 3.34: Baum mit maximaler Knotenzahl

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

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

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

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

Damit kann man ausrechnen:

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

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

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

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

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

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

□

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

## Implementierungen

### (a) über Arrays

Voraussetzung für diese Implementierung ist, dass alle Knoten höchstens einen festen Grad  $d$  haben. Dann können Zeiger auf die Söhne eines Knotens in einem Array gespeichert werden, so dass  $\text{sons}[i]$  einen Zeiger auf den  $i$ -ten Sohn enthält.



Abbildung 3.35: Array-Implementierung allgemeiner Bäume

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

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

### (b) über Binärbäume

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

Eine entsprechende Typdeklaration dazu sieht so aus:

```

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

```

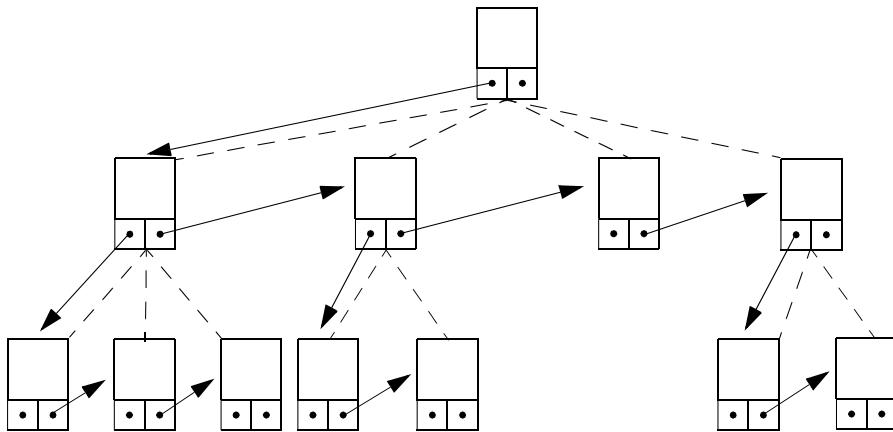


Abbildung 3.36: Binärbaum-Implementierung allgemeiner Bäume

### 3.7 Weitere Aufgaben

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

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

Spezifizieren Sie eine entsprechende Algebra!

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

$$\text{alpha: } a \times b \rightarrow c \times d$$

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

**Aufgabe 3.7:** Implementieren Sie die Operationen *empty*, *front*, *next*, *bol*, *eol* und *insert* des Datentyps *list*<sub>2</sub> in Java. Die Listenelemente sollen das Interface *Elem* realisieren.

**Aufgabe 3.8:** Unter Benutzung des Datentyps *list*<sub>1</sub> sollen Polynome der Form

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

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

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

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

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

**Aufgabe 3.10:** Angenommen, Sie hätten eine vollständige Java-Implementierung des im Text angegebenen Datentyps *list*<sub>2</sub> zur Verfügung. Geben Sie eine Implementierung der Datentypen

- (a) *stack* und
- (b) *queue*

auf dieser Basis an.

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

- (a) *contains*: *tree* × *elem* → *bool*
- (b) *ancestors*: *tree* × *elem* → *elemset*

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

**Aufgabe 3.12:** Die Knoten eines binären Baumes seien durchnummieriert von 1 bis  $n$ . Angenommen, wir haben Arrays *preorder*, *inorder* und *postorder*, die zu jedem Knoten  $i$  die Position des Knotens in der zugehörigen Ordnung enthalten. Beschreiben Sie einen Algorithmus, der für zwei Knoten mit Nummern  $i$  und  $j$  bestimmt, ob  $i$  Vorfahr von  $j$  ist. Erklären Sie, warum Ihr Algorithmus funktioniert. Sind tatsächlich alle drei Ordnungen notwendig?

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

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

durch einen binären Baum unter Verwendung eines Stacks. Die Ergebnisliste soll unter Verwendung des Datentyps *list<sub>2</sub>* aufgebaut werden.

## 3.8 Literaturhinweise

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

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

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



## 4 Datentypen zur Darstellung von Mengen

Die Darstellung von Mengen ist offensichtlich eine der grundlegendsten Aufgaben überhaupt. Wir haben im letzten Kapitel bereits einige Bausteine kennengelernt, die zur Darstellung von Mengen eingesetzt werden können (Listen, Bäume). In diesem Kapitel werden verschiedene Datentypen für Mengen betrachtet, die sich durch ihre Operationssätze unterscheiden; es geht nun darum, die Grundbausteine geeignet auszuwählen und zu verfeinern, um spezielle Operationen effizient zu unterstützen.

Wir betrachten zunächst das relativ einfache Problem, Mengen so darzustellen, dass die klassischen Operationen Vereinigung, Durchschnitt und Differenz effizient ausgeführt werden können. Der Hauptteil des Kapitels ist einem Datentyp gewidmet, der das Einfügen und Entfernen von Elementen zusammen mit einem Test auf Enthaltssein anbietet, bekannt als *Dictionary*. Dazu betrachten wir zunächst einige einfache, aber nicht zufriedenstellende Implementierungsstrategien und dann *Hashing*, *binäre Suchbäume* und *AVL-Bäume*. Aus algorithmischer Sicht sind dies Lösungen des Problems des *Suchens* auf einer Menge. Zwei weitere Datentypen schließen das Kapitel ab, nämlich *Priority Queues* (Mengen mit den Operationen des Einfügens und der Entnahme des Minimums) und ein Typ zur Verwaltung von Partitionen von Mengen mit den Operationen Verschmelzen zweier Teilmengen und Auffinden einer Teilmenge, zu der ein gegebenes Element gehört. Die Implementierungen dieser beiden Typen bilden wiederum wichtige Bausteine für Graph- und Sortieralgorithmen der folgenden Kapitel.

Neben der Vorstellung spezieller Algorithmen und Datenstrukturen geht es in diesem Kapitel auch darum, die Analyse von Algorithmen, insbesondere die Durchschnittsanalyse, genauer kennenzulernen und die dazugehörige Rechentechnik einzuüben. Wir haben uns bemüht, anspruchsvollere Rechnungen recht ausführlich darzustellen, um auch dem mathematisch nicht so Geübten den Zugang zu erleichtern.

### 4.1 Mengen mit Durchschnitt, Vereinigung, Differenz

Wir betrachten zunächst Mengen mit den klassischen Operationen Durchschnitt, Vereinigung und Differenz ( $\cap$ ,  $\cup$  und  $\setminus$ ). Zum Aufbau solcher Mengen wird man daneben sicherlich Operationen *empty* und *insert* brauchen; wir sehen eine weitere Operation *enumerate* vor, die uns alle Elemente einer Menge auflistet. Damit erhält man folgende Algebra:

```

algebra set1
sorts      set, elem, list
ops       empty      :           → set
            insert     : set × elem   → set
            union      : set × set   → set
            intersection : set × set   → set
            difference  : set × set   → set
            enumerate   : set         → list
sets        ...
functions   ...
end set1.

```

Wir verzichten darauf, die Trägermengen und Funktionen zu spezifizieren; die zentralen Operationen *union*, *intersection* und *difference* sind ja wohlbekannt.

### Implementierungen

Wir nehmen an, dass die darzustellenden Mengen einem linear geordneten Grundwertebereich entstammen. Das ist gewöhnlich der Fall (weil z. B.  $S \subset \text{integer}$  oder  $S \subset D$  für irgendeinen geordneten atomaren Wertebereich  $D$ ). Andernfalls definiert man willkürlich irgendeine Ordnung, z. B. die lexikographische Ordnung auf einer Menge von  $k$ -Tupeln. Für die Repräsentation der Mengen bieten sich verschiedene Möglichkeiten an:

#### (a) Bitvektor

Falls die darzustellenden Mengen Teilmengen eines genügend kleinen, endlichen Wertebereichs  $U = \{a_1, \dots, a_N\}$  sind, so eignet sich eine *Bitvektor*-Darstellung ( $U$  steht für “Universum”).

```

type set1 = array [1..N] of bool;
var s : set1;

```

Dabei gilt  $a_i \in s$  genau dann, wenn  $s[i] = \text{true}$ .

Dies entspricht der Technik der Mengendarstellung für den vordefinierten *set*-Typ aus Kapitel 2, wie er beispielsweise von PASCAL oder Modula-2 angeboten wird. Allerdings ist dort die Größe des “Universums”  $U$  durch die Länge eines Speicherwortes beschränkt.

Für die Komplexität der Operationen gilt bei dieser Art der Mengenimplementierung:

<i>insert</i>	O(1)
<i>empty, enumerate</i>	O( $N$ )
<i>union, intersection, difference</i>	O( $N$ )

Man beachte, dass der Aufwand proportional zur Größe des Universums  $U$  ist, nicht proportional zur Größe der dargestellten Mengen.

### (b) Ungeordnete Liste

Diese Implementierung, bei der jede Menge durch eine Liste ihrer Elemente dargestellt wird, ist nicht besonders effizient, da die uns interessierenden Operationen Durchschnitt, Vereinigung und Differenz für zwei Listen der Größen  $n$  und  $m$  jeweils Komplexität  $O(n \cdot m)$  haben. Man muss z. B. für die Durchschnittsbildung für jedes Element der ersten Liste die gesamte zweite Liste durchlaufen, um zu testen, ob das Element ebenfalls in der zweiten Liste vorkommt.

### (c) Geordnete Liste

Das ist die für den allgemeinen Fall beste Darstellung. Sie ist insbesondere dann von Interesse, wenn die zu behandelnden Mengen wesentlich kleiner sind als das Universum  $U$ , da die Komplexität der Operationen bei dieser Implementierung nicht von der Größe des Universums, sondern von der Mächtigkeit der Mengen abhängt. Die zugehörigen Algorithmen besitzen folgende Komplexitäten:

<i>empty</i>	: Liste initialisieren	O(1)
<i>insert</i>	: Richtige Position in Liste suchen, dort einfügen	O( $n$ )
<i>union,</i> <i>intersection,</i> <i>difference</i>	“Paralleler” Durchlauf durch die beiden beteiligten Listen (s. unten)	O( $n + m$ )
<i>enumerate</i>	: Durchlauf	O( $n$ )

Als Beispiel für die Implementierung der zentralen Operationen betrachten wir die Durchschnittsbildung.

**Beispiel 4.1:** Durchschnittsbildung mit parallelem Durchlauf.

Hierbei wird zunächst das erste Element der ersten Liste gemerkt und in der zweiten Liste solange von ihrem Beginn an gesucht, bis ein Element gefunden wird, das gleich oder größer dem gemerkten Element ist. Ist das in der zweiten Liste gefundene Element gleich dem gemerkten, so gehört dieses zum Durchschnitt, jedoch aufgrund der Ordnung

keiner seiner Vorgänger. Nun wird das aktuelle Element aus der zweiten Liste gemerkt und in der ersten Liste sequentiell nach einem Element gesucht, das größer oder gleich diesem ist. So wird fortgefahren, bis beide Listen erschöpft sind. In der folgenden Abbildung ist in der ersten und zweiten Zeile jeweils eine der beteiligten Listen dargestellt, in der dritten Zeile der Durchschnitt beider durch die Listen gegebenen Mengen. Die Ziffern geben die Reihenfolge an, in der die Zeiger fortgeschaltet werden.

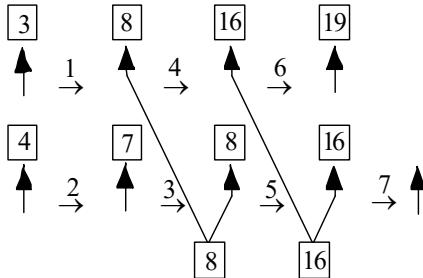


Abbildung 4.1: Durchschnittsbildung mit parallelem Durchlauf

Wir implementieren diese Funktionalität auf der Basis des Datentyps *list<sub>2</sub>* (Abschnitt 3.1) als Erweiterung der Klasse *List*.

Der Algorithmus zur Durchschnittsbildung überprüft je zwei Listenelemente auf Gleichheit sowie auf Bestehen einer Kleiner-Beziehung. Deshalb ergänzen wir das Interface *Elem* für Listenelemente um die Methoden *isEqual* und *isLess*:

```

interface Elem
{
    String toString();
    boolean isEqual(Elem e);
    boolean isLess(Elem e);
}
  
```

Damit kann der Algorithmus zur Durchschnittsbildung als Methode *intersection* der Klasse *List* wie folgt implementiert werden:

```

public List intersection(List l2)
{
    Pos p1 = next(front());
    Pos p2 = l2.next(l2.front());
    List l = new List(); Pos p = l.front();
  
```

```

while( $!(p_1 == \text{null} \parallel p_2 == \text{null})$ )
{
    if (retrieve( $p_1$ ).isEqual( $l_2.\text{retrieve}(p_2)$ ))
    {
         $l.\text{insert}(p, \text{retrieve}(p_1));$ 
         $p = l.\text{next}(p);$ 
         $p_1 = \text{next}(p_1); p_2 = l_2.\text{next}(p_2);$ 
    }
    else
    {
        if (retrieve( $p_1$ ).isLess( $l_2.\text{retrieve}(p_2)$ ))
             $p_1 = \text{next}(p_1);$ 
        else
             $p_2 = l_2.\text{next}(p_2);$ 
    }
}
return  $l;$ 
}

```

Um die Komplexität dieser Operation anzugeben, muss man sich nur klarmachen, dass in jedem Schleifendurchlauf mindestens ein Element einer Liste “verbraucht” wird. Nach  $n+m$  Durchläufen sind also beide Listen erschöpft. Die Komplexität der Operation *intersection* ist daher  $O(n + m)$ .  $\square$

Vereinigung und Differenz lassen sich ganz analog mit parallelem Durchlauf und der gleichen Zeitkomplexität berechnen.

**Selbsttestaufgabe 4.1:** Formulieren Sie eine Methode, die testet, ob zwischen zwei Mengen, die als geordnete Listen dargestellt werden, eine Inklusion (Teilmengenbeziehung) besteht. Die Richtung der Inklusion sei beim Aufruf nicht festgelegt. Vielmehr soll die Methode einen Zeiger auf die größere Menge zurückliefern, bzw. *null*, falls in keiner Richtung eine Inklusion besteht.  $\square$

## 4.2 Dictionaries: Mengen mit INSERT, DELETE, MEMBER

Die meisten Anwendungen von Mengendarstellungen benötigen nicht die Operationen Durchschnitt, Vereinigung und Differenz. Der bei weitem am häufigsten gebrauchte Satz von Operationen enthält die *insert*-, *delete*- und *member*-Operationen. Ein Datentyp, der im Wesentlichen diese Operationen anbietet, heißt *Dictionary* (Wörterbuch).

Wir betrachten also in diesem Abschnitt einen Datentyp *set<sub>2</sub>* mit Operationen *empty*, *isempty*, *insert*, *delete* und *member*, bzw. seine Implementierungen. Der Datentyp entspricht gerade dem Einleitungsbeispiel (Abschnitt 1.2), wobei *contains* in *member* umbenannt wird und jetzt beliebige Grundmengen zugelassen sind.

#### 4.2.1 Einfache Implementierungen

Wir kennen bereits verschiedene einfache Implementierungsmöglichkeiten, die im Folgenden noch einmal mit ihren Komplexitäten zusammengefasst werden:

- (a) Sequentiell geordnete Liste im Array (Einleitungsbeispiel)

<i>insert, delete</i>	$O(n)$
<i>member</i>	$O(\log n)$
Platzbedarf	$O(N)$ ( $N$ Größe des Array)

Eine Menge kann bei dieser Implementierung nicht beliebig wachsen, da ihre maximale Mächtigkeit durch die Größe des Arrays beschränkt ist.

- (b) Ungeordnete Liste

<i>insert</i>	$O(1)$
mit Duplikateliminierung	$O(n)$
<i>delete, member</i>	$O(n)$
Platzbedarf	$O(n)$

- (c) Geordnete Liste

<i>insert, delete, member</i>	$O(n)$
Platzbedarf	$O(n)$

- (d) Bitvektor

<i>insert, delete, member</i>	$O(1)$
Platzbedarf	$O(N)$

Die Repräsentation als Bitvektor ist nur im Spezialfall anzuwenden, da die Größe des "Universums" beschränkt sein muss.

Alle einfachen Implementierungen haben also irgendwelche Nachteile. Ideal wäre eine Darstellung, die bei linearem Platzbedarf alle Operationen in konstanter Zeit realisiert, also:

<i>insert, delete, member</i>	$O(1)$
Platzbedarf	$O(n)$

In den folgenden Abschnitten versuchen wir, einer solchen Implementierung nahezukommen.

### 4.2.2 Hashing

Die Grundidee von *Hashverfahren* besteht darin, aus dem *Wert* eines zu speichernden Mengenelementes seine *Adresse* im Speicher zu berechnen. Den Speicher zur Aufnahme der Mengenelemente fasst man auf als eine Menge von *Behältern* ("buckets"), die etwa  $B_0, \dots, B_{m-1}$  nummeriert seien. Der Wertebereich  $D$ , aus dem Mengenelemente stammen können, kann beliebig groß sein; es gilt also gewöhnlich  $|D| \gg m$ . Wenn die zu speichernden Objekte eine komplexe innere Struktur besitzen (z. B. Personen-Records), so benutzt man eine Komponente oder eine Kombination von Komponenten als Wert, der die Abbildung auf den Speicher kontrolliert; dieser Wert heißt *Schlüssel* (z. B. Nachnamen von Personen). Eine *Hashfunktion* (vornehmer: Schlüsseltransformation) ist eine totale Abbildung

$$h : D \rightarrow \{0, \dots, m-1\}$$

wobei  $D$  der Schlüssel-Wertebereich ist. In typischen Anwendungen ist  $D$  etwa die Menge aller Zeichenketten der Maximallänge 20. Eine Hashfunktion sollte folgende Eigenschaften besitzen:

1. Sie sollte *surjektiv* sein, also alle Behälter erfassen.
2. Sie sollte die zu speichernden Schlüssel möglichst *gleichmäßig* über alle Behälter verteilen.
3. Sie sollte effizient zu berechnen sein.

**Beispiel 4.2:** Wir wollen die Monatsnamen über 17 Behälter verteilen. Eine einfache Hashfunktion, die Zeichenketten  $c_1 \dots c_k$  abbildet, benutzt den Zahlenwert der Binärdarstellung jedes Zeichens, der  $N(c_i)$  heiße:

$$h_0(c_1 \dots c_k) = \sum_{i=1}^k N(c_i) \bmod m$$

Für das Beispiel vereinfachen wir dies noch etwas, indem wir nur die ersten drei Zeichen betrachten und annehmen:  $N(A) = 1, N(B) = 2, \dots, N(Z) = 26$ . Also

$$h_1(c_1 \dots c_k) = (N(c_1) + N(c_2) + N(c_3)) \bmod 17.$$

Dabei behandeln wir Umlaute als 2 Zeichen, also ä = ae usw. Damit verteilen sich die Monatsnamen wie folgt:

0	November	9	Juli
1	April, Dezember	10	
2	März	11	Juni
3		12	August, Oktober
4		13	Februar
5		14	
6	Mai, September	15	
7		16	
8	Januar		

□

Wir sehen, dass bisweilen mehrere Schlüssel auf denselben Behälter abgebildet werden; dies bezeichnet man als *Kollision*. Hashverfahren unterscheiden sich in der Art der Kollisionsbehandlung bzw. der Auffassung von Behältern. Beim *offenen Hashing* nimmt man an, dass *ein* Behälter beliebig viele Schlüssel aufnehmen kann, indem z. B. eine verketzte Liste zur Darstellung der Behälter verwendet wird. Beim *geschlossenen Hashing* kann ein Behälter nur eine kleine konstante Anzahl  $b$  von Schlüsseln aufnehmen; falls mehr als  $b$  Schlüssel auf einen Behälter fallen, entsteht ein *Überlauf* (“overflow”). Der gewöhnlich betrachtete Spezialfall ist  $b = 1$ , dies ist der Fall, der zum Begriff Kollision geführt hat.

Wie wahrscheinlich sind Kollisionen?

Im Folgenden gehen wir davon aus, dass Ihnen der Inhalt des Abschnitts Grundlagen II des Anhangs “Mathematische Grundlagen” bekannt ist.

Wir nehmen an, dass eine “ideale” Hashfunktion vorliegt, die  $n$  Schlüsselwerte völlig gleichmäßig auf  $m$  Behälter verteilt,  $n < m$ . Bezeichne  $P_X$  die Wahrscheinlichkeit, dass Ereignis  $X$  eintritt. Offensichtlich gilt:

$$\begin{aligned} P_{\text{Kollision}} &= 1 - P_{\text{keine Kollision}} \\ P_{\text{keine Kollision}} &= P(1) \cdot P(2) \cdot \dots \cdot P(n) \end{aligned}$$

wobei  $P(i)$  die Wahrscheinlichkeit bezeichnet, dass der  $i$ -te Schlüssel auf einen freien Behälter abgebildet wird, wenn alle vorherigen Schlüssel ebenfalls auf freie Plätze abgebildet wurden.  $P_{\text{keine Kollision}}$  ist also die Wahrscheinlichkeit, dass alle Schlüssel auf freie Behälter abgebildet werden. Zunächst gilt

$$P(1) = 1$$

da zu Anfang noch kein Behälter gefüllt ist. Beim Einfügen des zweiten Elements ist ein Behälter gefüllt,  $m-1$  Behälter sind frei. Da jeder Behälter mit gleicher Wahrscheinlichkeit  $1/m$  getroffen wird, gilt

$$P(2) = \frac{m-1}{m}$$

Analog gilt für alle folgenden Elemente

$$P(i) = \frac{m-i+1}{m}$$

da jeweils bereits  $(i-1)$  Behälter belegt sind. Damit ergibt sich für die Gesamtwahrscheinlichkeit:

$$P_{Kollision} = 1 - \frac{m(m-1)(m-2)\dots(m-n+1)}{m^n}$$

Wir betrachten ein Zahlenbeispiel. Sei  $m = 365$ , dann ergibt sich:

$n$	$P_{Kollision}$
22	0.475
23	0.507
50	0.970

Tabelle 4.1: Kollisionswahrscheinlichkeiten

Dieser Fall ist als das “Geburtstagsparadoxon” bekannt: Wenn 23 Leute zusammen sind, ist die Wahrscheinlichkeit, dass zwei von ihnen am gleichen Tag Geburtstag haben, schon größer als 50%. Bei 50 Personen ist es fast mit Sicherheit der Fall.

Als Konsequenz aus diesem Beispiel ergibt sich, dass Kollisionen im Normalfall praktisch unvermeidlich sind.

Beim *offenen Hashing* wird jeder Behälter einfach durch eine beliebig erweiterbare Liste von Schlüsseln dargestellt; ein Array von Zeigern verwaltet die Behälter.

**var hashtable: array [0..m-1] of ↑listelem**

Für Beispiel 4.2 ist ein Teil einer solchen Hashtabelle in Abbildung 4.2 gezeigt.

Beim offenen Hashing stellen Kollisionen also kein besonderes Problem dar. Es ist auch nicht nötig, dass die Gesamtzahl der Einträge kleiner ist als die Anzahl der Tabellenplätze, wie beim geschlossenen Hashing mit  $b = 1$ . Zum Einfügen, Entfernen und Suchen eines Schlüssels  $k$  wird jeweils die Liste  $hashtable[i]$  mit  $i = h(k)$  durchlaufen.

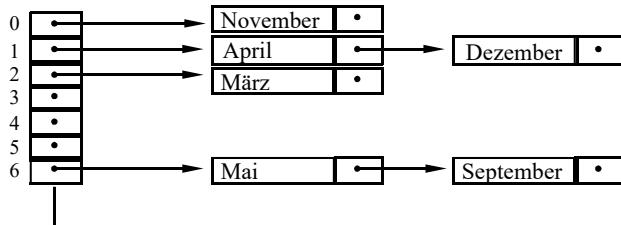


Abbildung 4.2: Offenes Hashing

Die durchschnittliche Listenlänge ist  $n/m$ . Wenn man  $n$  ungefähr gleich  $m$  wählt und falls die Hashfunktion die Schlüssel tatsächlich gleichmäßig über die verschiedenen Listen verteilt, so ist der erwartete Zeitaufwand für Suchen, Einfügen und Entfernen jeweils  $O(1 + n/m) = O(1)$ .

Andererseits gibt es keine Garantie, dass nicht etwa alle Schlüssel auf eine einzige Liste abgebildet werden. Der worst case ist bei allen Hashverfahren sehr schlecht, nämlich  $O(n)$ . Der Platzbedarf ist  $O(n+m)$ .

Offenes Hashing bzw. seine Implementierungstechnik mit getrennten Listen für jeden Behälter wird auch als *separate chaining* bezeichnet.

Beim *geschlossenen Hashing* ist die Zahl der Einträge  $n$  begrenzt durch die Kapazität der Tabelle  $m \cdot b$ . Wir betrachten den Spezialfall  $b = 1$ ; für  $b > 1$  lassen sich analoge Techniken entwickeln. Jede Zelle der Hashtabelle kann also genau ein Element aufnehmen. Dies wird gewöhnlich so implementiert, dass Elemente direkt als Array-Komponenten gespeichert werden.

```

var hashtable = array [0..m-1] of elem;
type elem = record key: keydomain;
            {weitere Information}
            end
  
```

Wir nehmen an, dass ein spezieller Wert des *keydomain* sich eignet, um auszudrücken, dass eine Zelle der Tabelle unbesetzt ist; wir werden später sehen, dass ein weiterer Wert benötigt wird, um darzustellen, dass eine Zelle besetzt war und das enthaltene Element gelöscht worden ist. Wir bezeichnen diese Werte mit *empty* und *deleted* (z. B. *empty* = <Leerstring>; *deleted* = “\*\*\*\*”). Falls solche Werte nicht vorhanden sind, weil alle Elemente des *keydomain* als Schlüssel auftreten können, so muss eine weitere Record-Komponente eingeführt werden.

Für das geschlossene Hashing haben Methoden zur Kollisionsbehandlung entscheidende Bedeutung. Die allgemeine Idee, genannt *rehashing* oder auch *offene Adressierung*, besteht darin, neben  $h = h_0$  weitere Hashfunktionen  $h_1, \dots, h_{m-1}$  zu benutzen, die in irgendeiner Reihenfolge für einen gegebenen Schlüssel  $x$  die Zellen  $h(x), h_1(x), h_2(x)$  usw. inspizieren. Sobald eine freie oder als gelöscht markierte Zelle gefunden ist, wird  $x$  dort eingetragen. Bei einer Suche nach  $x$  wird die gleiche Folge von Zellen betrachtet, bis entweder  $x$  gefunden wird oder das Auftreten der ersten freien Zelle in dieser Folge anzeigt, dass  $x$  nicht vorhanden ist. Daraus ergibt sich, dass man ein Element  $y$  nicht einfach löschen (und die Zelle als frei markieren) kann, da sonst ein Element  $x$ , das beim Einfügen durch Kollisionen an  $y$  vorbeigeleitet worden ist, nicht mehr gefunden werden würde. Statt dessen muss für jedes entfernte Element die betreffende Zelle als *deleted* markiert werden. Der benutzte Platz wird also nicht wieder freigegeben; deshalb empfiehlt sich geschlossenes Hashing nicht für sehr “dynamische” Anwendungen mit vielen Einfügungen und Lösch-Operationen.

Die Folge der Hashfunktionen  $h_0, \dots, h_{m-1}$  sollte so gewählt sein, dass für jedes  $x$  der Reihe nach sämtliche  $m$  Zellen der Tabelle inspiziert werden. Die einfachste Idee dazu heißt *lineares Sondieren* und besteht darin, der Reihe nach alle Folgezellen von  $h(x)$  zu betrachten, also

$$h_i(x) = (h(x) + i) \bmod m \quad 1 \leq i \leq m-1$$

Beim Einfügen aller Monatsnamen in ihrer natürlichen Reihenfolge ergibt sich die folgende Verteilung, wenn  $h(x)$  wie vorher gewählt wird und lineares Sondieren angewendet wird:

0 November		9 Juli	
1 April		10	
2 März	}	11 Juni	
3 Dezember		12 August	
4		13 Februar	}
5		14 Oktober	}
6 Mai	}	15	
7 September	}	16	
8 Januar			

Bei annähernd voller Tabelle hat lineares Sondieren eine Tendenz, lange Ketten von besetzten Zellen zu bilden. Wenn nämlich bereits eine Kette der Länge  $t$  existiert, so ist die Wahrscheinlichkeit, dass eine freie Zelle dahinter als nächste belegt wird,  $(t+1)/m$

im Vergleich zu  $1/m$  für eine freie Zelle, deren Vorgänger auch frei ist, da jeder Eintrag, der irgendeinen Behälter in der Kette trifft, auf das der Kette folgende freie Element verschoben wird.

Wir werden im Folgenden geschlossenes Hashing analysieren und dabei solche Tendenzen zur Kettenbildung zunächst ignorieren. Dieses Modell nennt man *ideales* oder *uniformes* Hashing.

### Analyse des “idealen” geschlossenen Hashing

Wir fragen nach den erwarteten Kosten des *Einfügens*, d. h. nach der erwarteten Zahl von Zelleninspektionen (“*Proben*”) beim Einfügen in eine Tabelle der Größe  $m$ , in die  $n$  Elemente bereits eingetragen sind.

(Vielleicht sollten Sie sich den Abschnitt Grundlagen II des Anhangs “Mathematische Grundlagen” noch einmal ansehen.)

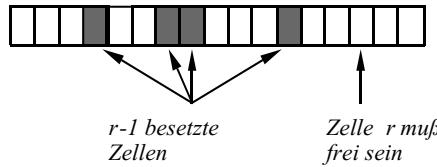
Die Anzahl aller Konfigurationen von  $n$  besetzten und  $m-n$  freien Zellen, also die Anzahl aller möglichen Auswahlen von  $n$  besetzten Zellen aus insgesamt  $m$  Zellen, ist

$$\binom{m}{n}$$

Wie groß ist die Wahrscheinlichkeit, dass beim Einfügen des  $(n+1)$ -ten Elementes  $x$  genau  $r$  Proben vorgenommen werden?  $x$  legt eine Folge von Zellen  $h_0(x), h_1(x), \dots$  fest, die untersucht werden. Es werden genau  $r$  Proben vorgenommen, wenn die ersten  $(r-1)$  Zellen  $h_0(x), \dots, h_{r-2}(x)$  besetzt sind und die  $r$ -te Zelle  $h_{r-1}(x)$  frei ist. Sei  $P_r$  die Wahrscheinlichkeit dafür, dass genau  $r$  Proben vorgenommen werden.

$$P_r = \frac{\#\text{“günstige” Ereignisse}}{\#\text{alle Ereignisse}} = \frac{\#\text{“günstige” Ereignisse}}{\binom{m}{n}}$$

“#” steht dabei für “Anzahl von”. Wieviel “günstige” Ereignisse gibt es, d. h. Ereignisse, in denen die ersten  $(r-1)$  untersuchten Zellen besetzt und die  $r$ -te untersuchte Zelle frei ist? Durch die Folge  $h_0(x), \dots, h_{r-1}(x)$  sind  $r$  Zellen als besetzt oder frei bereits festgelegt (Abbildung 4.3). Die restlichen  $n-(r-1)$  Elemente können auf beliebige Art auf die noch verbleibenden  $m-r$  Zellen verteilt sein.

Abbildung 4.3: Beim Einfügen sind  $r$  Proben nötig

Dazu gibt es genau

$$\binom{m-r}{n-(r-1)}$$

Möglichkeiten, die gerade die günstigen Ereignisse darstellen. Damit erhalten wir:

$$P_r = \frac{\binom{m-r}{n-r+1}}{\binom{m}{n}}$$

Der Erwartungswert für die Anzahl der Proben ist

$$\sum_{r=1}^m r \cdot P_r = \sum_{r=1}^m r \cdot \frac{\binom{m-r}{n-r+1}}{\binom{m}{n}} \quad (1)$$

Um das auszuwerten, müssen wir uns zunächst “bewaffnen”. Dazu sind einige Rechenregeln für den Umgang mit Binomialkoeffizienten in Grundlagen III angegeben.

Unser Ziel besteht darin, eine Form

$$\sum_{m=0}^n \binom{m}{k}$$

zu erreichen, da diese mit Hilfe der Gleichung III.5 ausgewertet werden kann. Der Nenner in der Summe in (1) ist unproblematisch, da in ihm die Summationsvariable  $r$  nicht vorkommt. Hingegen sollte im Zähler

$$r \cdot \binom{m-r}{n-r+1}$$

der Faktor  $r$  verschwinden und der untere Index  $n-r+1$  sollte irgendwie konstant, also unabhängig von  $r$  werden. Das letztere Ziel kann man erreichen durch Anwendung von III.1.

$$\text{III.1: } \binom{n}{k} = \binom{n}{n-k}$$

Damit ergibt sich

$$\binom{m-r}{n-r+1} = \binom{m-r}{m-r-n+r-1} = \binom{m-r}{m-n-1} \quad (2)$$

Um den Faktor  $r$  zu beseitigen, versuchen wir, ihn in den Binomialkoeffizienten hineinzuziehen, das heißt, wir wollen III.3 ausnutzen.

$$\text{III.3: } n \cdot \binom{n-1}{k-1} = k \cdot \binom{n}{k}$$

Dazu muss man aus dem Faktor  $r$  einen Faktor  $m-r+1$  machen, da sich dann mit III.3 ergibt:

$$(m-r+1) \cdot \binom{m-r}{m-n-1} = (m-n) \cdot \binom{m-r+1}{m-n} \quad (3)$$

Dies kann man wiederum erreichen, indem man in (1) eine Summe

$$\sum (m+1) \cdot P_r$$

addiert und wieder abzieht. Damit ergibt sich folgende Rechnung:

$$\begin{aligned} \sum r \cdot P_r &= \sum (m+1) \cdot P_r - \sum (m+1) \cdot P_r + \sum r \cdot P_r \\ &= (m+1) \cdot \sum P_r - \sum (m+1-r) \cdot P_r \\ &= m+1 - \sum_{r=1}^m (m-r+1) \binom{m-r}{m-n-1} / \binom{m}{n} \end{aligned}$$

Dabei haben wir die Tatsache ausgenutzt, dass  $\sum P_r = 1$  ist und das Ergebnis (2) eingebaut. Jetzt benutzen wir (3):

$$\begin{aligned}
 &= m+1 - \sum_{r=1}^m (m-n) \binom{m-r+1}{m-n} / \binom{m}{n} \\
 &= m+1 - \frac{m-n}{\binom{m}{n}} \cdot \sum_{r=1}^m \binom{m-r+1}{m-n} \tag{4}
 \end{aligned}$$

Wenn man die Summe ausschreibt, sieht man:

$$\sum_{r=1}^m \binom{m-r+1}{m-n} = \binom{m+1-m}{m-n} + \dots + \binom{m+1-1}{m-n} = \sum_{k=1}^m \binom{k}{m-n}$$

Diese Summe hat fast die Form, die in III.5 benötigt wird, es fehlt der Summand für  $k=0$ . Da aber gilt

$$\binom{0}{m-n} = 0$$

kann man diesen Summanden hinzufügen und endlich III.5 ausnutzen:

$$\text{III.5: } \sum_{m=0}^n \binom{m}{k} = \binom{n+1}{k+1}$$

$$\sum_{k=1}^m \binom{k}{m-n} = \sum_{k=0}^m \binom{k}{m-n} = \binom{m+1}{m-n+1}$$

Damit lässt sich der Ausdruck in (4) weiter ausrechnen:

$$= m+1 - \frac{m-n}{\binom{m}{n}} \cdot \binom{m+1}{m-n+1} \tag{5}$$

Wir wenden noch einmal III.1 an:

$$\text{III.1: } \binom{n}{k} = \binom{n}{n-k}$$

$$\binom{m+1}{m-n+1} = \binom{m+1}{m+1-m+n-1} = \binom{m+1}{n}$$

und setzen dies in (5) ein:

$$= m+1 - (m-n) \cdot \frac{\binom{m+1}{n}}{\binom{m}{n}} \quad (6)$$

Das ist nach Definition der Binomialkoeffizienten (siehe II.6):

$$\begin{aligned} &= m+1 - (m-n) \cdot \frac{\frac{(m+1)^n}{n!}}{\frac{m^n}{n!}} \\ &= m+1 - (m-n) \cdot \frac{m+1}{m-n+1} \\ &= (m+1) \cdot \left(1 - \frac{m-n}{m-n+1}\right) \\ &= \frac{m+1}{m-n+1} \end{aligned}$$

Das sind also die erwarteten *Kosten des Einfügens* des  $(n+1)$ -ten Elementes. Die gleichen Kosten entstehen beim *erfolglosen Suchen* in einer Tabelle der Größe  $m$  mit  $n$  Einträgen, da die Suche beim Auffinden der ersten freien Zelle abgebrochen wird.

Die *Kosten einer erfolgreichen Suche* nach dem  $k$ -ten eingefügten Element entsprechen den Kosten für das Einfügen des  $k$ -ten Elementes, ebenso die *Kosten für das Entfernen* des  $k$ -ten Elementes. Gemittelt über alle  $n$  eingefügten Elemente sind die erwarteten Kosten für das erfolgreiche Suchen oder Entfernen

$$\begin{aligned} &\frac{1}{n} \cdot \sum_{k=1}^n \frac{m+1}{m-(k-1)+1} \\ &= \frac{m+1}{n} \cdot \sum_{k=1}^n \frac{1}{m-k+2} \\ &= \frac{m+1}{n} \cdot \left( \frac{1}{m-n+2} + \frac{1}{m-n+3} + \dots + \frac{1}{m+1} \right) \quad (1) \end{aligned}$$

Um weiterzurechnen, benötigen wir Kenntnisse über *harmonische Zahlen* aus Grundlagen IV. Damit lässt sich (1) weiter auswerten:

$$= \frac{m+1}{n} \cdot (H_{m+1} - H_{m-n+1})$$

$$\approx \frac{m+1}{n} \cdot \ln \frac{m+1}{m-n+1}$$

Mit einem “Auslastungsfaktor”  $\alpha := n/m$  ergibt sich: Die erwarteten Kosten (Anzahl der Proben) sind

$$\approx \frac{1}{1-\alpha} =: C_n'$$

für das Einfügen und die erfolglose Suche, und

$$\approx \frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha} =: C_n$$

für das Entfernen und die erfolgreiche Suche<sup>1</sup>. Einige Zahlenwerte für  $C_n$  und  $C_n'$  bei verschiedenen Auslastungsfaktoren finden sich in Tabelle 4.2.

Auslastung	$C_n'$	$LIN_n'$	$C_n$	$LIN_n$
$\alpha = 20\%$	1,25	1,28	1,12	1,125
50%	2	2,5	1,38	1,5
80%	5	13	2,01	3
90%	10	50,5	2,55	5,5
95%	20	200,5	3,15	10,5

Tabelle 4.2: Auslastung und Kosten beim geschlossenen Hashing

Wir haben die Kosten für *eine* Operation auf einer mit  $n$  Elementen gefüllten Tabelle der Größe  $m$  betrachtet.

$$C_n = \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

---

1. Die Bezeichnungen  $C_n$  und  $C_n'$  entsprechen denen in [Knuth 1998].

beschreibt natürlich auch die *durchschnittlichen* Kosten für das Einfügen der ersten  $n$  Elemente; das war ja gerade die Herleitung. Die *Gesamtkosten* für den Aufbau der Tabelle sind entsprechend

$$n \cdot \frac{m+1}{n} \cdot \ln \frac{m+1}{m-n+1} \approx m \cdot \ln \frac{m}{m-n}$$

Wenn man die Effekte, die durch Kettenbildung entstehen, einbezieht, ergeben sich schlechtere Werte als beim idealen Hashing. Eine Analyse für diesen Fall, das *lineare Sondieren*, ist in [Knuth 1998] durchgeführt, die entsprechenden Werte sind:

$$LIN_n \approx \frac{1}{2} \cdot \left( 1 + \frac{1}{1-\alpha} \right) \quad (\text{erfolgreiche Suche})$$

$$LIN_n' \approx \frac{1}{2} \cdot \left( 1 + \frac{1}{(1-\alpha)^2} \right) \quad (\text{erfolglose Suche})$$

Entsprechende Vergleichswerte finden sich in Tabelle 4.2. Lineares Sondieren wird also sehr schlecht, wenn die Hashtabelle zu mehr als 80% gefüllt ist.

### Kollisionsstrategien

Um beim Auftreten von Kollisionen Kettenbildung zu vermeiden, müssen geeignete Funktionen  $h_i(x)$  gefunden werden. Wir betrachten verschiedene Möglichkeiten:

#### (a) Lineares Sondieren (Verallgemeinerung)

$$h_i(x) = (h(x) + c \cdot i) \bmod m \quad 1 \leq i \leq m-1$$

Dabei ist  $c$  eine Konstante. Die Zahlen  $c$  und  $m$  sollten teilerfremd sein, damit alle Zellen getroffen werden. Dieses Verfahren bietet keine Verbesserung gegenüber  $c = 1$ , es entstehen Ketten mit Abstand  $c$ .

#### (b) Quadratisches Sondieren

$$h_i(x) = (h(x) + i^2) \bmod m$$

Diese Grundidee kann man noch etwas verfeinern: man wähle

$$\left. \begin{array}{rcl} h_0(x) & = & h(x) \\ h_1(x) & = & h(x) + 1^2 \\ h_2(x) & = & h(x) - 1^2 \\ h_3(x) & = & h(x) + 2^2 \\ h_4(x) & = & h(x) - 2^2 \end{array} \right\} \text{mod } m$$

Etwas genauer sei

$$\left. \begin{array}{l} h_{2i-1}(x) = (h(x) + i^2) \text{ mod } m \\ h_{2i}(x) = (h(x) - i^2 + m^2) \text{ mod } m \end{array} \right\} 1 \leq i \leq \frac{m-1}{2}$$

Hier haben wir im zweiten Fall  $m^2$  addiert, um sicherzustellen, dass die modulo-Funktion auf ein positives Argument angewandt wird. Man wähle dabei  $m = 4 \cdot j + 3$ ,  $m$  Primzahl. Dann wird jede der  $m$  Zahlen getroffen (ein Ergebnis aus der Zahlentheorie [Radke 1970]). Quadratisches Sondieren ergibt keine Verbesserung für *Primärkollisionen* ( $h_0(x) = h_0(y)$ ), aber es vermeidet Clusterbildung bei *Sekundärkollisionen* ( $h_0(x) = h_k(y)$  für  $k > 0$ ), das heißt die Wahrscheinlichkeit für die Bildung längerer Ketten wird herabgesetzt.

### (c) Doppel-Hashing

Man wähle zwei Hashfunktionen  $h$ ,  $h'$ , die voneinander *unabhängig* sind. Das soll Folgendes bedeuten: Wir nehmen an, dass für jede der beiden Hashfunktionen eine Kollision mit Wahrscheinlichkeit  $1/m$  auftritt, also für zwei Schlüssel  $x$  und  $y$  gilt:

$$P(h(x) = h(y)) = \frac{1}{m} \quad P(h'(x) = h'(y)) = \frac{1}{m}$$

Die Funktionen  $h$  und  $h'$  sind unabhängig, wenn eine Doppelkollision nur mit der Wahrscheinlichkeit  $1/m^2$  auftritt:

$$P(h(x) = h(y) \wedge h'(x) = h'(y)) = \frac{1}{m^2}$$

Dann definieren wir eine Folge von Hashfunktionen

$$h_i(x) = (h(x) - i \cdot h'(x)) \text{ mod } m, \text{ für } i \geq 0.$$

Dabei muss  $h'(x) \neq 0$  sein und es darf  $m$  nicht teilen (das ist sicher, wenn  $m$  Primzahl ist).

Das ist endlich eine wirklich gute Methode. Experimente zeigen, dass ihre Kosten von idealem Hashing praktisch nicht unterscheidbar sind. Es ist allerdings nicht leicht, Paare von Hashfunktionen zu finden, die *beweisbar* voneinander unabhängig sind (einige sind in [Knuth 1998] angegeben). In der Praxis wird man sich oft mit “intuitiver” Unabhängigkeit zufrieden geben.

### Hashfunktionen

Für die Wahl der Basis-Funktion  $h(x)$  bieten sich z. B. die Divisionsmethode oder die Mittel-Quadrat-Methode an.

#### (a) Divisionsmethode

Dies ist die einfachste denkbare Hashfunktion. Seien die natürlichen Zahlen der Schlüsselbereich, dann wählt man

$$h(k) = k \bmod m$$

wobei  $m$  die maximale Anzahl von Einträgen ist. Ein Nachteil dabei ist, dass aufeinanderfolgende Zahlen  $k, k+1, k+2, \dots$  auf aufeinanderfolgende Zellen abgebildet werden; das kann störende Effekte haben.

#### (b) Mittel-Quadrat-Methode

Sei  $k$  dargestellt durch eine Ziffernfolge  $k_r k_{r-1} \dots k_1$ . Man bilde  $k^2$ , dargestellt durch  $s_{2r} s_{2r-1} \dots s_1$  und entnehme einen Block von mittleren Ziffern als Adresse  $h(k)$ . Die mittleren Ziffern hängen von *allen* Ziffern in  $k$  ab. Dadurch werden aufeinanderfolgende Werte besser gestreut.

**Beispiel 4.3:** Die Abbildung einiger Werte von  $k$  für  $m = 100$  ist in Tabelle 4.3 gezeigt.

$k$	$k \bmod m$	$k^2$	$h(k)$
127	27	16129	12
128	28	16384	38
129	29	16641	64

Tabelle 4.3: Mittel-Quadrat-Methode

□

Für die Abbildung von Zeichenketten muss man zunächst die Buchstabencodes aufsummieren.

**Zusammenfassung:** Hash-Verfahren haben ein sehr schlechtes worst-case-Verhalten ( $O(n)$  für die drei Dictionary-Operationen), können aber (mit etwas Glück) sehr gutes Durchschnittsverhalten zeigen ( $O(1)$ ). Alle Hash-Verfahren sind relativ einfach zu implementieren. Bei allgemeinen dynamischen Anwendungen sollte *offenes Hashing* gewählt werden, da hier auch Entfernen von Elementen und Überschreiten der festen Tabellengröße problemlos möglich ist. Sobald die Tabellengröße um ein Vielfaches überschritten wird, sollte man *reorganisieren*, das heißt, eine neue, größere Tabelle anlegen und dort alle Elemente neu eintragen. *Geschlossenes Hashing* eignet sich im Wesentlichen nur für Anwendungen, bei denen die Gesamtzahl einzufügender Elemente von vornherein beschränkt ist und keine oder nur sehr wenige Elemente entfernt werden müssen. Beim geschlossenen Hashing sollte eine Auslastung von 80% nicht überschritten werden.

Man muss für jede Anwendung überprüfen, ob eine gewählte Hashfunktion die Schlüssel gleichmäßig verteilt. Ein Nachteil von Hash-Verfahren ist noch, dass die Menge der gespeicherten Schlüssel nicht effizient in sortierter Reihenfolge aufgelistet werden kann.

**Selbsttestaufgabe 4.2:** Konstruieren Sie die Hash-Tabelle, die sich nach dem Einfügen aller Monatsnamen ergibt, mit derselben Hashfunktion, die in Beispiel 4.2 verwendet wurde und  $m = 17$ . Verwenden Sie diesmal jedoch quadratisches Sondieren als Kollisionsstrategie (nur in der Grundform, also mit positiven Inkrementen).  $\square$

### 4.2.3 Binäre Suchbäume

Wir erhalten eine weitere effiziente Dictionary-Implementierung, indem wir in geeigneter Weise die Elemente einer darzustellenden Menge den Knoten eines binären Baumes zuordnen.

**Definition 4.4:** Sei  $T$  ein Baum. Wir bezeichnen die Knotenmenge von  $T$  ebenfalls mit  $T$ . Eine *Knotenmarkierung* ist eine Abbildung

$$\mu: T \rightarrow D$$

für irgendeinen geordneten Wertebereich  $D$ .

**Definition 4.5:** Ein knotenmarkierter binärer Baum  $T$  heißt *binärer Suchbaum* genau dann, wenn für jeden Teilbaum  $T$  von  $T$ ,  $T = (T_l, y, T_r)$ , gilt:

$$\begin{aligned} \forall x \in T_l: \mu(x) &< \mu(y) \\ \forall z \in T_r: \mu(z) &> \mu(y) \end{aligned}$$

Das heißt, alle Schlüssel im linken Teilbaum sind kleiner als der Schlüssel in der Wurzel, alle Schlüssel im rechten Teilbaum sind größer.

**Beispiel 4.6:** Der in Abbildung 4.4 gezeigte Baum entsteht, wenn lexikographische Ordnung benutzt wird und die Monatsnamen in ihrer natürlichen Reihenfolge eingefügt werden.

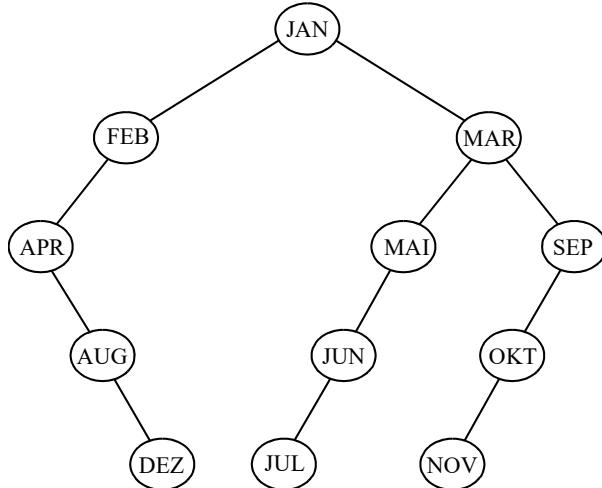


Abbildung 4.4: Monatsnamen im binären Suchbaum

Bei der umgekehrten Einfügereihenfolge entsteht der in Abbildung 4.5 gezeigte Baum.

Übrigens liefert ein *inorder*-Durchlauf die eingetragenen Elemente in Sortierreihenfolge, also hier in alphabetischer Reihenfolge.  $\square$

Zur Darstellung der Baumstruktur benutzen wir folgende Deklarationen:

```

type tree =  $\uparrow$ node;
node = record key : elem;
           left, right :  $\uparrow$ node
         end
  
```

Die entsprechenden Klassendefinitionen kennen wir bereits aus Abschnitt 3.5:

```

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

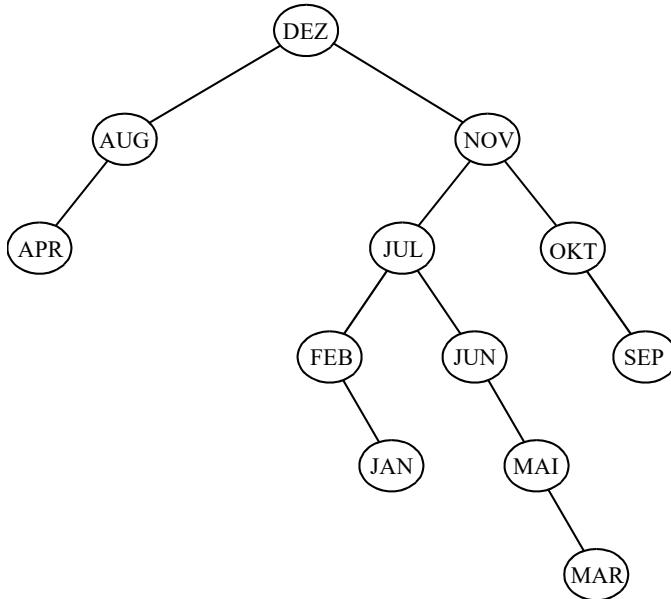


Abbildung 4.5: Binärer Suchbaum bei umgekehrter Einfügereihenfolge

```

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

Nun werden wir die Dictionary-Operationen *member*, *insert* und *delete* als Methoden der Klasse *Node* implementieren. Eine Methode, die überprüft, ob ein gegebenes Element im Baum vorkommt, kann wie folgt formuliert werden:

```

boolean member (Elem x)
{
    if(x isEqual(key))
        return true;
    else
        if(x isLess(key))
            if(left == null) return false; else return left.member(x);
            else // x > key
                if(right == null) return false; else return right.member(x);
}
  
```

Auf der Basis dieses Verfahrens können die Algorithmen zum Einfügen und Löschen formuliert werden:

```
algorithm insert (t, x)
{füge ein neues Element x in den Baum t ein}
suche nach x im Baum t;
if x nicht gefunden
then sei p mit p = nil der Zeiger, an dem die Suche erfolglos endete; erzeuge einen
neuen Knoten mit Eintrag x und lass p darauf zeigen.
end if.
```

Diese Einfügestrategie ist in der Methode *insert* realisiert:

```
public Node insert(Elem x)
{
    if(x isEqual(key))
        return this;
    else
    {
        if(x.isLess(key))
            if(left == null)
            {
                left = new Node(null, x, null);
                return left;
            }
            else return left.insert(x);
        else
            if(right == null)
            {
                right = new Node(null, x, null);
                return right;
            }
            else return right.insert(x);
    }
}
```

In dieser Implementierung liefert die Methode *insert* als Rückgabewert einen Zeiger auf den (neuen oder bereits vorhandenen) Knoten, der das Element *x* enthält.

Das Entfernen eines Elementes ist etwas komplexer, da auch Schlüssel in inneren Knoten betroffen sein können und man die Suchbaumstruktur aufrecht erhalten muss:

```

algorithm delete ( $t, x$ )
{lösche Element  $x$  aus Baum  $t$ }
suche nach  $x$  im Baum  $t$ ;
if  $x$  gefunden im Knoten  $p$ 
then if  $p$  ist Blatt
    then  $p$  entfernen; Zeiger auf  $p$  auf  $nil$  setzen
    else if  $p$  hat nur einen Sohn
        then  $p$  entfernen; Zeiger auf  $p$  auf  $p$ 's Sohn zeigen lassen
        else ( $p$  hat zwei Söhne) in  $p$ 's Teilbaum den Knoten  $q$  bestimmen, der
            das kleinste Element  $y > x$  enthält; im Knoten  $p$  Schlüssel  $x$  durch  $y$ 
            ersetzen; den Schlüssel  $y$  aus dem Teilbaum mit Wurzel  $q$  entfernen
    end if
end if
end if.

```

**Beispiel 4.7:** Wir betrachten in Abbildung 4.6 eine Folge von Löschoperationen, die jeweils an den Pfeilen notiert sind. Nach dem Löschen des Elements 14 wird der Teilbaum mit der Wurzel 33 an der Stelle angehängt, an der sich vorher das Element 14 befand. Das Löschen des Elements 19 ist eine Folgeoperation des Löschens der Wurzel 7, da 19 das kleinste Element mit  $x > 7$  im Baum ist.

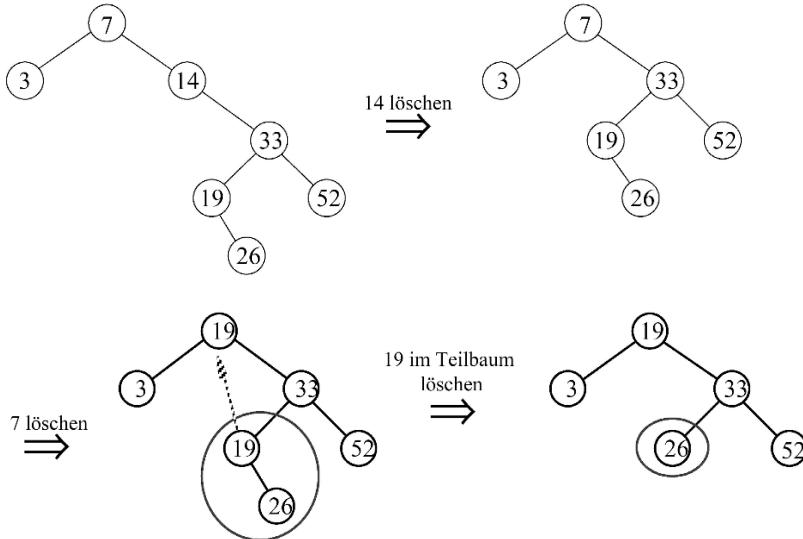


Abbildung 4.6: Löschen im binären Suchbaum

□

Um den Algorithmus *delete* zu realisieren, ist es nützlich, eine Prozedur *deletemin* zu haben, die den minimalen Schlüsselwert in einem nichtleeren Baum ermittelt und gleichzeitig diesen Wert löscht.

In der Implementierung von *deletemin* stehen wir vor dem Problem, dass im Vaterknoten des zu löschenen Elementes der Zeiger auf dieses Element auf *null* gesetzt werden muss, *nachdem* wir es im rekursiven Abstieg gefunden haben. Dazu verwenden wir einen kleinen Trick: wenn wir den zu löschenen Knoten erreicht haben, setzen wir sein *key*-Attribut auf *null*. Wieder zurück in der nächsthöheren Aufrufebene überprüfen wir, ob der *key*-Wert des Sohnes auf *null* steht. In diesem Fall setzen wir den Zeiger auf diesen Sohn um.

```
private Elem deletemin()
{
    ELEM result;
    if (left == null) // der aktuelle Knoten enthält das minimale Element
    {
        result = key;
        key = null;
    }
    else
    {
        result = left.deletemin();
        if (left.key == null) left = left.right;
    }
    return result;
}
```

Auf dieser Grundlage kann man dann relativ einfach die Methode *delete* formulieren. Sie löscht den Knoten mit dem als Parameter übergebenen Element aus dem Baum. Wie schon bei *deletemin* markieren wir auch hier den zu löschenen Knoten, indem wir sein *key*-Attribut auf *null* setzen. Die Methode *delete* liefert die neue Wurzel des Baumes zurück. Diese unterscheidet sich genau dann von der bisherigen Wurzel, wenn das zu löschende Element in der Wurzel lag und diese weniger als zwei Söhne hatte.

```
public Node delete(ELEM x)
{
    if(x.IsLess(key))
    {
        if(left != null) left = left.delete(x);
        return this;
    }
}
```

```

else if(key.IsLess(x))
{
    if(right != null) right = right.delete(x);
    return this;
}
else // key == x
{
    if((left == null) && (right == null)) return null;
    else if(left == null) return right;
    else if(right == null) return left;
    else // der aktuelle Knoten hat zwei Söhne
    {
        key = right.deletemin();
        if(right.key == null) right = right.right;
        return this;
    }
}
}
}

```

Die meisten Methoden der Klasse *Tree* rufen lediglich die gleichnamigen Methoden des Attributes *root* (vom Typ *Node*) auf und reichen deren Rückgabewert durch. Die Methode *Tree.delete* muss allerdings unbedingt das Attribut *root* auf den Rückgabewert des Aufrufes *root.delete()* umsetzen, um eine eventuelle Änderung der Wurzel zu berücksichtigen:

```

public void delete(Elem x)
{
    root = root.delete(x);
}

```

Alle drei Algorithmen *insert*, *delete* und *member* folgen jeweils einem einzigen Pfad im Baum von der Wurzel zu einem Blatt oder inneren Knoten; der Aufwand ist proportional zur Länge dieses Pfades. Wir haben in Abschnitt 3.5 schon gesehen, dass die maximale Pfadlänge  $O(\log n)$  in einem balancierten und  $O(n)$  in einem degenerierten Baum betragen kann. Ein degenerierter binärer Suchbaum entsteht insbesondere, wenn die Folge einzufügender Schlüssel bereits sortiert ist. So entsteht z. B. aus der Einfügereihenfolge

3 7 19 22 40 51

beginnend mit dem leeren Baum ein zur linearen Liste entarteter Baum (Abbildung 4.7). In diesem Fall ist der Aufwand für den Aufbau des gesamten Baumes

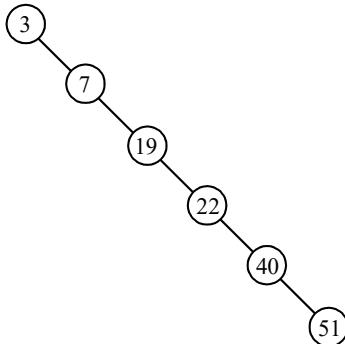


Abbildung 4.7: Entarteter binärer Suchbaum

$$1 + 2 + \dots + n = \frac{n(n+1)}{2} = O(n^2),$$

die durchschnittlichen Kosten für eine Einfügung sind  $O(n)$ . Auf einem derart degenerierten Baum brauchen alle drei Operationen  $O(n)$  Zeit. Das worst-case-Verhalten von binären Suchbäumen ist also schlecht. Wie steht es mit dem Durchschnittsverhalten?

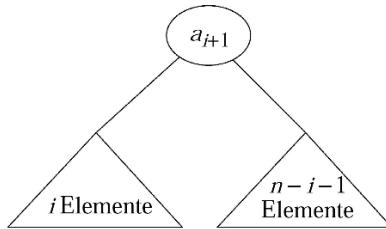
### Durchschnittsanalyse für binäre Suchbäume

Wir fragen nach der durchschnittlichen Anzahl von Knoten eines Pfades (der Einfachheit halber nennen wir dies im Folgenden die mittlere Pfadlänge, während nach Definition die Pfadlänge um 1 niedriger ist als die Knotenzahl) in einem “durchschnittlichen” binären Suchbaum. Ein durchschnittlicher Suchbaum sei gemäß folgenden Annahmen definiert:

1. Er sei nur durch Einfügungen entstanden.
2. In Bezug auf die Einfügereihenfolge seien alle Permutationen der Menge der gespeicherten Schlüssel  $\{a_1, \dots, a_n\}$  gleich wahrscheinlich.

Sei  $a_1 \dots a_n$  die sortierte Folge der Schlüssel  $\{a_1, \dots, a_n\}$ . Bezeichne  $P(n)$  die gesuchte mittlere Pfadlänge in einem durchschnittlichen Baum mit  $n$  Schlüsseln.

Für die Wahl des ersten Elementes  $b_1$  bezüglich der Einfügereihenfolge  $b_1 \dots b_n$  sind alle Elemente aus  $a_1 \dots a_n$  gleich wahrscheinlich. Sei  $a_j = a_{i+1}$  das erste gewählte Element. Dann hat der entstehende Baum die in Abbildung 4.8 gezeigte Gestalt.

Abbildung 4.8: Binärer Suchbaum mit  $n$  Elementen

Der linke Teilbaum wird ein “zufälliger” Baum sein mit Schlüsseln  $\{a_1, \dots, a_i\}$ , der rechte ein zufälliger Baum mit Schlüsseln  $\{a_{i+2}, \dots, a_n\}$ . Die mittlere Pfadlänge in *diesem* Baum ist

$$\frac{i}{n} \cdot (P(i)+1) + \frac{n-i-1}{n} \cdot (P(n-i-1)+1) + \frac{1}{n} \cdot 1$$

Im ersten Term beschreibt  $P(i)+1$  die mittlere Pfadlänge zu Schlüsseln des linken Teilbaumes; da dieser Teilbaum  $i$  Schlüssel enthält, geht diese Pfadlänge mit Gewicht  $i/n$  in die Mittelwertbildung für den Gesamtbaum ein. Der zweite Term beschreibt analog den Beitrag des rechten Teilbaumes, der letzte Term den Beitrag der Wurzel. Gemittelt über alle  $n$  möglichen Wahlen von  $a_{i+1}$  ergibt das (also  $0 \leq i \leq n-1$ ):

$$P(n) = \frac{1}{n^2} \cdot \sum_{i=0}^{n-1} \left[ i \cdot (P(i)+1) + (n-i-1) \cdot (P(n-i-1)+1) + 1 \right] \quad (1)$$

Für die folgenden Rechnungen mit Summenformeln sehen Sie sich besser erst den Abschnitt Grundlagen III des Anhangs “Mathematische Grundlagen” an.

Wegen

$$\sum_{i=0}^{n-1} (n-i-1) \cdot (P(n-i-1)+1) = \sum_{k=0}^{n-1} k \cdot (P(k)+1)$$

kann man die beiden wesentlichen Summanden in (1) zusammenfassen:

$$= \frac{1}{n^2} \cdot \left( 2 \cdot \sum_{i=0}^{n-1} i \cdot (P(i)+1) + \sum_{i=0}^{n-1} 1 \right)$$

$$\begin{aligned}
 &= \frac{1}{n^2} \cdot \left( 2 \cdot \sum_{i=0}^{n-1} i \cdot P(i) + 2 \cdot \sum_{i=0}^{n-1} i + n \right) \\
 &= \frac{1}{n^2} \cdot \left( 2 \cdot \sum_{i=0}^{n-1} i \cdot P(i) + 2 \cdot \frac{(n-1) \cdot n}{2} + n \right) \\
 &= 1 + \frac{2}{n^2} \cdot \sum_{i=0}^{n-1} i \cdot P(i)
 \end{aligned}$$

Wir erhalten also eine Rekursionsgleichung, in der der Wert für  $P(n)$  auf eine Summe aller Werte von  $P(i)$  mit  $i < n$  zurückgreift. Das sieht ziemlich unerfreulich aus. Vielleicht hilft es, die Summenbildung in der Rekursionsgleichung durch eine neue Variable auszudrücken. Definiere

$$S_n := \sum_{i=0}^n i \cdot P(i) \quad (2)$$

Dann ist

$$P(n) = 1 + \frac{2}{n^2} \cdot S_{n-1} \quad (3)$$

Nun gilt

$$\begin{aligned}
 S_n &= n \cdot P(n) + S_{n-1} \quad \text{wegen (2)} \\
 &= n + \frac{2}{n} \cdot S_{n-1} + S_{n-1} \quad \text{wegen (3)}
 \end{aligned}$$

Wir erhalten also für  $S_n$  die Rekursionsgleichung mit Anfangswerten:

$$S_n = \frac{n+2}{n} \cdot S_{n-1} + n$$

$$S_1 = 1$$

$$S_0 = 0$$

Diese neue Rekursionsgleichung sieht schon etwas angenehmer aus. Wir können sie mit Techniken lösen, die in Grundlagen V eingeführt werden.

Die verallgemeinerte Form dieser Rekursionsgleichung ist

$$a_n S_n = b_n S_{n-1} + c_n \left( \text{hier: } a_n = 1, b_n = \frac{n+2}{n}, c_n = n \right)$$

Man setze

$$U_n := s_n a_n S_n$$

und wähle dazu

$$s_n := \frac{n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2}{(n+2) \cdot (n+1) \cdot n \cdot \dots \cdot 4} = \frac{2 \cdot 3}{(n+2) \cdot (n+1)} \text{ für } n \geq 2$$

$$s_1 = 1$$

Dann gilt

$$\begin{aligned} U_n &= s_1 b_1 S_0 + \sum_{i=1}^n \frac{2 \cdot 3 \cdot i}{(i+1) \cdot (i+2)} \\ &= 6 \cdot \sum_{i=1}^n \frac{i}{(i+1) \cdot (i+2)} \quad \text{wegen } S_0 = 0 \end{aligned}$$

Ausgeschrieben sieht diese Summe so aus:

$$\begin{aligned} &\sum_{i=1}^n \frac{i}{(i+1) \cdot (i+2)} \\ &= \frac{1}{2 \cdot 3} + \frac{2}{3 \cdot 4} + \frac{3}{4 \cdot 5} + \dots + \frac{n}{(n+1) \cdot (n+2)} \\ &\leq 1 \cdot \frac{1}{3} + 1 \cdot \frac{1}{4} + 1 \cdot \frac{1}{5} + \dots + 1 \cdot \frac{1}{n+2} \\ &= H_{n+2} - \frac{3}{2} \quad (\text{siehe Grundlagen IV}) \end{aligned}$$

Diesen Wert kann man einsetzen, um schließlich  $P(n)$  zu erhalten:

$$\begin{aligned}
 U_n &\leq 6 \cdot \left( H_{n+2} - \frac{3}{2} \right) \\
 S_n &= \frac{U_n}{S_n} \leq (n+2) \cdot (n+1) \cdot \left( H_{n+2} - \frac{3}{2} \right) \\
 P(n) &= 1 + \frac{2}{n^2} S_{n-1} \\
 &\leq 1 + \frac{2 \cdot (n^2 + n)}{n^2} H_{n+1} - \frac{3 \cdot (n^2 + n)}{n^2}
 \end{aligned}$$

Für große Werte von  $n$  nähert sich dies dem Wert

$$\lim_{n \rightarrow \infty} P(n) \leq 1 + 2 \cdot \ln(n+1) + 2\gamma - 3$$

Wegen

$$\log x = \frac{\ln x}{\ln 2}$$

ergibt sich schließlich

$$P(n) \approx 2 \cdot \ln 2 \cdot \log n + \text{const}$$

Die Proportionalitätskonstante  $2 \cdot \ln 2$  hat etwa den Wert 1.386.

Das ist also der erwartete Suchaufwand. Die mittlere Pfadlänge zu einem Blatt, die beim Einfügen und Löschen von Bedeutung ist, unterscheidet sich nicht wesentlich. Damit ist in einem so erzeugten "zufälligen" Baum der erwartete oder durchschnittliche Aufwand für alle drei Dictionary-Operationen  $O(\log n)$ .

Diese Aussage gilt allerdings nicht mehr, wenn sehr lange, gemischte Folgen von Einfüge- und Löschoperationen betrachtet werden. Dadurch, dass im Löschalgorithmus beim Entfernen des Schlüssels eines inneren Knotens jeweils der Nachfolgerschlüssel die freiwerdende Position einnimmt, wandert (z. B. in der Wurzel) der Schlüsselwert allmählich nach oben und der Baum wird mit der Zeit linkslastig. Dies konnte analytisch in [Culberson 1985] gezeigt werden: Wenn in einem zufällig erzeugten Suchbaum mit  $n$  Schlüsseln eine Folge von mindestens  $n^2$  Update-Operationen ausgeführt wird, so hat der dann entstandene Baum eine erwartete Pfadlänge von  $\Theta(\sqrt{n})$  (ein Update besteht aus dem Einfügen und dem Entfernen je eines zufällig gewählten Schlüssels). Als Folgerung daraus sollte man die Löschprozedur so implementieren, dass jeweils zufällig aus-

gewählt wird, ob der Vorgänger oder der Nachfolger des zu löschenen Schlüssels die freie Position einnimmt.

#### 4.2.4 AVL-Bäume

Obwohl binäre Suchbäume im Durchschnitt gutes Verhalten zeigen, bleibt doch der nagende Zweifel, ob bei einer gegebenen Anwendung nicht der sehr schlechte worst case eintritt (z. B. weil die Elemente sortiert eintreffen). Wir betrachten nun eine Datenstruktur, die mit solchen Unsicherheiten radikal Schluss macht und auch im schlimmsten Fall eine Laufzeit von  $O(\log n)$  für alle drei Dictionary-Operationen garantiert: den *AVL-Baum* (benannt nach den Erfindern Adelson-Velskii und Landis [1962]). Es gibt verschiedene Arten *balancierter Suchbäume*, der AVL-Baum ist ein Vertreter.

Die Grundidee besteht darin, eine *Strukturinvariante* für einen binären Suchbaum zu formulieren und diese unter Updates (also Einfügen, Löschen) aufrecht zu erhalten. Die Strukturinvariante ist eine Abschwächung des Balanciertheitskriteriums vollständig balancierter Bäume, in denen ja lediglich der letzte Level nur teilweise besetzt sein durfte. Sie lautet:

**Definition 4.8:** Ein *AVL-Baum* ist ein binärer Suchbaum, in dem sich für jeden Knoten die Höhen seiner zwei Teilbäume höchstens um 1 unterscheiden.

Falls durch eine Einfüge- oder Löschoperation diese Strukturinvariante verletzt wird, so muss sie durch eine *Rebalancieroperation* wiederhergestellt werden. Diese Operationen manipulieren jeweils einige Knoten in der Nähe der Wurzel eines *aus der Balance geratenen* Teilbaumes, um den einen Teilbaum etwas anzuheben, den anderen abzusenken, und so eine Angleichung der Höhen zu erreichen.

#### Updates

Eine Einfüge- oder Löschoperation wird zunächst genauso ausgeführt wie in einem gewöhnlichen binären Suchbaum. Eine *Einfügung* fügt in jedem Fall dem Baum ein neues Blatt hinzu. Dieses Blatt gehört zu allen Teilbäumen von Knoten, die auf dem Pfad von der Wurzel zu diesem Blatt liegen; durch die Einfügung können die Höhen all dieser Teilbäume *um 1 wachsen*. Sei die *aktuelle Position* im Baum nach dieser Einfügung *das neu erzeugte Blatt*.

Eine *Löschoperation* vernichtet irgendwo im Baum einen Knoten; das kann ein Blatt oder ein innerer Knoten sein, und es kann der Knoten des eigentlich zu entfernenden Elementes sein oder des Folgewertes. Sei die *aktuelle Position* nach einer Löschopera-

tion der *Vater des gelöschten Knotens*. Genau die Teilbäume auf dem Pfad von der Wurzel zu diesem Vaterknoten sind betroffen; ihre Höhe könnte sich *um 1 verringert* haben.

Für jede der beiden Update-Operationen schließt sich nun eine *Rebalancierphase* an: Man läuft von der aktuellen Position aus den Pfad zur Wurzel zurück. In jedem Knoten wird dessen Balance überprüft; falls der Knoten aus der Balance geraten ist, wird das durch eine der im Folgenden beschriebenen Rebalancieroperationen korrigiert. Wir können also davon ausgehen, dass alle Teilbäume unterhalb des gerade betrachteten (aus der Balance geratenen) Knotens selbst balanciert sind.

### Rebalancieren

**Fall (a):** Wir nehmen an, der gerade betrachtete Knoten (Teilbaum) sei durch eine *Einfügung* aus der Balance geraten. Also einer seiner Teilbäume ist um 1 gewachsen. O. B. d. A.<sup>2</sup> sei der rechte Teilbaum um 2 höher als der linke (bzw. tiefer, wenn wir das untere Ende betrachten). Der andere Fall (linker Teilbaum tiefer) ist symmetrisch und kann analog behandelt werden.

Man kann noch einmal zwei Fälle (a1) und (a2) unterscheiden, je nachdem, ob innerhalb des rechten Teilbaums der rechte (*C*) oder der linke Teilbaum (*B*) (siehe Abbildung 4.9) gewachsen ist.

#### Fall (a1):

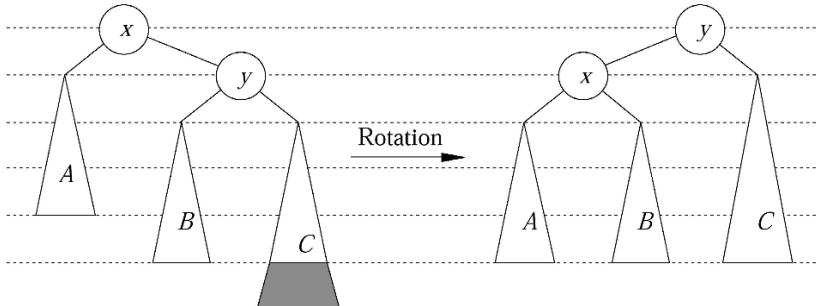


Abbildung 4.9: Anschließend Zustand ok, Höhe unverändert

In diesem Fall reicht eine *einfache Rotation* gegen den Uhrzeigersinn aus, um die Balance wiederherzustellen. Die Ordnungseigenschaften bleiben bei einer solchen

---

2. Das heißt: "Ohne Beschränkung der Allgemeinheit", eine bei Mathematikern beliebte Abkürzung.

Operation erhalten, wie man durch inorder-Auflistung der Komponenten leicht überprüfen kann.

Betrachten wir den anderen möglichen Fall:

**Fall (a2):**

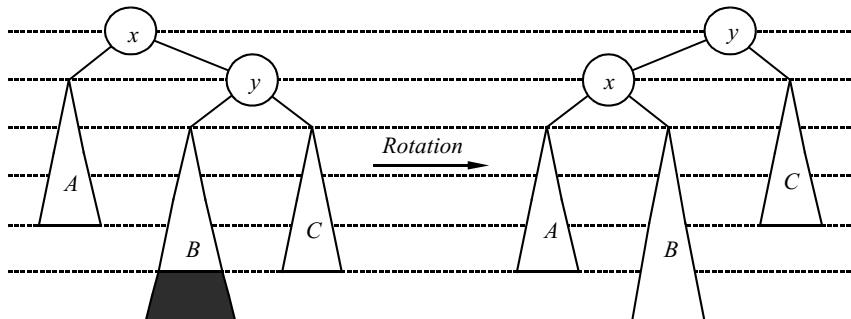


Abbildung 4.10: Anschließend Zustand nicht ok, Rotation reicht nicht

In einem solchen Fall kann eine Operation wie oben die Balance offensichtlich nicht wiederherstellen. Wir sehen uns Teilbaum  $B$  im Fall (a2) genauer an:

**Fall (a2.1):**

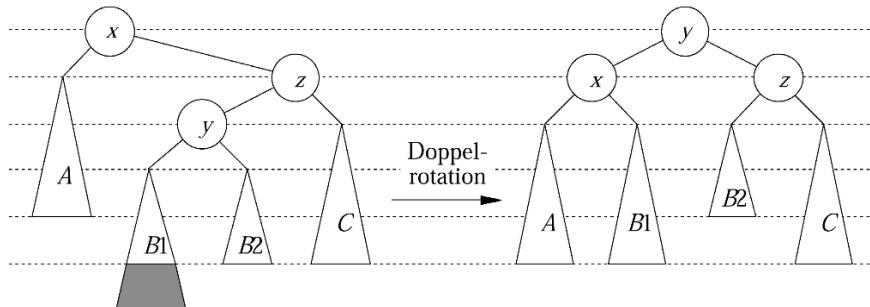


Abbildung 4.11: Anschließend Zustand ok, Höhe unverändert

Eine *Doppelrotation*, in der zunächst nur der Teilbaum mit der Wurzel  $z$  im Uhrzeigersinn rotiert wird und erst danach der gesamte Baum (diesmal gegen den Uhrzeigersinn) rotiert wird, stellt die Balance wieder her. Auch in dem anderen Fall, dass  $B2$  der größere Teilbaum ist, funktioniert diese Methode:

**Fall (a2.2):**

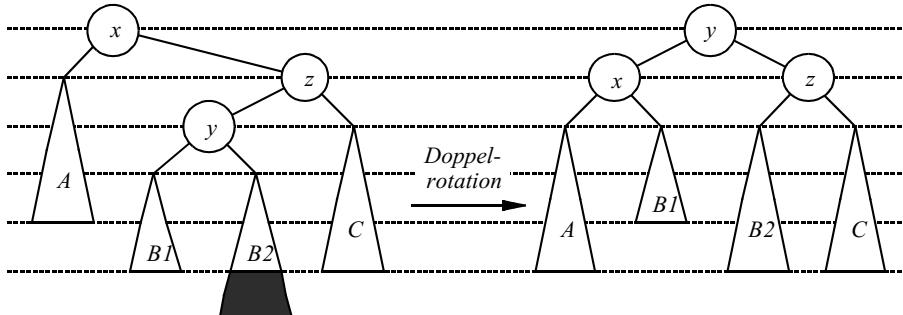


Abbildung 4.12: Anschließend Zustand ok, Höhe unverändert

**Satz 4.9:** Nach einer Einfügung genügt eine einzige Rotation oder Doppelrotation, um die Strukturinvariante des AVL-Baumes wiederherzustellen.

**Beweis:** Eine Rotation oder Doppelrotation im ersten aus der Balance geratenen Knoten  $p$  auf dem Pfad von der aktuellen Position zur Wurzel sorgt dafür, dass der Teilbaum mit Wurzel  $p$  die gleiche Höhe hat wie vor der Einfügung. Also haben auch alle Teilbäume, deren Wurzeln Vorfahren von  $p$  sind, die gleiche Höhe und keiner dieser Knoten kann jetzt noch unbalanciert sein.  $\square$

**Fall (b):** Wir nehmen nun an, dass der gerade betrachtete Knoten durch eine Löschoperation aus der Balance geraten sei. O. B. d. A. sei der rechte Teilbaum tiefer, und zwar um 2. Hier lassen sich drei Fälle unterscheiden (nach den möglichen Formen des tieferen Teilbaumes):

Fall (b1):

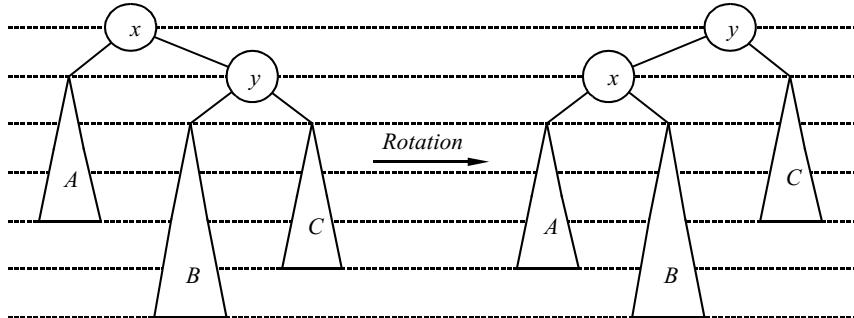


Abbildung 4.13: Anschließend Zustand nicht ok

Fall (b2):

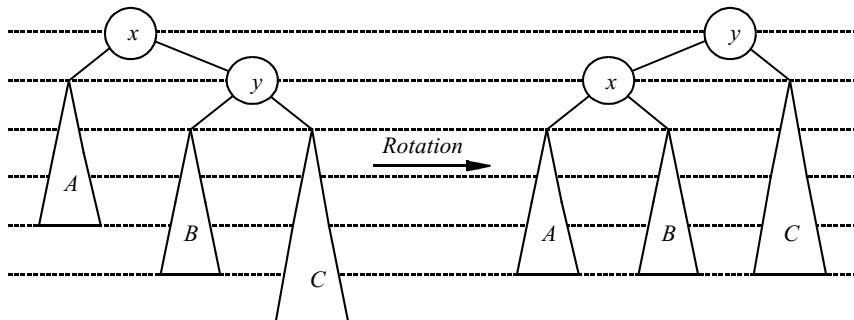


Abbildung 4.14: Anschließend Zustand ok, Höhe um 1 vermindert

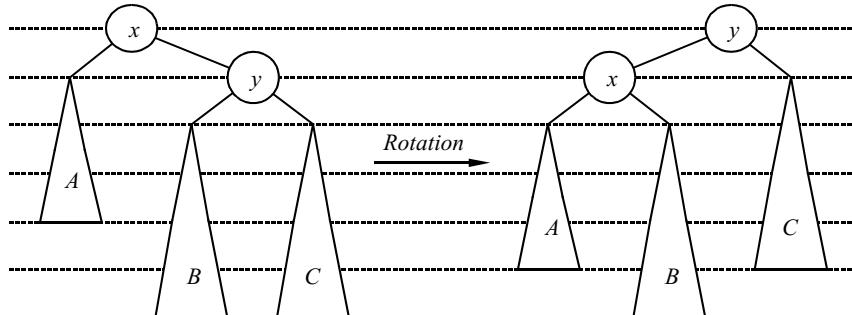
**Fall (b3):**

Abbildung 4.15: Anschließend Zustand ok, Höhe unverändert

Wie man sieht, kann im zweiten und dritten Fall die Balance durch eine Einfachrotation analog zum Einfügen wiederhergestellt werden. Im ersten Fall muss der “kritische” Teilbaum  $B$  genauer untersucht werden, dabei lassen sich wieder die drei Fälle unterscheiden:

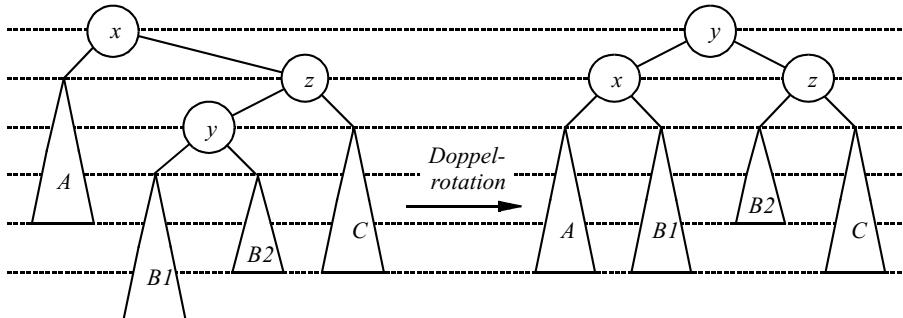
**Fall (b1.1):**

Abbildung 4.16: Anschließend Zustand ok, Höhe um 1 vermindert

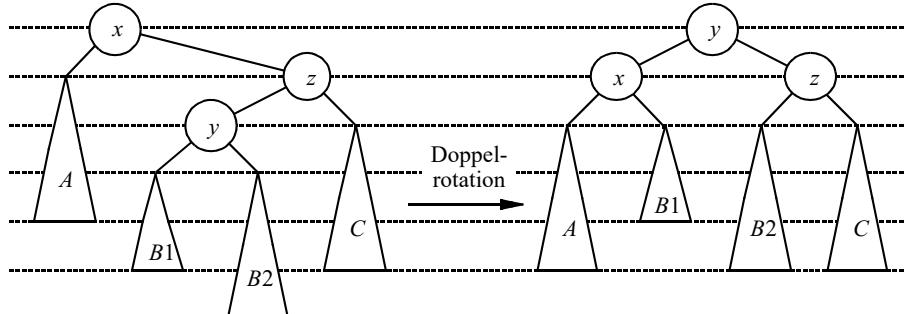
**Fall (b1.2):**

Abbildung 4.17: Anschließend Zustand ok, Höhe um 1 vermindert

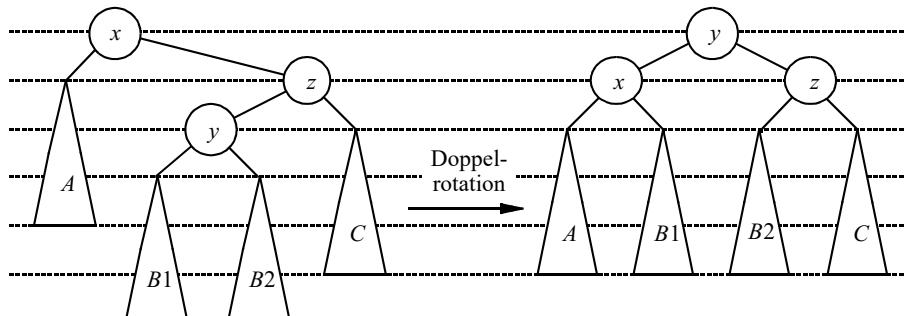
**Fall (b1.3):**

Abbildung 4.18: Anschließend Zustand ok, Höhe um 1 vermindert

In diesen Fällen kann stets die Balance durch eine Doppelrotation wiederhergestellt werden.

**Satz 4.10:** Ein durch eine Löschoperation aus der Balance geratener Teilbaum kann durch eine Rotation oder Doppelrotation wieder ausgeglichen werden. Es kann aber notwendig sein, auch Vorgängerteilbäume bis hin zur Wurzel zu rebalancieren.

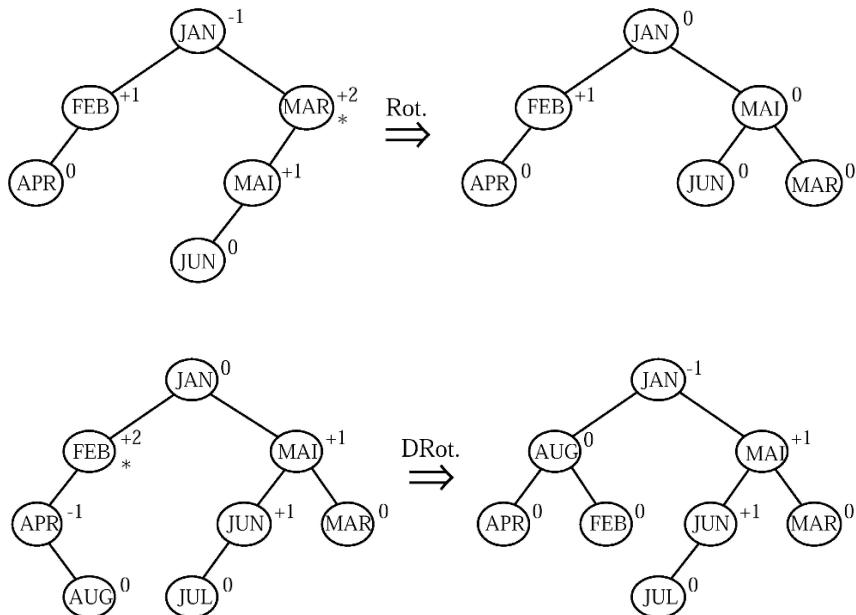
**Beweis:** Ergibt sich aus der obigen Fall-Analyse. Dadurch, dass die Höhe des betroffenen Teilbaumes durch die Ausgleichsoperation gesunken ist, können auch Vorgänger unbalanciert sein.  $\square$

Damit sind die Update-Algorithmen (bis auf Implementierungsdetails) komplett. In der Implementierung wird man in jedem Knoten des Suchbaumes zusätzlich die *Balance* (oder die Höhe) abspeichern und unter Updates aufrecht erhalten, die definiert ist als

$$\text{balance}(p) = \text{height}(p.\text{left}) - \text{height}(p.\text{right}),$$

die also im balancierten Fall die Werte  $\{-1, 0, +1\}$  annehmen kann und bei einem unbalancierten Knoten auch noch  $-2$  und  $+2$ .

**Beispiel 4.11:** Wir fügen die Monatsnamen in einen anfangs leeren AVL-Baum ein. Wann immer Rebalancieren nötig ist, wird der Baum neu gezeichnet. Die Balance wird im Beispiel durch  $“+n”$  oder  $“-n”$  an jedem Knoten eingetragen. Knoten, die die Strukturinvariante verletzen und damit ein Rebalancieren auslösen, werden mit  $“*”$  gekennzeichnet.



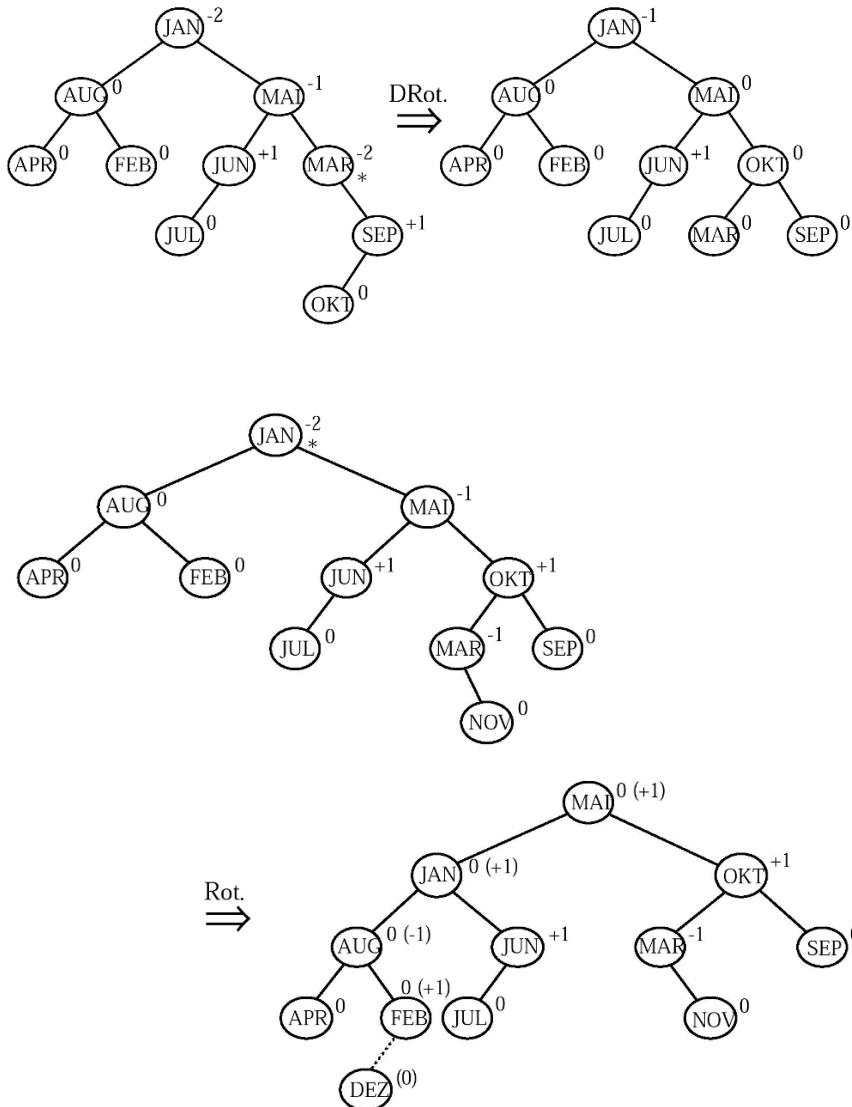


Abbildung 4.19: Erzeugen eines AVL-Baumes der Monatsnamen

Man wendet also jeweils eine Rotation an, wenn bei einem aus der Balance geratenen Knoten ein äußerer Teilbaum "am tiefsten hängt" und eine Doppelrotation, wenn der tiefste der mittlere Teilbaum ist.  $\square$

**Selbsttestaufgabe 4.3:** Ermitteln Sie den AVL-Baum, der beim Einfügen der Zahlen 0...15 in einen anfangs leeren Baum entsteht.  $\square$

Eine Implementierung des AVL-Baumes lässt sich z. B. auf der Basis einer erweiterten Definition der Klasse *Node* vornehmen:

```
class Node
{
    ELEM key;
    int height;
    Node left, right;
    ...
    int balance() {...};      /* liefert für einen gegebenen AVL-Baum seine
                               Balance im Bereich -2..+2 zurück */
    Node insert(ELEM e) {...}; /* fügt e in den AVL-Baum t ein und liefert eine
                               balancierte Version von t zurück - rekursive Methode */
}
```

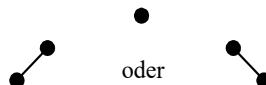
**Selbsttestaufgabe 4.4:** Formulieren Sie die Methode *insert* für den AVL-Baum.  $\square$

Für die Analyse des AVL-Baumes beobachtet man, dass alle drei Algorithmen (Suchen, Einfügen, Entfernen) jeweils einem Pfad von der Wurzel zu einem Blatt folgen und dann evtl. den Pfad vom Blatt zurück zur Wurzel laufen und dabei Rotationen oder Doppelrotationen vornehmen. Jede solche Operation benötigt nur  $O(1)$  Zeit; der Gesamtaufwand für jede der drei Dictionary-Operationen ist daher  $O(h)$ , wobei  $h$  die Höhe des AVL-Baumes ist. Wie hoch wird ein AVL-Baum mit  $n$  Knoten im schlimmsten Fall?

Sei  $N(h)$  die minimale Anzahl von Knoten in einem AVL-Baum der Höhe  $h$ .

$$N(0) = 1$$

$$N(1) = 2$$



oder

Allgemein erhält man einen minimal gefüllten AVL-Baum der Höhe  $h$ , indem man einen Wurzelknoten mit jeweils einem Baum der Höhe  $h-1$  und einem Baum der Höhe  $h-2$  kombiniert, also gilt:

$$N(h) = 1 + N(h-1) + N(h-2)$$

Das erinnert an die Definition der Fibonacci-Zahlen, die in Grundlagen VI eingeführt werden, nämlich  $F_0 = 0$ ,  $F_1 = 1$ ,  $F_k = F_{k-2} + F_{k-1}$ . Wir vergleichen die Werte:

$k$	0	1	2	3	4	5	6	7	8	9	10	11	12
$F_k$	0	1	1	2	3	5	8	13	21	34	55	89	144
$N(k)$	1	2	4	7	12	20	33	54	88	143			

**Hypothese:**  $N(k) = F_{k+3} - 1$

**Beweis:** Den Induktionsanfang findet man in der obigen Tabelle. Der Induktionsschluss ist:

$$\begin{aligned} N(k+1) &= 1 + N(k) + N(k-1) \\ &= 1 + F_{k+3} - 1 + F_{k+2} - 1 \\ &= F_{k+4} - 1 \end{aligned}$$

□

Also hat ein AVL-Baum der Höhe  $h$  mindestens  $F_{h+3} - 1$  Knoten. Das heißt, ein AVL-Baum mit  $n$  Knoten hat höchstens die Höhe  $h$ , die bestimmt ist durch:

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

Beachten Sie, dass mit diesen Ungleichungen der Wert von  $h$  für ein gegebenes  $n$  definiert wird. Es genügt nicht zu sagen " $N(h) \leq n$ ", da es viele  $h$  gibt, die das erfüllen.

$$F_{h+3} \leq n + 1$$

$$\frac{1}{\sqrt{5}} \Phi^{h+3} - \frac{1}{2} \leq \frac{1}{\sqrt{5}} (\Phi^{h+3} - \hat{\Phi}^{h+3}) \leq n + 1$$

$$\frac{1}{\sqrt{5}} \Phi^{h+3} \leq n + \frac{3}{2}$$

$$\log_{\Phi} \frac{1}{\sqrt{5}} + h + 3 \leq \log_{\Phi} \left( n + \frac{3}{2} \right)$$

$$\begin{aligned}
 h &\leq \log_{\Phi} n + const \\
 &= \log_{\Phi} 2 \cdot \log_2 n + const \\
 &= \frac{\ln 2}{\ln \Phi} \cdot \log_2 n + const \\
 &= 1.4404 \log_2 n + const
 \end{aligned}$$

Also ist die Höhe  $O(\log n)$  und die Proportionalitätskonstante etwa 1.44, das heißt, ein AVL-Baum ist höchstens um 44% höher als ein vollständig ausgeglichener binärer Suchbaum. Der AVL-Baum realisiert damit alle drei Dictionary-Operationen in  $O(\log n)$  Zeit und braucht  $O(n)$  Speicherplatz.

### 4.3 Priority Queues: Mengen mit INSERT, DELETEMIN

Priority Queues sind Warteschlangen, in die Elemente gemäß einer “Priorität” eingeordnet werden; es wird jeweils das Element mit “höchster” Priorität entnommen. Wir nehmen allerdings an, dass die höchste Priorität im intuitiven Sinne das Element mit dem niedrigsten - gewöhnlich numerischen - Prioritätswert besitzt. Ein Anwendungsbeispiel aus dem täglichen Leben ist etwa der Warterraum einer Krankenhaus-Ambulanz; Patienten in akuter Gefahr werden vorgezogen. In der Informatik spielen Prioritäts-Warteschlangen besonders in Betriebssystemen eine Rolle. Verschiedene Prozesse (laufende Programme) besitzen unterschiedliche Priorität und erhalten entsprechend Ressourcen, z. B. CPU-Zeit.

Aus der Sicht der Verwaltung von Mengen sind die wesentlichen Operationen offensichtlich das Einfügen eines Elementes mit gegebenem Zahlenwert (Priorität) in eine Menge und das Entnehmen des minimalen Elementes. Man muss allerdings beachten, dass mehrere Objekte mit gleicher Priorität in einer Warteschlange vorkommen können, und so haben wir es genau genommen mit *Multimengen* zu tun (auch Multisets, Bags genannt = Mengen mit Duplikaten). Wir notieren Multimengen etwa so:

$$\begin{aligned}
 \{|1, 3, 3, 7, 7, 7, 10|\} &=: M \\
 \{|1, 2, 3\} \cup \{|2, 3, 4\} &= \{|1, 2, 2, 3, 3, 4\} \\
 \{|1, 2, 3, 3\} \setminus \{|3\} &= \{|1, 2, 3\}
 \end{aligned}$$

Die Mengenoperationen haben also eine entsprechend modifizierte Bedeutung. Man kann eine Multimenge auch als eine Menge von Paaren darstellen, wobei ein Paar jeweils den Wert eines Elementes der Multimenge und die Anzahl seines Auftretens angibt. Die obige Multimenge M hätte dann die Darstellung

$$\{(1, 1), (3, 2), (7, 3), (10, 1)\}$$

Notation: Sei  $\mathcal{M}(S)$  die Menge aller endlichen Multimengen, deren Elemente aus  $S$  stammen. Ein Datentyp für Priority Queues lässt sich dann so spezifizieren:

```

algebra pqueue
sorts      pqueue, elem
ops       empty      : pqueue      → pqueue
            isempty    : pqueue      → bool
            insert     : pqueue × elem → pqueue
            deletemin : pqueue      → pqueue × elem
sets      pqueue =  $\mathcal{M}(\text{elem})$ 
functions
            empty      =  $\emptyset$ 
            isempty(p) = ( $p = \emptyset$ )
            insert(p, e) =  $p \cup \{ | e | \}$ 
            deletemin(p) =  $\begin{cases} (p', e) \text{ mit } e = \min(p) \text{ und} \\ \quad p' = p \setminus \{ | e | \} & \text{falls } p \neq \emptyset \\ \text{undefiniert} & \text{sonst} \end{cases}$ 
end pqueue.

```

Dabei ist die Definition einer mehrsortigen Algebra gegenüber der Einleitung leicht erweitert worden, so dass Funktionen auch mehrere Ergebnisse liefern können. Wir nehmen an, dass man in einem Algorithmus oder Programm eine solche Funktion benutzen kann, indem man ihre Ergebnisse einem Tupel von Variablen zuweist, z. B.

$$(q, m) := \text{deletemin}(p)$$

## Implementierung

Priority Queues lassen sich effizient mit zur Darstellung von Multimengen leicht modifizierten *AVL-Bäumen* realisieren; dann können beide Operationen in  $O(\log n)$  Zeit ausgeführt werden. Es gibt aber eine einfachere Implementierung mit *partiell geordneten Bäumen*, die wir uns hier ansehen wollen und die insbesondere Einsatz bei Sortieralgorithmen (*Heapsort*, siehe Abschnitt 5.3) findet.

**Definition 4.12:** Ein *partiell geordneter Baum* ist ein knotenmarkierter binärer Baum  $T$ , in dem für jeden Teilbaum  $T'$  mit Wurzel  $x$  gilt:

$$\forall y \in T': \mu(x) \leq \mu(y).$$

In der Wurzel steht also jeweils das Minimum eines Teilbaums. Analog kann man natürlich auch einen partiell geordneten Baum definieren, in dessen Wurzel das Maximum steht. Ein partiell geordneter Baum wird auch häufig als *Heap* (Haufen) bezeichnet; eine

engere Definition des Begriffes bezeichnet als Heap die Array-Einbettung eines partiell geordneten Baumes (Abschnitt 3.5, Implementierung (b)).

**Beispiel 4.13:** Abbildung 4.20 zeigt einen partiell geordneten Baum, der die Multimenge  $\{4, 6, 6, 7, 10, 10, 12, 13, 13, 19\}$  darstellt.

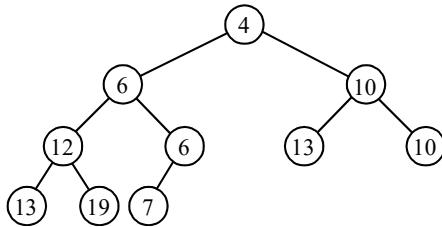


Abbildung 4.20: Partiell geordneter Baum

□

In einem Heap ist die Folge der Knotenmarkierungen auf einem Pfad monoton steigend. Wir betrachten im Folgenden *links-vollständige* partiell geordnete Bäume. Das soll heißen, alle Ebenen bis auf die letzte sind voll besetzt (vollständiger Baum), und auf der letzten Ebene sitzen die Knoten so weit links wie möglich.

Einfügen in einen Heap kann dann mit folgendem Verfahren vorgenommen werden:

```

algorithm insert ( $h, e$ )
  {füge Element  $e$  in den Heap  $h$  ein}
  erzeuge einen neuen Knoten  $q$  mit Eintrag  $e$ ; füge  $q$  auf der ersten freien Position
  der untersten Ebene ein (falls die unterste Ebene voll besetzt ist, beginne eine
  neue Ebene);
  sei  $p$  der Vater von  $q$ ;
  while  $p$  existiert and  $\mu(q) < \mu(p)$  do
    vertausche die Einträge in  $p$  und  $q$ ; setze  $q$  auf  $p$  und  $p$  auf den Vater von  $p$ .
  end while.
  
```

**Beispiel 4.14:** Wenn in den Heap aus dem vorigen Beispiel das neue Element 5 eingefügt wird, dann wird es zunächst ein Sohn des Elements 6, steigt jedoch dann solange im Heap auf, bis es ein Sohn der Wurzel geworden ist (Abbildung 4.21). □

Zu zeigen ist, dass dieses Einfügeverfahren korrekt ist, das heißt, einen partiell geordneten Baum liefert. Dazu betrachten wir eine einzelne Vertauschung, die innerhalb der Schleife vorgenommen wird (Abbildung 4.22).

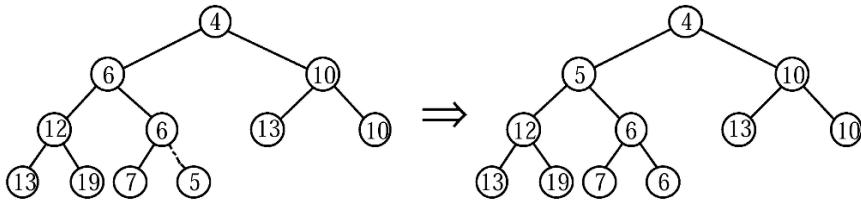


Abbildung 4.21: Einfügen des Elementes 5

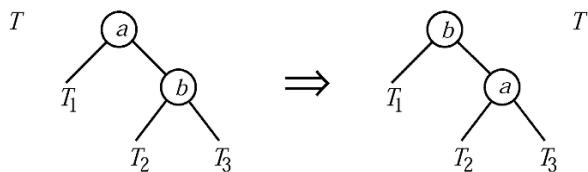


Abbildung 4.22: Vertauschung beim Einfügen

Es gilt  $b < a$ , denn sonst würde  $b$  nicht mit  $a$  vertauscht.  $b$  ist das neu eingefügte Element.  $a$  ist minimal bzgl.  $T_1$ ,  $T_2$  und  $T_3$  (das heißt,  $a$  ist minimal in der Menge aller dort enthaltenen Schlüssel), deshalb ist  $a$  auch minimal bzgl.  $T_1$ ,  $T_2$  und  $T_3$ . Weiter gilt  $b < a$ , deshalb ist  $b$  minimal bzgl.  $T_1$ ,  $T_2$ ,  $T_3$  und  $a$ . Also ist  $T'$  ein partiell geordneter Baum.

Die zweite wichtige Operation auf einem Heap ist das Entnehmen des minimalen Elements. Dies wird folgendermaßen realisiert:

```

algorithm deletemin (h)
{lösche das minimale Element aus dem Heap h und gib es aus}
entnimm der Wurzel ihren Eintrag und gib ihn als Minimum aus; nimm den Ein-
trag der letzten besetzten Position im Baum (lösche diesen Knoten) und setze
ihn in die Wurzel; sei p die Wurzel und seien q, r ihre Söhne;
while q oder r existieren and ( $\mu(p) > \mu(q)$  or  $\mu(p) > \mu(r)$ ) do
    vertausche den Eintrag in p mit dem kleineren Eintrag der beiden Söhne;
    setze p auf den Knoten, mit dem vertauscht wurde, und q, r auf dessen Söhne
end while.

```

Warum ist das korrekt? Wir betrachten wieder eine Vertauschung innerhalb der Schleife.

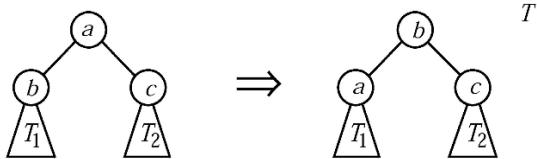


Abbildung 4.23: Vertauschung bei Entnahme des Minimums

$b$  ist minimal bzgl.  $T_1$ ,  $c$  bzgl.  $T_2$ ,  $a$  ist das neu in die Wurzel eingefügte Element. Es gilt:  $b < a$  (sonst würde die Operation nicht durchgeführt) und  $b \leq c$  (Auswahl des kleineren Sohnes). Also ist  $b$  minimal bzgl.  $a$ ,  $c$ ,  $T_1$  und  $T_2$ .  $a$  ist möglicherweise nicht minimal bzgl.  $T_1$ . Das wird aber weiterbehandelt, bis gilt  $a \leq b$  und  $a \leq c$  (bezogen auf einen tiefen Teilbaum), womit dann der ganze Baum wieder partiell geordnet ist.

Beide Operationen folgen einem Pfad im Baum; der Baum ist balanciert. Mit dem in einen Array eingebetteten Heap lassen sich beide Operationen deshalb in  $O(\log n)$  Zeit realisieren. Die Implementierung ist einfach und auch praktisch sehr effizient.

**Selbsttestaufgabe 4.5:** Schreiben Sie die Methoden *insert*, *deletemin* für einen Heap im Array. □

#### 4.4 Partitionen von Mengen mit MERGE, FIND

Wir betrachten einen Datentyp, mit dem sich eine *Partition* einer Menge, also eine Zerlegung der Menge in disjunkte Teilmengen, verwalten lässt. Eine solche Zerlegung entspricht bekanntlich einer *Äquivalenzrelation* auf den Elementen der Grundmenge. Der Datentyp soll nun speziell die Lösung des folgenden Problems unterstützen:

Gegeben sei eine Partition einer Menge  $S$  und eine Folge von “Äquivalenz-Anweisungen” der Form

$$a_1 \equiv b_1, a_2 \equiv b_2, \dots, a_m \equiv b_m$$

für  $a_i, b_i \in S$ . Man verschmelze bei jeder Anweisung  $a_i \equiv b_i$  die Äquivalenzklassen von  $a_i$  und  $b_i$ . Dabei soll zu jeder Zeit die Äquivalenzklasse für ein beliebiges  $x \in S$  effizient ermittelt werden können.

**Beispiel 4.15:** Wir betrachten die Änderungen der Partition  $\{1\} \{2\} \{3\} \{4\} \{5\} \{6\}$  unter einer Folge von Äquivalenzanweisungen:

```

1 ≡ 4
{1, 4} {2} {3} {5} {6}
2 ≡ 5
{1, 4} {2, 5} {3} {6}
2 ≡ 4
{1, 2, 4, 5} {3} {6}

```

□

Eine Datenstruktur, die eine effiziente Lösung dieses Problems erlaubt, hat verschiedene Anwendungen. Eine davon ist die Implementierung bestimmter Graph-Algorithmen, die wir im nächsten Kapitel besprechen.

Die wesentlichen Operationen, die benötigt werden, sind *merge* und *find*. *Merge* verschmilzt zwei Teilmengen, das heißt, bildet die Vereinigung ihrer Elemente (Teilmengen werden auch *Komponenten* genannt). *Find* stellt für ein gegebenes Element fest, zu welcher Komponente es gehört. Ein entsprechender Datentyp lässt sich spezifizieren, wie in Abbildung 4.24 gezeigt.  $F(elem)$  bezeichnet dabei wiederum die Menge aller endlichen Teilmengen von  $elem$ .

## Implementierungen

### (a) Implementierung mit Arrays

Wir betrachten den Spezialfall  $elem = \{1, \dots, n\}$  und  $compname = \{1, \dots, n\}$  und benutzen zwei Arrays der Größe  $n$ . Der erste Array, indiziert mit Komponentennamen, erlaubt uns den Zugriff auf alle Elemente dieser Komponente, er enthält zu jedem Komponentennamen den Namen eines Elementes, das zu dieser Komponente gehört. Der zweite Array, indiziert mit Elementen (oder Elementnamen), erlaubt es, zu einem Element die enthaltende Komponente zu finden. Er speichert außer dem Namen der enthaltenden Komponente noch den Namen eines weiteren Elements dieser Komponente, oder Null, falls bereits alle Elemente dieser Komponente erfasst sind. Auf diese Weise wird eine verkettete Liste innerhalb des Arrays (wie in Abschnitt 3.1.2, Implementierung (d)) gebildet.

```

type elem      = 1..n;
compname = 1..n;

var components = array[compname] of
  record
    ... ; firstelem: 0..n
  end;

```

```

algebra    partition
sorts      partition, compname, elem
ops        empty          :  $\rightarrow \text{partition}$ 
              addcomp       :  $\text{partition} \times \text{compname} \times \text{elem} \rightarrow \text{partition}$ 
              merge         :  $\text{partition} \times \text{compname} \times \text{compname} \rightarrow \text{partition}$ 
              find          :  $\text{partition} \times \text{elem} \rightarrow \text{compname}$ 

sets       Sei  $CN$  eine beliebige unendliche Menge (von ‘‘Komponentennamen’’)
              compname =  $CN$ 
              partition =  $\{(c_i, S_i) \mid n \geq 0, 1 \leq i \leq n, c_i \in CN, S_i \in F(\text{elem})\}$ 
                            $| i \neq j \Rightarrow c_i \neq c_j \wedge S_i \cap S_j = \emptyset\}$ 

functions   empty      =  $\emptyset$ 
              Sei  $p = \{(c_1, S_1), \dots, (c_n, S_n)\}$ ,  $C = \{c_1, \dots, c_n\}$ ,  $S = \bigcup_{i=1}^n S_i$ 
              Sei  $a \in CN \setminus C$ ,  $x \notin S$ . Sonst ist addcomp undefiniert.
              addcomp ( $p, a, x$ ) =  $p \cup \{(a, \{x\})\}$ 
              Seien  $a = c_i, b = c_j \in C, d \in (CN \setminus C) \cup \{a, b\}$ 
              merge ( $p, a, b$ ) =  $(p \setminus \{(a, S_i), (b, S_j)\}) \cup \{(d, S_i \cup S_j)\}$ 
              Sei  $x \in S$ :
              find ( $p, x$ ) =  $a$  so dass  $(a, S') \in p$  und  $x \in S'$ 
end partition.

```

---

Abbildung 4.24: Algebra *partition*

```

var elems      = array[elem] of
                  record
                      comp: compname;
                      nexelem: 0..n
                  end;

```

**Beispiel 4.16:** Die Partition  $\{1, 2, 4, 5\}, \{3\}, \{6\}$  könnte so dargestellt sein, wie in Abbildung 4.25 gezeigt, wobei ein Wert 0 dem Zeiger *nil* entspricht:

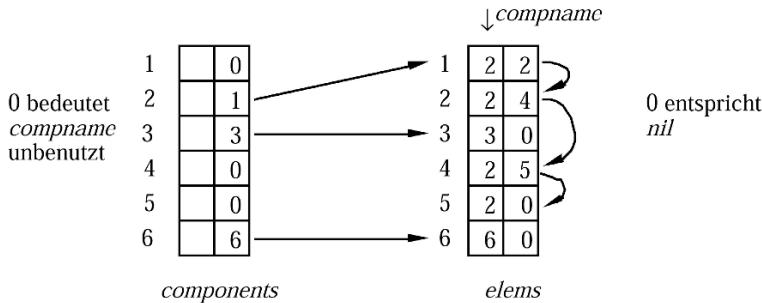


Abbildung 4.25: Darstellung einer Partition

Mit den hier benutzten Komponentennamen könnte man die Partition also in der Form

$\{(2, \{1, 2, 4, 5\}), (3, \{3\}), (6, \{6\})\}$

aufschreiben. Abbildung 4.25 enthält ein ungenutztes Feld im Array *components*, dessen Bedeutung unten erklärt wird.  $\square$

Offensichtlich braucht *find* nur  $O(1)$  Zeit. Die Operation *merge* wird mit dieser Datenstruktur folgendermaßen implementiert:

```

algorithm merge ( $p, a, b$ )
{verschmelze die Komponenten  $a$  und  $b$  der Partition  $p$ }
  durchlaufe entweder für  $a$  oder für  $b$  die zugehörige Liste im Array elems,
  nehmen wir an für  $a$ ;
  setze für jedes Element den Komponentennamen auf  $b$ ;
  sei  $j$  der Index des letzten Elementes dieser Liste;
   $elems[j].nextelem := components[b].firstelem;$ 
   $components[b].firstelem := components[a].firstelem;$ 
   $components[a].firstelem := 0$ 
end merge.

```

Am Ende werden also noch die Listen verkettet und die Komponente  $a$  gelöscht; der Name für die Vereinigung von  $a$  und  $b$  ist nun  $b$ .

Der Zeitbedarf für *merge* ist proportional zur Anzahl der Elemente der durchlaufenen Liste, also  $O(n)$ .

Betrachten wir nun eine Folge von  $n-1$  *merge*-Anweisungen (mehr gibt es nicht, da dann alle Elemente in einer einzigen Komponente sind). Im schlimmsten Fall könnte der  $i$ -te *merge*-Schritt eine Komponente  $a$  der Größe 1 mit einer Komponente  $b$  der Größe  $i$  ver-

schmelzen und dazu  $b$  durchlaufen mit Aufwand  $O(i)$ . Der Gesamtaufwand wäre dann etwa

$$\sum_{i=1}^{n-1} i = O(n^2)$$

Ein einfacher Trick vermeidet das: Man verwaltet die Größen der Komponenten mit und durchläuft beim Verschmelzen die *kleinere* Liste, wählt also als neuen Namen den der größeren Komponente. Das bisher ungenutzte Feld im Array *components* sei zu diesem Zweck deklariert als *count*:  $0..n$ .

*Analyse:* Jedes Element, das umbenannt wird, findet sich anschließend in einer Komponente mindestens der doppelten Größe wieder. Jedes Element kann daher höchstens  $O(\log n)$  mal umbenannt werden. Der Gesamtaufwand für  $n-1$  *merge*-Anweisungen ist proportional zur Anzahl der Umbenennungen, also  $O(n \log n)$ .

Diese Art der Analyse, bei der wir keine gute Schranke für eine einzelne Operation, aber eine gute obere Schranke für eine Folge von  $n$  Operationen erreichen können, kommt des Öfteren vor. Man sagt, die *amortisierte* worst-case Laufzeit pro *merge*-Operation (bzgl. einer Folge von  $n$  Operationen) ist  $O(\log n)$ .

### (b) Implementierung mit Bäumen

In dieser Implementierung wird eine Komponente jeweils durch einen Baum dargestellt, dessen Knoten die Elemente der Komponente darstellen und in dem Verweise jeweils vom Sohn zum Vater führen. Zusätzlich braucht man einen Array, um die Elementknoten direkt zu erreichen.

```
type node = record father: ↑node; ... end
        elem  = 1..n
        compname = ↑node
var elems : array[elem] of ↑node
```

Als "Namen" einer Komponente benutzen wir einen Verweis auf die Wurzel des zugehörigen Baumes.

**Beispiel 4.17:** Die im vorigen Beispiel verwendete Partition nimmt folgende Gestalt an:

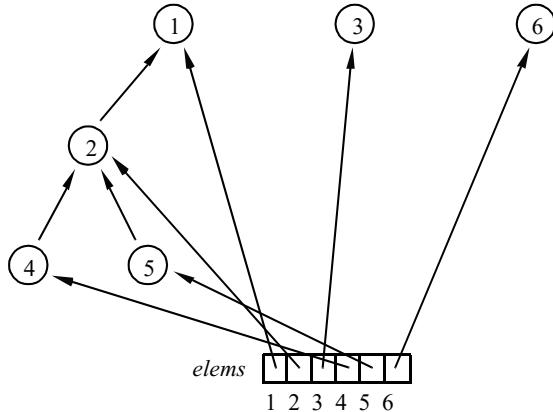


Abbildung 4.26: Partition aus Abbildung 4.25

Auf dieser Datenstruktur sehen die beiden Algorithmen so aus:

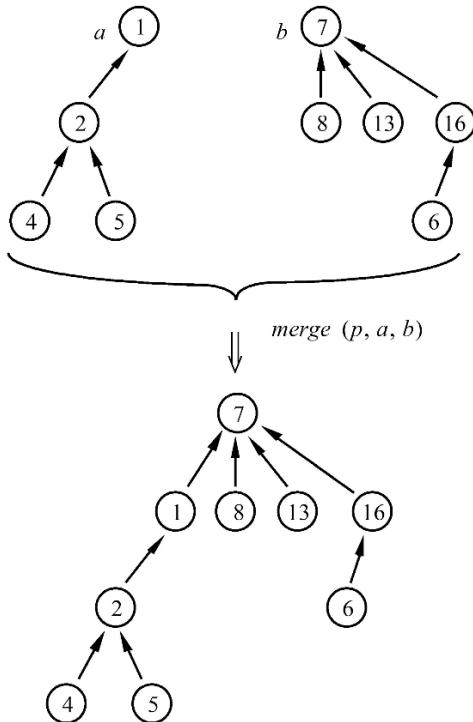
*find* ( $p, x$ ): Man lokalisiert über  $\text{elems}$  den Knoten für  $x$ , läuft von dort zur Wurzel und gibt den Zeiger auf die Wurzel zurück. Da diese Operation genau einem Pfad im Baum folgt, ist der Aufwand  $O(h)$ .

*merge* ( $p, a, b$ ): Einer der Knoten  $a, b$  wird zum Sohn des anderen gemacht. Da diese Operation immer genau zwei Zugriffe benötigt, ist ihr Aufwand  $O(1)$ .

**Beispiel 4.18:** In Abbildung 4.27 wird das Verschmelzen zweier Komponenten  $a$  und  $b$  gezeigt. Der Array  $\text{elems}$  ist weggelassen, da sein Inhalt durch die Operation nicht verändert wird.  $\square$

Ein ähnlicher Trick wie schon bei der Array-Implementierung hilft, die Höhe zu beschränken: Man verwaltet in einem zusätzlichen Feld in der Wurzel die Größe der Komponente und macht jeweils die Wurzel der kleineren Komponente zum Sohn der Wurzel der größeren. Dadurch geschieht Folgendes: In jedem Knoten der kleineren Komponente steigt der Abstand zur Wurzel um 1 und dieser Knoten befindet sich nun in einer doppelt so großen Komponente wie vorher. Wenn zu Anfang jedes Element eine eigene Komponente bildete, kann deshalb durch eine Folge von *merge*-Schritten der Abstand eines Elementes zur Wurzel (also die Tiefe) höchstens  $\log n$  werden.

Der Aufwand der Operationen ist also für *find*  $O(\log n)$  und für *merge*  $O(1)$ .

Abbildung 4.27: Verschmelzen der Komponenten  $a$  und  $b$ 

### Letzte Verbesserung: Pfadkompression

Eine Folge von  $n$  *find*-Operationen könnte bei dieser Implementierung  $O(n \log n)$  Zeit brauchen. Die Idee der *Pfadkompression* besteht darin, dass man bei jedem *find*-Aufruf alle Knoten auf dem Pfad zur Wurzel direkt zu Söhnen der Wurzel macht.

**Beispiel 4.19:** Abbildung 4.28 zeigt, wie durch die Operation *find* ( $p, 5$ ) alle Teilbäume, deren Wurzeln auf dem Pfad vom Element 5 zur Wurzel des Baumes passiert werden, direkt zu Söhnen dieser Wurzel gemacht werden.

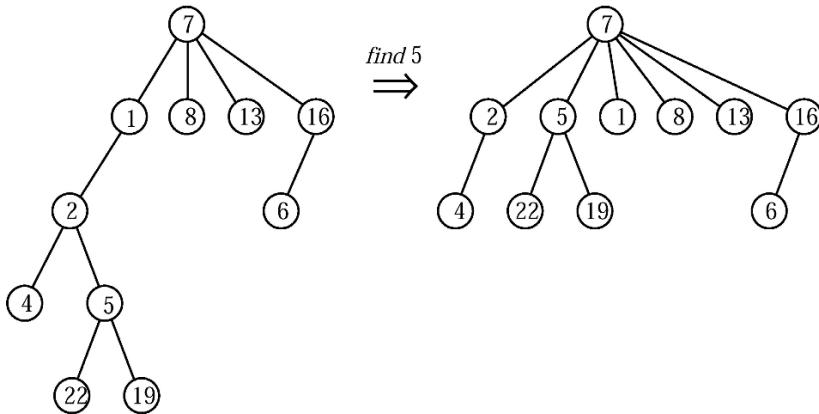


Abbildung 4.28: Pfadkompression

□

Man kann zeigen, dass der Aufwand für  $n$  *find*-Operationen damit reduziert wird auf  $O(n \cdot G(n))$ , wobei  $G(n)$  eine extrem langsam wachsende Funktion ist (es gilt  $G(n) \leq 5^{\sqrt{n}}$   $\forall n \leq 2^{65536}$ ). Aus praktischer Sicht ist der Aufwand also linear. Eine Analyse findet sich in [Aho et al. 1974].

## 4.5 Weitere Aufgaben

**Aufgabe 4.6:** Schreiben Sie Algorithmen für die Operationen des Datentyps “Dictionary” unter Verwendung einer offenen Hashtabelle.

**Aufgabe 4.7:**

- (a) Fügen Sie die Schlüsselfolge 16, 44, 21, 5, 19, 22, 8, 33, 27, 30 gemäß der Hashfunktion

$$h(k) = k \bmod m \quad \text{für } m = 11$$

in eine geschlossene Hashtabelle mit  $b = 1$  ein. Stellen Sie das Ergebnis graphisch dar.

1. Verwenden Sie eine lineare Kollisionsstrategie

$$h_i(k) = (h(k) + c * i) \bmod m \quad \text{mit } c = 1$$

2. Verwenden Sie eine quadratische Kollisionsstrategie

$$h_i(k) = (h(k) + i^2) \bmod m$$

3. Verwenden Sie Doppelhashing

$$h_i(k) = (h(k) + h'(k) * i^2) \bmod m$$

Bestimmen Sie hierfür ein geeignetes  $h'$ .

- (b) Geben Sie für Aufgabenteil (a.1) und  $m = 14$  möglichst allgemein alle  $c$  an, die sich als unbrauchbar erweisen. Begründen Sie Ihre Lösung.

**Aufgabe 4.8:** Schreiben Sie Algorithmen für die Operationen des Datentyps “Dictionary” unter Verwendung einer geschlossenen Hashtabelle mit  $b = 1$ . Verwenden Sie hierfür Hashfunktionen nach der Mittel-Quadrat-Methode und Doppelhashing als Kollisionsstrategie. Die Grundmenge der einzutragenden Elemente seien die positiven ganzen Zahlen.

**Aufgabe 4.9:** Formulieren Sie einen Algorithmus, der eine *Bereichssuche* auf einem binären Suchbaum durchführt. Das heißt, es wird ein Intervall bzw. eine untere und obere Grenze aus dem Schlüsselwertebereich angegeben; die Aufgabe besteht darin, alle im Baum gespeicherten Werte auszugeben, die im Suchintervall enthalten sind. Wie ist die Laufzeit Ihres Algorithmus?

**Aufgabe 4.10:** Bestimmen Sie die erwarteten Kosten einer erfolgreichen binären Suche in einem Array.

*Hinweis:* Stellen Sie den Arrayinhalt als Baum dar, den möglichen Verzweigungen bei der Suche entsprechend.

**Aufgabe 4.11:** In einem binären Suchbaum seien als Schlüsselwerte ganze Zahlen gespeichert. Es sollen Anfragen der folgenden Form unterstützt werden: “Ermittle für ein beliebiges ganzzahliges Suchintervall die Summe der darin enthaltenen Schlüsselwerte.”

- (a) Welche zusätzliche Information muss in jedem Knoten verwaltet werden, damit Anfragen dieser Art in  $O(\log n)$  Zeit durchgeführt werden können? Wie sieht dann der Anfragealgorithmus aus?
- (b) Geben Sie entsprechend modifizierte Algorithmen für das Einfügen und Löschen an, die die zusätzliche Information mitändern.

**Aufgabe 4.12:** In einer Variante des binären Suchbaumes werden alle zu verwaltenden Schlüssel in den Blättern gespeichert; die inneren Knoten dürfen beliebige Schlüsselwerte enthalten, die sich als “Wegweiser” eignen, das heißt, die Suchbaumeigenschaft nicht verletzen. Die Blätter enthalten zusätzlich Zeiger auf das linke und rechte Nachbarblatt, bilden also eine doppelt verkettete Liste.

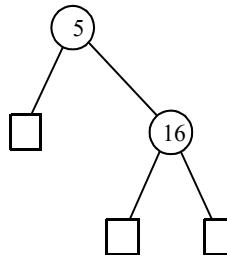
- (a) Geben Sie entsprechende Knotendeklarationen an und schreiben Sie eine modifizierte Methode für *insert*.
- (b) Wie würde man auf dieser Struktur eine Bereichssuche (Aufgabe 4.9) durchführen?

**Aufgabe 4.13:** Sei das *Gewicht* eines Baumes die Anzahl seiner Knoten. Nehmen wir an, man fordert, dass in einem binären Suchbaum in jedem Knoten das Gewicht des einen Teilbaums höchstens doppelt so groß ist wie das Gewicht des anderen. Kann man damit eine logarithmische Höhe garantieren?

**Aufgabe 4.14:** Wir präzisieren die Fragestellung aus Aufgabe 4.13 wie folgt: Zunächst gehen wir von einem vollständig mit Blättern versehenen Baum aus, in dem nur die inneren Knoten Schlüssel enthalten (wie in Abbildung 3.25). Das bedeutet, dass jeder innere Knoten genau zwei Söhne hat. Das *Gewicht*  $W(T)$  eines Baumes  $T$  sei nun definiert als die Anzahl seiner Blätter (die bekanntlich um 1 größer ist als die Anzahl seiner inneren Knoten). Die *Balance* eines inneren Knotens  $p$  sei definiert als

$$\rho(p) = \frac{W(T_l)}{W(T)}$$

wobei  $p$  die Wurzel des Baumes  $T$  mit linkem Teilbaum  $T_l$  sein soll. Die Wurzel des in der folgenden Abbildung gezeigten Baumes hätte also Balance 1/3, der rechte Sohn Balance 1/2.



Nehmen wir an, man verlangt, dass für jeden inneren Knoten  $p$  eines solchen binären Suchbaums die Balance im Bereich  $1/4 \leq \rho(p) \leq 3/4$  läge. Könnte man mit den vom AVL-Baum her bekannten Restrukturierungsoperationen (Rotationen, Doppelrotationen) dieses Balanciertheitskriterium unter Einfügungen aufrecht erhalten?

**Aufgabe 4.15:** Inwieweit ist in der Array-Implementierung von Partitionen (Abschnitt 4.4, Implementierung (a)) die Annahme, dass Komponentennamen und Mengennamen

jeweils durch  $\{1, \dots, n\}$  dargestellt werden, wesentlich? Lässt sich die Implementierung verallgemeinern?

## 4.6 Literaturhinweise

Eine sehr eingehende Beschreibung und mathematische Analyse von Hashverfahren findet man bei Knuth [1998]. Die Idee der offenen Adressierung geht zurück auf Peterson [1957]; daher stammt auch die Analyse des idealen Hashing (dort “uniform hashing” genannt). Die weitergehende Analyse des linearen Sondierens ist von Knuth [1998]. Übersichtsarbeiten zu Hashing sind [Morris 1968], [Maurer und Lewis 1975] und [Severance und Duhne 1976]. Eine Übersicht zu Hashfunktionen, die auch experimentell miteinander verglichen werden, findet man bei [Lum *et al.* 1971]. Techniken des dynamischen und erweiterbaren Hashing, die in diesem Buch nicht behandelt werden, sind in [Enbody und Du 1988] dargestellt.

Die Ursprünge binärer Suchbäume liegen wiederum im Dunkeln; erste Veröffentlichungen dazu sind [Windley 1960], [Booth und Colin 1960] und [Hibbard 1962]. In diesen Arbeiten wird auch jeweils auf etwas unterschiedliche Art die Durchschnittsanalyse für binäre Suchbäume durchgeführt. Hibbard [1962] hat auf den Zusammenhang mit der Analyse des durchschnittlichen Verhaltens von Quicksort hingewiesen. Die Tatsache, dass binäre Suchbäume mit der Zeit doch entarten, wenn auch Löschungen vorgenommen werden (falls jeweils das größere Element die Lücke füllt), wurde von Culberson [1985] gezeigt. AVL-Bäume stammen, wie erwähnt, von Adelson-Velskii und Landis [1962]; dort wurde auch schon die Analyse der maximalen Höhe durchgeführt. Die Aufgaben 4.14 und 4.15 führen zu gewichtsbalancierten Bäumen (BB[ $\alpha$ ]-Bäume, Bäume beschränkter Balance), die von Nievergelt und Reingold [1973] beschrieben wurden. Die Theorie der *fringe analysis* ist eine Methodik zur Analyse des durchschnittlichen Verhaltens von Suchbäumen; einen Überblick dazu gibt Baeza-Yates [1995].

Der Begriff “Priority Queue” für eine Datenstruktur mit den genannten Operationen wurde von Knuth [1998] eingeführt. Die Heap-Implementierung wurde zuerst von Williams [1964] zur Realisierung von Heapsort eingesetzt.

Die Grundidee der Lösung des UNION-FIND-Problems mit Bäumen durch Verschmelzen unter Beachtung des Gewichts wird von Knuth [1998] M.D. McIlroy zugeschrieben, die Idee der Pfadkompression A. Tritter. Die Analyse des Algorithmus stammt von Hopcroft und Ullman [1973]. Der Fall mit gewichtetem Verschmelzen und Pfadkompression wurde von Tarjan [1975] analysiert (siehe auch [Aho *et al.* 1974]). Inzwischen gibt es

eine Fülle von Resultaten zu derartigen Problemen, die von Galil und Italiano [1991] zusammenfassend dargestellt werden.



## 5 Sortieralgorithmen

Das Sortieren einer Menge von Werten über einem geordneten Wertebereich (z. B. *int*, *real*, *string*), das heißtt, die Berechnung einer geordneten Folge aus einer ungeordneten Folge dieser Werte, ist ein zentrales und intensiv studiertes algorithmisches Problem. Sortieralgorithmen haben viele direkte Anwendungen in der Praxis, finden aber auch häufig Einsatz als Teilschritte in Algorithmen, die ganz andere Probleme lösen. Zum Beispiel für die Plane-Sweep- und Divide-and-Conquer-Algorithmen in Kapitel 8 ist Sortieren eine wesentliche Voraussetzung.

Das *Sortierproblem* lässt sich etwas präziser so formulieren: Gegeben sei eine Folge  $S = s_1 \dots s_n$  von Records, die eine *key*-Komponente eines linear geordneten Datentyps besitzen. Man berechne eine Folge  $S' = s_{i_1} \dots s_{i_n}$  als Permutation der Folge  $S$ , sodass

$$s_{i_1}.key \leq s_{i_2}.key \leq \dots \leq s_{i_n}.key$$

Es ist durchaus erlaubt, dass derselbe Schlüsselwert in mehreren Records vorkommt.

Man kann Sortieralgorithmen nach verschiedenen Kriterien klassifizieren:

- *intern/extern*: Man spricht von einem internen Verfahren, wenn alle Records gleichzeitig im Hauptspeicher gehalten werden können. Ein externes Verfahren lädt jeweils nur eine Teilmenge der Datensätze.
- methodisch:
  - Sortieren durch Einfügen
  - Sortieren durch Auswählen
  - Divide-and-Conquer
  - Fachverteilen
  - ...
- *nach Effizienz*: Einfache Verfahren haben eine Laufzeit von  $O(n^2)$ , gute Methoden erreichen  $O(n \log n)$ . Bei manchen Methoden unterscheiden sich noch Durchschnitts- und worst-case-Verhalten.
- *im Array oder nicht*: Bei internen Verfahren ist man häufig an Realisierungen interessiert, die die Ausgangs- und Zielfolge im Array darstellen. Solche Methoden sind einfach zu implementieren und brauchen keinen zusätzlichen Platz für Zeiger. Besonders beliebt sind Verfahren, die nur einen einzigen Array benötigen und das Ergebnis durch Vertauschen innerhalb dieses Arrays erzielen, man sagt, solche Verfahren sortieren *in situ*. Denn wenn etwa zwei Arrays gebraucht werden, kann man nur eine Menge der halben Größe im Hauptspeicher sortieren.

- *allgemeine / eingeschränkte Verfahren:* Manche Methoden (Sortieren durch Fachverteilen) lassen sich nur auf eine Folge von Records mit speziellen Eigenschaften anwenden.

In vielen Sortieralgorithmen, insbesondere beim Sortieren im Array, sind die wesentlichen Operationen der *Vergleich* zweier Schlüssel und das *Vertauschen* von zwei Datensätzen. Deshalb werden oft in der Analyse diese Operationen gezählt und somit als Kostenmaß benutzt.

Ein Sortierverfahren heißt *stabil*, wenn sichergestellt ist, dass die Reihenfolge von Sätzen mit gleichem Schlüsselwert nicht verändert wird, also in der Ergebnisfolge die gleiche ist wie in der Ausgangsfolge.

## 5.1 Einfache Sortierverfahren: Direktes Auswählen und Einfügen

In diesem Abschnitt betrachten wir einfache Verfahren mit einer Laufzeit von  $O(n^2)$ . Methodisch behandeln wir Sortieren durch Auswählen und durch Einfügen. Zu diesen Methoden gibt es auch “raffinierte” Implementierungen mit  $O(n \log n)$  Laufzeit; diese werden in Abschnitt 5.3 besprochen.

Sei  $S$  die gegebene Folge von Sätzen. Wir benutzen zur Beschreibung der beiden Strategien zwei Folgen *SORTED* und *UNSORTED*.

```
algorithm SelectionSort ( $S$ )
{sortiere die Folge  $S$  durch Auswählen}
 $UNSORTED := S$ ;  $SORTED := \emptyset$ ;
while  $UNSORTED \neq \emptyset$  do
    entnimm  $UNSORTED$  das Minimum und hänge es an  $SORTED$  an.
end while.

algorithm InsertionSort ( $S$ )
{sortiere die Folge  $S$  durch Einfügen}
 $UNSORTED := S$ ;  $SORTED := \emptyset$ ;
while  $UNSORTED \neq \emptyset$  do
    entnimm  $UNSORTED$  das erste Element und füge es an der richtigen Position in
     $SORTED$  ein.
end while.
```

Diese algorithmischen Schemata gelten auch für die verfeinerten Methoden des Abschnitt 5.3. Wir sprechen von *direktem* Auswählen oder Einfügen, wenn sie ohne Weiteres im Array realisiert werden.

Wenden wir uns nun der Implementierung zu. Alle in diesem Kapitel vorgestellten Methoden sind als *statische* Methoden implementiert. Sie manipulieren keine Attribute ihrer Klasse, sondern arbeiten ausschließlich auf den übergebenen Parametern. Hilfsmethoden sind als *private* definiert, so dass sie nicht im Zugriff des Benutzers liegen.

**Beispiel 5.1:** Die Klasse, innerhalb derer wir im Folgenden unsere Sortiermethoden implementieren, heiße *SortAlgs*.

```
public class SortAlgs
{
    private static void utility(...) {...}
    public static void selectionSort(...) {...}
    ...
}
```

Aus einem Programm, das die Klasse *SortAlgs* verwendet, kann man die Methode *selectionSort* nun problemlos aufrufen:

```
SortAlgs.selectionSort(...)
```

Den entsprechenden Aufruf der Methode *utility* würde der Compiler jedoch als unzulässig zurückweisen.  $\square$

Zur Implementierung des Algorithmus *SelectionSort* definieren wir zunächst die Hilfsmethode *swap*, die wir im Verlauf dieses Kapitels noch mehrfach verwenden werden. Sie vertauscht in einem Array *S* mit Komponenten vom Typ *Elem* die Elemente an den Positionen *n* und *m*.

```
private static void swap(Elem[] S, int n, int m)
{
    Elem tmp = S[n];
    S[n] = S[m]; S[m] = tmp;
}
```

Nun können wir den Algorithmus *SelectionSort* wie folgt implementieren:

```
public static void selectionSort(Elem[] S)
{
    Elem min; int minindex;
```

```

for(int i = 0; i < S.length - 1; i++)
{
    min = S[i]; minindex = i;
    for(int j = i + 1; j < S.length; j++)
    {
        if(S[j].isLess(min))
        {
            min = S[j]; minindex = j;
        }
    }
    swap(S, i, minindex);
}
}

```

} Auffinden des Minimums der S[i]..S[n]

Im Wesentlichen werden beim ersten Durchlauf der Schleife  $n$  Operationen durchgeführt (innere Schleife), beim 2. Durchlauf  $n-1$  usw., also

$$n + (n-1) + (n-2) + \dots + 1 = \sum_{i=1}^n i = \frac{n(n+1)}{2} = O(n^2)$$

Durchschnitts- und worst-case-Verhalten sind gleich. Wenn man Vergleiche und Vertauschungen zählt, ergibt sich:

$$\begin{aligned} C(n) &= O(n^2) && (\text{Comparisons}) \\ X(n) &= O(n) && (\text{Exchanges}) \end{aligned}$$

Betrachten wir nun die Methode *insertionSort*.

```

public static void insertionSort(Elem[] S)
{
    Elem r;
    int j;
    for(int i = 1; i < S.length; i++)
    {
        r = S[i];
        j = i - 1;
        while(j >= 0 && r.isLess(S[j]))
        {
            S[j + 1] = S[j];
            j--;
        }
        S[j + 1] = r;
    }
}

```

} größere Werte jeweils um ein Feld nach oben schieben

Im worst case wird die innere Schleife jeweils bis ganz vorne ( $S[0]$ ) durchlaufen. Das tritt bei einer fallend geordneten Ausgangsfolge ein. Der Aufwand ist dann etwa

$$1+2+\dots+n = \sum_{i=1}^n i = O(n^2)$$

Im Durchschnitt kann man erwarten, dass die innere Schleife etwa die Hälfte der niedrigeren Array-Positionen durchläuft. Der Aufwand beträgt dann

$$\frac{1}{2} \cdot (1+2+\dots+n) = \frac{1}{2} \sum_{i=1}^n i = \frac{n \cdot (n+1)}{4} = O(n^2)$$

$$\begin{aligned} C_{avg}(n) &= O(n^2), & C_{worst}(n) &= O(n^2) \\ M_{avg}(n) &= O(n^2), & M_{worst}(n) &= O(n^2) \quad (\text{Move-Operationen}) \end{aligned}$$

**Selbsttestaufgabe 5.1:** Ein weiteres bekanntes Sortierverfahren, genannt *BubbleSort*, beruht auf der Idee, zwei benachbarte Elemente eines Arrays zu vertauschen, wenn sie in der falschen Reihenfolge sind. Formulieren Sie eine entsprechende Sortiermethode und analysieren Sie ihre Laufzeit.  $\square$

**Selbsttestaufgabe 5.2:** Die Methode *insertionSort* kann geringfügig verbessert werden, wenn beim Einfügen des  $i$ -ten Elements die Tatsache ausgenutzt wird, dass die  $(i-1)$  ersten Elemente bereits sortiert sind, indem für das Auffinden der Position, an der das  $i$ -te Element eingefügt werden muss, binäres Suchen verwendet wird. Modifizieren Sie die Methode entsprechend und erklären Sie, warum diese Verbesserung nur geringfügig ist.  $\square$

## 5.2 Divide-and-Conquer-Methoden: Mergesort und Quicksort

Die folgenden beiden Algorithmen erreichen ein Laufzeitverhalten von  $O(n \log n)$ , Mergesort im worst case, Quicksort im Durchschnitt (Quicksort braucht im worst case  $O(n^2)$ ). Leider sortiert Mergesort nicht in situ, jedenfalls nicht auf offensichtliche Art (trickreiche Implementierungen, die das doch erreichen, sind in den Literaturhinweisen zitiert). Mergesort ist vor allem als externes Sortierverfahren bekannt; diesen Aspekt betrachten wir im letzten Kapitel. Quicksort ist als das im Durchschnitt schnellste Verfahren überhaupt bekannt; es gibt aber seit einiger Zeit eine Heapsort-Variante, die für große  $n$  schneller ist (siehe Abschnitt 5.3).

Beide Algorithmen benutzen das *Divide-and-Conquer*-Paradigma, das sich allgemein so formulieren lässt:

```

if die Objektmenge ist klein genug
then löse das Problem direkt
else
  Divide: Zerlege die Menge in mehrere Teilmengen (wenn möglich, gleicher
  Größe).
  Conquer: Löse das Problem rekursiv für jede Teilmenge.
  Merge: Berechne aus den für die Teilmengen erhaltenen Lösungen eine
  Lösung des Gesamtproblems.
end if

```

Wie wir sehen werden, benutzt Mergesort einen trivialen Divide-Schritt und leistet die eigentliche Arbeit im Merge-Schritt; Quicksort “arbeitet” im Divide-Schritt und hat dafür einen trivialen Merge-Schritt. Sei  $S = s_1 \dots s_n$  die Eingabesequenz.

```

algorithm MergeSort ( $S$ )
{sortiere  $S$  durch Verschmelzen}
if  $|S| = 1$  then return  $S$ 
else
  Divide:  $S_1 := s_1 \dots s_{\lfloor n/2 \rfloor}; S_2 := s_{\lfloor n/2 \rfloor + 1} \dots s_n;$ 
  Conquer:  $S_1' := \text{MergeSort}(S_1); S_2' := \text{MergeSort}(S_2);$ 
  Merge: return Merge ( $S_1', S_2'$ )
end if.

```

*Merge* ( $S', S''$ ) ist dabei eine Prozedur, die zwei sortierte Folgen zu einer einzigen sortierten Folge verschmilzt. Wir haben bereits in Abschnitt 4.1 gesehen, dass das in einem parallelen Durchlauf in  $O(k+m)$  Zeit möglich ist, wobei  $k$  und  $m$  die Listenlängen sind, für zwei Listen der Länge  $n/2$  also in  $O(n)$  Zeit. Der Algorithmus Quicksort lässt sich so formulieren:

```

algorithm QuickSort ( $S$ )1
{sortiere die Folge  $S$  mit Quicksort}
if  $|S| = 1$  then return  $S$ 
else
  Divide: Wähle irgendeinen Schlüsselwert  $x = s_j.key$  aus  $S$  aus. Berechne eine
  Teilfolge  $S_1$  aus  $S$  mit den Elementen, deren Schlüsselwert kleiner als
   $x$  ist und eine Teilfolge  $S_2$  mit Elementen  $\geq x$ .
  Conquer:  $S_1' := \text{QuickSort}(S_1); S_2' := \text{QuickSort}(S_2);$ 
  Merge: return concat ( $S_1', S_2'$ )
end if.

```

1. Achtung: Hier wird die Grundstruktur von Quicksort gezeigt. Diese muss noch etwas verfeinert werden, um Terminierung sicherzustellen, siehe unten.

Betrachten wir zunächst Implementierungen des MergeSort-Algorithmus. Die direkte Implementierung mit verketteten Listen braucht  $O(n)$  Zeit für das Zerlegen einer Liste der Länge  $n$  und  $O(n)$  Zeit für das Verschmelzen zweier Listen mit insgesamt  $n$  Elementen. Wir erhalten die Rekursionsgleichungen

$$T(n) = \begin{cases} O(1) & \text{falls } n = 1 \\ O(n) + 2 \cdot T(n/2) + O(n) & \text{falls } n > 1 \end{cases}$$

↑ Divide      ↑ Conquer      ↑ Merge

Eine Implementierung im Array (mit 2 Arrays) ist ebenfalls möglich. Jede Folge wird dabei als Teilbereich im Array dargestellt. Ein entsprechender Ablauf ist in Abbildung 5.1 gezeigt. Beachten Sie, dass dort alle Rekursionsebenen gemeinsam dargestellt werden!

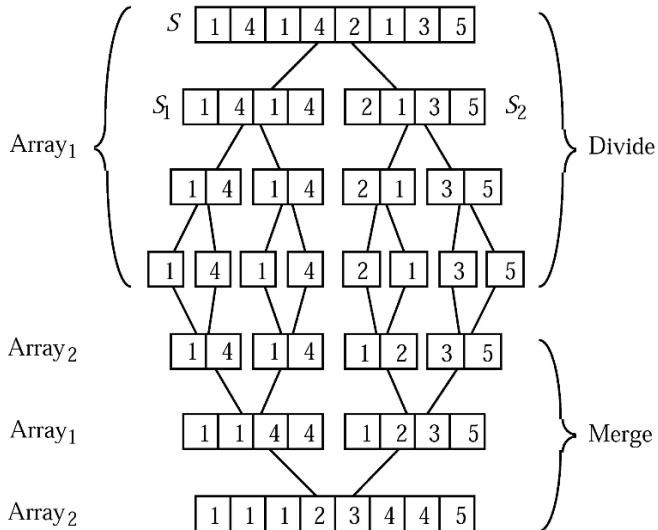


Abbildung 5.1: Mergesort-Implementierung mit zwei Arrays

Hierbei ist Teilen in konstanter Zeit möglich, Verschmelzen in linearer Zeit, also erhalten wir

$$T(n) = \begin{cases} O(1) & \text{falls } n = 1 \\ O(1) + 2 \cdot T(n/2) + O(n) & \text{falls } n > 1 \end{cases}$$

↑ Divide      ↑ Conquer      ↑ Merge

Für beide Implementierungen von Mergesort sind die Rekursionsgleichungen Spezialfälle der allgemeineren Form

$$T(n) = \begin{cases} a & n=1 \\ 2 \cdot T(n/2) + f(n) + c & n>1 \end{cases}$$

die bei sehr vielen Divide-and-Conquer-Algorithmen gilt. Um diese Gleichungen zu verstehen und zu lösen, betrachten wir den Baum der rekursiven Aufrufe und notieren an jedem Knoten die dort entstehenden Kosten. Zur Vereinfachung sei  $n = 2^k$ , also  $k = \log n$ .

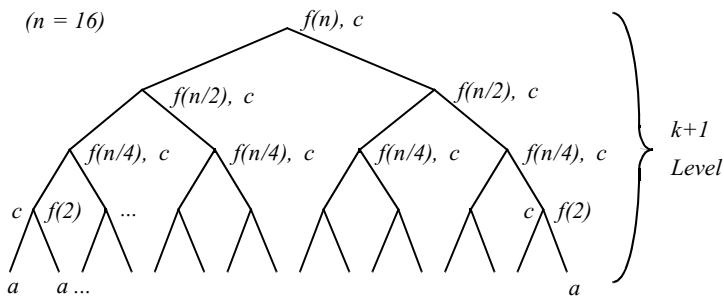


Abbildung 5.2: Baum der rekursiven Aufrufe

Der Aufwand für alle  $c$ - und  $a$ -Beiträge ist  $c \cdot (n-1) + a \cdot n$ , also  $O(n)$ . Wenn  $f(n)$  mindestens linear wächst, das heißt, wenn gilt

$$i \cdot f(n/i) \leq f(n),$$

dann können wir den Aufwand für die  $f(n)$ -Beiträge so abschätzen:

$$\begin{aligned} \text{Level 0: } & f(n) \\ 1: & 2 \cdot f(n/2) \leq f(n) \\ 2: & 4 \cdot f(n/4) \leq f(n) \\ \dots & \\ (\log n)-1: & n/2 \cdot f(2) \leq f(n) \end{aligned} \left. \right\} \log n \text{ Level}$$

Von diesem allgemeinen Ergebnis kann man die Komplexität verschiedener Spezialfälle direkt ablesen:

1.  $f(n) = 0 \Rightarrow T(n) = O(n)$
2.  $f(n) = b \cdot n \Rightarrow T(n) = O(n \log n)$
3.  $f(n) = b \cdot n \cdot \log n \Rightarrow T(n) = O(n \log^2 n)$

Wir überprüfen für den 3. Fall einmal die Voraussetzung, dass  $f(n)$  mindestens linear wächst:

$$i \cdot \left( b \cdot \frac{n}{i} \cdot \log \frac{n}{i} \right) = b \cdot n \cdot \log \frac{n}{i} \leq b \cdot n \cdot \log n$$

Das Ergebnis der Analyse lässt sich so zusammenfassen:

**Satz 5.2:** Ein balancierter DAC-Algorithmus (das heißt, die Teilmengen sind jeweils etwa gleich groß) hat eine Laufzeit

- $O(n)$  falls Divide- und Merge-Schritt jeweils nur  $O(1)$  Zeit brauchen,
- $O(n \log n)$  falls Divide- und Merge-Schritt in linearer Zeit ( $O(n)$ ) durchführbar sind.

□

Als direkte Konsequenz ergibt sich eine Laufzeit von  $O(n \log n)$  für Mergesort.

*Quicksort* kann man ebenfalls direkt mit Listen implementieren; solche Implementierungen sind besonders in logischen Programmiersprachen (PROLOG) beliebt. Der Divide-Schritt erfordert  $O(n)$  Zeit, der Merge-Schritt (Konkatenation zweier Listen) ist evtl. sogar in  $O(1)$  realisierbar. Erreicht Quicksort damit eine Laufzeit von  $O(n \log n)$ ? Leider nicht im worst case, da nicht sichergestellt ist, dass im Divide-Schritt eine balancierte Aufteilung erreicht wird. Bei der bisher beschriebenen Version ist es sogar möglich, dass in jedem Divide-Schritt das Minimum als  $x$ -Wert gewählt wird; dadurch wird Liste  $S_1$  leer und  $S_2 = S$  und Quicksort terminiert nicht! Die bisher beschriebene Version ist also nicht korrekt und muss wie folgt verfeinert werden:

```

algorithm QuickSort ( $S$ )
if  $|S| = 1$  then return  $S$ 
else if alle Schlüssel in  $S$  sind gleich
    then return  $S$ 
    else
        Divide: Wähle  $x = s_j.key$ , sodass  $x$  nicht minimal ist in  $S$ .
        usw.
    
```

Dies ist die korrekte Version von Quicksort. Im worst case wird eine Folge der Länge  $n$  in eine Folge der Länge 1 und eine der Länge  $n-1$  zerlegt. Der Aufrufbaum kann also entarten:

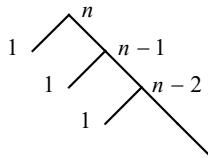


Abbildung 5.3: Aufrufbaum von Quicksort im worst case

Da der Aufwand in jedem Knoten linear ist, entsteht ein Gesamtaufwand von

$$b \cdot n + b \cdot (n-1) + \dots + b \cdot 1 = b \cdot \frac{n(n+1)}{2} = O(n^2) \quad (\text{für eine Konstante } b)$$

Da in jedem Divide-Schritt die zu zerlegende Folge unsortiert ist, muss für das Zerlegen jedes Element dieser Folge mit  $x$  verglichen werden. Das heißt, bei *jeder* Implementierung, mit Listen oder Arrays oder wie auch immer, erfordert der Divide-Schritt  $\Omega(n)$  Zeit, daher gilt:

**Satz 5.3:** Für jede Quicksort-Implementierung ist die Laufzeit im worst case  $\Omega(n^2)$ .  $\square$

Wir betrachten nun genauer die Implementierung von Quicksort im Array. Sei wieder  $S$  ein Array der Größe  $n$ , der zu Anfang die ungeordnete Folge von Records enthält. Jede zu verarbeitende Folge wird anhand ihrer Indexgrenzen  $(i, j)$  im Array dargestellt. Das erste Teilproblem besteht darin, einen trennenden Schlüsselwert  $x$  zu finden, falls nicht alle Schlüssel im Teilarray gleich sind. Dieser sollte nicht minimal sein, um Terminierung sicherzustellen. Die folgende Methode *findx* durchsucht den Teilarray von links nach rechts nach zwei unterschiedlichen Schlüsseln und gibt den Index des größeren Schlüssels zurück, wenn zwei verschiedene existieren, andernfalls -1.

Bei Aufruf von *findx*  $(i, j)$  gelte  $j > i$ . Dies ist eine Zusicherung, auf die sich die Implementierung der Prozedur *findx* verlässt (vgl. Meyer [1990]).

```

private static int findx(Elem[] S, int i, int j)
/* liefert den Index eines nicht minimalen Schlüssels im Teilarray (i,j),
falls existent, sonst -1 */
{
    int k = i + 1;
    while(k <= j && S[k].isEqual(S[k - 1])) k++;
    if(k > j) return -1;
    else if(S[k - 1].isLess(S[k])) return k; else return k -1;
}

```

Das zweite Teilproblem besteht darin, innerhalb des Teilarrays  $(i, j)$  für ein gegebenes  $x$  Records so zu vertauschen, dass alle Schlüssel  $< x$  links von allen Schlüsseln  $\geq x$  stehen. Dazu lässt man einen Zeiger  $l$  von  $i$  aus aufwärts und einen Zeiger  $r$  von  $j$  aus abwärts wandern. Zeiger  $l$  stoppt beim ersten Schlüssel  $\geq x$ , Zeiger  $r$  beim ersten Schlüssel  $< x$ . Nun werden  $S[l]$  und  $S[r]$  vertauscht. Das wird wiederholt, bis  $l$  und  $r$  sich treffen. Zu jeder Zeit stehen links von  $l$  nur Schlüssel  $< x$  und rechts von  $r$  nur Schlüssel  $\geq x$ . Wenn  $l$  und  $r$  sich treffen, ist die Aufteilung des gegebenen Teilarrays daher komplett.

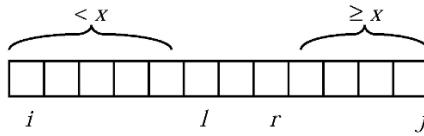


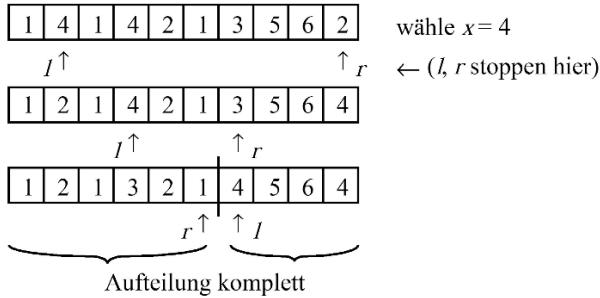
Abbildung 5.4: Partitionieren des Teilarrays  $(i, j)$

Diese Strategie wird in der folgenden Methode *partition* realisiert.

```
private static int partition(Elem[] S, int i, int j, Elem x)
/* zerlegt den Teilarray  $(i, j)$  anhand des Schlüsselwertes  $x$  und gibt den Index
des Elementes  $\geq x$  zurück */
{
    int l = i; int r = j;
    while(l < r)
    {
        while(S[l].isLess(x)) l++;
        while(!S[r].isLess(x)) r--;
        if(l < r) swap(S, l, r);
    }
    return l;
}
```

Die Korrektheit der Methode *partition* kann man sich so klarmachen: Aufgrund der Wahl von  $x$  gibt es mit Sicherheit jeweils einen Schlüssel  $< x$  und einen  $\geq x$  im Teilarray, deshalb können  $l$  und  $r$  nicht über die Grenzen hinauslaufen. Sie können auch nicht auf dem gleichen Schlüssel stoppen, da ein Schlüssel nicht gleichzeitig  $< x$  und  $\geq x$  sein kann.

**Beispiel 5.4:** Abbildung 5.5 zeigt einen Ablauf der Methode *partition*, wobei als Trennwert  $x = 4$  gewählt wurde. □

Abbildung 5.5: Ablauf der Methode *partition*

Die Methode *quicksort* selbst lässt sich dann so formulieren:

```
private static void quickSort(Elem[] S, int i, int j)
{
    int k, xindex;
    if(i < j)
    {
        xindex = findx(S, i, j);
        if(xindex != -1)
        {
            /* Divide: */ k = partition(S, i, j, S[xindex]);
            /* Conquer: */ quickSort(S, i, k - 1); quickSort(S, k, j);
            /* Merge: nichts zu tun */
        }
    }
}
```

Dem Benutzer machen wir es natürlich noch etwas bequemer:

```
public static void quickSort(Elem[] S)
{
    quickSort(S, 0, S.length - 1);
}
```

Zum Aufruf der Methode *quickSort* reicht damit – wie bei allen anderen implementierten Sortierverfahren in diesem Kapitel – die Übergabe des zu sortierenden Arrays als Parameter.

Man sieht, dass die Laufzeit von *findx* und *partition* jeweils  $O(m)$  ist für einen Teilarray der Größe  $m$ , das heißt, der Divide-Schritt erfordert  $O(m)$  Zeit. Also ist die *worst-case Laufzeit* bei dieser Implementierung  $O(n^2)$ , aufgrund des Arguments, das zu Satz 5.3

führte. Man beachte, dass bisher nur klar war, dass die worst-case Laufzeit  $\Omega(n^2)$  beträgt!

### Durchschnittsanalyse für Quicksort

Wir untersuchen nun das durchschnittliche Verhalten von Quicksort und machen dazu folgende Annahmen:

1. In der zu sortierenden Folge seien alle Schlüssel paarweise verschieden.
2. Wenn  $a_1 \dots a_n$  die geordnete Folge der Schlüssel ist, dann wählt  $\text{findx}$  alle Elemente aus  $\{a_2, \dots, a_n\}$  mit gleicher Wahrscheinlichkeit  $1/(n-1)$ .

Unter diesen Annahmen sei  $T(n)$  die erwartete Laufzeit von Quicksort für einen Array der Länge  $n$ . Wenn  $x = a_{i+1}$  gewählt wird, dann ist die Laufzeit

$$\begin{array}{c} O(n) + T(i) + T(n-i) \\ \uparrow \quad \uparrow \quad \uparrow \\ \text{partition, findx} \quad \text{Quicksort}(1, i) \quad \text{Quicksort}(i+1, n) \end{array}$$

Wir erhalten  $T(n)$  durch Mittelung über alle Wahlen von  $x = a_{i+1}$ :

$$T(n) = \begin{cases} a & n=1 \\ b \cdot n + c + \frac{1}{n-1} \cdot \sum_{i=1}^{n-1} (T(i) + T(n-i)) & n>1 \end{cases}$$

Darin stellt  $a$  die konstanten Kosten in den Blättern des Aufrufbaumes von Quicksort dar,  $b \cdot n$  die linearen und  $c$  die konstanten Kosten in inneren Knoten. Wie auch immer der Aufrufbaum balanciert ist, die konstanten Kosten sind insgesamt  $O(n)$ . Zur Vereinfachung berechnen wir diese Kosten außerhalb der Rekursionsgleichung. Sei

$$T'(n) = \begin{cases} 0 & n=1 \\ b \cdot n + \frac{2}{n-1} \cdot \sum_{i=1}^{n-1} T'(i) & n>1 \end{cases}$$

sodass  $T(n) = T'(n) + O(n)$ . Dabei haben wir die beiden Terme in der Summe durch Indexwechsel zusammengefasst wie in Abschnitt 4.2.3 (überhaupt verläuft diese Analyse sehr ähnlich). Wir berechnen nun  $T'(n)$ . Sei

$$S_n := \sum_{i=1}^n T'(i)$$

Dann gilt

$$T'(n) = b \cdot n + \frac{2}{n-1} \cdot S_{n-1}$$

$$S_n = T'(n) + S_{n-1}$$

$$= b \cdot n + \frac{2}{n-1} \cdot S_{n-1} + S_{n-1}$$

$$S_n = \frac{n+1}{n-1} \cdot S_{n-1} + b \cdot n$$

Die ganz ähnliche Rekursionsgleichung

$$S_n = \frac{n+2}{n} \cdot S_{n-1} + n$$

wurde in Abschnitt 4.2.3 (Binäre Suchbäume) schon gelöst. Hier erhält man ganz analog

$$S_n = b \cdot n \cdot (n+1) \cdot (H_{n+1} - 1)$$

Damit gilt

$$T'(n) = b \cdot n + \frac{2 \cdot b \cdot n \cdot (n-1)}{n-1} \cdot (H_n - 1) = b \cdot n \cdot (2 \cdot H_n - 1)$$

Wegen  $\lim_{n \rightarrow \infty} H_n = \ln n + \gamma$  folgt

$$T'(n) = 2 \cdot b \cdot n \cdot \ln n - \Theta(n)$$

$$T(n) = O(n \log n)$$

Die erwarteten Kosten für Quicksort sind also  $O(n \log n)$ . Durch den Einsatz der Konstanten  $a, b, c$  haben wir eine ‘pauschale’ Analyse für die Gesamtkosten aller Operationen gemacht. Man kann nach dem gleichen Schema separat Analysen für die Anzahl der Vergleiche oder Vertauschungen machen; dann ist  $b = 1$  und die oben präzise ermittelte Proportionalitätskonstante erlangt Aussagekraft, z. B.

$$C(n) \approx 2 \cdot \ln 2 \cdot n \log n + O(n) = 1.386 \cdot n \log n + O(n)$$

Die Analyse beruhte auf der Annahme, dass alle Elemente der zu sortierenden Folge mit gleicher Wahrscheinlichkeit als Split-Element  $x$  ausgewählt werden. Man muss sich fragen, ob eine gegebene Implementierung dies tatsächlich leistet. Unsere Prozedur *findx*

wird im Allgemeinen den ersten oder zweiten Schlüssel von links im Teilarray auswählen. Unter der Annahme, dass alle zu sortierenden Folgen mit gleicher Wahrscheinlichkeit auftreten, entspricht das der obigen Annahme. Man beachte aber, dass Quicksort mit dieser Auswahlprozedur ausgesprochen schlecht (sogar auf die schlechtestmögliche Art) auf eine bereits sortierte Eingabefolge reagiert. Deshalb sollte man in der Praxis diese spezielle Implementierung nicht wählen.

Eine Möglichkeit besteht darin, die Auswahl tatsächlich zufällig mithilfe eines Zufallszahlengenerators durchzuführen. Dann gibt es in der Praxis keine besonders ungünstigen Eingabefolgen mehr. Theoretisch gibt es sie natürlich immer noch, das heißt der worst case wird nicht verbessert; solche Folgen treten aber nicht mit erhöhter Wahrscheinlichkeit auf, wie etwa eine sortierte Folge.

Ein noch besseres Verhalten ergibt sich mit einer Variante, die wir *Clever Quicksort* nennen wollen; dabei werden zufällig drei verschiedene Elemente der Folge ausgewählt und das Element mit dem *mittleren* Wert als Split-Element benutzt. Bei diesem Verfahren tendiert der Split-Wert dazu, eher in der Mitte zu liegen. Die Analyse einer entsprechenden Rekursionsgleichung gelang Kemp [Kemp 1989]; es ergibt sich eine erwartete Anzahl von Vergleichen [Wegener 1990b]:

$$C(n) \approx 1.188n \log(n-1) - 2.255n + 1.188 \log(n-1) + 2.507$$

Diese Anzahl von Vergleichen liegt nur um 18.8% über dem optimalen Wert (siehe Abschnitt 5.4). Clever Quicksort ist damit ein hervorragendes und auch in der Praxis beliebtes Verfahren.

Quicksort hat noch ein Problem: der Platzbedarf für den Rekursionsstack kann  $O(n)$  werden. Man kann Quicksort aber so modifizieren, dass dieser zusätzliche Platzbedarf nur  $O(\log n)$  beträgt (Aufgabe 5.4).

**Selbsttestaufgabe 5.3:** Verfolgen Sie die Berechnung von Quicksort für die Eingabefolge 17, 24, 3, 1, 12, 7, 9, 5, 2, 0, 24, 42, 49, 46, 11 und geben Sie die Arrayausprägung nach jedem Sortierschritt an. □

**Selbsttestaufgabe 5.4:** Beschreiben Sie die Modifikationen, die in der Methode *quickSort* vorzunehmen sind, damit der Stack nur logarithmisch wächst. □

### 5.3 Verfeinertes Auswählen und Einfügen: Heapsort und Baumsortieren

Die grundlegenden Strategien für das Sortieren durch Auswählen (SelectionSort) und Einfügen (InsertionSort) wurden schon in Abschnitt 5.1 besprochen. Die kritischen Operationen, d. h. die Auswahl des Minimums bzw. das Einfügen in eine geordnete Folge, wurden dort jeweils in  $O(n)$  Zeit realisiert. Das führte zu  $O(n^2)$ -Algorithmen. Die Verfahren dieses Abschnitts unterstützen diese Operationen jeweils durch Einsatz geeigneter Datenstrukturen. *Heapsort* benutzt einen partiell geordneten Baum bzw. Heap (siehe Abschnitt 4.3), um das Minimum in  $O(\log n)$  Zeit zu finden und zu entnehmen. Beim *Baumsortieren* wird das Einfügen in eine geordnete Folge durch einen AVL-Baum realisiert; nachdem alle Elemente eingefügt sind, liefert ein Inorder-Durchlauf die sortierte Ergebnisfolge. Beide Verfahren erreichen somit eine worst-case Laufzeit von  $O(n \log n)$ . Da Baumsortieren auf recht komplexe dynamische Datenstrukturen angewiesen ist (neben dem AVL-Baum eignen sich auch viele andere balancierte Baumstrukturen), ist es in der Praxis nicht so interessant. Man sollte sich aber darüber im Klaren sein, dass dies eine asymptotisch optimale Sortiermethode ist (wir werden im nächsten Abschnitt zeigen, dass  $\Omega(n \log n)$  eine untere Schranke für die worst-case Komplexität allgemeiner Sortierverfahren ist).

#### Standard-Heapsort

Wegen der effizienten Implementierbarkeit eines partiell geordneten Baumes im Array ist Heapsort auch in der Praxis interessant; es ist das beste bekannte Sortierverfahren, das  $O(n \log n)$  im worst-case garantiert. Die Grundidee sollte bereits klar sein:

1. Die  $n$  zu sortierenden Elemente werden in einen Heap eingefügt: Komplexität  $O(n \log n)$ .
2. Es wird  $n$ -mal das Minimum aus dem Heap entnommen: Komplexität  $O(n \log n)$ .

Wir verfeinern diese Idee noch etwas, um den Aufbau des Heap zu beschleunigen und um in situ sortieren zu können. Sei wieder  $S[i..k]$  eine im Array implementierte Folge von Records, die eine *key*-Komponente eines linear geordneten Datentyps besitzen.

**Definition 5.5:** Ein Teilarray  $S[i..k]$ ,  $1 \leq i \leq k \leq n$ , heißt *Teilheap*

$$\Leftrightarrow \forall j \in [i, \dots, k]: \quad S[j].key \leq S[2j].key \quad \text{falls } 2j \leq k \\ \text{und} \quad S[j].key \leq S[2j+1].key \quad \text{falls } 2j+1 \leq k$$

Wenn  $S[1..n]$  ein Teilheap ist, dann ist  $S[1..n]$  auch ein Heap (das heißt die Array-Einbettung eines partiell geordneten Baumes). Der Ablauf von Heapsort lässt sich dann so beschreiben:

1. Aufbau des Heap:  $S[\lfloor n/2 \rfloor + 1..n]$  ist bereits ein Teilheap (weil diese Knoten keine Söhne mehr haben und somit die Bedingung trivialerweise erfüllt ist).

```
for  $i := \lfloor n/2 \rfloor$  downto 1 do
    erweitere den Teilheap  $S[(i+1)..n]$  zu einem Teilheap  $S[i..n]$  durch "Einsinken
    lassen" von  $S[i]$ 
end for
```

2. Baue die sortierte Folge rückwärts am hinteren Ende des Arrays auf (wenn die sortierte Folge nachher aufsteigend geordnet im Array stehen soll, verwende man an Stelle eines Minimum-Heaps einen Maximum-Heap).

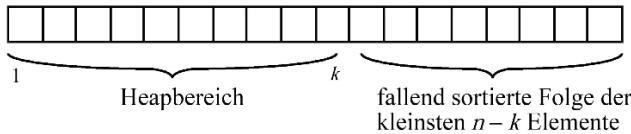


Abbildung 5.6: Heapsort

Dazu wird in jedem Schritt das Minimum  $S[1]$  mit  $S[k]$  vertauscht und dadurch der Heap-Bereich auf  $S[1..k-1]$  reduziert.  $S[2..k-1]$  ist weiterhin ein Teilheap. Durch Einsinken von  $S[1]$  wird  $S[1..k-1]$  wieder zu einem (Teil-) Heap.

Wir benutzen eine Methode *reheap* ( $S, i, k$ ), die  $S[i]$  einsinken lässt, um aus einem Teilheap  $S[(i+1)..k]$  einen Teilheap  $S[i..k]$  zu machen. Da Arrays in Java mit dem Index 0 beginnen, verringern wir bei Arrayzugriffen in der Implementierung den Index aus dem Algorithmus jeweils um 1. Diese Stellen sind durch Unterstreichung kenntlich gemacht.

```

private static void reheap(Elem[] S, int i, int k)
{
    int j = i; int son;
    boolean endloop = false;
    do
    {
        if(2 * j > k) break;
        else
        {
            if(2 * j + 1 <= k)
                if(S[2 * j - 1].isLess(S[2 * j + 1 - 1])) son = 2 * j;      // 1. Vergleich
                else son = 2 * j + 1;
            else /* 2 * j = k */ son = 2 * j;
            // son ist der Sohn, mit dem evtl. zu vertauschen ist.
            if(S[son - 1].isLess(S[j - 1]))                                // 2. Vergleich
            {
                swap(S, j - 1, son - 1); j = son;
            }
            else { endloop = true; }
        }
        while(!endloop);
    }
}

```

*Reheap* braucht zwei Schlüsselvergleiche auf jeder Ebene des Baumes, also maximal  $2 \cdot \log n$  Vergleiche. Der Gesamtaufwand für *reheap* ist natürlich  $O(\log n)$ .

```

public static void heapSort(Elem[] S)
{
    int n = S.length;
    for(int i = n / 2; i >= 1; i--) reheap(S, i, n);      // Phase 1
    for(int i = n; i >= 2; i--)
    {
        swap(S, 1 - 1, i - 1);
        reheap(S, 1, i - 1);
    }
}

```

### Analyse von Heapsort

In Phase (1) (Aufbau des Heap) wird die Schleife  $(n/2)$ -mal durchlaufen, jeweils mit Aufwand  $O(\log n)$ . In Phase (2) (Auswahlphase) wird die Schleife  $(n-1)$ -mal durchlaufen. Also ist der Aufwand im *worst case*  $O(n \log n)$ .

Bei genauerem Hinsehen entdecken wir, dass der Aufwand in Phase (1) nur  $O(n)$  ist. Wir notieren dazu den Aufwand für *reheap* an jedem Knoten des in den Array eingebetteten Baumes (der zu Anfang ja noch kein Heap ist), wobei wir als Einheit einen Schleifen-durchlauf benutzen:

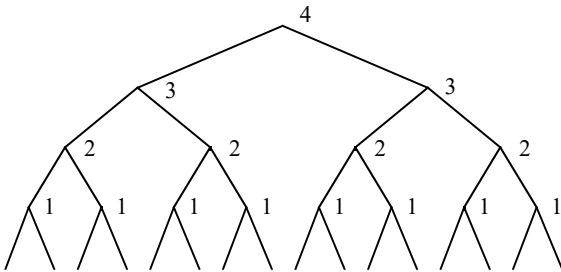


Abbildung 5.7: Aufwand für *reheap*

Offensichtlich ist dieser Aufwand im Allgemeinen

$$\frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \dots$$

somit ist

$$\frac{n}{2} \cdot \underbrace{\sum_{i=0}^{\infty} \frac{i}{2^i}}_{=2}$$

eine obere Schranke. Dass die Summe den Wert 2 hat, ist in Aufgabe 5.5 zu zeigen. Diese Beobachtung hat eine interessante Anwendung: Da der Heap-Aufbau nur  $O(n)$  Zeit braucht, kann man *Heapsort* dazu benutzen, um in  $O(n + k \log n)$  die  $k$  kleinsten Elemente einer Menge in Sortierreihenfolge zu erhalten. Wenn  $k \leq n/\log n$  ist, kann dieses Problem also in  $O(n)$  Zeit gelöst werden.

Da *reheap* auf jeder Ebene des Heap-Baumes zwei Vergleiche benötigt, ist die Gesamtzahl der Vergleiche

$$C_{\text{worst}}(n) = 2 \cdot n \cdot \log n + O(n)$$

Wie sieht es im Durchschnitt aus? Wir machen keine präzise Analyse, aber beobachten, dass in einem vollständigen binären Baum 50% der Knoten auf der untersten Ebene liegen, 25% auf der vorletzten Ebene usw. Wenn ein einsinkendes Element auf allen Positionen des Heap mit gleicher Wahrscheinlichkeit landet, dann durchläuft es im

Durchschnitt fast einen kompletten Pfad bis zu einem Blatt. Daher liegt die durchschnittliche Anzahl der Vergleiche nahe bei

$$2 \cdot n \cdot \log n + O(n)$$

Das ist der Grund für das schlechte Durchschnittsverhalten von Heapsort im Vergleich zu Quicksort, das ja im Durchschnitt nur

$$\approx 1,386 \cdot n \cdot \log n + O(n)$$

Vergleiche benötigt.

In der Methode *Reheap* werden auf jeder Ebene mit den zwei Vergleichen zwei Entscheidungen getroffen:

1. Welcher Pfad ist im Heap zu verfolgen (also mit welchem Sohn des aktuellen Knotens ist der Pfad fortzusetzen)?
2. Muss das bearbeitete Element noch tiefer einsinken?

Wir haben gerade gesehen, dass die zweite Entscheidung fast immer positiv ausfällt.

### Bottom-Up-Heapsort

Eine Heapsort-Variante wurde von Wegener [1990a, 1990b] vorgeschlagen. Die Idee des Bottom-Up-Heapsort besteht darin, die beiden Entscheidungen zu trennen und *reheap* so zu organisieren:

1. Durchlaufe den Heap top-down, um einen Pfad zu bestimmen, auf dem das einsinkende Element  $e$  irgendwo liegen bleiben muss. Dazu sind in jedem Knoten lediglich die beiden Söhne zu vergleichen und der kleinere auszuwählen. Dieser Pfad heiße *Zielpfad*; er endet in einem Blatt  $b$ .
2. Durchlaufe den Zielpfad von  $b$  aus bottom-up und vergleiche jeden Schlüssel mit  $e$ . Abbruch, sobald ein Schlüssel  $q < e$  gefunden wird.
3. Entnimm das Wurzelement  $e$ ; schiebe alle Elemente auf dem Zielpfad bis hin zu  $q$  um eine Position nach oben; füge  $e$  auf der alten Position von  $q$  ein.

**Beispiel 5.6:** Abbildung 5.8 zeigt das Einsinken eines Elementes 14 mit der neuen *reheap*-Strategie.  $\square$

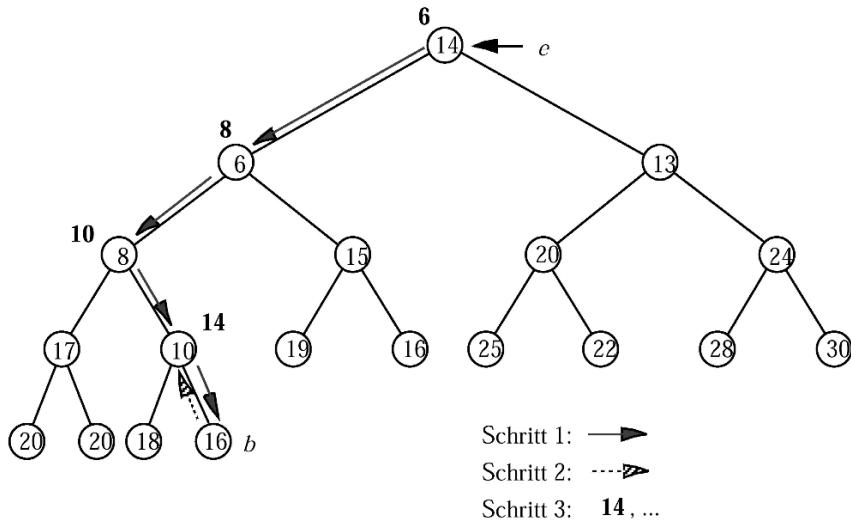


Abbildung 5.8: Einsinken bei Bottom-Up-Heapsort

Das lässt sich so implementieren:

```

private static void reheap (Elem [] S, int i, int k)
{
    /* Schritt 1: */
    int j = i; Elem r, t;
    while(2 * j < k)
        if(S[2 * j - 1].isLess(S[2 * j + 1 - 1])) j = 2 * j; // Vergleich
        else j = 2 * j + 1;
        if(2 * j == k) j = k; // j entspricht nun b

    /* Schritt 2: */
    while(S[i - 1].isLess(S[j - 1])) j = j / 2; // Vergleich; j entspricht q

    /* Schritt 3: */
    r = S[j - 1]; S[j - 1] = S[i - 1]; j = j / 2;
    while(j >= i)
    {
        t = r; r = S[j - 1]; S[j - 1] = t; // vertausche S[j - 1] mit r
        j = j / 2;
    }
}

```

Der Vorteil gegenüber Standard-Heapsort besteht darin, dass die Schleife in Schritt 2 meist nach einem oder zwei Durchläufen abbricht; die Anzahl der Vergleiche für Heapsort rückt damit im Durchschnitt viel näher an

$$\begin{aligned} 1 \cdot n \cdot \log n + O(n) \text{ als an} \\ 2 \cdot n \cdot \log n + O(n). \end{aligned}$$

Experimente bestätigen die Analysen und zeigen, dass Bottom-Up-Heapsort besser ist als Standard-Quicksort, falls  $n \geq 400$  und besser als Clever Quicksort, falls  $n \geq 16000$ . Hinzu kommt das gute  $O(n \log n)$  worst-case Verhalten von Heapsort gegenüber  $O(n^2)$  bei Quicksort.

**Selbsttestaufgabe 5.5:** Beweisen Sie die in der Analyse von Heapsort verwendete Gleichung

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$$

Orientieren Sie sich dabei an der in Grundlagen I des Anhangs ‘‘Mathematische Grundlagen’’ gezeigten Technik.  $\square$

## 5.4 Untere Schranke für allgemeine Sortierverfahren

Wir haben nun verschiedene Sortieralgorithmen kennengelernt, die eine Laufzeit von  $O(n \log n)$  haben. Es erhebt sich die Frage, ob man noch viel Mühe investieren soll, um bessere Algorithmen zu finden. Gibt es vielleicht ein Verfahren, das in  $O(n)$  sortiert? Mit anderen Worten, wir suchen nach einer unteren Schranke für die Komplexität des Sortierproblems.

$\Omega(n)$  ist offensichtlich eine untere Schranke, da man sicherlich jeden Schlüsselwert einmal ansehen muss. Gibt es eine höhere untere Schranke?

Um das zu untersuchen, müssen wir etwas präziser festlegen, was wir unter einem ‘‘allgemeinen’’ Sortierverfahren verstehen wollen. Ein allgemeines Verfahren soll in der Lage sein, eine beliebige Menge von Werten über einem linear geordneten Wertebereich zu sortieren, also etwa Zeichenketten, reelle Zahlen usw. Wir nehmen daher an, dass ein allgemeines Verfahren nur eine einzige Möglichkeit hat, auf Schlüsselwerte zuzugreifen, nämlich einen Vergleichsoperator  $<$  (oder  $\leq$ ) auf sie anzuwenden. Weitere Information über die interne Struktur der Werte sei nicht vorhanden. Wir können uns vorstellen, der Schlüsselwertebereich sei ein ‘‘abstrakter’’ Datentyp, der nur eine Vergleichsoperation nach außen hin anbietet. Allgemeine Verfahren in diesem Sinne heißen *Schlüsselver-*

*gleichs*-Verfahren. Wir wollen nun eine untere Schranke für sämtliche (bekannten oder unbekannten) Algorithmen in dieser Klasse bestimmen.

Sei  $A$  ein beliebiger Algorithmus aus dieser Klasse. Nehmen wir an,  $A$  werde auf eine Folge  $S = s_1 \dots s_n$  angesetzt (wir nehmen zur Vereinfachung an, dass sämtliche Schlüssel in  $s_1 \dots s_n$  verschieden sind). Wir „beobachten“ nun während des Ablaufs lediglich die durchgeföhrten Vergleiche und ignorieren alle anderen Operationen. Sobald  $A$  terminiert, hat er eine Permutation der Ausgangsfolge hergestellt, nämlich gerade die, die  $S$  in die sortierte Folge überführt. Wir notieren noch, welche Permutation erzeugt worden ist. Das Verhalten von  $A$ , angesetzt auf diese spezielle Folge  $S$ , lässt sich dann z. B. so notieren:

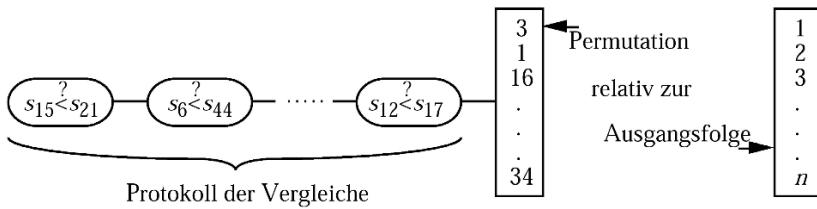


Abbildung 5.9: Schlüsselvergleichs-Verfahren

Wir machen noch die weitere Annahme, dass der Ablauf von  $A$  für festes  $n$  nur von den Ergebnissen der durchgeföhrten Schlüsselvergleiche abhängt. Dann lässt sich das Verhalten von  $A$  für sämtliche Folgen der Länge  $n$  in Form eines *Entscheidungsbaumes* darstellen. Der Baum beschreibt einfach die Menge aller möglichen Abläufe von  $A$ ; jeder Ablauf entspricht einem Pfad.

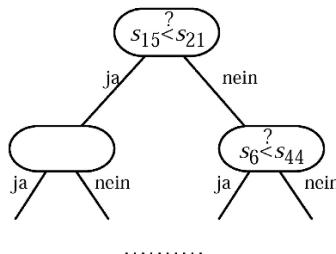


Abbildung 5.10: Entscheidungsbaum beim Schlüsselvergleichs-Verfahren

Ein vollständiger Entscheidungsbaum, der das Verhalten irgendeines konkreten Algorithmus für  $n = 3$  wiedergibt, könnte etwa so aussehen, wie in Abbildung 5.11 gezeigt:

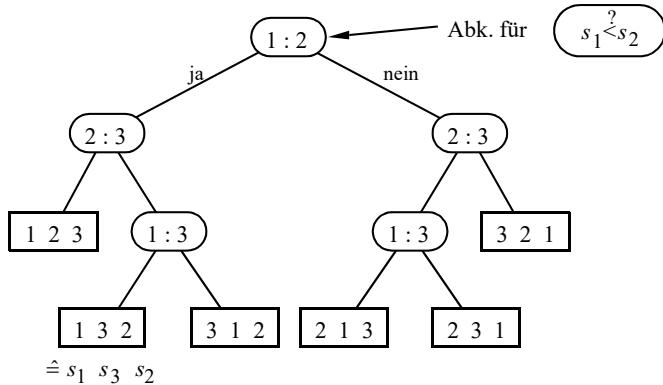


Abbildung 5.11: Vollständiger Entscheidungsbaum (Beispiel)

Die Eingabefolge  $s_1 \dots s_n$  ist eine beliebige Permutation der geordneten Folge  $s_{i_1} \dots s_{i_n}$ ; um die Eingabefolge in die geordnete Folge zu überführen, muss ein Sortieralgorithmus daher in der Lage sein, sämtliche Permutationen der Eingabefolge zu erzeugen bzw. über den Entscheidungsbaum anzusteuern. Es gibt  $n!$  Permutationen, deshalb hat der Entscheidungsbaum für Folgen der Länge  $n$  genau  $n!$  Blätter.

Es mag sein, dass ein Algorithmus überflüssige Vergleiche durchführt; das führt zu Knoten im Entscheidungsbaum, die nur einen Sohn haben, da der andere Zweig niemals erreicht wird. So gilt z. B. in Abbildung 5.12  $s_1 < s_2 \wedge s_2 < s_3 \Rightarrow s_1 < s_3$ , daher kann der gezeigte Zweig nicht erreicht werden.

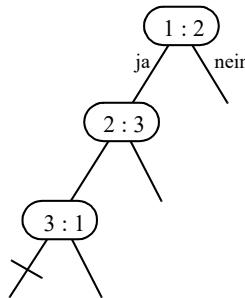


Abbildung 5.12: Unerreichbarer Zweig des Entscheidungsbaumes

Es folgt:

**Lemma 5.7:** Ein Entscheidungsbaum ohne redundante Vergleiche für Folgen der Länge  $n$  ist ein binärer Baum mit genau  $n!$  Blättern.  $\square$

**Satz 5.8:** Jeder Schlüsselvergleichs-Sortieralgorithmus benötigt  $\Omega(n \log n)$  Vergleiche im worst-case.

**Beweis:** Der Entscheidungsbaum für diesen Algorithmus ist ein binärer Baum mit  $n!$  Blättern, also mit mindestens  $2 \cdot n! - 1$  Knoten insgesamt. Um eine untere Schranke für das Verhalten des Algorithmus im schlimmsten Fall zu bestimmen, müssen wir *balancede* Entscheidungsbäume betrachten, denn in diesen ist die Länge des längsten Pfades (entsprechend dem schlimmsten Fall) minimal. Die minimale Höhe eines Baumes mit  $m$  Knoten ist (Abschnitt 3.5)

$$h = \lfloor \log m \rfloor$$

Also ist in diesem Fall die minimale Höhe

$$\begin{aligned} h &= \lfloor \log(2n!-1) \rfloor \\ &= 1 + \lfloor \log(n!-1/2) \rfloor \\ &= \lceil \log(n!-1/2) \rceil \\ &= \lceil \log n! \rceil \end{aligned}$$

Es gibt daher einen Pfad mit  $h$  Kanten, also mit genau  $h$  Vergleichen. Wir schätzen den Wert von  $h$  ab:

$$\begin{aligned} n! &\geq \left(\frac{n}{2}\right)^{n/2} \\ \log n! &\geq \frac{n}{2} \log \frac{n}{2} \\ &= \Omega(n \log n) \end{aligned}$$

Man kann  $h$  präziser bestimmen mit der *Stirling'schen Formel* (siehe z. B. Graham *et al.* [1994]), die besagt:

$$n! \approx \sqrt{2\pi \cdot n} \cdot \left(\frac{n}{e}\right)^n$$

Damit kann man ausrechnen:

$$\begin{aligned} \ln n! &\approx \ln \sqrt{2\pi \cdot n} + \ln \sqrt{n} + n \ln n - n \ln e \\ &= n \ln n - n + \frac{\ln n}{2} + O(1) \\ \log n! &\approx n \log n - \frac{n}{\ln 2} + \frac{\log n}{2} + O(1) \quad (\text{wegen } \log_2 x = \frac{\ln x}{\ln 2}) \end{aligned}$$

Also ist die minimale Anzahl der Vergleiche  $n \cdot \log n - O(n)$ . □

## 5.5 Sortieren durch Fachverteilten: Bucket sort und Radixsort

Wenn die Schlüsselwerte bestimmten Einschränkungen unterliegen und auch andere Operationen als Vergleiche angewandt werden können, so ist es möglich, schneller als in  $\Omega(n \log n)$  Zeit zu sortieren, z. B. in  $O(n)$ .

**Beispiel 5.9:** Sei  $S = s_1 \dots s_n$ , sei  $keytype = 1..n$ , und es sei bekannt, dass in  $S$  keine Duplikate vorkommen. Sei  $S$  wieder im Array repräsentiert. Wir benutzen einen zweiten Array gleicher Art, also

**var**  $S, T$ : array[1..n] **of** recordtype

Dann erzeugt die Anweisung

```
for  $i := 1$  to  $n$  do
     $T[S[i].key] := S[i]$ 
end for;
```

die entsprechende sortierte Folge in  $T$  und braucht dazu nur  $O(n)$  Zeit.  $\square$

Diese Idee lässt sich etwas verallgemeinern. Sei nun  $keytype = 0..m-1$  und seien Duplikate erlaubt. Wir benutzen eine Menge von Behältern (Buckets)  $B_0, \dots, B_{m-1}$ . Jedes Bucket lässt sich als Liste von Records implementieren; die Buckets werden über einen Array verwaltet. Die Datenstruktur ist also genau die gleiche wie beim offenen Hashing (vgl. Abschnitt 4.2.2, Abbildung 4.2).

```
algorithm BucketSort ( $S$ )
  1. for  $i := 0$  to  $m-1$  do  $B_i := \emptyset$  end for;
  2. for  $i := 1$  to  $n$  do
    füge  $s_i$  in  $B_{s_i.key}$  ein
    end for;
  3. for  $i := 0$  to  $m-1$  do
    schreibe die Elemente von  $B_i$  in die Ergebnisfolge
    end for
end BucketSort;
```

Das Einfügen eines Records in ein Bucket ist in  $O(1)$  Zeit möglich. Also braucht BucketSort  $O(n + m)$  Zeit, wenn  $m = O(n)$ , somit  $O(n)$  Zeit.

Man kann sogar noch weiter verallgemeinern: Sei  $keytype = 0..n^k-1$ . Direkte Anwendung von BucketSort hätte dann eine Laufzeit von  $O(n + n^k) = O(n^k)$ . Wir können nun aber BucketSort in mehreren Phasen anwenden und noch immer eine Laufzeit von  $O(n)$  erreichen (bzw.  $O(k \cdot n)$ , wenn wir  $k$  nicht als Konstante auffassen). Sei z. B.  $keytype = 0..n^2-1$  und sei  $m = n$ , das heißt, wir benutzen Buckets  $B_0, \dots, B_{n-1}$ . Dann werden zwei Sortierphasen durchgeführt:

1. Phase: BucketSort, wobei  $s_i$  in  $B_j$  eingefügt wird mit  $j = s_i.key \bmod n$ .
2. Phase: Durchlaufe für BucketSort die Ergebnisliste der ersten Phase und hänge jeweils  $s_i$  an  $B_j$  an mit  $j = s_i.key \bmod n$ .

**Beispiel 5.10:** Sei  $n = 10$ , die Werte aus dem Bereich  $0..99$ . Zu sortieren sei die Folge

3 18 24 6 47 7 56 34 98 60

Nach dem gerade geschilderten Verfahren ergeben sich folgende Sortierphasen:

1. Phase:	$B_0$	$B_1$	$B_2$	$B_3$	$B_4$	$B_5$	$B_6$	$B_7$	$B_8$	$B_9$
	60		3	24		6	47	18		
			34		56		7	98		

Ergebnisfolge: 60 3 24 34 6 56 47 7 18 98

2. Phase:  $B_0 \ B_1 \ B_2 \ B_3 \ B_4 \ B_5 \ B_6 \ B_7 \ B_8 \ B_9$   
           3   18   24   34   47   56   60                 98  
           6  
           7

Ergebnisfolge: 3 6 7 18 24 34 47 56 60 98

□

Es ist hierbei wesentlich für alle Phasen außer der ersten, dass Records ans Ende einer Bucketliste angefügt werden. Man benutze eine Listenimplementierung, die Anhängen ans Ende in  $O(1)$  Zeit erlaubt.

Offensichtlich fasst das beschriebene Verfahren Schlüsselwerte als Ziffernfolgen zur Basis  $m$  auf und sortiert in jeder Phase nach einer Ziffer, nämlich zuerst nach der letzten Ziffer, dann nach der vorletzten usw. Diese Technik heißt *Radixsortieren*. Es ist dabei nicht notwendig, dass alle “Ziffern” über dem gleichen Wertebereich gebildet werden, man kann auch in jeder Phase einen anderen Array zur Organisation der Buckets verwenden.

**Beispiel 5.11:** Eine Menge von Datensätzen könnte z. B. folgende Datumsdarstellung beinhalten:

```
type Satz = record Jahr : 1900..2099;
               Monat : 1..12;
               Tag : 1..31; ... (weitere Information)
end
```

Solche Records kann man etwa mit Radixsort in drei Phasen nach Datum sortieren, zuerst nach Tag, dann nach Monat, dann nach Jahr. □

Wenn Radixsort  $n$  Records zu sortieren hat, deren Schlüssel jeweils Tupel der Form  $(t_1, t_2, \dots, t_k)$  sind mit  $t_i \in T_i$  und  $|T_i|$  die Kardinalität des Wertebereichs  $T_i$  angibt, dann ist der Gesamtaufwand

$$O\left(k \cdot n + \sum_{i=1}^k |T_i|\right)$$

Wenn etwa gilt  $|T_i| \leq n \ \forall i$  und wir  $k$  als Konstante auffassen, dann ist der Gesamtaufwand  $O(n)$ .

Radixsortieren eignet sich z. B. recht gut, wenn die Schlüssel Zeichenketten fester, geringer Länge sind; wir fassen dann jeden Buchstaben als eine Ziffer zur Basis 26 auf.

**Selbsttestaufgabe 5.6:** Was für ein Sortierverfahren würden Sie vorschlagen, wenn bekannt ist, dass die Eingabefolge zum größten Teil sortiert vorliegt und nur an deren Ende einige wenige unsortierte Werte angehängt sind? Begründen Sie Ihr Verfahren.  $\square$

## 5.6 Weitere Aufgaben

**Aufgabe 5.7:** Ein weiteres Sortierverfahren *Shellsort* ist in der folgenden Methode *shell-sort* realisiert.

```
public static void shellsort(Elem[] S)
{
    int j;
    int incr = S.length / 2;
    while(incr > 0)
    {
        for(int i = incr; i < S.length; i++)
        {
            j = i - incr;
            while(j >= 0)
            {
                if(S[j + incr].isLess(S[j]))
                {
                    swap(S, j, j + incr);
                    j = j - incr;
                }
                else j = -1;
            }
            incr = incr / 2;
        }
    }
}
```

- (a) Zeigen Sie an einem Beispiel, wie das Verfahren arbeitet.
- (b) Begründen Sie, warum Shellsort auch dann noch korrekt arbeitet, wenn im angegebenen Algorithmus eine andere monoton fallende Folge für die Werte von *incr* verwendet wird, solange das letzte Glied dieser Folge 1 ist.

**Aufgabe 5.8:** Sind Quicksort und Mergesort stabil? Ist das oben gezeigte Shellsort stabil? Begründen Sie Ihre Antworten.

**Aufgabe 5.9:** Einer der beiden in Aufgabe 5.8 angegebenen Divide-and-Conquer-Algorithmen ist nicht stabil. Dieses Verhalten lässt sich beheben, indem man vor dem Sortie-

ren eine Schlüsseltransformation anwendet, die *alle* Schlüssel verschieden macht, diese neuen Schlüssel sortiert und anschließend eine Rücktransformation anwendet. Ursprünglich gleiche Schlüssel werden verschieden gemacht, indem sie anhand ihrer Position unterschieden werden. Als Werte für die Schlüssel seien alle positiven natürlichen Zahlen ohne Null zulässig.

- (a) Geben Sie für den fraglichen Algorithmus eine Transformationsfunktion an.
- (b) Geben Sie zu Ihrer Transformation eine Rücktransformation an, die nach der Sortierung angewendet werden kann, um die ursprünglichen Schlüsselwerte zu erhalten.

**Aufgabe 5.10:** Nehmen Sie an, die einzigen Werte, die als Schlüsselwerte auftreten, sind 0 und 1. Wie wirkt sich dies auf das worst-case-Verhalten von InsertionSort aus, wie auf das von Quicksort?

- (a) Geben Sie für beide Algorithmen eine Eingabe an, die den worst case beschreibt und geben Sie die Laufzeit in diesen Fällen an.
- (b) Geben Sie einen Algorithmus an, der die Sortierung in  $O(n)$  herstellt.

**Aufgabe 5.11:** In einem Array seien Records mit den Schlüsselwerten rot, grün und blau enthalten. Es soll eine Folge dieser Records hergestellt werden, in der am Anfang alle roten, dann alle grünen, zuletzt alle blauen Schlüssel vorkommen. Die einzigen erlaubten Zugriffsoperationen auf den Array sind dabei das Feststellen des Schlüsselwerts eines Records und das Vertauschen zweier Records. Formulieren Sie einen Algorithmus, der den Array *in situ* in  $O(n)$  Zeit wie angegeben sortiert. Zeigen Sie, dass Ihr Algorithmus in  $O(n)$  Schritten terminiert und dass der Array dann tatsächlich sortiert ist.

**Aufgabe 5.12:** Ein extrem einfacher Sortieralgorithmus beruht auf der Idee, für jedes Element in der Eingabefolge auszuzählen, wieviele Schlüsselwerte kleiner sind als sein eigener Schlüsselwert. Hierfür enthalte die Recordstruktur der Eingabeelemente ein zusätzliches Feld *count*. Sie können annehmen, dass keine Duplikate von Schlüsselwerten in der Eingabe vorkommen.

- (a) Formulieren Sie einen solchen Algorithmus.
- (b) Vergleichen Sie seine worst-case-Laufzeit mit der anderer Sortierverfahren.

## 5.7 Literaturhinweise

Ein Kompendium für Sortierverfahren ist das Kapitel 5 des Buches von Knuth [1998]; dort werden auch viele Varianten nicht-optimaler Verfahren (z. B. BubbleSort, Shaker-Sort, ShellSort) eingehend untersucht. Dieses Buch bietet ebenfalls einen Abriss der

Geschichte von Sortierverfahren, die bis ins 19. Jahrhundert zurückreicht (Holleriths Sortiermaschinen für Lochkarten). Die ersten Computerprogramme zum Sortieren wurden wohl von John von Neumann 1945 eingesetzt; er benutzte interne Mischsortiertechniken (siehe [Knuth 1998]).

Sortieren durch direktes Einfügen (mit binärer Suche, wie in Aufgabe 5.2) wurde schon von Steinhaus [1958] beschrieben. ShellSort (Aufgabe 5.7) wurde nach seinem Erfinder benannt [Shell 1959]. MergeSort-Varianten, die *in situ* sortieren, wurden von Kronrod [1969] und Pardo [1977] entwickelt. Diese Techniken wurden, insbesondere in praktischer Hinsicht, weiter verbessert von Huang und Langston [1988] und von Dvorak und Durian [1988].

Quicksort stammt von Hoare [1962], eine detaillierte Studie findet sich in der Dissertation von Sedgewick [1978]. In manchen Anwendungen treten viele gleiche Schlüssel (Duplikate) in einer zu sortierenden Folge auf, deshalb ist man an Sortieralgorithmen interessiert, die dies ausnutzen und möglichst einen “gleitenden” Übergang in ihrer Komplexität vom Spezialfall nur gleicher Schlüssel ( $O(n)$ ) zum allgemeinen Fall ( $O(n \log n)$ ) erzielen. Derartige Quicksort-Varianten wurden von Wegner [1985] studiert; man kann z. B. erreichen, dass die mittlere Laufzeit bei  $O(n \log N + n)$  liegt, wobei  $N$  die Anzahl verschiedener Schlüssel in der Eingabefolge ist.

Der klassische Heapsort-Algorithmus ist von Williams [1964]; die beschriebene Strategie des Heap-Aufbaus in  $O(n)$  Zeit stammt von Floyd [1964]. Durchschnittsanalysen für Heapsort gelangen Doberkat [1984]. Eine Variante, die Vorsortierung ausnutzt, wurde von Dijkstra [1982] entwickelt, sie braucht nur  $O(n)$  Zeit für eine bereits sortierte Folge und  $O(n \log n)$  für eine beliebige Folge.

Bottom-Up-Heapsort wurde von Wegener [1990a, 1990b] als Verbesserung einer Idee von Carlsson [1987] entwickelt. Carlsson schlug bereits vor, die Bestimmung des Zielpfades von der Entscheidung zu trennen, an welcher Stelle das einsinkende Element bleiben soll; er verwendet dann binäre Suche auf dem gefundenen Zielpfad, was aber der Bottom-Up-Strategie unterlegen ist.

Die Bestimmung einer unteren Schranke für allgemeine Sortierverfahren mithilfe eines Entscheidungsbaumes stammt bereits von Ford und Johnson [1959]. Derartige Techniken zur Bestimmung unterer Schranken sind inzwischen verallgemeinert worden. So betrachtet Schmitt [1983] *rationale Entscheidungsbäume*, in denen die Entscheidungen in den Knoten vom Vergleich der Werte rationaler Funktionen über den Schlüsselwerten abhängen können; eine rationale Funktion ist mithilfe der arithmetischen Operationen  $+$ ,  $-$ ,  $*$ ,  $/$  aufgebaut. Eine weitere Verallgemeinerung sind *algebraische Entscheidungsbäume* [BenOr 1983].

Techniken des Sortierens durch Fachverteilen wie BucketSort und RadixSort werden ebenfalls in [Knuth 1998] behandelt.

## 6 Graphen

Ein *Graph* stellt eine Menge von Objekten zusammen mit einer Beziehung (Relation) auf diesen Objekten dar. Wir betrachten einige Beispiele:

- (a) Objekte: Personen; Beziehung: Person  $A$  kennt Person  $B$ .
- (b) Spieler eines Tennisturniers;  $A$  spielt gegen  $B$ .
- (c) Städte; es gibt eine Autobahn zwischen  $A$  und  $B$ .
- (d) Stellungen im Schachspiel; Stellung  $A$  lässt sich durch einen Zug in Stellung  $B$  überführen.

Die Bezeichnung Graph stammt von der üblichen “graphischen” Darstellung: Objekte werden als Knoten, Beziehungen als Kanten dargestellt. Eine Beziehung wird im Allgemeinen durch eine *gerichtete* Kante (einen Pfeil) dargestellt. Der Spezialfall einer *symmetrischen* Beziehung  $R$ , das heißt,  $aRb \Rightarrow bRa$ , wird durch eine *ungerichtete* Kante dargestellt. Entsprechend unterscheidet man *gerichtete* und *ungerichtete* Graphen.

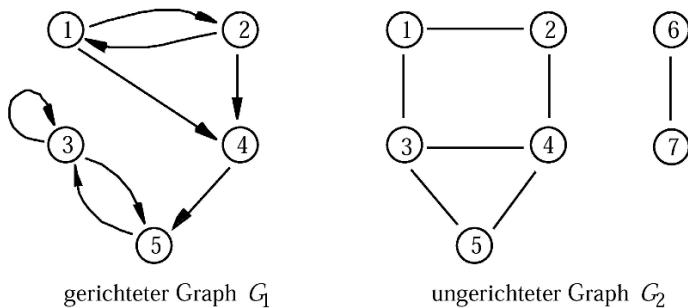


Abbildung 6.1: Beispiele für Graphen

In den obigen Beispielen (b) und (c) sind die Beziehungen symmetrisch und können durch ungerichtete Graphen dargestellt werden, in (a) und (d) hingegen nicht. Da eine symmetrische Beziehung ein Spezialfall einer Beziehung ist, lässt sich ein ungerichteter Graph immer durch einen gerichteten Graphen darstellen, z. B. ist  $G_3$  eine gerichtete Darstellung des obigen ungerichteten Graphen  $G_2$ .

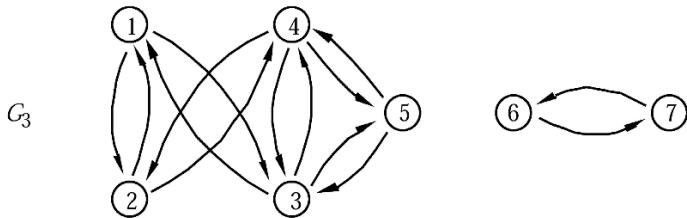


Abbildung 6.2: Gerichtete Darstellung eines ungerichteten Graphen

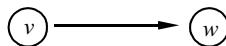
Das Beispiel zeigt auch, dass die Platzierung von Knoten und Kanten in der graphischen Darstellung natürlich keine Rolle spielt. Das heißt, es gibt viele verschiedene Darstellungen desselben Graphen. Wir führen in Abschnitt 6.1 zunächst die Grundbegriffe anhand gerichteter Graphen ein. Abschnitt 6.2 zeigt die wichtigsten Speicherdarstellungen für Graphen, und Abschnitt 6.3 behandelt Techniken, um Graphen systematisch zu durchlaufen.

## 6.1 Gerichtete Graphen

**Definition 6.1:** Ein *gerichteter Graph (Digraph = directed graph)* ist ein Paar  $G = (V, E)$ , wobei gilt:

- (i)  $V$  ist eine endliche, nichtleere Menge (die Elemente werden *Knoten* genannt),
- (ii)  $E \subseteq V \times V$  (die Elemente heißen *Kanten*).

Eine Kante ist also ein Paar von Knoten; Kante  $(v, w)$  wird graphisch wie folgt dargestellt:

Abbildung 6.3: Kante  $(v, w)$ 

Man sagt dann, Kante  $(v, w)$  ist *inzident* mit  $v$  und  $w$ ; Knoten  $v$  und  $w$  sind *benachbart*, *adjazent* oder *Nachbarn*.

Die bereits besprochenen Bäume sind Spezialfälle von Graphen. Viele der dort eingeführten Begriffe treten genauso oder ähnlich hier wieder auf. Der *Grad* eines Knotens ist die Gesamtzahl eingehender und ausgehender Kanten; entsprechend spricht man vom *Eingangsgrad* und *Ausgangsgrad*. Ein *Pfad* ist eine Folge von Knoten  $v_1, \dots, v_n$ , so dass für  $1 \leq i \leq n-1$  gilt:  $(v_i, v_{i+1}) \in E$ . Die *Länge* des Pfades ist auch hier die Anzahl der

Kanten auf dem Pfad, also  $n-1$ . Ein Pfad heißt *einfach*, wenn alle Knoten auf dem Pfad paarweise verschieden sind. Dabei ist als Ausnahme  $v_1 = v_n$  erlaubt. Ein einfacher Pfad, dessen Länge mindestens 1 ist und in dem der erste und der letzte Knoten identisch sind ( $v_1 = v_n$ ), heißt (einfacher) *Zyklus*. Ein einzelner Knoten  $v_1$  stellt einen Pfad der Länge 0 dar.

**Beispiel 6.2:** Im Graphen  $G_1$ , der in Abbildung 6.4 noch einmal gezeigt ist, gibt es einen einfachen Pfad  $<1, 2, 4, 5, 3>$ . Die Folge  $<1, 2, 1, 2, 4>$  beschreibt einen nicht-einfachen Pfad,  $<1, 2, 1>$ ,  $<3, 3>$  und  $<3, 5, 3>$  sind Zyklen.  $\square$

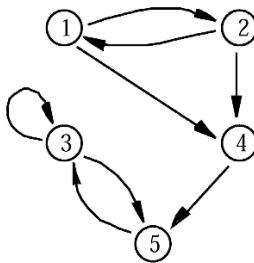


Abbildung 6.4: Graph  $G_1$  aus Abbildung 6.1

Ein *Teilgraph* eines Graphen  $G = (V, E)$  ist ein Graph  $G' = (V', E')$  mit  $V' \subseteq V$  und  $E' \subseteq E$ .

Zwei Knoten  $v, w$  eines gerichteten Graphen heißen *stark verbunden* (*strongly connected*), wenn es einen Pfad von  $v$  nach  $w$  und einen Pfad von  $w$  nach  $v$  gibt. Eine *stark verbundene Komponente* (oder *starke Komponente*) ist ein Teilgraph mit maximaler Knoten- und Kantenzahl, in dem alle Knoten paarweise stark verbunden sind. Für eine starke Komponente  $G' = (V', E')$  gilt also: Es gibt keinen Teilgraphen  $G'' = (V'', E'')$  mit  $V \supseteq V'' \supset V'$  oder  $E \supseteq E'' \supset E'$ , der ebenfalls eine starke Komponente ist. Im obigen Beispiel sind



starke Komponenten, jedoch ist folgender Teilgraph keine:



Ein gerichteter Graph, der nur aus einer einzigen starken Komponente besteht, heißt *stark verbunden*.

Knoten und/oder Kanten eines Graphen können weitere Informationen zugeordnet bekommen; wir sprechen dann von einem *markierten* Graphen. Ähnlich wie bei markierten Bäumen bezeichne  $\mu$  eine Knotenmarkierungsfunktion und  $\nu$  eine Kantenmarkierungsfunktion,  $\nu(e)$  ergibt also die Beschriftung der Kante  $e$ . Besonders interessant sind den Kanten zugeordnete *Kosten*, die z. B. in einem Verkehrsnetz Entferungen oder Reisezeiten darstellen können. Die Kosten eines Pfades sind die Summe der Kosten aller beteiligten Kanten.

**Beispiel 6.3:** In dem Graphen  $G_4$  mit markierten Kanten (Abbildung 6.5) sind die Kosten auf einem Pfad von Knoten 1 zu 4 immer 21, auch über den “Umweg” mit Knoten 3.  $\square$

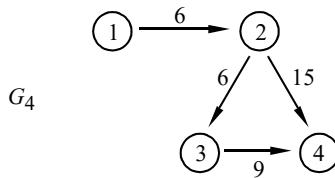


Abbildung 6.5: Markierter Graph

## 6.2 (Speicher-) Darstellungen von Graphen

Es gibt verschiedene Datenstrukturen zur Darstellung von Graphen. Die Art der auszuführenden Operationen bestimmt die Wahl einer Datenstruktur für einen bestimmten Algorithmus.

### (a) Adjazenzmatrix

Sei  $V = \{1, \dots, n\}$ . Die *Adjazenzmatrix* für  $G = (V, E)$  ist eine boolesche  $n \times n$ -Matrix  $A$  mit

$$A_{ij} = \begin{cases} \text{true} & \text{falls } (i, j) \in E \\ \text{false} & \text{sonst} \end{cases}$$

Meist werden *true* und *false* durch 1 und 0 dargestellt.

**Beispiel 6.4:** Abbildung 6.6 zeigt die Adjazenzmatrix für  $G_1$ .  $\square$

	$j = 1$	$2$	$3$	$4$	$5$
$i = 1$	0	1	0	1	0
2	1	0	0	1	0
3	0	0	1	0	1
4	0	0	0	0	1
5	0	0	1	0	0

Abbildung 6.6: Adjazenzmatrix für  $G_1$ 

Bei kantenmarkierten Graphen kann man direkt die Kantenbeschriftung statt der booleschen Werte in die Matrix (*markierte Adjazenzmatrix*) eintragen; ein geeignetes Element des Wertebereichs von  $v$  muss dann ausdrücken, dass keine Kante vorhanden ist.

**Beispiel 6.5:** Die Matrix für  $G_4$  ist in Abbildung 6.7 gezeigt; hier bezeichnet ein Eintrag  $\infty$  eine nicht vorhandene Kante.  $\square$

	1	2	3	4
1	$\infty$	6	$\infty$	$\infty$
2	$\infty$	$\infty$	6	15
3	$\infty$	$\infty$	$\infty$	9
4	$\infty$	$\infty$	$\infty$	$\infty$

Abbildung 6.7: Adjazenzmatrix für  $G_4$ 

Ein *Vorteil* der Darstellung eines Graphen in einer Adjazenzmatrix ist die Möglichkeit, in einer Laufzeit von  $O(1)$  festzustellen, ob eine Kante von  $v$  nach  $w$  existiert, das heißt, man sollte diese Repräsentation wählen, wenn in den beabsichtigten Algorithmen Tests auf das Vorhandensein von Kanten häufig vorkommen. Ein *Nachteil* ist der hohe Platzbedarf von  $O(n^2)$ . Vor allem, wenn die Anzahl der Kanten relativ klein ist im Vergleich zum Quadrat der Knotenzahl, wird Speicherplatz verschwendet. Das ist z. B. in planaren Graphen der Fall, bei denen die Anzahl der Kanten nur linear mit der Anzahl der Knoten wächst. Das Auffinden aller  $k$  Nachfolger eines Knotens benötigt  $O(n)$  Zeit, was auch relativ ungünstig ist. Schließlich brauchen Algorithmen schon zur Initialisierung der Matrix eine Laufzeit von  $\Theta(n^2)$ .

### (b) Adjazenzlisten

Man verwaltet für jeden Knoten eine Liste seiner (Nachfolger-) Nachbarknoten. Über einen Array der Länge  $n = |\mathcal{V}|$  ist jede Liste direkt zugänglich.

**Beispiel 6.6:** Adjazenzlisten für  $G_1$

□

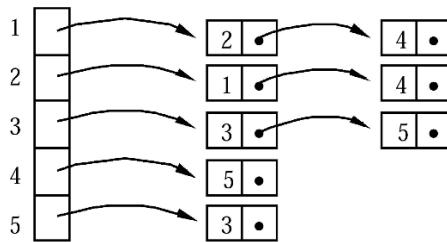


Abbildung 6.8: Adjazenzlisten für  $G_1$

Bei markierten Graphen können die Listenelemente weitere Einträge enthalten.

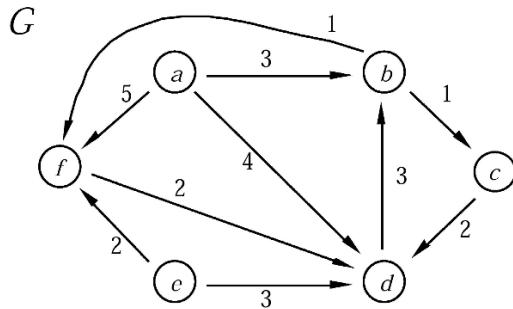
Der Vorteil dieser Darstellung ist der geringere Platzbedarf, der nämlich  $O(n+e)$  beträgt für  $n = |\mathcal{V}|, e = |\mathcal{E}|$ . Man kann alle  $k$  Nachfolger eines Knotens in Zeit  $O(k)$  erreichen. Ein "wahlfreier" Test, ob  $v$  und  $w$  benachbart sind, kann aber nicht mehr in konstanter Zeit durchgeführt werden, da die Adjazenzliste von  $v$  durchlaufen werden muss, um das Vorhandensein von  $w$  zu überprüfen.

Falls zu einem Knoten seine Vorgänger-Nachbarknoten aufgesucht werden müssen, kann man noch *inverse* Adjazenzlisten verwalten, die zu jedem Knoten eine Liste seiner Vorgänger enthalten.

Es gibt natürlich weitere Möglichkeiten, Graphen darzustellen, aber Adjazenzmatrizen und Adjazenzlisten sind die beiden wichtigsten. Beide Strukturen werden meist *statisch* benutzt, also zu Beginn oder vor dem Lauf des interessierenden Algorithmus aufgebaut und dann nicht mehr verändert. Updates (*insert, delete*) spielen hier nicht so eine große Rolle wie bei den *dynamischen* Datenstrukturen der vorherigen Kapitel.

**Selbsttestaufgabe 6.1:** Geben Sie die Repräsentation des Graphen  $G$  in der folgenden Abbildung an durch

- (a) eine Adjazenzmatrix, die die Kantenmarkierung enthält;
- (b) Adjazenzlisten, deren Elemente die Kantenmarkierung enthalten.



□

### 6.3 Graphdurchlauf

Ein erstes Problem, auf dessen Lösung weitere Graph-Algorithmen aufzubauen, besteht darin, systematisch alle Knoten eines Graphen  $G$  aufzusuchen. Wir nehmen zunächst vereinfachend an, alle Knoten des Graphen seien von einer *Wurzel*  $r$  aus erreichbar. Einen solchen Graphen nennt man *Wurzelgraphen*. Es gibt zwei wesentliche Arten, Graphen zu durchlaufen, nämlich den *Tiefendurchlauf* und den *Breitendurchlauf*. Beide lassen sich auf einfache Art mit Hilfe eines Baumes erklären, den wir die Expansion von  $G$  in  $r$  nennen und der im Allgemeinen unendlich ist.

**Definition 6.7:** Die *Expansion*  $X(v)$  eines Graphen  $G$  in einem Knoten  $v$  ist ein Baum, der wie folgt definiert ist:

- (i) Falls  $v$  keine Nachfolger hat, so ist  $X(v)$  der Baum, der nur aus dem Knoten  $v$  besteht.



- (ii) Falls  $v_1, \dots, v_k$  die Nachfolger von  $v$  sind, so ist  $X(v)$  der Baum  $(v, X(v_1), \dots, X(v_k))$ .

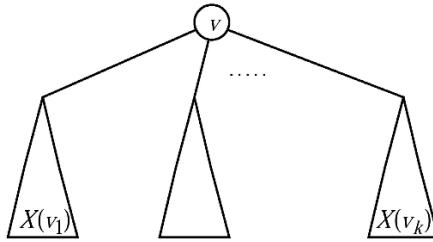
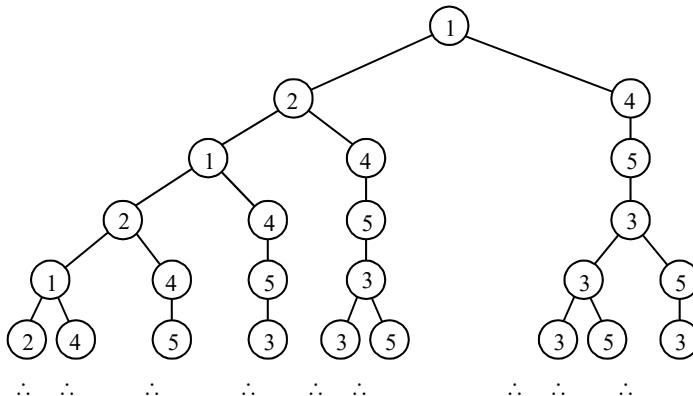


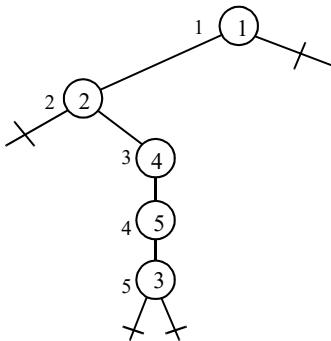
Abbildung 6.9: Expansion eines Graphen (Prinzip)

**Beispiel 6.8:** Für  $G_1$  ist die Expansion im Knoten 1 in Abbildung 6.10 gezeigt.  $\square$

Abbildung 6.10: Expansion des Graphen  $G_1$ 

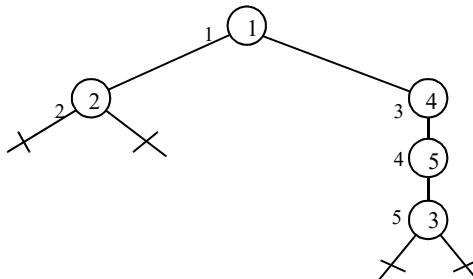
Dieser Baum stellt also einfach die Menge aller vom Knoten 1 ausgehenden Pfade in  $G$  dar. Der Baum ist unendlich, falls  $G$  Zyklen enthält. Ein *Tiefendurchlauf* (*depth-first-traversal*) des Graphen  $G$  entspricht nun gerade einem *Preorder*-Durchlauf der Expansion von  $G$ , der jeweils in einem bereits besuchten Knoten von  $G$  abgebrochen wird.

**Beispiel 6.9:** Für den Graphen  $G_1$  wird mit einem Tiefendurchlauf der in Abbildung 6.11 gezeigte Teilbaum besucht. Ziffern neben den Knoten stellen die Besuchsreihenfolge dar; durchgestrichene Kanten bedeuten Abbruch an einem schon erreichten Knoten.  $\square$

Abbildung 6.11: Tiefendurchlauf für  $G_1$ 

Ein *Breitendurchlauf (breadth-first-traversal)* besucht die Knoten der Expansion *ebenenweise*, also zuerst die Wurzel, danach alle Knoten auf Level 1, dann alle auf Level 2 usw. Das bedeutet, dass im Graphen zunächst alle über einen Pfad der Länge 1, dann der Länge 2 usw. erreichbaren Knoten besucht werden. Bei schon besuchten Knoten wird wie beim Tiefendurchlauf ebenfalls abgebrochen.

**Beispiel 6.10:** Breitendurchlauf für  $G_1$  (Abbildung 6.12). □

Abbildung 6.12: Breitendurchlauf für  $G_1$ 

Jede der Durchlaufarten definiert zu einem Wurzelgraphen also einen Baum, genannt *depth-first-* bzw. *breadth-first-Spannbaum*. In einem allgemeinen Graphen (ohne die Bedingung, dass alle Knoten von einem Wurzelknoten aus erreichbar sein müssen) können bei einem Durchlauf unbesuchte Knoten verbleiben; wenn das der Fall ist, beginnt jeweils in einem solchen Knoten ein neuer Durchlauf. Entsprechend werden *spannende Wälder* erzeugt.

Einen Algorithmus für den Tiefendurchlauf kann man rekursiv so formulieren:

```
algorithm dfs (v)
{Tiefendurchlauf vom Knoten v aus}
if v wurde noch nicht besucht
then verarbeite v;
markiere v als besucht;
for each Nachfolger vi von v do dfs (vi) end for
end if.
```

Ob *v* bereits besucht wurde, merkt man sich in einem zusätzlichen Array

*visited* : **array**[1..*n*] **of** *bool*

Die Operation “verarbeite *v*” hängt von der jeweiligen Anwendung ab. Um einen Graphen vollständig zu durchlaufen, sei der Algorithmus *dfs* eingebettet in eine Umgebung:

```
for v := 1 to n do visited[v] := false end for;
for v := 1 to n do dfs (v) end for;
```

Die zweite Schleife ist notwendig, da – wie oben erwähnt – im Allgemeinen nicht alle Knoten von einem Wurzelknoten (hier 1) erreichbar sind.

Auf einer Adjazenzlistendarstellung braucht ein solcher Durchlauf offensichtlich  $O(n+e)$  Zeit.

Um den Breitendurchlauf zu realisieren, benutzt man eine Schlange, in der jeweils die Nachfolger eines Knotens gemerkt werden, während man seine Brüder bearbeitet. Den entsprechenden Datentyp *Queue* kennen wir bereits aus Abschnitt 3.3.

```
algorithm bfs (v)
{Breitendurchlauf vom Knoten v aus}
füge in die leere Queue q den Knoten v ein.
markiere v als besucht;
while q enthält Elemente do
entnimm q das erste Element =: w;
verarbeite w;
for each Nachfolger wi von w do
if wi wurde noch nicht besucht
then
hänge wi an q an;
markiere wi als besucht;
end if
end for
end while.
```

Wie der Algorithmus *dfs* muss auch der Algorithmus *bfs* für jeden unbesuchten Knoten des Graphen aufgerufen werden, um das Erreichen jedes Knotens sicherzustellen:

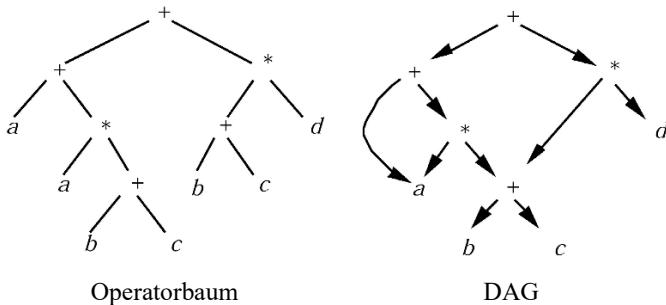
```

for  $v := 1$  to  $n$  do
    if  $v$  wurde noch nicht besucht then  $bfs(v)$  end if
end for;

```

Wieder entsteht in der for-each-Schleife ein Aufwand, der insgesamt proportional zur Anzahl der Kanten im Graphen ist. Der Gesamtaufwand für die anderen Operationen ist  $O(n)$ . Also hat auch der Breitendurchlauf auf einer Adjazenzlisten-Darstellung eine Laufzeit von  $O(n + e)$ , da alle Knoten genau einmal in die Queue aufgenommen werden.

**Selbsttestaufgabe 6.2:** Arithmetische Ausdrücke können durch einen Operatorbaum oder durch einen gerichteten azyklischen Graphen (DAG - *directed acyclic graph*) repräsentiert werden, in dem gemeinsame Teilausdrücke nur einmal auftreten. Ein Baum und ein DAG für den Ausdruck  $a + a * (b + c) + (b + c) * d$  sind in der folgenden Abbildung zu sehen:



Geben Sie einen Algorithmus an, der aus einem Operatorbaum mit den Operatoren  $+$  und  $*$  sowie Operanden aus der Menge  $\{a, \dots, z\}$  einen solchen DAG konstruiert. Wie groß ist die Zeitkomplexität?  $\square$

## 6.4 Literaturhinweise

Graphentheorie ist ein seit langem intensiv studiertes Teilgebiet der Mathematik. Das wohl erste Resultat stammt von Euler [1736]. Er löste das berühmte *Königsberger Brückenproblem*, das in der Frage besteht, ob es möglich ist, sämtliche durch Brücken verbundene Stadtteile und Inseln Königsbergs auf einem Rundweg so zu besuchen, dass jede Brücke nur einmal passiert wird – offensichtlich ein Graphenproblem. Inzwischen gibt es eine Fülle von Lehrbüchern zur Graphentheorie und zu Graph-Algorithmen, von denen hier nur einige genannt werden können [Harary 1994], [Papadimitriou und Steiglitz 1998], [Mehlhorn 1984b], [Gibbons 1985], [West 2001], [Wilson 2010], [Jungnickel

2012] (deutsche Ausgabe [Jungnickel 1994]). Eine kurze Übersicht zu Graph-Algorithmen bietet [Khuller und Raghavachari 1996].



## 7 Graph-Algorithmen

Wir betrachten in Abschnitt 7.1 einen der wichtigsten Graph-Algorithmen, nämlich den Algorithmus von Dijkstra zur Bestimmung kürzester Wege von einem Startknoten aus. Abschnitt 7.2 beschreibt den Algorithmus von Floyd, der kürzeste Wege zwischen allen Paaren von Knoten des Graphen ermittelt. In Abschnitt 7.3 betrachten wir *Kontraktionshierarchien*, eine relativ neue Technik zur Vorverarbeitung (*preprocessing*) von Graphen mit dem Ziel, Kürzeste-Wege-Suchen zu beschleunigen. Abschnitt 7.4 und Abschnitt 7.5 behandeln die Berechnung der *transitiven Hülle* sowie die Ermittlung *stark zusammenhängender Komponenten*. Abschnitt 7.6 bietet eine kurze, etwas präzisere Einführung ungerichteter Graphen. Schließlich zeigt Abschnitt 7.7 den Algorithmus von Kruskal zur Berechnung eines *minimalen Spannbaums* in einem ungerichteten Graphen.

### 7.1 Bestimmung kürzester Wege von einem Knoten zu allen anderen

Gegeben sei ein gerichteter Graph, dessen Kanten mit positiven reellen Zahlen (Kosten) beschriftet sind. Die Aufgabe besteht nun darin, für einen beliebigen Startknoten  $v$  die kürzesten Wege zu allen anderen Knoten im Graphen zu berechnen. Die Länge eines Weges (oder Pfades) ist dabei definiert als die Summe der Kantenkosten. Dieses Problem ist bekannt als das *single source shortest path*-Problem.

Ein Algorithmus zur Lösung dieses Problems ist der *Algorithmus von Dijkstra*, der auf folgender Idee beruht. Ausgehend vom Startknoten  $v$  lässt man innerhalb des Graphen  $G$  einen Teilgraphen wachsen; der Teilgraph beschreibt den bereits erkundeten Teil von  $G$ . Innerhalb des Teilgraphen gibt es zwei Arten von Knoten und zwei Arten von Kanen. Der anschaulichkeit halber stellen wir uns vor, dass diese Arten farblich unterschieden werden. Die Knoten können grün oder gelb gefärbt sein. *Grüne Knoten* sind solche, in denen bereits alle Kanten zu Nachfolgern betrachtet wurden; die Nachfolger liegen daher mit im bereits erkundeten Teilgraphen, können also grün oder gelb sein. Bei *gelben Knoten* sind die ausgehenden Kanten noch nicht betrachtet worden; gelbe Knoten bilden also den Rand oder die Peripherie des Teilgraphen. Die Kanten innerhalb des Teilgraphen sind gelb oder rot; die *roten Kanten* bilden innerhalb des Teilgraphen einen *Baum der kürzesten Wege*. In jedem Knoten des Teilgraphen wird der Abstand zu  $v$  verwaltet (über den bisher bekannten kürzesten Weg). Der Teilgraph wächst nun, indem in jedem Schritt der gelbe Knoten mit minimalem Abstand von  $v$  ins Innere des Teilgraphen übernommen, also grün gefärbt wird; seine Nachfolgerknoten werden, soweit sie noch nicht im Teilgraphen lagen, zu neuen gelben Knoten. Für gelbe Knoten, die so erneut

erreicht werden, sind die für sie bisher bekannten kürzesten Pfade (rote Kanten) ggf. zu korrigieren.

**Beispiel 7.1:** Gegeben sei folgender Graph  $G$ . Alle Knoten und Kanten sind noch ungefärbt.  $A$  sei der Startknoten.

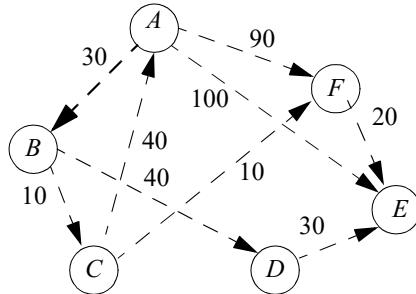


Abbildung 7.1: Ausgangsgraph  $G$

*Tipp:* Kennzeichnen Sie bitte selbst in den folgenden Graphen die Knoten und Kanten farblich entsprechend, dann sind sie besser zu unterscheiden.

Zu Anfang ist nur  $A$  gelb und wird im ersten Schritt in einen grünen Knoten umgewandelt, während  $B, E$  und  $F$  als seine Nachfolger gelb und die Kanten dorthin rot werden:

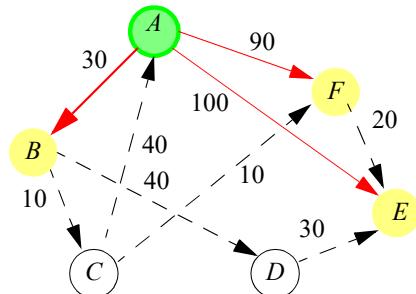


Abbildung 7.2: Erster Schritt

Dann wird  $B$  als gelber Knoten mit minimalem Abstand zu  $A$  grün gefärbt; gleichzeitig werden die noch nicht besuchten Knoten  $C$  und  $D$  zu gelben:

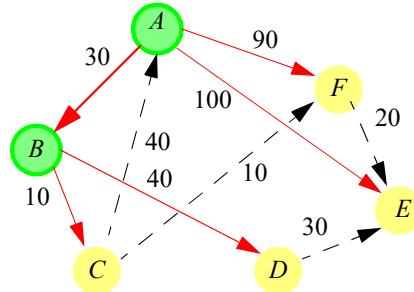


Abbildung 7.3: Zweiter Schritt

Als nächster gelber Knoten mit minimalem Abstand zu  $A$  wird  $C$  grün:

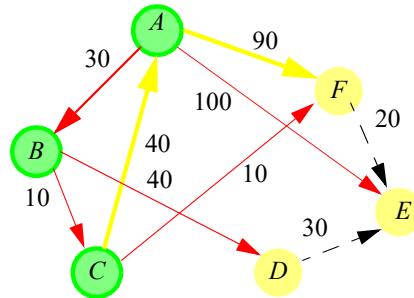


Abbildung 7.4: Dritter Schritt

Hier ist  $F$  über  $C$  zum zweiten Mal erreicht worden. Der aktuelle Weg dorthin ist aber der bislang kürzeste. Deshalb wird die rote Kante ( $A, F$ ) in eine gelbe umgewandelt, und die Kante ( $C, F$ ) wird rote Kante zu  $F$ . Auch die Kante ( $C, A$ ) wird gelb gefärbt. Der Baum der roten Kanten ist im Moment:

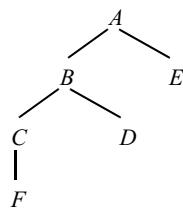


Abbildung 7.5: Aktueller Baum der roten Kanten

Der Endzustand des Graphen und des Baumes ist folgender:

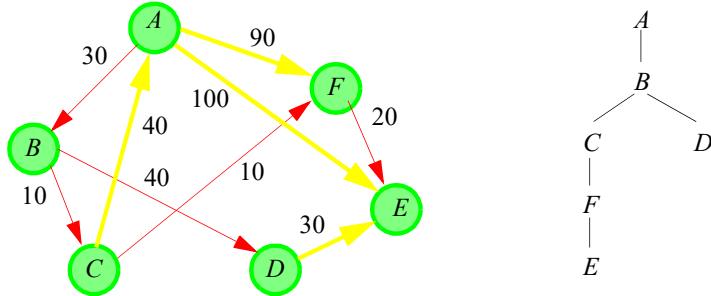


Abbildung 7.6: Endzustand des Graphen und des Baumes der roten Kanten

□

Dieser Algorithmus lässt sich dann wie folgt formulieren. Bezeichne  $dist(w)$  den Abstand des Knotens  $w$  vom Startknoten  $v$ , und seien  $GELB$  und  $GRÜN$  die beschriebenen Knotenmengen. Bezeichne  $succ(w)$  die Menge der Nachfolger (-Nachbarn) des Knotens  $w$  in  $G$ . Sei  $cost(w, w')$  das Kostenmaß der Kante  $(w, w')$ .

```

algorithm Dijkstra ( $v$ )
  {berechne alle kürzesten Wege vom Knoten  $v$  aus}
   $GRÜN := \emptyset$ ;  $GELB := \{v\}$ ;  $dist(v) := 0$ ;
  while  $GELB \neq \emptyset$  do
    wähle  $w \in GELB$ , so dass  $\forall w' \in GELB: dist(w) \leq dist(w')$ ;
    färbe  $w$  grün;
    for each  $w_i \in succ(w)$  do
      if  $w_i \notin (GELB \cup GRÜN)$  then noch nicht besuchter Knoten
        färbe  $w_i$  rot;
        färbe  $w_i$  gelb;  $dist(w_i) := dist(w) + cost(w, w_i)$ 
      elsif  $w_i \in GELB$  then  $w_i$  erneut erreicht
        if  $dist(w_i) > dist(w) + cost(w, w_i)$ 
          färbe  $w_i$  rot;
          färbe die bisher rote Kante zu  $w_i$  gelb;
           $dist(w_i) := dist(w) + cost(w, w_i)$ 
        else färbe  $(w, w_i)$  gelb
        end if
      else färbe  $(w, w_i)$  gelb
      end if
    end for
  end while.
  
```

Bei der Terminierung des Algorithmus sind also alle Knoten grün, die von  $v$  aus erreichbar sind. Es können ungefärbte Knoten und Kanten verbleiben, die dann von  $v$  aus nicht erreichbar sind.

Man muss nun noch zeigen, dass der Algorithmus von Dijkstra korrekt ist, d. h. tatsächlich kürzeste Wege berechnet. Dazu beweisen wir zunächst die folgenden Lemmata:

**Lemma 7.2:** Zu jeder Zeit ist für jeden Knoten  $w \in GELB$  der rote Pfad zu  $w$  minimal unter allen gelbroten Pfaden zu  $w$ , das heißt solchen, die nur gelbe oder rote Kanten haben.

**Beweis:** Wir führen Induktion über die Folge grün gefärbter Knoten durch.

*Induktionsanfang:*

Die Behauptung gilt für  $v$ , da noch keine Kante gefärbt ist.

*Induktionsschluss:*

Annahme: Die Aussage gilt für  $GRÜN$  und  $GELB$ .

Jetzt wird  $w \in GELB$  grün gefärbt. Seien  $w_1, \dots, w_n$  die Nachfolger von  $w$ . Für die  $w_i$  sind folgende Fälle zu unterscheiden:

(a)  $w_i$  wurde zum erstenmal erreicht, war also bisher ungefärbt und wird jetzt gelb.

Der einzige gelbrote Pfad zu  $w_i$  hat die Form  $v \Rightarrow w \rightarrow w_i$  (bezeichne “ $\Rightarrow$ ” einen Pfad, “ $\rightarrow$ ” eine einzelne Kante). Der Pfad  $v \Rightarrow w$  ist nach der Induktionsannahme minimal. Also ist  $v \Rightarrow w_i$  minimal.

(b)  $w_i$  ist gelb und wurde erneut erreicht.

Der bislang rote Pfad zu  $w_i$  war  $v \Rightarrow x \rightarrow w_i$ , der neue rote Pfad zu  $w_i$  ist der kürzere Pfad unter den Pfaden

- (1)  $v \Rightarrow x \rightarrow w_i$  und
- (2)  $v \Rightarrow w \rightarrow w_i$ .

Der Pfad (1) ist minimal unter allen gelbroten Pfaden, die nicht über  $w$  führen, (2) ist minimal unter allen, die über  $w$  führen (wie in (a)), also ist der neue Pfad minimal unter allen gelbroten Pfaden.

□

**Lemma 7.3:** Wenn ein Knoten grün gefärbt wird, dann ist der rote Pfad zu ihm der kürzeste von allen Pfaden im Graphen  $G$ .

**Beweis:** Der Beweis benutzt wieder Induktion über die Folge grün gefärbter Knoten.

*Induktionsanfang:*

$v$  wird zuerst grün gefärbt, es gibt noch keinen roten Pfad.

*Induktionsschluss:*

Annahme: Die Aussage gilt für *GRÜN* und *GELB*. Nun wird  $w$  als gelber Knoten mit minimalem Abstand zu  $v$  grün gefärbt. Der rote Pfad zu  $w$  ist nach Lemma 7.2 minimal unter allen gelbroten Pfaden. Nehmen wir an, es gibt einen kürzeren Pfad zu  $w$  als diesen; er muss dann eine nicht gelbrote Kante enthalten. Da ungefärbte Kanten nur über gelbe Knoten erreichbar sind, muss es einen gelben Knoten  $\overline{w}$  geben, über den dieser Pfad verläuft, er hat also die Form

$$v \Rightarrow \overline{w} \rightarrow \Rightarrow w.$$

$\uparrow$  ungefärbte Kante

Da aber  $w$  minimalen Abstand unter allen gelben Knoten hat, muss schon der Pfad  $v \Rightarrow \overline{w}$  mindestens so lang sein wie  $v \Rightarrow w$ , also ist  $v \Rightarrow \overline{w} \Rightarrow w$  nicht kürzer. Somit erhält man einen *Widerspruch* zur Annahme, es gebe einen kürzeren Pfad, und die Behauptung ist bewiesen.  $\square$

**Satz 7.4:** Der Algorithmus von Dijkstra berechnet die kürzesten Pfade zu allen von  $v$  erreichbaren Knoten.

**Beweis:** Nach Ablauf des Algorithmus sind alle erreichbaren Knoten grün gefärbt. Die Behauptung folgt dann aus Lemma 7.3.  $\square$

## Implementierungen des Algorithmus von Dijkstra

### (a) mit einer Adjazenzmatrix

Sei  $V = \{1, \dots, n\}$  und sei  $cost(i, j)$  die Kosten-Adjazenzmatrix mit Einträgen  $\infty$  an den Matrixelementen, für die keine Kante existiert. Man benutzt weiter:

```
type node = 1..n;
var dist : array[node] of real;
var father : array[node] of node;
var green : array[node] of bool;
```

Der Array  $father$  stellt den Baum der roten Kanten dar, indem zu jedem Knoten sein Vaterknoten festgehalten wird. Jeder Schritt (in dem ein Knoten hinzugefügt bzw. grün gefärbt wird) besteht dann aus folgenden Teilschritten:

- Der gesamte Array  $dist$  wird durchlaufen, um den gelben Knoten  $w$  mit minimalem Abstand zu finden. Der Aufwand hierfür ist  $O(n)$ .
- Die Zeile  $cost(w, -)$  der Matrix wird durchlaufen, um für alle Nachfolger von  $w$  ggf. den Abstand und den Vater zu korrigieren, was ebenfalls einen Aufwand von  $O(n)$  ergibt.

Die Zeitkomplexität ist daher insgesamt  $O(n^2)$ , da die obigen Teilschritte  $n$  mal durchgeführt werden.

Diese Implementierung ist ineffizient, wenn  $n$  nicht sehr klein oder  $e \approx n^2$  ist.

**Selbsttestaufgabe 7.1:** Formulieren Sie die Methode *Dijkstra* für die Implementierung mit einer Adjazenzmatrix. Nehmen Sie vereinfachend an, dass der Startknoten der Knoten 1 sei (in einer Implementierung per Array in Java also der Knoten mit dem Index 0) und der Graph zusammenhängend ist.  $\square$

### (b) mit Adjazenzlisten und als Heap dargestellter Priority Queue

Der Graph sei durch Adjazenzlisten mit Kosteneinträgen dargestellt. Es gebe Arrays  $dist$  und  $father$  wie unter (a). Zu jeder Zeit besitzen grüne und gelbe Knoten Einträge in  $dist$  und  $father$ . Weiterhin sind gelbe Knoten mit ihrem Abstand vom Ausgangsknoten als Ordnungskriterium in einer als Heap (im Array) dargestellten Priority Queue repräsentiert. Heap-Einträge und Knoten sind miteinander doppelt verkettet, d. h. der Heap-Eintrag enthält die Knotennummer, und ein weiterer Array

```
var heapaddress : array[node] of 1..n
```

enthält zu jedem Knoten seine Position im Heap. Ein Einzelschritt des Algorithmus von Dijkstra besteht dann aus folgenden Teilen:

1. Entnimm den gelben Knoten  $w_i$  mit minimalem Abstand aus der Priority Queue. Das erfordert einen Aufwand von  $O(\log n)$ .
2. Finde in der entsprechenden Adjazenzliste die  $m_i$  Nachfolger von  $w_i$ . Hier ist der Aufwand  $O(m_i)$ .
  - a. Für jeden “neuen” gelben Nachfolger erzeuge einen Eintrag in der Priority Queue. – Kosten  $O(\log n)$ .
  - b. Für jeden “alten” gelben Nachfolger korrigiere ggf. seinen Eintrag in der Priority Queue. Seine Position dort ist über  $heapaddress$  zu finden. Da sein Abstandswert bei der Korrektur sinkt, kann der Eintrag im Heap ein Stück nach oben wandern. Auch hier entstehen Kosten  $O(\log n)$ . Die Heap-Adressen der vertauschten Einträge im Array  $heapaddress$  können in  $O(1)$  Zeit geändert werden.

Der Aufwand für (a) und (b) beträgt insgesamt  $O(m_i \log n)$ .

Es gilt:  $\sum m_i = e$  mit  $e = |E|$ . Über alle Schritte des Algorithmus summiert ist der Aufwand für (2)  $O(e \log n)$ . Der Aufwand für (1) ist ebenfalls  $O(e \log n)$ , da ein Element nur aus der Priority Queue entnommen werden kann, wenn es vorher eingefügt wurde. Also ist der Gesamtaufwand bei dieser Implementierung  $O(e \log n)$ , der Platzbedarf ist  $O(n + e)$ .

## 7.2 Bestimmung kürzester Wege zwischen allen Knoten im Graphen

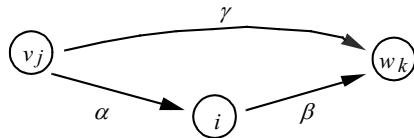
Wir betrachten die Bestimmung der kürzesten Wege zwischen allen Paaren von Knoten, bekannt als das “*all pairs shortest path*”-Problem. Man kann zu seiner Lösung natürlich den Algorithmus von Dijkstra iterativ auf jeden Knoten des Graphen anwenden. Es gibt hierfür aber den wesentlich einfacheren *Algorithmus von Floyd*. Dieser Algorithmus wird gewöhnlich anhand der Kostenmatrix-Darstellung des Graphen erklärt. Wir geben hier eine allgemeinere Formulierung an, die noch unabhängig von der tatsächlichen Repräsentation ist.

Der Algorithmus berechnet eine Folge von Graphen  $G_0, \dots, G_n$  (unter der Annahme, dass der Ausgangsgraph  $G$   $n$  Knoten hat, die von 1 bis  $n$  durchnummieriert sind). Graph  $G_i$  entsteht jeweils durch Modifikation des Graphen  $G_{i-1}$ . Jeder Graph  $G_i$  ist wie folgt definiert:

- (i)  $G_i$  hat die gleiche Knotenmenge wie  $G$ .
- (ii) Es existiert in  $G_i$  eine Kante  $v \xrightarrow{\alpha} w \Leftrightarrow$  es existiert in  $G$  ein Pfad von  $v$  nach  $w$ , in dem als Zwischenknoten nur Knoten aus der Menge  $\{1, \dots, i\}$  verwendet werden. Der kürzeste derartige Pfad hat Kosten  $\alpha$ . – Dabei bezeichne die Notation  $v \xrightarrow{\alpha} w$  eine Kante  $(v, w)$  mit  $v(v, w) = \alpha$ .

Sei  $G_0 = G$ . Man beachte, dass  $G_0$  bereits die obige Spezifikation der  $G_i$  erfüllt. Der  $i$ -te Schritt des Algorithmus von Floyd berechnet  $G_i$  aus  $G_{i-1}$  wie folgt:

Seien  $v_1, \dots, v_r$  die Vorgänger von Knoten  $i$  im Graphen  $G_{i-1}$  und seien  $w_1, \dots, w_s$  die Nachfolger. Betrachte alle Paare  $(v_j, w_k)$  mit  $j = 1, \dots, r$  und  $k = 1, \dots, s$ .

Abbildung 7.7: Der  $i$ -te Schritt des Algorithmus von Floyd

- (a) Falls noch keine Kante von  $v_j$  nach  $w_k$  existiert, so erzeuge eine Kante

$$v_j \xrightarrow{\alpha+\beta} w_k .$$

- (b) Falls schon eine Kante  $v_j \xrightarrow{\gamma} w_k$  existiert, ersetze die Markierung  $\gamma$  dieser Kante durch  $\alpha + \beta$ , falls  $\alpha + \beta < \gamma$ .

Mit vollständiger Induktion kann man leicht zeigen, dass  $G_i$  wiederum die Spezifikation erfüllt. Denn in  $G_{i-1}$  waren alle kürzesten Pfade bekannt und durch Kanten repräsentiert, die als Zwischenknoten nur Knoten aus  $\{1, \dots, i-1\}$  benutzten. Jetzt sind in  $G_i$  alle Pfade bekannt, die Knoten  $\{1, \dots, i\}$  benutzen.

Deshalb sind nach  $n$  Schritten, bei Beendigung des Algorithmus Floyd, die Kosten aller kürzesten Wege in  $G$  in  $G_n$  repräsentiert.

**Beispiel 7.5:** Wir wenden diesen Algorithmus auf unseren Beispielgraphen  $G$  aus Abbildung 7.1 an:

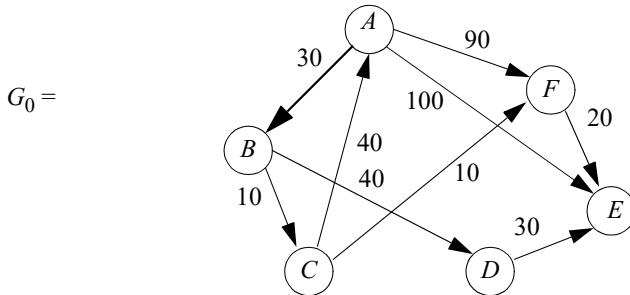


Abbildung 7.8: Ausgangsgraph

Im ersten Schritt werden alle Paare von Vorgängern  $\{C\}$  und Nachfolgern  $\{B, E, F\}$  des Knotens  $A$  betrachtet. Die Kanten  $(C, B)$  und  $(C, E)$  werden dabei mit entsprechender Kostenmarkierung neu eingefügt.

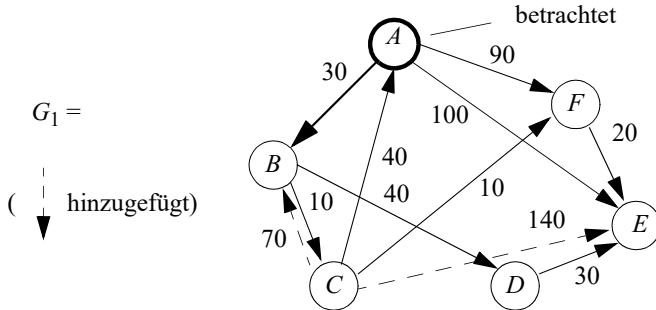


Abbildung 7.9: Erster Schritt

Danach sind die Vorgänger  $\{A, C\}$  und die Nachfolger  $\{C, D\}$  von  $B$  an der Reihe.

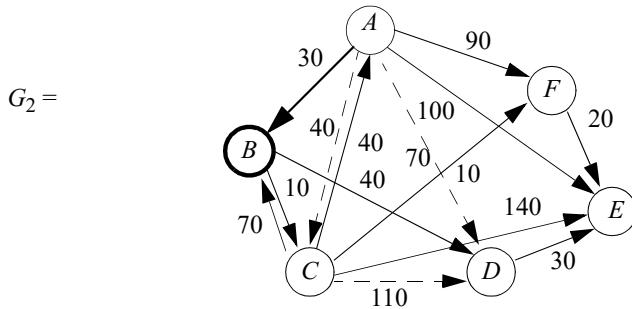


Abbildung 7.10: Zweiter Schritt

□

### Implementierung des Algorithmus von Floyd

#### (a) mit der Kostenmatrix-Darstellung

Hiermit ergibt sich eine sehr einfache Implementierung.

Sei  $C_{ij}$  die Kostenmatrix für den Ausgangsgraphen  $G$ . Sei  $C_{ii} = 0$  für alle  $i$  und sei  $C_{ij} = \infty$ , falls es in  $G$  keine Kante von Knoten  $i$  nach Knoten  $j$  gibt.

```

public static void Floyd(float[][] A, float[][] C)
{
    int n = A.length;      // Zusicherung: A und C sind quadratisch und gleich groß
    for(int j = 0; j < n; j++)
        for(int k = 0; k < n; k++) A[j][k] = C[j][k];
    for(int i = 0; i < n; i++)      // Dies sind die Haupt-“Schritte” des Algorithmus
        for(int j = 0; j < n; j++)
            for(int k = 0; k < n; k++)
                if(A[j][i] + A[i][k] < A[j][k])      //  $\alpha + \beta < \gamma$ 
                {
                    A[j][k] = A[j][i] + A[i][k];
                    /* Hier wird später noch eine Anweisung eingefügt */
                }
}

```

Die Laufzeit für diese Implementierung ist offensichtlich  $O(n^3)$ .

### (b) mit Adjazenzlisten

Hier würde man Vorgänger- und Nachfolger-Adjazenzlisten verwenden. In jedem Schritt müssen alle Elemente der Vorgängerliste des Knotens  $i$  mit allen Elementen der Nachfolgerliste kombiniert werden. Für  $r$  Vorgänger und  $s$  Nachfolger ist der Aufwand dieses Schrittes also  $O(rs)$ . Man könnte hoffen, dass sich für Graphen mit geringem Eingangs- und Ausgangsgrad eine (auch asymptotische) Laufzeitverbesserung gegenüber der Adjazenzmatrix-Darstellung ergibt. Das ist aber – außer bei sehr speziellen Klassen von Graphen – nicht der Fall, da der Algorithmus selbst noch laufend neue Kanten erzeugt und so die Eingangs- und Ausgangsgrade der Knoten erhöht. Deshalb ist die einfachere Adjazenzmatrix-Implementierung im Allgemeinen vorzuziehen.

**Selbsttestaufgabe 7.2:** Die Distanz-Matrix  $A$  ist für beliebige Kantenmarkierungen (also auch negative!) folgendermaßen definiert:

$$A_{ij} = \begin{cases} \text{die Länge des kürzesten Weges von } i \text{ nach } j, \\ \quad \text{falls ein Weg existiert} \\ +\infty \\ -\infty \quad \text{falls kein Weg existiert} \\ \quad \text{falls ein Zyklus mit negativen} \\ \quad \text{Kosten auf einem Weg von } i \text{ nach } j \text{ existiert} \end{cases}$$

- (a) Zeigen Sie dass der Algorithmus von Floyd (mit Adjazenzmatrix) nicht korrekt ist, falls negative Kantenmarkierungen zugelassen sind.
- (b) Ergänzen Sie den Algorithmus so, dass er am Schluss die korrekten Werte ermittelt hat.

□

Der Algorithmus berechnet bisher nur die Kosten aller kürzesten Pfade, nicht die Pfade selbst. Eine einfache Ergänzung liefert auch die Pfade: Man beschriftet die Kante  $(v_j, w_k)$  zusätzlich mit der Information, über welchen “mittleren” Knoten der kürzeste Weg verläuft.

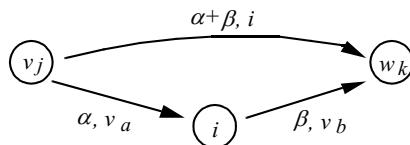


Abbildung 7.11: Erweiterung der Kantenmarkierung um die Pfadinformation

Diese Information wird dann folgendermaßen benutzt: “Um den kürzesten Pfad von  $v_j$  nach  $w_k$  zu laufen, laufe man rekursiv den kürzesten Pfad von  $v_j$  nach  $i$  und dann den von  $i$  nach  $w_k$ .”

Man realisiert diese Idee wie folgt. In der Kostenmatrix-Implementierung benutzt man einen weiteren Array  $P$  derselben Größe wie  $C$ . Der Kopf der Methode *Floyd* wird dazu entsprechend modifiziert:

```
public static void Floyd(float[][] A, int[][] P, float[][] C)
```

$P$  wird überall mit -1 initialisiert. An der Stelle \* wird folgende Anweisung in die Methode *Floyd* eingefügt:

```
 $P[j][k] = i;$ 
```

Anhand der von *Floyd* berechneten Arrays  $A$  und  $P$  lässt sich anschließend der kürzeste Pfad für zwei beliebige Knoten in  $O(m)$  Zeit rekonstruieren, wobei  $m$  die Länge dieses Pfades ist.

**Selbsttestaufgabe 7.3:** Schreiben Sie zum Algorithmus von Floyd eine Methode  $path(P, i, j)$ , die aus dem Array  $P$  die Folge der Knoten auf dem kürzesten Pfad von  $i$  nach  $j$  ermittelt, wie gerade skizziert. □

### 7.3 Berechnung kürzester Wege mittels Kontraktionshierarchien

Die bisher betrachteten Algorithmen zur Berechnung kürzester Wege sind fundamental und stammen schon aus den Anfangszeiten des Computers (Algorithmus von Dijkstra 1959, Floyd 1962).

Inzwischen gibt es sehr populäre Anwendungen, die die Berechnung kürzester Wege in sehr großen Graphen erfordern, nämlich die Planung einer Fahrtroute in einem Navigationssystem oder z.B. in Google Maps. Ein Straßennetz ist ein gerichteter Graph, in dem die Knoten Kreuzungen, Einmündungen oder Straßenenden und die Kanten die verbindenden Straßenabschnitte darstellen. Als Kantenkosten kann man die Distanz entlang der Wegstrecke oder auch geschätzte Fahrzeiten verwenden. Ein solches Netz, z.B. abgeleitet aus OpenStreetMap-Daten, hat schon für Deutschland ca. 8 Millionen Knoten und 19 Millionen Kanten (Tabelle 7.1).

Für die Berechnung des kürzesten Weges von  $A$  nach  $B$  kann man den Algorithmus von Dijkstra mit Startknoten  $A$  benutzen und ihn abbrechen lassen, sobald  $B$  erreicht (genauer: „grün gefärbt“) wird. Wie ist der Aufwand in einem solchen Fall? Der Algorithmus von Dijkstra lässt im Wesentlichen vom Startknoten aus einen Kreis der besuchten Knoten wachsen; bei gleichmäßiger Verteilung von Knoten ist der Aufwand in etwa proportional zur Kreisfläche, also quadratisch in der Entfernung zum Ziel  $B$ . Schlimmstenfalls muss man das gesamte Netz, also viele Millionen Knoten und Kanten absuchen. Selbst bei heutigen schnellen Rechnern liegt die Laufzeit damit für eine lange Strecke eher im Minuten- als im gewünschten (Sub-) Sekundenbereich.

Es gibt eine Variante des Algorithmus von Dijkstra, genannt A\*, die eine zielorientiertere Suche des kürzesten Weges von  $A$  nach  $B$  ermöglicht. Dabei wird als nächster Knoten immer derjenige expandiert, für den die Summe aus der Distanz vom Startknoten und der geschätzten Distanz zum Ziel minimal ist. Als geschätzte Distanz zum Ziel kann man einfach die euklidische Distanz verwenden. Der Algorithmus A\* ist schneller als der Algorithmus von Dijkstra für diese Anwendung, aber nicht um Größenordnungen schneller. Anstelle eines Kreises betrachtet er die Knoten in einer Ellipse.

Nach Anwendung des Algorithmus von Floyd auf das Straßennetz hätten wir eine Datenstruktur, die die kürzesten Wege zwischen allen Paaren von Knoten enthält. Man könnte sehr schnell auf die Kante von  $A$  nach  $B$  zugreifen und daraus den Pfad in  $O(m)$  Zeit rekonstruieren (vgl. Selbsttestaufgabe 7.3), wobei  $m$  die Länge des Pfades ist. Leider ist die Laufzeit zur Konstruktion der Datenstruktur  $O(n^3)$  und der Speicherplatzbedarf  $O(n^2)$ , mit  $n$  als Anzahl der Knoten, da zu jedem Paar von Knoten eine Kante konstruiert worden ist. Damit ist diese Möglichkeit für große Netze ebenfalls nicht zu gebrauchen.

Glücklicherweise hat es in den letzten Jahren, besonders in der ersten Dekade des zweiten Jahrtausends, entscheidende Fortschritte gegeben. In diesem Abschnitt betrachten wir den Ansatz der *Kontraktionshierarchien* (*Contraction Hierarchies*).

Die grundsätzliche Vorgehensweise ist wie folgt:

- Zu einem gegebenen Graphen  $G$  konstruiert man eine Kontraktionshierarchie  $CH(G)$ . Dabei werden  $G$  einige neue Kanten hinzugefügt.
- In  $CH(G)$  kann man einen kürzesten Weg von  $A$  nach  $B$  sehr viel schneller berechnen als in  $G$ .

Wir betrachten die Konstruktion von  $CH(G)$  und die Kürzeste-Wege-Suche darin in den folgenden beiden Abschnitten.

### 7.3.1 Berechnung einer Kontraktionshierarchie für einen Graphen

Gegeben sei wieder ein gerichteter Graph, dessen Kanten mit Kosten beschriftet sind, also  $G = (V, E, \mu)$  mit  $\mu: E \rightarrow \mathbb{R}^+$ . Eine äquivalente Notation ist  $G = (V, \hat{E})$ , wobei  $\hat{E} = \{(p, q, \alpha) \mid (p, q) \in E \wedge \mu((p, q)) = \alpha\}$ . Damit können wir kostenmarkierte Kanten als Tripel schreiben.

Vorbereitend werden Schleifen im Graphen, also Kanten der Form  $(p, p, \alpha)$ , entfernt, da sie in kürzesten Wegen nicht vorkommen können.

Eine Kontraktionshierarchie wird in einer Folge von Kontraktionsschritten berechnet. In jedem Schritt wird der gegebene Graph transformiert. Dabei wird ein Knoten  $v$  mit seinen inzidenten Kanten aus dem Graphen entfernt und es werden ggf. einige neue Kanten hinzugefügt. Eine neue Kante verbindet einen Vorgänger  $p$  von  $v$  mit einem Nachfolger  $q$  von  $v$  und sie wird genau dann eingefügt, wenn der kürzeste Weg von  $p$  nach  $q$  über  $v$  verläuft, die Distanz beim Löschen von  $v$  also größer würde. Ggf. vorhandene weitere Kanten (“Parallelkanten”) von  $p$  nach  $q$  mit höheren Kosten werden gelöscht.

Formal lässt sich dies so beschreiben: Für einen gegebenen Graphen  $G = (V, \hat{E})$  bezeichne  $spc(v, w, G)$  die Kosten eines kürzesten Pfades von  $v$  nach  $w$  in  $G$ . Seien

$$In(v) := \{(p, v, \alpha) \in \hat{E}\}$$

$$Out(v) := \{(v, q, \alpha) \in \hat{E}\}$$

die Mengen der eingehenden und ausgehenden Kanten von  $v$ . Der Graph, den wir aus dem Graphen  $G$  nach Löschen des Knotens  $v$  mit seinen eingehenden und ausgehenden Kanten erhalten, sei

$$del(G, v) := (V \setminus \{v\}, \hat{E} \setminus (In(v) \cup Out(v))).$$

Die Kanten, die beim Löschen von  $v$  neu einzufügen sind, seien definiert als

$$C_{new}(G, v) := \{ (p, q, \gamma) \mid (p, v, \alpha) \in \hat{E} \wedge (v, q, \beta) \in \hat{E} \wedge \gamma = \alpha + \beta \wedge \\ spc(p, q, del(G, v)) > \gamma \}$$

Die Menge der noch zu löschen Parallelkanten sei

$$Parallel(G, v) := \{ (p, q, \alpha) \in \hat{E} \mid \exists (p, q, \beta) \in C_{new}(G, v) \}$$

Die Kontraktion von  $G$  an  $v$  sei definiert als

$$C(G, v) := del(G, v) \cup C_{new}(G, v) \setminus Parallel(G, v)$$

Der Algorithmus zur Berechnung einer Kontraktionshierarchie wählt in jedem Schritt einen Knoten  $v$  des aktuellen Graphen  $G$  und berechnet für ihn die Kontraktion  $C(G, v)$ . Die in diesem Schritt neu erzeugten Kanten werden einer Menge  $\hat{E}_{new}$  hinzugefügt. Außerdem merkt sich der Algorithmus, in welcher Reihenfolge die Knoten kontrahiert wurden. Der Algorithmus terminiert, wenn alle Knoten kontrahiert wurden. Anschließend werden die Knoten in der Reihenfolge ihrer Kontraktion umnummeriert und die Kanten entsprechend korrigiert.

**algorithm**  $CH(G)$

{ Eingabe ist  $G = (V, \hat{E})$  - ein kostenmarkierter Graph.  
Ausgabe ist  $G'''$  - eine Kontraktionshierarchie für  $G$  }

```

 $\hat{E}_{new} := \emptyset;$ 
 $L :=$  die leere Liste;
 $G' := G;$ 
while  $G' \neq (\emptyset, \emptyset)$  do
    wähle einen Knoten  $v \in G'$ ;
     $\hat{E}_{new} := \hat{E}_{new} \cup C_{new}(G', v);$ 
     $G' := C(G', v);$ 
     $L := append(L, v);$ 
end while
 $G' := (V, \hat{E} \cup \hat{E}_{new});$ 
verwende die Indizes in der Liste  $L = v_{i1}, \dots, v_{in}$  als neue Knotenidentifikatoren,  
also  $v_1 := v_{i1}, v_2 := v_{i2}$ , und korrigiere die Kanten in  $G'$  entsprechend. Ergebnis  
der Umnummerierung sei  $G'''$ ;
return  $G'''$ 
end CH.

```

Ergebnis ist der Ausgangsgraph, erweitert um die Menge neu berechneter Kanten  $\hat{E}_{new}$ , mit neuen Knotennummern. Einen so berechneten Graphen bezeichnen wir als *Kontraktionshierarchie*.

Der Algorithmus verwendet implizit Aufrufe des Algorithmus von Dijkstra, um die Kosten des kürzesten Weges  $spc(p, q, del(G, v))$  zu berechnen. Da sehr viele Aufrufe dieser Funktion auftreten, ist es gut, die Kosten pro Suche zu begrenzen. Dazu kann man etwa die Suchtiefe beschränken (z.B. nur bis zu einer Pfadlänge von  $k = 5$  vom Startknoten aus) oder auch die Distanz (Abbruch, wenn der Zielknoten bis zur Distanz  $\gamma$  nicht erreicht wurde). Es ist nicht tragisch, wenn durch eine solche Beschränkung ein existierender kürzerer Weg übersehen wird; dies führt nur dazu, dass eine Kante zuviel eingefügt wird, was bei der Verwendung der Kontraktionshierarchie keine Fehler verursacht.

Der Algorithmus ist so formuliert, dass Knoten nicht einfach in einer vorgegebenen Reihenfolge kontrahiert werden (etwa der ihrer ursprünglichen Nummerierung), sondern in jedem Kontraktionsschritt gewählt werden. Tatsächlich hat die Wahl einer geschickten Reihenfolge dramatische Auswirkungen sowohl auf die Laufzeit als auch auf die Qualität des Ergebnisses (worin die Qualität liegt, sehen wir im nächsten Abschnitt).

Bei der Wahl des nächsten Knotens können verschiedene Heuristiken gewählt werden. Grundsätzlich ist anzustreben, die Zahl neu erzeugter Kanten zu minimieren. Mögliche Heuristiken dafür sind z.B.

- Kontrahiere  $v$  jetzt, falls die potentielle Zahl neu erzeugter Kanten geringer ist als die gelöschter Kanten, also falls  $|In(v)| \cdot |Out(v)| < |In(v)| + |Out(v)|$ . Andernfalls verschiebe die Kontraktion von  $v$  auf einen späteren Zeitpunkt.
- Kontrahiere  $v$  jetzt, falls die wirkliche Zahl neu erzeugter Kanten geringer ist als die gelöschter Kanten. Dazu müssen die neu zu erzeugenden Kanten zunächst berechnet werden unter Verwendung der Funktion  $spc$ . Diese Berechnung war vergebens, wenn die Kontraktion verschoben wird. Trotzdem lohnt sich der Aufwand, wie praktische Experimente zeigen.
- Kontrahiere stets Knoten, für die die Differenz aus der Anzahl neu erzeugter und gelöschter Kanten minimal ist.

Aus dem Gesagten wird klar, dass eine präzise Laufzeitanalyse kaum möglich ist.

### 7.3.2 Suche eines kürzesten Weges in einer Kontraktionshierarchie

Warum lassen sich nun in einer Kontraktionshierarchie kürzeste Wege schneller finden?

Gegeben sei die Kontraktionshierarchie als Graph  $G = (V, E, \mu)$  mit  $V = \{1, \dots, n\}$ . Tatsächlich spielen die Knotennummern eine entscheidende Rolle. Da es keine Kanten der Form  $(p, p)$  gibt, kann man jede Kante  $(p, q) \in E$  klassifizieren als

- *Aufwärtskante*, falls  $p < q$
- *Abwärtskante*, falls  $p > q$

Damit wird die Menge der Kanten  $E$  zerlegt in die Menge der Aufwärtskanten  $E_{\uparrow}$  und die Menge der Abwärtskanten  $E_{\downarrow}$ . Also  $G = (V, E_{\uparrow} \cup E_{\downarrow}, \mu)$ .

Es gilt nun der folgende erstaunliche Satz:

**Satz 7.6:** Sei  $G = (V, E_{\uparrow} \cup E_{\downarrow}, \mu)$  eine Kontraktionshierarchie und seien  $v, w \in V, v \neq w$ . Falls es in  $G$  einen kürzesten Weg von  $v$  nach  $w$  gibt, so existiert auch ein kürzester Weg von  $v$  nach  $w$  der Form  $v = v_0, \dots, v_k, \dots, v_m = w$ ,  $0 \leq k \leq m$ , für den gilt:

$$\begin{aligned} 0 \leq i < k &\Rightarrow (v_i, v_{i+1}) \in E_{\uparrow} \wedge \\ k \leq i < m &\Rightarrow (v_i, v_{i+1}) \in E_{\downarrow} \end{aligned}$$

□

Das heißt, es existiert ein kürzester Weg, der vom Startknoten aus bis zu einem “höchsten” Knoten nur über Aufwärtskanten verläuft, von dort aus bis zum Zielknoten nur über Abwärtskanten.

**Beweis:** Wir führen einen Widerspruchsbeweis. Nehmen wir also an, der Satz gilt nicht. Dann existiert ein kürzester Weg von  $v$  nach  $w$ , aber kein kürzester Weg der gewünschten Form. Betrachten wir also einen kürzesten Weg  $v = v_0, \dots, v_m = w$  von  $v$  nach  $w$ . Da er nicht die vom Satz behauptete Form hat, muss er ein lokales Minimum enthalten, also eine Teilfolge  $v_i, v_{i+1}, v_{i+2}$  mit  $v_i > v_{i+1}$  und  $v_{i+1} < v_{i+2}$ . Diese Situation ist in Abbildung 7.12 illustriert.

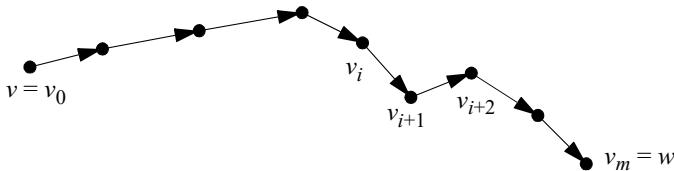


Abbildung 7.12: Ein kürzester Pfad von  $v$  nach  $w$  mit einem lokalen Minimum

Da Knoten  $v_{i+1}$  kleiner ist als  $v_i$  und auch als  $v_{i+2}$ , ist er vor diesen beiden kontrahiert worden. Das heißt, zum Zeitpunkt, als  $v_{i+1}$  kontrahiert wurde, existierten im Graphen diese beiden Knoten und demnach auch die Kanten  $(v_i, v_{i+1})$  und  $(v_{i+1}, v_{i+2})$ . Knoten  $v_i$  war ein Vorgänger von  $v_{i+1}$  und Knoten  $v_{i+2}$  ein Nachfolger. Gemäß Algorithmus *CH* wurde an dieser Stelle entschieden, ob eine Kante  $(v_i, v_{i+2})$  einzufügen war.

Falls die Kante  $(v_i, v_{i+2})$  erzeugt wurde, dann existiert im Graphen der Pfad  $v = v_0, \dots, v_i, v_{i+2}, \dots, v_m = w$ , ein gleich langer kürzester Weg von  $v$  nach  $w$ , der dieses lokale

Minimum nicht besitzt. Wir müssten dann diesen Pfad weiterbetrachten, der zunächst die vom Satz behauptete Form nicht verletzt (falls er nicht noch ein anderes lokales Minimum besitzt). Mit dieser Argumentation können wir jedes lokale Minimum “beseitigen”, für das eine solche Kante erzeugt wurde.

Falls die Kante  $(v_i, v_{i+2})$  nicht erzeugt wurde, existierte im Graphen zum Zeitpunkt der Kontraktion eine kürzere Verbindung von  $v_i$  nach  $v_{i+2}$  als die über  $v_{i+1}$ . Diese kürzere Verbindung existiert auch jetzt im Graphen (da alle jemals vorhandenen Kanten enthalten sind). Das bedeutet, dass der betrachtete Pfad  $v = v_0, \dots, v_m = w$  kein kürzester Weg ist, ein Widerspruch zur Annahme.

Insgesamt haben wir gezeigt, dass es nicht möglich ist, ein Gegenbeispiel zu finden, also einen kürzesten Weg mit einem lokalen Minimum, zu dem es nicht auch einen kürzesten Weg ohne dieses Minimum gäbe. Es folgt die Aussage des Satzes.  $\square$

Die im Satz gezeigte Eigenschaft von Kontraktionshierarchien kann man auf verschiedene Arten ausnutzen, um kürzeste Wege zu berechnen, nämlich:

1. Berechnen und Verschneiden von Nachfolger- und Vorgängermengen
2. Hub Labeling
3. Bidirektionale Dijkstra-Suche

### Berechnen und Verschneiden von Nachfolger- und Vorgängermengen

Dieser Algorithmus hat die folgenden Schritte:

1. Berechne mit dem Algorithmus von Dijkstra alle kürzesten Wege vom Startknoten  $v$  aus im *Aufwärtsgraphen*  $G\uparrow = (V, E\uparrow, \mu)$ . Dabei wird die Menge der Knoten *Forward*( $v$ ) ermittelt, die man von  $v$  aus aufwärts laufend erreichen kann. Außerdem werden die kürzesten Wege zu diesen Knoten im Baum der kürzesten Wege dargestellt und die Distanz vom Startknoten im Knoten gespeichert. Die Elemente von *Forward*( $v$ ) seien Tripel der Form  $(p, P, d)$  wobei  $p$  der von  $v$  aus erreichte Knoten ist,  $P$  der kürzeste Weg<sup>1</sup> von  $v$  nach  $p$ , und  $d$  die Distanz von  $v$  nach  $p$ . Der Pfad  $P$  hat die Form  $p_1, \dots, p_k$  mit  $v = p_1$  und  $p_k = p$ .
2. Sei  $E_{\downarrow reverse} := \{(p, q) \mid (q, p) \in E\downarrow\}$  mit  $\mu_{reverse}(p, q) = \mu(q, p)$ . Berechne mit dem Algorithmus von Dijkstra alle kürzesten Wege vom Zielknoten  $w$  aus im Graphen  $G_{\downarrow reverse} = (V, E_{\downarrow reverse}, \mu_{reverse})$ . In diesem Graphen sind alle abwärts führenden Kanten herumgedreht, so dass man vom Zielknoten aus aufwärts laufend als

---

1. Dies ist eine vereinfachende Notation. Genauer enthalten Tripel nicht den kürzesten Weg  $P$ , sondern einen Zeiger auf den Vorgänger des Knotens im Baum der kürzesten Wege, aus dem sich der Pfad  $P$  in Linearzeit rekonstruieren lässt.

Nachfolger alle ursprünglichen Vorgänger im Abwärtsgraphen finden kann. Die so ermittelte Menge von Knoten nennen wir  $Backward(w)$ .  $Backward(w)$  enthält analog zu  $Forward(v)$  Tripel der Form  $(q, P, d)$ .

3. Berechne eine Menge  $Cand$  als sog. Verbund von  $Forward(v)$  mit  $Backward(w)$ , d. h., bestimme alle Paare von Knoten  $(p_i, P_i, d_i) \in Forward(v)$ ,  $(q_j, P_j, d_j) \in Backward(w)$  mit  $p_i = q_j$ .  $Cand$  enthält Kandidaten für maximale Knoten entsprechend  $v_k$  aus Satz 7.6.
4. Sei  $((p_i, P_i, d_i), (q_j, P_j, d_j)) \in Cand$  ein Paar von Knoten, für das  $d_i + d_j$  minimal ist. Dann ist  $P_i \circ reverse(P_j)$  ein kürzester Weg von  $v$  nach  $w$  in der Kontraktionshierarchie  $G = (V, E_\uparrow \cup E_\downarrow, \mu)$ .<sup>2</sup>

Den Verbund in Schritt 3. kann man z. B. berechnen, indem man  $Forward(v)$  nach  $p_i$  und  $Backward(w)$  nach  $q_j$  sortiert und dann gemeinsame Werte  $p_i = q_j$  in einem parallelen Listendurchlauf ermittelt.

Der Algorithmus ist korrekt: er findet mit Sicherheit ein gemeinsames Element  $p_i = q_j$  aufgrund von Satz 7.6. Denn der “höchste” Knoten  $v_k$  aus Satz 7.6 ist von  $v$  aus vorwärts und von  $w$  aus rückwärts zu erreichen und liegt somit in der Menge  $Forward(v)$  und  $Backward(w)$ . Da alle gemeinsamen Knoten ermittelt werden, wird auch  $v_k$  gefunden. Da  $Forward(v)$  und  $Backward(w)$  durch Kürzeste-Wege-Suche ermittelt wurden, ist  $P_i \circ reverse(P_j)$  ein kürzester Weg.

Warum ist dieses doch etwas komplizierte Verfahren effizient? Das liegt daran, dass für einen beliebigen Knoten  $v$  eines sehr großen Netzwerks die Menge der “aufwärts” erreichbaren Knoten verschwindend klein ist. In eigenen Experimenten haben wir folgende Zahlen ermittelt (Tabelle 7.1). Aus OpenStreetMap-Daten (Stand Anfang 2017) wurde ein Straßennetz für Deutschland konstruiert.

Die Durchschnitts- und Maximalgrößen der Mengen  $Forward(v)$  und  $Backward(v)$  wurden jeweils auf einer zufällig ausgewählten Menge von 300 Startknoten berechnet.

Da der Aufwand des Algorithmus von Dijkstra grob proportional zur Anzahl erreichbarer Knoten ist, terminiert die Suche auch auf dem riesigen Graphen in Sekundenbruchteilen. Auch das Sortieren und der Verbund der Mengen mit jeweils ca. 2000 Elementen benötigen nur sehr wenig Zeit.

---

2. Darin bezeichnet  $\circ$  den Konkatenationsoperator für Folgen und  $reverse$  einen Operator zur Umkehrung von Folgen, also  $reverse(< v_1, \dots, v_n >) = < v_n, \dots, v_1 >$  (vgl. Abschnitt 3.1).

Anzahl Knoten des Ausgangsnetzes	8459045
Anzahl Kanten des Ausgangsnetzes	18765418
Anzahl der durch Berechnen der Kontraktionshierarchie hinzugefügten Kanten	15377664
Gesamtzahl Kanten der Kontraktionshierarchie	34143082
Durchschnittliche Anzahl von Knoten von $Forward(v)$	1792.67
Maximale Anzahl von Knoten von $Forward(v)$	2669
Durchschnittliche Anzahl von Knoten von $Backward(v)$	1823.30
Maximale Anzahl von Knoten von $Backward(v)$	2593

Tabelle 7.1: Experimente mit einer Kontraktionshierarchie für Deutschland

### Hub Labeling

Hub Labeling ist eigentlich nur eine einfache Variante der vorherigen Technik. Anstatt die Mengen  $Forward(v)$  und  $Backward(w)$  zur Anfragezeit zu berechnen, werden diese Mengen vorab direkt nach dem Aufbau der Kontraktionshierarchie für alle Knoten des Netzes berechnet und gespeichert. Natürlich ist der Zeitbedarf dafür und der Speicherplatzbedarf beträchtlich. Mit einigen Optimierungen beim Bau der Kontraktionshierarchie – also geschickter Wahl der Knotenreihenfolge – lassen sich die Größen der Forward- und Backward-Mengen auf jeweils einige Hundert Knoten reduzieren; diese kann man noch besonders kompakt abspeichern.

Bei der Kürzeste-Wege-Suche wird dann nur noch ein paralleler Durchlauf durch die beiden in Arrays dargestellten Mengen (die sog. *Label*) benötigt. Damit ist die Suche extrem schnell.

Die Größe der Forward- und Backward-Label macht vor allem die oben erwähnte Qualität des Ergebnisses der Kontraktion aus.

### Bidirektionale Variante des Algorithmus von Dijkstra

In den bisher beschriebenen Suchverfahren wurden zunächst die Mengen  $Forward(v)$  und  $Backward(w)$  jeweils mit dem Algorithmus von Dijkstra berechnet. Hier ist die Idee, die Vorrwärtssuche vom Startknoten aus (über Aufwärtskanten) und die Rückwärtssuche vom Zielknoten aus (über invertierte Abwärtskanten) miteinander zu verzahnen. Der Vorteil ist, dass man im Allgemeinen früher abbrechen kann, d. h., die beiden Mengen

müssen nicht komplett berechnet werden. Außerdem findet man gemeinsame Elemente (Menge  $Cand$  in Schritt 3.) schon während der Suche.

Eingabe für den Algorithmus *bidijkstra* (Abbildung 7.13), ist die Kontraktionshierarchie  $G = (V, E^\uparrow \cup E^\downarrow, \mu)$ , hier notiert als Graph  $G^* = (V, E^{*\uparrow} \cup E^{*\downarrow}, cost)$  mit  $E^{*\uparrow} := E^\uparrow$ ,  $E^{*\downarrow} := E^\downarrow_{\text{reverse}}$  und

$$\text{cost}(p, q) = \begin{cases} \mu(p, q) & \text{falls } (p, q) \in E^{*\uparrow} \\ \mu_{\text{reverse}}(p, q) & \text{falls } (p, q) \in E^{*\downarrow} \end{cases}$$

Der Grund für die Umbenennung liegt darin, dass wir im Algorithmus Vorwärts- und Rückwärtssuche einheitlich beschreiben und nur durch einen Index  $dir \in \{\uparrow, \downarrow\}$  unter-

**algorithm** *bidijkstra* ( $G^*, v, w$ )  
 { Eingaben sind eine Kontraktionshierarchie  $G^*$ , Startknoten  $v$  und Zielknoten  $w$   
 Ausgabe ist ein kürzester Weg von  $v$  nach  $w$  }

```

 $d_\uparrow := empty; d_\uparrow.init(\infty); d_\uparrow(v) := 0;$ 
 $d_\downarrow := empty; d_\downarrow.init(\infty); d_\downarrow(w) := 0;$ 
 $Q_\uparrow := \{(0, v)\};$ 
 $Q_\downarrow := \{(0, w)\};$ 
 $dmin := \infty;$ 
 $dir := \uparrow;$ 
```

```

while (  $Q_\uparrow \neq \emptyset$  or  $Q_\downarrow \neq \emptyset$  ) and (  $dmin > min(Q_\uparrow)$  or  $dmin > min(Q_\downarrow)$  ) do
  (  $_$ ,  $p$  ) :=  $Q_{dir}$ .deletemin();
  foreach  $(p, q) \in E^{*dir}$  do
    if  $d_{dir}(p) + cost(p, q) < d_{dir}(q)$  then
       $d_{dir}(q) := d_{dir}(p) + cost(p, q);$ 
       $Q_{dir}.update(d_{dir}(q), q)$ 
    end if;
    if  $d_\uparrow(q) + d_\downarrow(q) < dmin$  then  $dmin := d_\uparrow(q) + d_\downarrow(q)$  ;  $u := q$  end if
  end for;
  if  $Q_{inv(dir)} \neq \emptyset$  then  $dir := inv(dir)$  end if
end while;
return  $path(v, u) \circ reverse(path(u, w))$ 
end bidijkstra.
```

Abbildung 7.13: Algorithmus für bidirektionale Kürzeste-Wege-Suche

scheiden wollen. Mit der Funktion  $inv$  können wir die Richtung umkehren:  $inv(\uparrow) = \downarrow$  und  $inv(\downarrow) = \uparrow$ .

Der Algorithmus benutzt zwei Datenstrukturen, nämlich eine Abbildung  $d$  und eine Priority Queue  $Q$ . Von beiden gibt es zwei Instanzen, jeweils eine für die Vorwärts- und eine für die Rückwärtssuche, also  $d^\uparrow$  und  $d^\downarrow$  bzw.  $Q^\uparrow$  und  $Q^\downarrow$ .

Die Abbildung  $d$  verwaltet bisher bekannte Distanzen zu Knoten vom jeweiligen Startknoten aus. Mit  $d := empty$  wird die leere Abbildung erzeugt und wir schreiben  $d(u) := x$ , um  $u$  die Distanz  $x$  zuzuweisen und  $d(u)$ , um auf die Distanz von  $u$  zuzugreifen (anstelle der Notationen  $assign(d, u, x)$  bzw.  $apply(d, u)$  aus Abschnitt 3.4). Wir fügen eine neue Operation  $init$  hinzu, die allen Werten des *domain* einen Wert zuweist. Im Algorithmus werden damit alle Knotendistanzen als  $\infty$  initialisiert.

Man kann die Abbildung mit Hilfe irgendeiner effizienten Dictionary-Implementierung realisieren, z.B. einem AVL-Baum. Die *init*-Operation speichert einfach ihr Argument, das dann für jeden *domain*-Wert zurückgegeben wird, der nicht explizit in der Datenstruktur zu finden ist; die Initialisierung benötigt damit nur konstante Zeit.

Die Priority-Queue  $Q$  speichert nun Paare (*Distanz*, *Knoten*) anhand ihrer ersten Komponente. Eine Operation *update* erlaubt es einerseits, für nicht vorhandene Knoten ihre Distanz einzufügen und andererseits, für bereits vorhandene Knoten ihre Distanz zu aktualisieren. Die Operation *min* liefert den minimalen Wert der ersten Komponente. Die Implementierung so modifizierter Priority-Queues, die auch den Zugriff über Knotennummern erlauben, wurde bereits in Abschnitt 7.1 skizziert.

Der Algorithmus betrachtet in jedem Schritt alternierend die Nachfolger eines Knotens der Vorwärts- oder der Rückwärtssuche, solange beide Queues nicht leer sind, danach nur noch die der nichtleeren Queue. Sobald ein Knoten auf beiden Seiten erreicht wurde (die Summe der Distanzen ist endlich) wird eine Schranke  $d_{min}$  für die minimale Distanz vom Start- zum Zielknoten aktualisiert. Dabei wird der Knoten  $u$  gemerkt, über den die minimale Distanz realisiert wurde. Der Algorithmus terminiert, sobald entweder beide Queues leer sind oder in beiden Queues die Minimaldistanz größer ist als der bekannte kürzeste Weg. Der Pfad über  $u$  wird dann als kürzester Weg zurückgegeben.<sup>3</sup>

---

3. Dazu ist allerdings noch die Verwaltung der Kürzeste-Wege-Bäume, also das Merken des Vorgängers auf dem kürzesten Weg für jeden Knoten notwendig. Diese Aktionen haben wir der Übersichtlichkeit halber weggelassen.

### Pfadexpansion

Die drei beschriebenen Verfahren liefern jeweils den kürzesten Weg in der Kontraktionshierarchie. Natürlich ist man eigentlich interessiert am kürzesten Weg im ursprünglichen Graphen. Es ist aber kein großes Problem, diesen zu rekonstruieren. Dazu speichert man mit einer neu erzeugten Kante, die beim Kontrahieren (Löschen) des Knotens  $v$  erzeugt wurde, den Knoten  $v$  mit ab, also als Quadrupel (*Source, Target, Cost, Middle*) mit undefiniertem Mittelknoten für Originalkanten des Graphen. Damit gilt:

$$\text{path}(s, t, c, m) = \begin{cases} (s, t) & \text{falls } m = \perp \\ \text{path}(s, m, c_1, m_1) \circ \text{path}(m, t, c_2, m_2) & \text{sonst} \end{cases}$$

Das ist die gleiche Technik, die schon beim Algorithmus von Floyd zur Berechnung kürzester Pfade aus Kanten beschrieben wurde (Abschnitt 7.2, vgl. Abbildung 7.11).

## 7.4 Transitive Hülle

Manchmal interessiert man sich auch nur dafür, ob es zwischen je zwei Knoten  $i$  und  $j$  einen Pfad (irgendeiner Länge  $\geq 1$ ) gibt. Die Kantenbewertung wird dabei außer acht gelassen. Gegeben sei ein Graph  $G = (V, E)$ . Die *transitive Hülle* von  $G$  ist der Graph  $\bar{G} = (V, \bar{E})$ , wobei  $\bar{E}$  definiert ist durch

$$(v, w) \in \bar{E} \Leftrightarrow \text{es gibt in } G \text{ einen Pfad von } v \text{ nach } w$$

Man erhält einen Algorithmus zur Berechnung der transitiven Hülle als vereinfachte Version des Algorithmus von Floyd. Dieser ist bekannt als *Marshalls Algorithmus*.

**Selbsttestaufgabe 7.4:** Überlegen Sie sich eine vereinfachte Version des Algorithmus *Floyd*, die die transitive Hülle eines Graphen berechnet und implementieren Sie diese für die Adjazenzmatrix-Darstellung (Marshalls Algorithmus).  $\square$

## 7.5 Starke Komponenten

Der folgende Algorithmus berechnet die starken Komponenten eines gegebenen gerichteten Graphen  $G = (V, E)$ .

**algorithm** *StrongComponents* ( $G$ )

1. Mit einem Tiefendurchlauf durch  $G$  berechne alle Depth-First-Spannbäume von  $G$ . Nummeriere dabei die Knoten in der Reihenfolge der *Beendigung* ihrer rekursiven Aufrufe.
2. Konstruiere zu  $G$  einen “inversen” Graphen  $G_r$ , indem bei jeder Kante in  $G$  die Richtung umgekehrt wird.
3. Konstruiere die Depth-First-Spannbäume von  $G_r$ . Beginne dazu mit dem Knoten mit der höchsten Nummer aus Schritt 1. Wenn jeweils ein Spannbaum komplett ist, setze mit dem Knoten mit der verbliebenen höchsten Nummer fort.
4. Die Knotenmenge jedes in Schritt 3 entstandenen Spannbaumes bildet gerade die Knotenmenge einer starken Komponente in  $G$ .

**end** *StrongComponents*.

Wie man sieht, ist der oben beschriebene Tiefendurchlauf das entscheidende Hilfsmittel für diesen Algorithmus.

**Beispiel 7.7:** Sei  $G$  der Graph

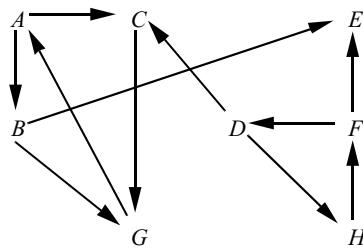


Abbildung 7.14: Ausgangsgraph

In den Einzelschritten des Algorithmus erhält man folgende Ergebnisse:

1. Depth-First-Spannbäume und Nummerierung

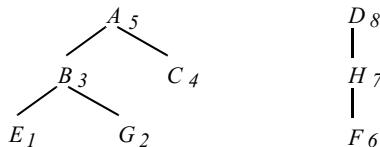


Abbildung 7.15: Erster Schritt

2. Inverser Graph  $G_r$

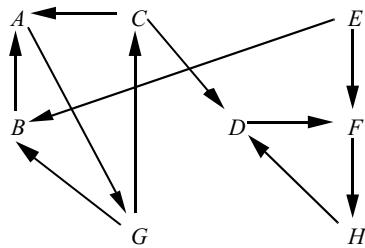


Abbildung 7.16: Zweiter Schritt

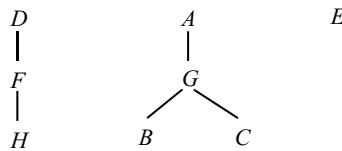
3. Spannbäume für  $G_r$ 

Abbildung 7.17: Dritter Schritt

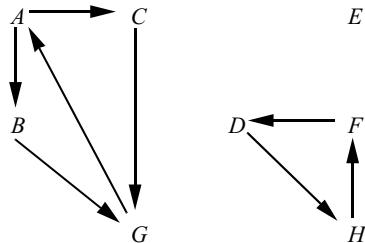
4. Starke Komponenten von  $G$ 

Abbildung 7.18: Ergebnis

□

Es muss natürlich noch bewiesen werden, dass der Algorithmus auch korrekt ist, das heißt, zu zeigen ist Folgendes:

**Lemma 7.8:** Die Knoten  $v$  und  $w$  liegen in derselben starken Komponente  $\Leftrightarrow v$  und  $w$  liegen in demselben  $G_r$ -Spannbaum.

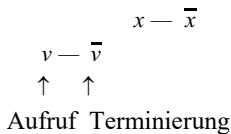
**Beweis:**

Beweisrichtung " $\Rightarrow$ ": Es gibt in  $G$  Pfade  $v \Rightarrow w$  und  $w \Rightarrow v$ , also auch in  $G_r$ . Sei o. B. d. A.  $v$  der erste Knoten (von  $v$  und  $w$ ), der beim Tiefendurchlauf in  $G_r$  besucht wird. Es gibt einen Pfad von  $v$  nach  $w$ , deshalb muss  $w$  im gleichen Spannbaum wie  $v$  liegen.

Beweisrichtung " $\Leftarrow$ ": Sei  $x$  die Wurzel des  $G_r$ -Spannbaumes, in dem  $v$  und  $w$  liegen. Dann gibt es einen Pfad  $x \Rightarrow v$  in  $G_r$  und somit gibt es einen Pfad  $v \Rightarrow x$  in  $G$ .

Der Knoten  $x$  hat eine höhere Nummer als  $v$ , denn  $v$  war in Schritt 3 noch nicht besucht, als der Aufruf für  $x$  erfolgte. Andernfalls wäre  $v$  vor  $x$  aufgerufen worden. In Schritt 1 terminierte also  $v$  vor  $x$ . Es gibt drei prinzipielle Möglichkeiten für die zeitliche Abfolge von Aufrufen und Beendigungen von  $v$  und  $x$ :

- (a)  $v$  wird beendet, bevor  $x$  aufgerufen wird.



- (b)  $x$  wird nach Aufruf, aber vor Beendigung von  $v$  aufgerufen.

$$\begin{array}{c} x - - \bar{x} \\ v - \bar{v} \end{array}$$

- (c)  $x$  wird vor Aufruf von  $v$  aufgerufen.

$$\begin{array}{c} x - - - \bar{x} \\ v - \bar{v} \end{array}$$

Wir untersuchen, welche Fälle tatsächlich möglich sind:

Fall (a): Nach der obigen Überlegung gibt es einen Pfad  $v \Rightarrow x$  in  $G$ . Die Abfolge (a) ist nur möglich, wenn es einen Zwischenknoten  $z$  gibt, der im gleichen  $G$ -Spannbaum liegt wie  $v$ , der vor  $v$  besucht wurde und über den der Pfad zu  $x$  führt.

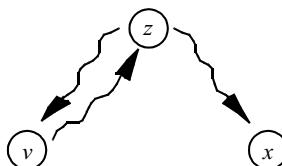


Abbildung 7.19: Aufbau des Graphen im Fall (a)

Dann muss aber die Laufzeit von  $z$  die von  $v$  und  $x$  einschließen, also folgende Abfolge auftreten:

$$\begin{array}{ccccccc} z & - & - & - & - & - & - & - & z \\ v & - & \overline{v} & & x & - & \overline{x} & \end{array}$$

In diesem Fall muss  $z$  in Schritt 1 aber eine höhere Nummer als  $x$  erhalten haben. Als  $x$  in Schritt 3 aufgerufen wurde, war  $z$  noch nicht besucht (denn  $v$  war noch nicht besucht). Also wäre  $z$  Wurzel des  $G_r$ -Spannbaumes geworden. Das ist ein Widerspruch zur Annahme, dass  $x$  Wurzel des Teilbaums ist.

Fall (b): Wenn  $x$  zur Laufzeit von  $v$  aufgerufen worden wäre, wäre  $x$  auch vorher terminiert.

Die Fälle (a) und (b) sind also nicht möglich, weil es einen Pfad  $v \Rightarrow x$  in  $G$  gibt. Es bleibt also nur Fall (c). Das heißt aber, es gibt auch einen Pfad  $x \Rightarrow v$  in  $G$ .

Analog zeigt man, dass es Pfade  $w \Rightarrow x$  und  $x \Rightarrow w$  gibt. Somit gibt es aber auch Pfade  $v \Rightarrow x \Rightarrow w$  und  $w \Rightarrow x \Rightarrow v$ . Also sind  $v$  und  $w$  stark verbunden und liegen in derselben Komponente.  $\square$

**Selbsttestaufgabe 7.5:** Suchen Sie mit Hilfe des Algorithmus *StrongComponents* die starken Komponenten aus dem Graphen  $G$  der Aufgabe 6.1. Geben Sie dabei auch die Zwischenergebnisse entsprechend dem Beispiel 7.7 an.  $\square$

**Selbsttestaufgabe 7.6:** Mit welcher Zeitkomplexität lässt sich der Algorithmus *StrongComponents* implementieren?  $\square$

## 7.6 Ungerichtete Graphen

**Definition 7.9:** Ein *ungerichteter Graph*  $G = (V, E)$  ist ein gerichteter Graph  $(V, E)$ , in dem die Relation  $E$  symmetrisch ist, d. h.

$$(v, w) \in E \Rightarrow (w, v) \in E.$$

Wir zählen die Kanten  $\{(v, w), (w, v)\}$  als *eine* Kante des ungerichteten Graphen und stellen sie graphisch folgendermaßen dar:



Die für gerichtete Graphen eingeführten Begriffe lassen sich im Allgemeinen übertragen. Ein *Zyklus* muss allerdings mindestens drei verschiedene Knoten enthalten, so dass  $v -$

$w$  kein Zyklus ist. Wann immer Knoten  $v$  mit  $w$  verbunden ist, so ist auch  $w$  mit  $v$  verbunden. Deshalb entfällt der Begriff ‘‘stark verbunden’’; ebenso sprechen wir einfach von *Komponenten* anstatt von starken Komponenten (eine Komponente ist ein bezüglich Knoten und Kanten maximaler Teilgraph, in dem alle Knoten paarweise verbunden sind). Ein ungerichteter Graph (oder einfach ein *Graph*) heißt verbunden, wenn er nur aus einer einzigen Komponente besteht.

**Selbsttestaufgabe 7.7:** Geben Sie einen  $O(n)$ -Algorithmus an, der testet, ob ein ungerichteter Graph mit  $n$  Knoten einen Zyklus enthält.  $\square$

Ein verbundener, azyklischer Graph heißt *freier Baum*. Man kann daraus einen normalen Baum erhalten, indem man einen beliebigen Knoten als Wurzel auswählt und alle Kanten von der Wurzel weg orientiert. Freie Bäume haben einige interessante Eigenschaften:

1. Ein freier Baum mit  $n \geq 1$  Knoten hat genau  $n - 1$  Kanten.
2. Wenn man einem freien Baum eine beliebige Kante hinzufügt, entsteht ein Zyklus.

Die Darstellungsarten (Adjazenz/Kosten-Matrix, Adjazenzlisten) sind die gleichen wie bei gerichteten Graphen. Natürlich sind die Matrix und die Listen nun symmetrisch. Auch die Durchlaufarten *Depth-first* und *Breadth-first* sind anwendbar.

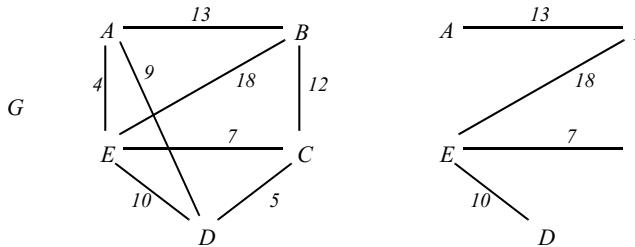
Die Berechnung verbundener *Komponenten* lässt sich hier ganz einfach durch einen Tiefendurchlauf (oder auch Breitendurchlauf) erledigen. Jeder entstehende Spannbaum enthält die Knoten einer Komponente.

Wir betrachten noch einen ausgewählten Algorithmus für ungerichtete Graphen.

## 7.7 Minimaler Spannbaum (Algorithmus von Kruskal)

Sei  $G = (V, E)$  ein verbundener Graph mit Kantenbewertung. Ein *Spannbaum* ist ein freier Baum, der alle Knoten in  $G$  enthält und dessen Kanten eine Teilmenge von  $E$  bilden. Die Kosten eines Spannbaumes ergeben sich als Summe der Kosten seiner Kanten. Gesucht ist für einen gegebenen Graphen  $G$  ein *minimaler Spannbaum*, also ein Spannbaum mit minimalen Kosten.

**Beispiel 7.10:** Abbildung 7.20 zeigt einen ungerichteten Graphen  $G$  und einen Spannbaum dazu, der allerdings nicht minimal ist.  $\square$



Ein Spannbaum (Kosten 48)

Abbildung 7.20: Graph mit (nicht-minimalem) Spannbaum

Minimale Spannbäume haben eine interessante Eigenschaft:

**Lemma 7.11:** Sei für  $G = (V, E)$   $\{U, W\}$  eine Zerlegung der Knotenmenge  $V$ . Sei  $(u, w)$  eine Kante in  $G$  mit minimalen Kosten unter allen Kanten  $\{(u', w') \mid u' \in U, w' \in W\}$ . Dann gibt es einen minimalen Spannbaum für  $G$ , der  $(u, w)$  enthält.

**Beweis:** Sei  $T = (V, \bar{E})$  ein beliebiger minimaler Spannbaum für  $G$ , der  $(u, w)$  nicht enthält. Dann gibt es in  $T$  mindestens eine, vielleicht aber auch mehrere Kanten, die  $U$  mit  $W$  verbinden.

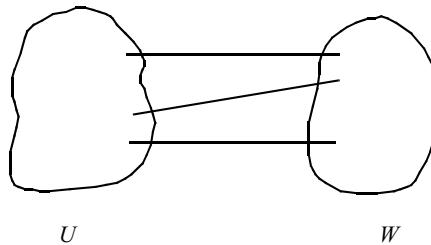


Abbildung 7.21: Beliebige Zerlegung

Der Spannbaum aus Abbildung 7.20 könnte z. B. zerlegt sein in  $U = \{A, C, E\}$  und  $W = \{B, D\}$  (Abbildung 7.22). Jede beliebige Zerlegung lässt sich also so darstellen, wie in Abbildung 7.21 gezeigt.

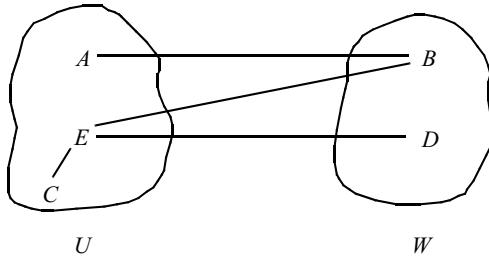
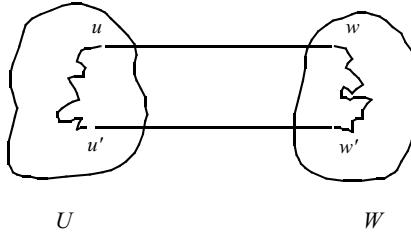


Abbildung 7.22: Mögliche Zerlegung des Spannbaums aus Abbildung 7.20

Wenn man nun  $T$  die Kante  $(u, w)$  hinzufügt, entsteht ein Graph  $T'$  mit einem Zyklus, der über  $u — w$  verläuft.  $u$  muss also noch auf einem anderen Weg mit  $w$  verbunden sein, der mindestens eine weitere Kante  $(u', w')$  enthält, die  $U$  mit  $W$  verbindet.

Abbildung 7.23: Zyklus über  $u — w$ 

Wenn man nun  $(u', w')$  aus  $T'$  löscht, entsteht ein Spannbaum  $T''$ , dessen Kosten höchstens so hoch sind wie die von  $T$ , da  $(u, w)$  minimale Kosten hat. Also ist  $T''$  ein minimaler Spannbaum, der  $(u, w)$  enthält.  $\square$

Dieses Lemma ist die Grundlage für den *Algorithmus von Kruskal* zur Konstruktion minimaler Spannbäume, der wie folgt vorgeht: Zu Anfang sei  $T$  ein Graph, der genau die Knoten von  $G$  enthält, aber keine Kanten. Die Kanten von  $G$  werden nun in der Reihenfolge steigender Kosten betrachtet. Wenn eine Kante zwei getrennte Komponenten von  $T$  verbindet, so wird sie in den Graphen  $T$  eingefügt und die Komponenten werden verschmolzen. Andernfalls wird die Kante ignoriert. Sobald  $T$  nur noch eine einzige Komponente besitzt, ist  $T$  ein minimaler Spannbaum für  $G$ .

Der Algorithmus benutzt eine Priority-Queue  $Q$  und eine Partition  $P$  (MERGE-FIND-Struktur, siehe Abschnitt 4.4). Sei  $G = (V, E)$  ein verbundener, kantenbewerteter Graph mit  $n$  Knoten.

```

algorithm Kruskal ( $G$ )
{berechne einen minimalen Spannbaum zum Graphen  $G$ }
initialisiere  $P$  so, dass jeder Knoten aus  $V$  eine eigene Komponente bildet;
sei  $T = (V, \emptyset)$ ;
füge alle Kanten aus  $E$  bzgl. ihrer Kosten in  $Q$  ein;
 $ncomp := n$ ;
while  $ncomp > 1$  do
     $(Q, (v, w)) := deletemin (Q)$ ;
     $a := find (P, v); b := find (P, w)$ ;
    if  $a \neq b$  then
         $insert (T, (v, w))$ ; {füge Kante  $(v, w)$  in den als Graphen dargestellten
        Baum  $T$  ein}
         $P := merge (P, a, b)$ ;
         $ncomp := ncomp - 1$ 
    end if
end while.

```

**Beispiel 7.12:** Beim Abarbeiten des Algorithmus mit dem Graphen  $G$  als Eingabe werden der Reihe nach die Kanten mit der Beschriftung 4, 5, 7 aufgenommen, diejenigen mit der Beschriftung 9 und 10 ignoriert, da sie keine Komponenten verbinden, und am Schluss die Kante mit der Beschriftung 12 aufgenommen. In Abbildung 7.24 sind im Spannbaum  $T$  die Kanten entsprechend ihrer Einfügereihenfolge markiert.  $\square$

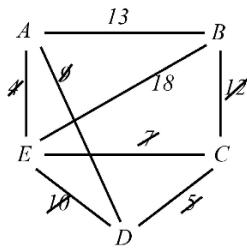
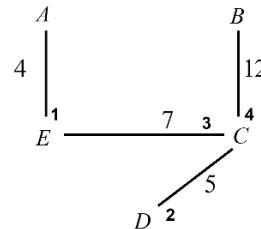
Graph  $G$ Spannbaum  $T$  (Kosten 28)

Abbildung 7.24: Algorithmus von Kruskal

Der Zusammenhang des Algorithmus mit dem Lemma ist folgender: Wenn die Kante  $(v, w)$  in  $T$  aufgenommen wird, verbindet sie zwei getrennte Komponenten  $C_a, C_b$ . Sei  $U = C_a$ ,  $W = V - C_a$ . Die Kante  $(v, w)$  hat minimale Kosten unter allen Kanten, die  $U$  mit  $W$  verbinden, denn alle billigeren Kanten sind schon betrachtet und haben  $C_a$  nicht "angeschlossen". Also ist  $T$  ein minimaler Spannbaum.

Die Analyse des Algorithmus ergibt für die einzelnen Schritte:

- Initialisierung von  $P$  und  $T$ :  $O(n)$
  - Initialisierung von  $Q$ :  $O(e \log e)$
  - Schleife
    - maximal  $e$  deletemin-Operationen:  $O(e \log e)$
    - maximal  $e$  find- und  $n$  merge-Operationen:
- Je nach Implementierung  $O(n \log n + e)$  oder  $O(e \log n)$

Es gilt:  $e \geq n - 1$  (da  $G$  ein verbundener Graph ist). Daher erhält man einen Gesamtaufwand von  $O(e \log e)$ .

## 7.8 Weitere Aufgaben

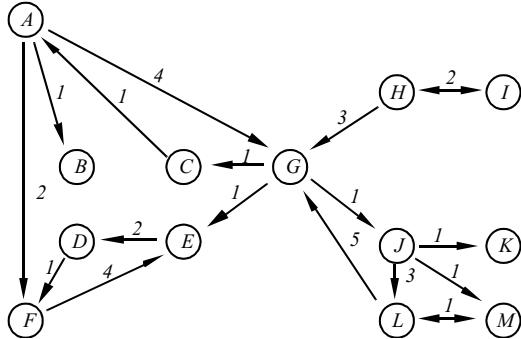
**Aufgabe 7.8:** Die Wurzel eines gerichteten Graphen ist ein Knoten  $r$ , von dem aus Pfade zu allen anderen Knoten des Graphen existieren. Schreiben Sie eine Methode, die feststellt, ob ein Graph eine Wurzel enthält.

**Aufgabe 7.9:** Gegeben sei ein gerichteter Graph mit  $e$  Kanten, sowie zwei Knoten des Graphen,  $s$  und  $t$ . Schreiben Sie eine Methode, die in  $O(e)$  Zeit eine maximale Menge knotendisjunkter Pfade von  $s$  nach  $t$  bestimmt.

Dabei gilt: Zwei Pfade zwischen  $s$  und  $t$  sind *knotendisjunkt*, wenn sie außer  $s$  und  $t$  keine gemeinsamen Knoten enthalten. Für die maximale Menge muss nur gelten, dass sich kein weiterer Pfad hinzufügen lässt, die maximale Anzahl von knotendisjunkten Pfaden muss *nicht* gefunden werden.

**Aufgabe 7.10:** Seien  $G = (V, E)$  ein zusammenhängender ungerichteter Graph und  $T$  ein depth-first-Spannbaum von  $G$ , der die Wurzel  $v$  hat. Sei  $H$  ein beliebiger Teilgraph von  $G$ . Beweisen oder widerlegen Sie, dass der Schnitt von  $T$  und  $H$  (beim Schnitt werden jeweils die Knoten- und Kantenmengen miteinander geschnitten) ein depth-first-Spannbaum von  $H$  ist.

**Aufgabe 7.11:** Gegeben sei folgender gerichteter Graph  $G$ :



Berechnen Sie die kürzesten Wege vom Knoten  $A$  zu allen anderen Knoten in  $G$  mit Hilfe des Algorithmus von Dijkstra. Geben Sie dabei einige entstehende Zwischengraphen an. Markieren Sie die Knoten und Kanten entsprechend den Angaben im Text farblich. Geben Sie am Schluss  $dist(w)$  für alle Knoten  $w$  an.

**Aufgabe 7.12:** Das Zentrum eines kantenmarkierten gerichteten Graphen soll gefunden werden. Sei  $v$  ein Knoten eines Graphen  $G = (V, E)$ . Die *Exzentrizität* von  $v$  ist definiert als

$$ecc(v) = \max \{ dist(w, v) \mid w \in V \}$$

falls  $dist(w, v)$  der kürzeste Abstand zwischen  $w$  und  $v$  ist. Das *Zentrum* von  $G$  ist der Knoten mit minimaler Exzentrizität, d. h. von allen Knoten ist er derjenige, von dem aus die Distanz zum von ihm aus am weitesten entfernten Knoten minimal ist.

Beschreiben Sie einen Algorithmus, der das Zentrum eines Graphen findet, und geben sie seine Laufzeit an.

**Aufgabe 7.13:** Beweisen Sie die Aussage aus Abschnitt 7.4: Ein freier Baum mit  $n$  Knoten hat genau  $n - 1$  Kanten.

**Aufgabe 7.14:** Ein Graph heißt *planar*, wenn er kreuzungsfrei in die Ebene eingebettet werden kann, das heißt, wenn man jedem Knoten  $v$  einen Punkt  $P(v)$  der Ebene zuordnen kann und jeder Kante  $(v, w)$  ein Liniensegment, das  $P(v)$  mit  $P(w)$  verbindet, so dass sich keine Liniensegmente kreuzen.

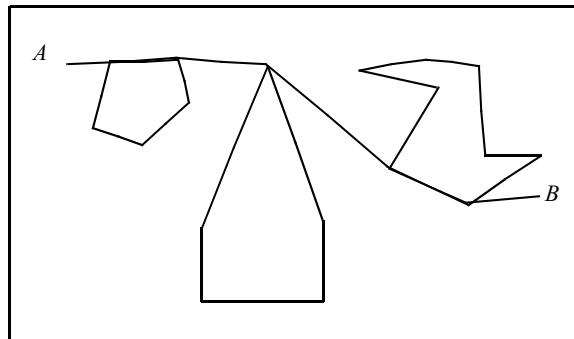
Eine *Dreieckspartition* sei ein spezieller planarer Graph, der wie folgt konstruiert wird: Ausgangspunkt ist ein Dreieck in der Ebene, dessen Ecken als Knoten und dessen Seiten als Kanten aufgefasst werden. Dieser Graph ist eine initiale Dreieckspartition (mit *einem* Dreieck). Eine gegebene Dreieckspartition kann man erweitern, indem man einen neuen

Knoten innerhalb eines ihrer Dreiecke plaziert und ihn über Kanten mit den drei Eckpunkten (Knoten) des umschließenden Dreiecks verbindet. Dies ist wiederum eine Dreieckspartition.

Beweisen Sie:

- (a) Eine Dreieckspartition hat genau  $3n - 6$  Kanten.
- (b) Ein beliebiger planarer Graph hat höchstens  $3n - 6$  Kanten.

**Aufgabe 7.15:** Gegeben seien ein Startpunkt  $A$  und ein Zielpunkt  $B$  in der Ebene. Ein Roboter soll sich von  $A$  nach  $B$  bewegen, muss dabei aber feste, polygonförmige Hindernisse in dieser Ebene umfahren. Keine zwei Polygone berühren einander oder schneiden sich. Der Roboter kann als punktförmig angesehen werden, d. h. er kann zwischen allen Polygonen durchfahren. Folgende Skizze soll ein Beispiel für ein solches Szenario sein:



Entwickeln Sie einen Algorithmus, der den kürzesten Weg bestimmt. Versuchen Sie, im Text vorgestellte Algorithmen hier einzubauen. Sie können auch davon ausgehen, dass z. B. ein Algorithmus vorhanden ist, der testet, ob Polygone oder Geraden sich schneiden.

Der Algorithmus muss nicht bis ins letzte Detail angegeben werden; es reicht, wenn intuitiv klar ist, wie die einzelnen Schritte im Detail auszuführen wären.

## 7.9 Literaturhinweise

Graphentheorie ist ein seit langem intensiv studiertes Teilgebiet der Mathematik. Das wohl erste Resultat stammt von Euler [1736]. Er löste das berühmte *Königsberger Brückenproblem*, das in der Frage besteht, ob es möglich ist, sämtliche durch Brücken verbundene Stadtteile und Inseln Königsbergs auf einem Rundweg so zu besuchen, dass

jede Brücke nur einmal passiert wird – offensichtlich ein Graphenproblem. Inzwischen gibt es eine Fülle von Lehrbüchern zur Graphentheorie und zu Graph-Algorithmen, von denen hier nur einige genannt werden können [Harary 1994], [Papadimitriou und Steiglitz 1998], [Mehlhorn 1984b], [Gibbons 1985], [West 2001], [Wilson 2010], [Jungnickel 2012] (deutsche Ausgabe [Jungnickel 1994]). Eine kurze Übersicht zu Graph-Algorithmen bietet [Khuller und Raghavachari 1996].

Der Algorithmus zur Berechnung aller kürzesten Wege von einem Knoten aus stammt von Dijkstra [1959] (in der Adjazenzmatrix-Implementierung); die Benutzung eines Heaps wurde von Johnson [1977] vorgeschlagen. Das beste bekannte Resultat für dieses Problem mit einer Laufzeit von  $O(e + n \log n)$  stammt von Fredman und Tarjan [1987]. Eine Variante des Algorithmus von Dijkstra ist der im Bereich der Künstlichen Intelligenz bekannte A\*-Algorithmus ([Hart *et al.* 1968], siehe auch [Nilsson 1982]). Anstelle des Knotens mit minimalem Abstand vom Startknoten wird dort in jedem Schritt der Knoten mit minimalem geschätztem Abstand vom Zielknoten hinzugenommen (“grün gefärbt”). Zur Schätzung wird eine “Heuristikfunktion” benutzt.

Der Algorithmus zur Berechnung kürzester Wege zwischen allen Knoten im Graphen stammt von Floyd [1962]. Eine bessere *durchschnittliche* Laufzeit von  $O(n^2 \log n)$  wurde von Moffat und Takaoka [1987] erreicht unter Benutzung eines früheren Resultats von Spira [1973].

Kontraktionshierarchien wurden von Geisberger in seiner Diplomarbeit (!) erfunden [Geisberger 2008]. Dort finden sich auch zahlreiche Untersuchungen von Heuristiken für die Knotenauswahl mit dem Ziel, die Anzahl neu erzeugter Kontraktionskanten wie auch die Größe der erzeugten Labelmengen zu minimieren. Unsere Darstellung stützt sich vor allem auf [Geisberger *et al.* 2012]. Es gibt auch eine frei verfügbare Implementierung [Geisberger *et al.* 2008].

Hub Labeling wird in [Abraham *et al.* 2011] detailliert beschrieben; es werden zahlreiche Optimierungstechniken entwickelt, insbesondere um Labelgrößen zu minimieren und Label kompakt darzustellen. Für das Straßennetz von Europa gelingt es, eine durchschnittliche Labelgröße von nur 85 Einträgen zu erreichen. In [Abraham *et al.* 2012] werden die Ideen des Hub Labeling in eine Datenbankumgebung übertragen. Damit lassen sich Kürzeste-Wege-Anfragen und verschiedene andere Anfragetypen, die auf Distanzen in Netzen beruhen, elegant in SQL ausdrücken.

Es gibt zahlreiche weitere Arbeiten, in denen Netze vorverarbeitet werden, um Kürzeste-Wege-Suche zu beschleunigen, von denen wir nur noch *Transit Node Routing* [Bast *et al.* 2007] und [Samet *et al.* 2008] erwähnen. Ein aktueller Überblicksartikel zur Kürzeste-Wege-Suche ist [Madkour *et al.* 2017].

Der in Aufgabe 7.4 behandelte Algorithmus zur Berechnung der transitiven Hülle ist von Warshall [1962]. Der Algorithmus zur Berechnung starker Komponenten orientiert sich an [Aho *et al.* 1983] und [Sharir 1981].

Der Algorithmus zur Berechnung eines minimalen Spannbaumes stammt von Kruskal [1956], ein weiterer früher Algorithmus ist von Prim [1957]. Andere Algorithmen wurden von Yao [1975], Gabow *et al.* [1986] und Fredman und Tarjan [1987] vorgeschlagen.



## 8 Geometrische Algorithmen

In diesem Kapitel betrachten wir Algorithmen und Datenstrukturen zur Lösung geometrischer Probleme. Solche Probleme werden etwa seit Mitte der siebziger Jahre des vorigen Jahrhunderts systematisch innerhalb des Forschungsgebietes der *algorithmischen Geometrie (computational geometry)* studiert, das so definiert ist:

“Algorithmische Geometrie ist die Studie der algorithmischen Komplexität elementarer geometrischer Probleme.”

Man betrachtet also grundlegende Probleme und versucht, obere und untere Schranken für Laufzeit und Platzbedarf zu bestimmen. Bekanntlich erfolgt die Ermittlung oberer Schranken durch Angabe eines geeigneten Algorithmus.

Warum soll man sich für die Lösung geometrischer Probleme interessieren? Der Grund liegt darin, dass dies häufig die abstrakt formulierten Versionen praktisch interessanter Probleme sind. Bei Graph-Algorithmen z. B. verhält es sich ähnlich: Das Finden kürzester Wege in einem Graphen kann etwa der Planung von Reiserouten oder Flugverbindungen entsprechen oder die Berechnung eines minimalen Spannbaumes der Ermittlung eines Stromleitungs- oder Kommunikationsnetzes mit minimalen Kosten. Einige geometrische Probleme sind die folgenden:

1. Gegeben sei eine Menge von Punkten in der Ebene. Bestimme alle Punkte, die innerhalb eines spezifizierten Rechtecks liegen.
2. Gegeben sei eine Menge von Rechtecken in der Ebene. Bestimme alle Paare sich schneidender Rechtecke.
3. Gegeben sei eine Menge von Polyedern im Raum. Entscheide, welche ihrer Kanten oder Kantenstücke aus einer bestimmten Richtung sichtbar bzw. verdeckt sind.
4. Gegeben sei eine Menge von Punkten  $S$  im  $k$ -dimensionalen Raum sowie ein Query-Punkt  $p$  (Query = Anfrage). Bestimme den Punkt in  $S$ , der  $p$  am nächsten liegt.

Die in dieser Art abstrakt formulierten und gelösten Probleme haben Anwendungen in ganz verschiedenen Bereichen der Informatik:

1. Datenbanken: Ein Datensatz mit  $k$  Attributen kann als Punkt im  $k$ -dimensionalen Raum aufgefasst werden. Problem (1) könnte z. B. dem Auffinden aller Datensätze von Personen entsprechen, deren Alter zwischen 20 und 30 Jahren und deren Gehalt zwischen 2000 und 3000 € liegt.

2. VLSI-Entwurf: Beim Entwurf hochintegrierter Schaltungen werden die verschiedenen Schichten eines Chips jeweils durch Rechteckmengen spezifiziert. Problem (2) tritt auf, wenn die Korrektheit eines solchen Entwurfs überprüft werden soll.
3. Computer-Graphik: Das beschriebene Problem (3) ist der entscheidende Schritt bei der Berechnung einer “realistischen” Ansicht einer dreidimensionalen Szene, die auf einem Graphikbildschirm dargestellt werden soll.
4. Erkennung gesprochener Sprache: Ein von einem Menschen gesprochenes Wort kann mit Hilfe entsprechender Elektronik analysiert und in eine Reihe von “features” (z. B. die Intensität pro Frequenzband etc.) zerlegt werden. Das Vokabular des Erkennungsprogrammes kann dann als Punktmenge im  $k$ -dimensionalen Raum dargestellt werden. Die Suche nach dem Punkt, der einem Query-Punkt am nächsten liegt – Problem (4) – entspricht der Suche nach dem vermutlich ausgesprochenen Wort des Vokabulars.

Ohne es so zu sehen, haben wir uns bereits mit mindestens *einem* geometrischen Problem intensiv beschäftigt:

Gegeben eine Punktmenge  $S$  im eindimensionalen Raum und ein Query-Punkt  $p$ , stelle fest, ob  $p$  in  $S$  enthalten ist.

Das war gerade die *member*-Operation des Dictionary-Datentyps (Abschnitt 4.2). Dieses Problem kann folgendermaßen geometrisch dargestellt werden:

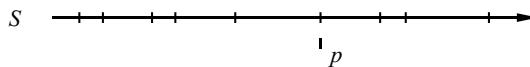


Abbildung 8.1: Punkt in Punktmenge enthalten?

Die geometrische Sicht führt zu verschiedenen naheliegenden Verallgemeinerungen. Elementare Objekte im eindimensionalen Raum sind neben Punkten auch Intervalle.

- (a) Finde alle Punkte von  $S$  innerhalb eines Query-Intervalls  $i_q = [x_l, x_r]$ .

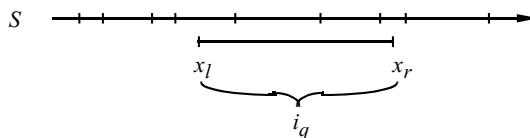


Abbildung 8.2: Im Intervall enthaltene Punkte

- (b) Sei  $S$  eine Menge von Intervallen. Finde alle Intervalle in  $S$ , die einen Query-Punkt  $p$  enthalten.

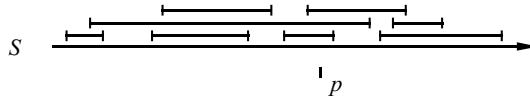


Abbildung 8.3: Intervalle, die einen Punkt enthalten

Analoge Probleme gibt es dann natürlich auch in höherdimensionalen Räumen, z. B.

- (c) Sei  $S$  eine Menge von Rechtecken in der Ebene. Finde alle Rechtecke in  $S$ , die
- einen Punkt  $p$  enthalten,
  - ein Query-Rechteck  $r$  schneiden.

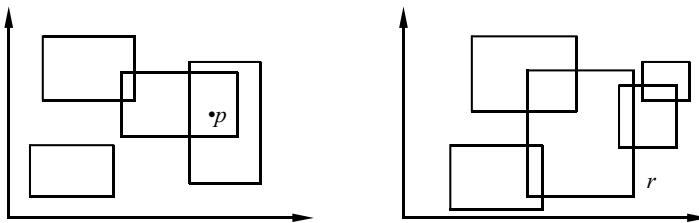


Abbildung 8.4: Suchen auf Rechteckmengen

Wir sehen, dass das *member*-Problem nur ein winziger (obgleich sehr wichtiger) Spezialfall aus einer viel größeren Klasse von Problemen ist. Die gerade genannten Probleme lassen sich als *Suchprobleme* auf *orthogonalen Objekten* klassifizieren. Dabei liegen zwei zentrale Unterscheidungen zugrunde. Die erste betrifft *Mengenprobleme* und *Suchprobleme*.

Ein *Mengenproblem* hat die Form: Berechne irgendeine interessierende Eigenschaft einer gegebenen Objektmenge  $S$ . Beispiele dafür sind etwa:

- Berechne alle Paare sich schneidender Rechtecke in einer Rechteckmenge  $S$ .
- Berechne die Kontur der insgesamt von  $S$  bedeckten Fläche.

Ein *Suchproblem* hat die Form: Gegeben eine Objektmenge  $S$  und ein weiteres (Query-)Objekt  $q$ ; ermittle alle Objekte in  $S$ , die zu  $q$  in irgendeiner interessierenden Beziehung stehen. Eine Lösung für ein Suchproblem besteht darin, die Menge  $S$  in einer Datenstruktur so zu organisieren, dass effizient gesucht werden kann.

Die zweite wichtige Unterscheidung betrifft *orthogonale* und *beliebig orientierte* Objekte:

**Definition 8.1:** Ein *k-dimensionales geometrisches Objekt* ist eine zusammenhängende<sup>1</sup> Teilmenge des  $\mathbb{R}^k$ . Ein solches Objekt heißt *orthogonal*, wenn es als kartesisches Produkt von  $k$  Intervallen beschrieben werden kann. Dabei darf jedes Intervall zu einem Punkt entarten.

Im eindimensionalen Raum gibt es genau zwei Arten von orthogonalen Objekten, nämlich Punkte und Intervalle; in der Ebene sind es Punkte, horizontale und vertikale Liniensegmente sowie Rechtecke (Abbildung 8.5).

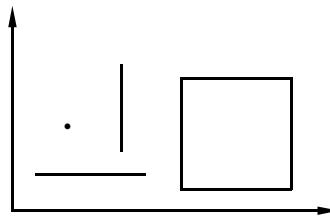


Abbildung 8.5: Orthogonale Objekte in der Ebene

Von *beliebig orientierten Objekten* sprechen wir, wenn die begrenzenden Kanten oder Flächen beliebige (insbesondere auch beliebig viele verschiedene) Richtungen<sup>2</sup> haben dürfen. Zum Beispiel die folgenden Probleme betreffen beliebig orientierte Objekte:

- Finde alle Polyeder (im  $\mathbb{R}^3$ ), die einen Query-Punkt enthalten.
- Berechne die Aufteilung eines Polygons in Dreiecke.

Punktmengen treten in beiden Bereichen auf. Natürlich sind Punkte immer auch orthogonale Objekte; wenn aber bei der Formulierung oder Lösung eines Problems auf Punktmengen beliebige Richtungen eine Rolle spielen, ordnen wir das Problem eher in die zweite Kategorie ein. Das ist etwa bei den folgenden Problemen für Punktmengen der Fall:

- Berechne die konvexe Hülle einer Punktmenge (das kleinste konvexe Polygon, das alle Punkte enthält).

---

1. Ein Objekt heißt *zusammenhängend*, falls sich zwei beliebig gewählte Punkte daraus stets durch eine Kurve innerhalb des Objekts verbinden lassen.  
 2. In diesem Text wird “Orientierung” gleichbedeutend mit “Richtung” gebraucht.

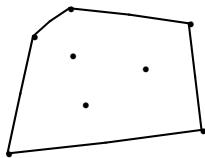


Abbildung 8.6: Konvexe Hülle

- Finde den Punkt mit minimalem Abstand zu einem Query-Punkt (*nearest-neighbour*-Problem).

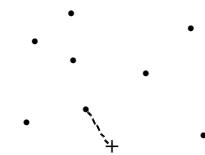


Abbildung 8.7: Minimaler Abstand

Im Rahmen dieses einführenden Buches zu Datenstrukturen und Algorithmen kann naturgemäß nur eine sehr knappe Einführung in das Gebiet der algorithmischen Geometrie gegeben werden. Anstatt viele Teilgebiete davon kurz und oberflächlich zu behandeln, konzentrieren wir uns auf einen bestimmten Bereich, nämlich die Behandlung von Mengen orthogonaler Objekte (meist in der Ebene) in Mengen- und Suchproblemen. Daran lassen sich gut einige grundlegende Techniken und Datenstrukturen der algorithmischen Geometrie demonstrieren.

Die beiden wichtigsten Techniken zur Behandlung von Mengenproblemen sind *Plane-Sweep* und *Divide-and-Conquer*. Wir betrachten beide Techniken, zum Teil für gleiche Probleme, in den Abschnitten 8.1 und 8.2. Plane-Sweep reduziert ein  $k$ -dimensionales Mengenproblem auf ein  $(k-1)$ -dimensionales Suchproblem. Deshalb werden einige elementare Datenstrukturen für Suchprobleme schon aus Plane-Sweep-Algorithmen heraus motiviert und dort auch eingeführt, während Suchprobleme allgemein in Abschnitt 8.3 behandelt werden. In Abschnitt 8.4 skizzieren wir den grundlegenden Plane-Sweep-Algorithmus zur Behandlung beliebig orientierter Objekte in der Ebene.

## 8.1 Plane-Sweep-Algorithmen für orthogonale Objekte in der Ebene

In diesem Abschnitt führen wir die Plane-Sweep-Technik am Beispiel des wohl einfachsten damit lösbarer Problems ein, nämlich des orthogonalen Segmentschnitt-Problems. Anschließend wird das Rechteckschnitt-Problem (Problem 2 der Einleitung) mit der gleichen Technik behandelt. Es zeigt sich, dass die Lösung des Segmentschnitt-Problems bereits eine Teillösung dieses Problems darstellt; ergänzend wird noch eine Lösung des sogenannten Punkteinschluss-Problems benötigt. Zur Unterstützung des entsprechenden Plane-Sweeps braucht man eine Datenstruktur zur dynamischen Darstellung einer Intervallmenge, den Segment-Baum. Schließlich zeigen wir, wie mit Plane-Sweep das Maßproblem gelöst werden kann, das darin besteht, die Größe der von einer Menge sich schneidender Rechtecke insgesamt bedeckten Fläche zu ermitteln.

### 8.1.1 Das Segmentschnitt-Problem

Als erstes Mengenproblem betrachten wir das sehr elementare *Segmentschnitt-Problem*:

Gegeben eine Menge horizontaler und vertikaler Liniensegmente (oder kurz: Segmente) in der Ebene, finde alle Paare sich schneidender Segmente.

Wir nehmen zunächst vereinfachend an, dass sich alle Segmente in allgemeiner Lage befinden. Das soll heißen, dass keine zwei Endpunkte verschiedener Segmente auf der gleichen  $x$ - oder  $y$ -Koordinate liegen. Dadurch werden auch Schnitte zwischen zwei horizontalen bzw. zwei vertikalen Segmenten ausgeschlossen, so dass die Aufgabe darauf beschränkt ist, Schnitte zwischen horizontalen und vertikalen Segmenten zu ermitteln. Der allgemeinere Fall, der gleiche  $x$ - oder  $y$ -Koordinaten zulässt, wird in Aufgabe 8.1 besprochen.

Ein trivialer Algorithmus löst das Problem in  $O(n^2)$  Zeit durch paarweisen Vergleich aller Segmente. Wenn man nur  $n$  als Parameter für die Komplexität des Problems heranzieht, ist das optimal, da  $\Theta(n^2)$  Schnitte existieren können. Schon in der Einleitung (Kapitel 1) wurde argumentiert, dass man auch  $k$ , die Anzahl vorhandener Schnitte, als Parameter heranziehen sollte. Wir beschreiben im Folgenden einen Algorithmus mit einer Laufzeit von  $O(n \log n + k)$ .

Plane-Sweep ist eine sehr populäre Methode der algorithmischen Geometrie, um Mengenprobleme zu lösen. Die Idee besteht darin, z. B. eine vertikale Gerade, genannt *Sweepline*, von links nach rechts durch die Ebene zu schieben und dabei den Schnitt der Geraden mit der Objektmenge zu „beobachten“. Natürlich kann man ebenso gut eine horizontale Gerade von oben nach unten oder von unten nach oben schieben. Manche Algorithmen benötigen mehrere Sweeps in unterschiedliche Richtungen, z. B. zuerst von

links nach rechts, dann von rechts nach links, wobei im zweiten Sweep die beim ersten Mal gewonnene Information ausgenutzt wird. Die Idee lässt sich übertragen in den dreidimensionalen Raum, wo dann eine (Hyper-) Ebene durch den Raum geschoben wird. Im Falle des Segmentschnitt-Problems können wir die folgenden Beobachtungen machen:

- Jedes horizontale Segment schneidet die kontinuierlich bewegte Sweepline während eines bestimmten Zeitintervalls in einer festen y-Koordinate. Zu jedem Zeitpunkt existiert also eine Menge aktuell die Sweepline schneidender Segmente (ggf. leer), die charakterisiert ist durch eine Menge von y-Koordinaten.
- Jedes vertikale Segment schneidet die Sweepline zu einem bestimmten Zeitpunkt in einem y-Intervall. Die horizontalen Segmente, die dieses vertikale Segment schneiden, sind gegeben durch zwei Eigenschaften:
  1. Sie gehören zur Menge aktuell die Sweepline schneidender Segmente.
  2. Ihre y-Koordinate liegt im y-Intervall des vertikalen Segments.

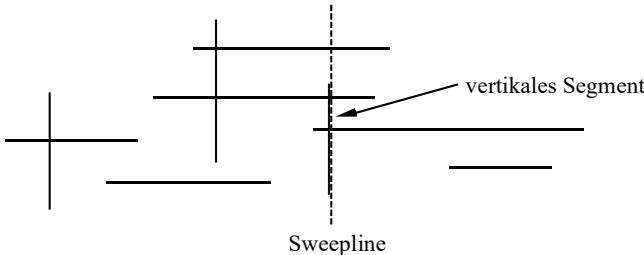


Abbildung 8.8: Segmentschnitt-Problem

Damit ist die Idee des Plane-Sweep-Algorithmus offensichtlich:

- Erhalte während des Sweeps die Menge der aktuell die Sweepline schneidenden Segmente anhand ihrer Menge von y-Koordinaten aufrecht.
- Wann immer ein vertikales Segment getroffen wird, ermittle alle vorhandenen y-Koordinaten, die in seinem y-Intervall liegen.

Die kontinuierliche Bewegung der Sweepline wird simuliert, indem man alle x-Koordinaten bestimmt, bei denen auf der Sweepline „etwas passiert“. In unserem Fall sind das die linken und rechten Enden horizontaler Segmente sowie die x-Koordinaten vertikaler Segmente. Diese x-Koordinaten werden dann schrittweise „abgefahren“. Zu einem Plane-Sweep-Algorithmus gehören daher immer zwei Datenstrukturen, nämlich

1. eine Struktur zur Darstellung der aktuell die Sweepline schneidenden Objekte, die *Sweepline-Status-Struktur*, und

2. eine Struktur zur Darstellung der noch zu bearbeitenden Haltepunkte der Sweep-line, manchmal die *Sweep-Event-Struktur* genannt. Dies ist eine in Sweep-Richtung geordnete Folge von Objekten.

Die Sweep-line-Status-Struktur muss offensichtlich dynamisch sein, das heißt, das Einfügen und Löschen von Objekten unterstützen. Die Event-Struktur ist in manchen Algorithmen statisch; das bedeutet, dass alle Haltepunkte einmal zu Anfang ermittelt und zur Initialisierung der Event-Struktur benutzt werden. Dies ist gewöhnlich bei der Behandlung orthogonaler Objekte der Fall, so auch für das Segmentschnitt-Problem. In anderen Algorithmen werden dynamisch während des Sweeps noch neue Haltepunkte erkannt und in die Event-Struktur eingefügt; ein Beispiel dafür findet sich in Abschnitt 8.4.

Um die Beschreibung der folgenden Algorithmen zu vereinfachen, führen wir einige Notationen ein. Wir wollen es uns ersparen, ständig programmiersprachlich Record-Typen einzuführen und arbeiten deshalb etwas abstrakter mit Tupeln. Ein horizontales Segment sei dargestellt als ein Tripel  $h = (x_1, x_2, y)$ , ein vertikales Segment als  $v = (x, y_1, y_2)$ , ein Rechteck als  $r = (x_1, x_2, y_1, y_2)$ . Bei einem Tupel  $t = (x_1, \dots, x_n)$  notieren wir mit  $t_i$  den Zugriff auf die  $i$ -te Komponente, also  $t_i = x_i$ . Für ein horizontales Segment  $h$  beschreibt daher  $h_3$  seine y-Koordinate. Für eine Tupelmenge  $S$  bezeichne  $\pi_i(S)$  die Projektion auf die  $i$ -te Komponente. Für eine Menge horizontaler Segmente  $H$  ist  $\pi_3(H)$  daher die Menge aller y-Koordinaten von Segmenten in  $H$ .

Wir benötigen im Folgenden immer wieder Strukturen, die eine Menge von Objekten anhand eines (meist geometrischen) Schlüssels organisieren, z. B. eine Menge vertikaler Segmente anhand ihrer x-Koordinate, eine Menge von Rechtecken anhand ihrer y-Intervalle usw. Wir beschreiben dann die Objektmenge als Menge von Paaren (Schlüssel, Objekt). Für eine solche Menge  $S$  erhalten wir z. B. mit  $\pi_1(S)$  die Menge aller vorkommenden Schlüsselwerte. Wir sprechen von derartigen Objektmengen in einem Kontext, in dem es darum geht, sie anhand des jeweiligen Schlüssels zu organisieren, trotzdem einfach als Menge von y-Koordinaten, Menge von y-Intervallen usw.

Der Algorithmus lässt sich dann so beschreiben:

**algorithm** *SegmentIntersectionPS* ( $H, V$ )

{Eingabe ist eine Menge horizontaler Segmente  $H$  und eine Menge vertikaler Segmente  $V$ , berechne mit Plane-Sweep die Menge aller Paare  $(h, v)$  mit  $h \in H$  und  $v \in V$  und  $h$  schneidet  $v$ }

*Schritt 1:* Sei  $S = \{ (x_1, (x_1, x_2, y)) \mid (x_1, x_2, y) \in H \}$   
 $\quad \cup \{ (x_2, (x_1, x_2, y)) \mid (x_1, x_2, y) \in H \}$   
 $\quad \cup \{ (x, (x, y_1, y_2)) \mid (x, y_1, y_2) \in V \};$

( $S$  ist also eine gemischte Menge von horizontalen und vertikalen Segmenten, in der jedes horizontale Segment einmal anhand des linken und einmal anhand des rechten Endpunkts dargestellt ist. Diese Menge beschreibt die Sweep-Event-Struktur.)  
Sortiere  $S$  nach der ersten Komponente, also nach x-Koordinaten;

*Schritt 2:* Sei  $Y$  eine Menge horizontaler Segmente, deren y-Koordinate als Schlüssel verwendet wird (Sweepline-Status-Struktur);  
 $Y := \emptyset$ ;  
durchlaufe  $S$ : das gerade erreichte Objekt ist

- (a) linker Endpunkt eines horizontalen Segments  $h = (x_1, x_2, y)$ :  
 $Y := Y \cup \{ (y, (x_1, x_2, y)) \}$  (füge  $h$  in  $Y$  ein)
- (b) rechter Endpunkt von  $h = (x_1, x_2, y)$ :  
 $Y := Y \setminus \{ (y, (x_1, x_2, y)) \}$  (entferne  $h$  aus  $Y$ )
- (c) ein vertikales Segment  $v = (x, y_1, y_2)$ :  
 $A := \pi_2(\{ w \in Y \mid w_1 \in [y_1, y_2] \})$ ;  
(finde alle Segmente in  $Y$ , deren y-Koordinate im y-Intervall von  $v$  liegt)  
gib alle Paare in  $A \times \{v\}$  aus

**end** *SegmentIntersectionPS.*

Damit ist das Segmentschnitt-Problem auf das folgende eindimensionale Teilproblem reduziert: Gib eine Datenstruktur an, die eine Menge von Koordinaten (bei diesem Problem y-Koordinaten) speichert und die folgenden Operationen erlaubt:

- Einfügen einer Koordinate,
- Entfernen einer Koordinate,
- Auffinden aller gespeicherten Koordinaten, die in einem Query-Intervall liegen.

Mit ‘‘Koordinate’’ bezeichnen wir einen ‘‘eindimensionalen Punkt’’. Eine Koordinate ist nichts anderes als ein Wert eines geordneten Wertebereichs.

Dieses Problem lässt sich noch mit einer Standard-Datenstruktur lösen: Wir benutzen z. B. eine Variante des AVL-Baumes, in der alle Koordinaten in den Blättern stehen, innere Knoten also nur ‘‘Pfadinformation’’ enthalten, und die Blätter zusätzlich zu einer Liste verkettet sind. Eine solche Struktur ist in Abbildung 8.9 gezeigt, sie wurde auch schon in Aufgabe 4.12 angesprochen. Man kann sich überlegen, dass Updates mit den gleichen Kosten wie im Standard-AVL-Baum realisiert werden können. Das Auffinden der Koordinaten im Intervall  $[y_1, y_2]$  erfolgt durch Suche nach dem am weitesten links liegenden Blatt, das rechts von  $y_1$  liegt. Anschließend läuft man an der Liste der verketteten Blätter entlang, bis das erste Blatt erreicht wird, das rechts von  $y_2$  liegt (Abbildung 8.9).

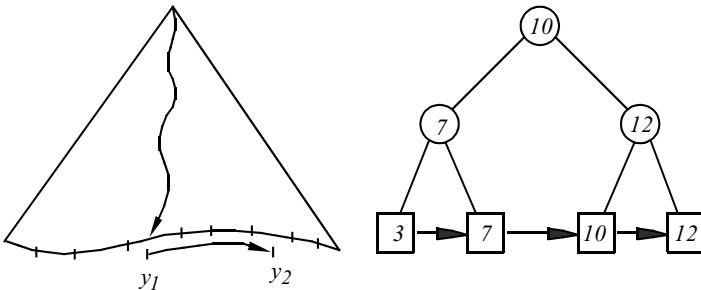


Abbildung 8.9: Variante des AVL-Baumes

Damit lässt sich also das Einfügen und Entfernen in  $O(\log n)$  Zeit durchführen. Die  $t$  Koordinaten in einem Intervall können in  $O(\log n + t)$  Zeit aufgefunden werden.

Für die Gesamtlaufzeit des Algorithmus ergibt sich folgendes: Sei  $n$  die Gesamtzahl von Segmenten, also  $n = |H| + |\mathcal{V}|$ . Schritt 1 benötigt  $O(n \log n)$  Zeit, im Wesentlichen für das Sortieren. In Schritt 2 werden  $O(n)$  Einfüge-, Entferne- und Query-Operationen durchgeführt, jede kostet  $O(\log n)$  Zeit, wenn man von den Termen  $t_i$  absieht ( $t_i$  ist die Anzahl der eingeschlossenen Koordinaten bei der  $i$ -ten Query). Falls  $k$  die Gesamtzahl sich schneidender Paare ist, so gilt  $k = \sum t_i$  und die Gesamtlaufzeit des Algorithmus ist  $O(n \log n + k)$ . Der Speicherplatzbedarf ist offensichtlich  $O(n)$ .

Man kann zeigen, dass für das Segmentschnittproblem  $\Omega(n \log n + k)$  bzw.  $\Omega(n)$  gleichzeitig untere Schranken für Zeit- und Platzbedarf darstellen; damit ist dieser Algorithmus optimal. Wir fassen zusammen:

Das Segmentschnitt-Problem kann für  $n$  Segmente mit  $k$  Schnittpaaren mittels Plane-Sweep in  $O(n \log n + k)$  Zeit und  $O(n)$  Platz gelöst werden.

**Selbsttestaufgabe 8.1:** Geben Sie einen Algorithmus an, der in einer Menge beliebiger horizontaler und vertikaler Liniensegmente effizient *alle* Paare sich schneidender Segmente berechnet. Das heißt, die im Text getroffene Einschränkung auf paarweise verschiedene Koordinaten wird hier aufgehoben, so dass z. B. auch Schnitte zwischen zwei horizontalen Segmenten oder solche, bei denen der Schnittpunkt gleich einem der Endpunkte ist, vorkommen können und ermittelt werden sollen. Sei weiter angenommen, dass sich in keinem Punkt der Ebene mehr als  $c$  Segmente schneiden, für eine kleine Konstante  $c$ .  $\square$

### 8.1.2 Das Rechteckschnitt-Problem

Wir betrachten ein weiteres Problem, das *Rechteckschnitt-Problem*:

Gegeben sei eine Menge achsenparalleler Rechtecke in der Ebene. Finde alle Paare von Rechtecken, die sich schneiden.

Während das Segmentschnitt-Problem relativ abstrakt erscheint, hat das Rechteckschnitt-Problem wichtige Anwendungen im VLSI-Entwurf. Interessanterweise lässt sich die Lösung des Rechteckschnitt-Problems teilweise auf die Lösung des Segmentschnitt-Problems zurückführen. Wir sind nämlich bereits in der Lage, alle Paare sich schneidender Rechtecke, *deren Kanten sich schneiden*, mithilfe des Segmentschnitt-Algorithmus aufzufinden.

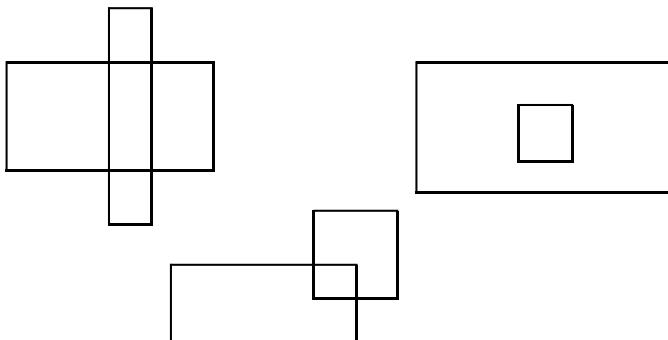


Abbildung 8.10: Rechteckschnitt-Problem

Das verbleibende Teilproblem besteht darin, solche Paare von Rechtecken aufzufinden, die einander ganz enthalten (zwei geometrische Objekte schneiden sich, wenn sie gemeinsame Punkte haben). Dazu wird in einem ersten Schritt der Rechteckmenge eine Menge von Punkten hinzugefügt, indem für jedes Rechteck ein beliebiger innerer Punkt als Repräsentant ausgewählt wird. Wir haben jetzt also eine gemischte Menge vor uns, die  $n$  Rechtecke und  $n$  innere Punkte enthält. Seien  $a, b$  zwei Rechtecke und  $p_a, p_b$  die zugehörigen inneren Punkte. Falls nun z. B.  $a$  ganz in  $b$  enthalten ist, so ist mit Sicherheit auch  $p_a$  in  $b$  enthalten. Diese Beobachtung erlaubt es, das betrachtete Problem (genannt *Rechteckeinschluss-Problem*) auf ein *Punkteinschluss-Problem* zu reduzieren. Falls  $b$   $p_a$  enthält, so ist es zwar auch möglich, dass  $a$  und  $b$  Kantenschnitte miteinander haben. Dies stört aber nicht, da in jedem Fall das Paar  $(a, b)$  als Paar sich schneidender Rechtecke auszugeben ist.

### Das Punkteinschluss-Problem und seine Plane-Sweep-Reduktion

Wir haben das Problem also zunächst reduziert auf das *Punkteinschluss-Problem*:

Gegeben eine gemischte Menge von Punkten und Rechtecken, finde alle Punktein schlüsse, das heißt alle Paare (Punkt, Rechteck), bei denen der Punkt im Rechteck liegt.

Dieses Problem soll wieder mit Plane-Sweep gelöst werden. Die vorbereitenden Schritte sind ähnlich wie beim Segmentschnitt-Algorithmus: Die linken und rechten Kanten aller Rechtecke werden ermittelt und zusammen mit den Punkten nach x-Koordinaten sortiert. Dies sind die “Stationen” des Sweeps, die den Inhalt der Sweep-Event-Struktur bilden.

Für die Beschreibung des Algorithmus nehmen wir an, dass eine linke Rechteckkante dargestellt sei als ein Tupel  $(x, \text{left}, (y_1, y_2), r)$ , eine rechte Rechteckkante als  $(x, \text{right}, (y_1, y_2), r)$  und ein Punkt als  $(x, \text{point}, y, r)$ . Darin sind  $\text{left}$ ,  $\text{right}$  und  $\text{point}$  Diskriminatorenwerte, die es erlauben, den Typ eines solchen Objekts zu bestimmen. Die letzte Komponente  $r$  ist der “Name” des Rechtecks, von dem die Kante oder der repräsentierende Punkt abstammen<sup>3</sup>. – Auf einer niedrigeren Abstraktionsebene kann man sich die Implementierung dieser Darstellung überlegen: Die Tupel könnten z. B. durch variante Records dargestellt werden, wobei  $\text{left}$ ,  $\text{right}$  und  $\text{point}$  als Werte für ein Diskriminatorenfeld die Unterscheidung der Varianten erlauben. Rechtecknamen könnten jeweils Zeiger auf Records mit der Rechteckdarstellung sein oder Indizes in einen Array aller Rechteckdarstellungen. Für die effiziente Implementierung des folgenden Algorithmus ist es wichtig, dass Rechtecknamen kopiert werden können, ohne die eigentliche Rechteckdarstellung zu vervielfachen. Auf dieser Ebene erscheinen einige Tupelkomponenten redundant, da sie durch Zugriff auf Komponenten des Rechteck-Records ersetzt werden können; man würde sie also nicht explizit abspeichern.

Wir betrachten nun den eigentlichen Plane-Sweep. Zu jedem Zeitpunkt schneidet die Sweepeline die Rechteckmenge in einer Menge  $I$  von y-Intervallen. Ein Punkt, der während des Sweeps angetroffen wird, liegt in allen Rechtecken, die aktuell die Sweepeline schneiden, und deren y-Intervall seine y-Koordinate enthält.

---

3. Wenn das “reine” Punkteinschlussproblem gelöst werden soll, entfällt die  $r$ -Komponente in der Punktdarstellung.

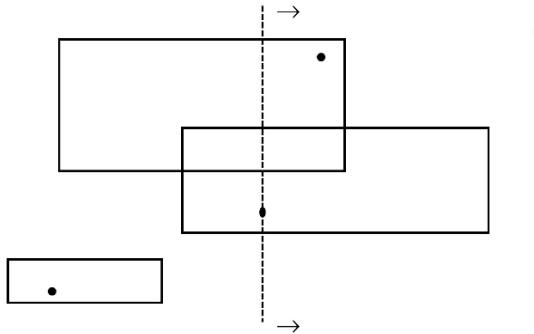


Abbildung 8.11: Punkteinschluss-Problem

Die Aktionen während des Sweeps sind also die folgenden:

- (a) falls eine linke Rechteckkante ( $x, \text{left}, (y_1, y_2), r$ ) angetroffen wird:

Füge das Intervall(-Objekt)  $([y_1, y_2], r)$  in  $I$  ein.

- (b) für eine rechte Rechteckkante ( $x, \text{right}, (y_1, y_2), r$ ):

Entferne das Intervall  $([y_1, y_2], r)$  aus  $I$ .

- (c) für einen Punkt ( $x, \text{point}, y, r$ ):

Finde alle Intervalle in  $I$ , die  $y$  enthalten. Genauer:

$$A := \pi_2(\{ ([y_1, y_2], r) \in I \mid y \in [y_1, y_2] \}); \text{ gib alle Paare in } A \times \{r\} \text{ aus.}$$

Damit ist das Punkteinschluss-Problem wiederum auf ein eindimensionales Suchproblem reduziert: Gesucht ist eine Datenstruktur, die eine Menge von Intervallen (als Schlüssel) speichert und die folgenden Operationen erlaubt:

- Einfügen eines Intervalls,
- Entfernen eines Intervalls,
- Auffinden aller Intervalle, die eine Query-Koordinate enthalten.

Der *Segment-Baum* ist eine solche Datenstruktur.

### Der Segment-Baum

Eine geordnete Menge von Koordinaten  $\{x_0, \dots, x_N\}$  teilt den eindimensionalen Raum in eine Sequenz “atomarer” Intervalle  $[x_0, x_1], [x_1, x_2], \dots, [x_{N-1}, x_N]$  auf (die unendlichen Intervalle links und rechts werden nicht betrachtet). Ein *Segment-Baum über*  $\{x_0, \dots, x_N\}$  ist ein binärer Baum minimaler Höhe, dessen Blättern die atomaren Intervalle zugeordnet sind. Ferner ist jedem inneren Knoten ein Intervall zugeordnet, das der Konkatenation der beiden Intervalle seiner Söhne entspricht. In der folgenden Darstellung ist für jeden Knoten das ihm zugeordnete Intervall gezeichnet:

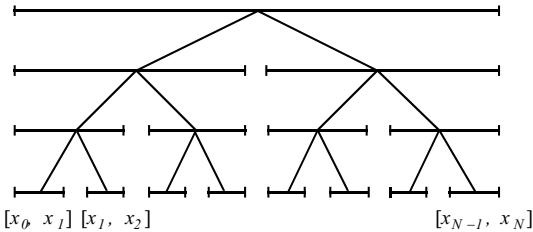


Abbildung 8.12: Segment-Baum

Diese zunächst leere Struktur dient als Rahmen, in den Intervalle eingetragen werden können, deren Endpunkte in der Menge  $\{x_0, \dots, x_N\}$  liegen (solche Intervalle nennen wir Intervalle über  $\{x_0, \dots, x_N\}$ ). Jeder Knoten des Segment-Baums hat eine zugeordnete Liste (die *Knotenliste*) von “Intervall-Namen”. Man könnte die Struktur eines Knotens also z. B. so definieren:

```
type seg_node = record
    bottom, top : integer;           {das Knotenintervall}
    entries      : interval_list;   {eingetragene Intervalle}
    left, right  : ↑seg_node
end
```

Für “Intervallnamen” gilt das Gleiche, was oben zu Rechtecknamen gesagt wurde. Bei der Anwendung des Segment-Baums für die Lösung des Punkteinschluss-Problems sind Intervallnamen tatsächlich Rechtecknamen. In den folgenden Beispielen verwenden wir Buchstaben  $a, b, c$  als Intervallnamen.

Ein Intervall  $([x_i, x_j], a)$  könnte im Prinzip in den Segment-Baum eingetragen werden, indem die Knotenliste jedes überdeckten Blattes (also der Blätter  $[x_i, x_{i+1}], [x_{i+1}, x_{i+2}], \dots, [x_{j-1}, x_j]$ ) um den Eintrag “ $a$ ” erweitert wird. Dabei würde ein einzelnes Intervall aber im Allgemeinen sehr viele Einträge erzeugen. Um dies zu vermeiden, werden Intervallnamen immer so hoch wie möglich im Baum eingetragen. Die Idee ist: Wann immer

zwei Söhne eines Knotens von einem Intervall “markiert” würden, markiere stattdessen den Vater. Dies führt zu folgender Definition:

**Definition 8.2:** Die Menge  $CN(i)$  (= *covered nodes*) der von einem Intervall  $i$  in einem Segment-Baum  $S$  kanonisch bedeckten Knoten ist definiert als

$$CN(i) := \{p \in S \mid \text{interval}(p) \subseteq i \wedge \neg \text{interval}(\text{father}(p)) \subseteq i\}.$$

Dabei bezeichnet  $S$  gleichzeitig wieder die Knotenmenge des Segment-Baums,  $\text{interval}(p)$  das Knotenintervall des Knotens  $p$  (bei der obigen Implementierung also das Intervall  $[p.\text{bottom}, p.\text{top}]$ ) und  $\text{father}(p)$  den Vaterknoten von  $p$ . Bei der Teilmengenbeziehung werden Intervalle als Punktmenzen aufgefasst.

Für ein eingetragenes Intervall  $a$  ergibt sich im Allgemeinen ein “gegabelter Pfad”, an dem Knoten liegen, in deren Intervall-Liste  $a$  eingetragen ist, wie in Abbildung 8.13 gezeigt.

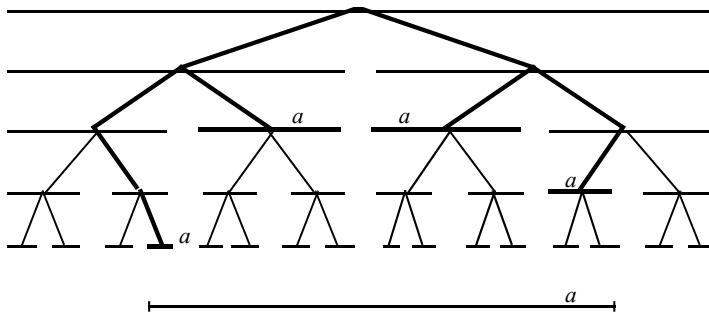


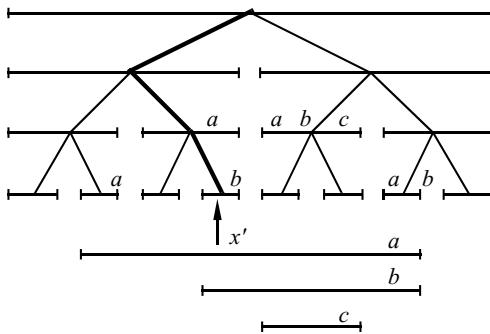
Abbildung 8.13: Intervall  $a$  im Segment-Baum

Die Gabelung muss nicht unbedingt schon in der Wurzel liegen. Abbildung 8.14 zeigt einen Segment-Baum, in dem drei Intervalle  $a$ ,  $b$  und  $c$  eingetragen sind.

Natürlich will man Intervalle nicht nur abspeichern, sondern auch Suchoperationen durchführen. Der Segment-Baum erlaubt gerade die oben geforderte Operation:

Gegeben sei eine Koordinate  $x'$ . Finde alle Intervalle, die  $x'$  enthalten.

Die Durchführung dieser Operation ist in Abbildung 8.14 skizziert: Man läuft entlang dem Pfad, der zu  $x'$  führt. Die Knotenlisten der Knoten auf dem Suchpfad enthalten genau die Intervalle, die  $x'$  enthalten; sie werden jeweils durchlaufen, um die Intervalle auszugeben.

Abbildung 8.14: Intervalle  $a$ ,  $b$  und  $c$  im Segment-Baum

Nun zur Analyse der Struktur: Der leere Baum hat  $N$  Blätter und  $N-1$  innere Knoten, benötigt also  $O(N)$  Speicherplatz und hat die Höhe  $O(\log N)$ . Man kann zeigen, dass ein beliebiges eingetragenes Intervall auf jeder Ebene des Baumes maximal zwei Einträge erzeugt (Aufgabe 8.16). Eine Menge von  $n$  Intervallen lässt sich also mit  $O(N + n \log N)$  Platz abspeichern (häufig ist  $N = \Theta(n)$  und der Platzbedarf ist dann  $O(n \log n)$ ). Das Einfügen oder Entfernen eines Intervalls ist in  $O(\log N)$  Zeit möglich. Für das Einfügen kann man sich dies klarmachen, indem man eine rekursive Prozedur formuliert, die die Knoten in  $CN(i)$  aufsucht, also gerade den ‘‘gegabelten Pfad’’ abläuft (Aufgabe 8.15). An jedem Knoten  $p$  in  $CN(i)$  wird das Intervall  $i$  in  $O(1)$  Zeit am Anfang der Liste  $p.entries$  eingetragen. Für das Entfernen ist die Lösung nicht so offensichtlich (Aufgabe 8.2). Das Auffinden der  $t$  Intervalle, die  $x'$  enthalten, erfolgt in  $O(\log N + t)$  Zeit.

**Selbsttestaufgabe 8.2:** Wie kann man dafür sorgen, dass aus einem Segment-Baum Intervalle in  $O(\log N)$  Zeit gelöscht werden können?

*Hinweis:* Es dürfen Hilfsdatenstrukturen benutzt werden, sofern dabei der Platzbedarf und die Zeit für die anderen Operationen asymptotisch nicht verschlechtert werden.  $\square$

Wir fassen die Eigenschaften des Segment-Baums zusammen: Sei  $X = \{x_0, \dots, x_N\}$  und  $N = \Theta(n)$ . Ein Segment-Baum über  $X$  erlaubt die Darstellung einer Intervallmenge  $I$  über  $X$  mit  $n$  Elementen in  $O(n \log n)$  Speicherplatz, so dass das Einfügen und Entfernen von Intervallen über  $X$  in  $O(\log n)$  Zeit möglich ist und die Intervalle, die eine Query-Koordinate enthalten, in  $O(\log n + t)$  Zeit ausgegeben werden können.

### Komplexität der Lösungen

Unter der Annahme, dass der Segment-Baum als Datenstruktur im obigen Plane-Sweep-Algorithmus für das Punkteinschluss-Problem eingesetzt wird, können wir nun diesen Algorithmus analysieren. Sei  $r$  die Anzahl vorhandener Rechtecke,  $p$  die Anzahl der Punkte und  $k$  die Anzahl der existierenden Punkteinschlüsse. Sei weiterhin  $n = r + p$ . Der Initialisierungsschritt des Algorithmus, in dem linke und rechte Rechteckkanten gebildet und zusammen mit den Punkten nach x-Koordinate sortiert werden – in dem also die Sweep-Event-Struktur aufgebaut wird – benötigt  $O(n \log n)$  Zeit. In diesem Algorithmus ist weiterhin eine Initialisierung der Sweep-line-Status-Struktur erforderlich, da man in den Segment-Baum nicht beliebige Intervalle eintragen kann. Es muss also zunächst die leere Rahmenstruktur aufgebaut werden. Dazu ist das zugrundeliegende Raster zu bestimmen, das wir oben  $\{x_0, \dots, x_N\}$  genannt hatten. In diesem Fall ergibt sich das Raster aus den vorkommenden y-Koordinaten der Rechtecke, hat also die Form  $Y = \{y_0, \dots, y_N\}$  mit  $N = 2r$ . Um das Raster zu berechnen, kann man einmal die Liste der Rechtecke durchlaufen, die vorkommenden y-Koordinaten extrahieren und diese dann sortieren, in insgesamt  $O(r \log r) = O(n \log n)$  Zeit. Man kann sich weiterhin leicht eine Prozedur überlegen (Aufgabe 8.18), die für ein Raster, also eine geordnete Folge von Koordinaten, in  $O(N)$  Zeit einen leeren Segment-Baum konstruiert. In dieser Anwendung ist  $O(N) = O(r) = O(n)$ . Die Initialisierungsschritte erfordern also insgesamt  $O(n \log n)$  Zeit.

Während des Sweeps werden je  $r$  linke und rechte Rechteckkanten und  $p$  Punkte angefahren. Jede der zugeordneten Operationen kann mithilfe des Segment-Baums in  $O(\log r) = O(\log n)$  Zeit durchgeführt werden, abgesehen von der Ausgabe der gefundenen Punkteinschlüsse. Der Zeitbedarf dafür ist linear, das heißt  $O(k)$ . Die Gesamtaufzeit dieses Plane-Sweep-Algorithmus ist damit  $O(n \log n + k)$ , der Platzbedarf ist  $O(n + r \log r)$ , da der Platzbedarf des Segment-Baums nur von der Größe der Rechteckmenge abhängt.

Der Rechteckschnitt-Algorithmus als Kombination von Segmentschnitt- und Punkteinschluss-Algorithmus sieht nun so aus (sei  $n$  die Anzahl der Rechtecke und  $k$  die Anzahl sich schneidender Paare):

1. Durchlaufe die Darstellung der Rechteckmenge und erzeuge für jedes Rechteck seine 4 Kanten. Wende darauf den Segmentschnitt-Algorithmus an (wobei Pseudo-Schnitte eines Rechtecks mit sich selbst zu ignorieren sind). Sei  $k'$  die Anzahl vorhandener Kantenschnitte. Dann benötigt dieser Schritt  $O(n \log n + k')$  Zeit und  $O(n)$  Speicherplatz.
2. Durchlaufe die Darstellung der Rechteckmenge und erzeuge für jedes Rechteck einen inneren Punkt als Repräsentanten. Wende auf die resultierende Menge von Punkten und Rechtecken den Punkteinschluss-Algorithmus an. Sei  $k''$  die Anzahl

der erzeugten Punkteinschlüsse. Dann benötigt dieser Schritt  $O(n \log n + k'')$  Zeit und  $O(n \log n)$  Speicherplatz.

Es gilt:

- (i)  $k' = O(k)$ , da zwei Rechtecke höchstens vier Kantenschnitte erzeugen.
- (ii)  $k'' = O(k)$ , da ein Paar sich schneidender Rechtecke höchstens zwei Punkteinschlüsse produziert und für jeden Punkteinschluss die zugehörigen Rechtecke sich schneiden.

Deshalb kann man insgesamt zusammenfassen:

Das Rechteckschnitt-Problem kann mittels Plane-Sweep in  $O(n \log n + k)$  Zeit und  $O(n \log n)$  Speicherplatz gelöst werden, wobei  $n$  die Anzahl der Rechtecke und  $k$  die Anzahl sich schneidender Paare ist.

Etwas unschön an dem so beschriebenen Algorithmus ist noch, dass ein Paar sich schneidender Rechtecke mehrfach ausgegeben werden kann. Man kann sich aber einen Trick überlegen, um solche Mehrfachausgaben zu unterdrücken, der die asymptotische Laufzeit und den Platzbedarf nicht erhöht.

**Selbsttestaufgabe 8.3:** Wie kann man im Rechteckschnitt-Algorithmus Mehrfachausgaben desselben Schnittpaars vermeiden, ohne asymptotisch die Laufzeit oder den Platzbedarf zu verschlechtern? □

Wir werden später noch sehen, dass sich der Speicherplatzbedarf durch Benutzung anderer Datenstrukturen zur Unterstützung des Plane-Sweep (Abschnitt 8.3) oder durch Anwendung von Divide-and-Conquer-Algorithmen (Abschnitt 8.2) auf  $O(n)$  reduzieren lässt.

### 8.1.3 Das Maßproblem

Als letzten Plane-Sweep-Algorithmus betrachten wir die Lösung des *Maßproblems*, das in zwei Dimensionen wie folgt definiert ist:

Gegeben eine Menge von Rechtecken, bestimme die Größe der insgesamt bedeckten Fläche (also den Flächeninhalt der Vereinigung aller Rechtecke).

Dieses Problem ist in Abbildung 8.15 illustriert.

### Plane-Sweep-Reduktion

Während wir gedanklich die Sweepline durch die gegebene Rechteckmenge schieben, können wir folgende Beobachtungen machen:

- Zu jedem Zeitpunkt schneidet die Sweepline die Rechteckmenge in einer Menge von  $y$ -Intervallen. Beim Passieren eines linken Rechteckendes ist das zugehörige  $y$ -Intervall in die Menge einzufügen und beim Passieren des rechten Endes zu entfernen.
- Zwischen zwei aufeinanderfolgenden Sweep-Positionen  $x_i$  und  $x_{i+1}$  gibt es eine wohldefinierte Menge  $Y$  aktuell vorhandener  $y$ -Intervalle (Abbildung 8.15). Die von Rechtecken bedeckte Fläche innerhalb des Streifens zwischen  $x_i$  und  $x_{i+1}$  ist gegeben durch das Produkt

$$(x_{i+1} - x_i) \cdot \text{measure}(Y)$$

wobei  $\text{measure}(Y)$  das eindimensionale (!) Maß der Intervall-Menge  $Y$  darstellt. Das gesuchte Maß der gesamten Rechteckmenge erhält man natürlich als Summe der Maße aller Streifen.

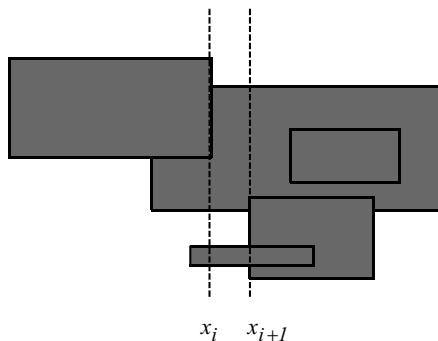


Abbildung 8.15: Maß-Problem

Dabei ist das Maß einer Intervallmenge  $I$  die Gesamtlänge der Vereinigung der Intervalle in  $I$ , wobei jedes Intervall als Punktmenge aufgefasst wird. (Die Vereinigung der Intervalle in  $I$  ist eine Menge disjunkter Intervalle  $I'$ ; das Maß von  $I$  ist die Summe der Längen aller Intervalle in  $I'$ .)

Damit ist durch den Plane-Sweep das Problem wieder auf das folgende eindimensionale Teilproblem reduziert: Gesucht ist eine Datenstruktur, die eine Menge von Intervallen speichert und die Operationen

- Einfügen eines Intervalls
  - Entfernen eines Intervalls
  - Abfrage des Maßes der gespeicherten Intervall-Menge

erlaubt.

## Ein modifizierter Segment-Baum

Wir kennen bereits eine Datenstruktur, die eine Menge von Intervallen darstellt und die ersten beiden Operationen erlaubt, nämlich den Segment-Baum. Tatsächlich lässt sich der Segment-Baum relativ einfach modifizieren, so dass er alle drei obigen Operationen unterstützt.

Wir erinnern uns, dass Intervalle in einen Segment-Baum eingetragen werden, indem sie gewisse Knoten (gerade die in der Menge  $CN(i)$ ) markieren, das heißt Einträge in deren Intervall-Liste erzeugen. Wir überlegen nun, wie groß für einen beliebigen Teilbaum mit Wurzelknoten  $p$  das Maß der *in ihm eingetragenen* Intervalle ist. Wir ignorieren also Intervalle, die den ganzen Teilbaum bedecken, aber nirgendwo in ihm eingetragen sind. Wir machen folgende Beobachtungen:

- Falls  $p$  ein Blatt ist, so ist das Maß der in diesem Teilbaum dargestellten Intervalle 0, falls  $p$ 's Intervall-Liste leer ist. Falls Intervalle eingetragen sind, so entspricht das Maß der Länge des  $p$  definierenden Intervalls *interval* ( $p$ ). Wieviele Intervalle eingetragen sind, spielt keine Rolle. Diese beiden Fälle sind in Abbildung 8.16 dargestellt. Wir zeichnen den Segment-Baum, wie gewohnt, über  $x$ -Koordinaten, obwohl er im Plane-Sweep für das Maßproblem y-Intervalle speichern soll.

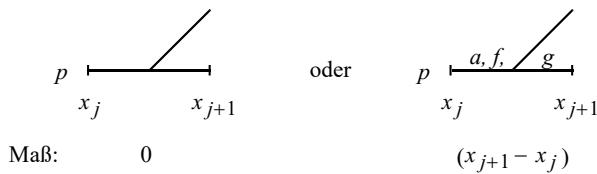


Abbildung 8.16: Blätter im modifizierten Segment-Baum

- Falls  $p$  ein innerer Knoten ist, so entspricht das Maß ebenfalls der Länge des definierten Intervalls  $\text{interval}(p)$ , falls Intervalle in  $p$ 's Intervall-Liste eingetragen sind (denn die gesamte Länge von  $\text{interval}(p)$  ist von diesen Intervallen bedeckt). Andernfalls erhält man das Maß durch Addition der Maße der beiden Teilbäume. Abbildung 8.17 zeigt die beiden Fälle.

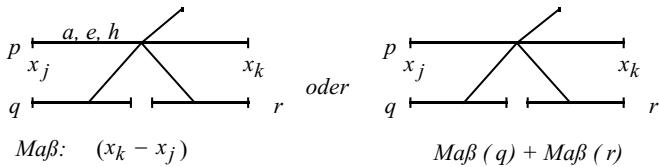


Abbildung 8.17: Innere Knoten im modifizierten Segment-Baum

Wir verändern die Baumstruktur nun wie folgt: Die einem Knoten  $p$  zugeordnete Intervall-Liste wird ersetzt durch einen einfachen Zähler  $p.count$ . Anstatt ein Intervall in die Liste einzutragen, erhöhen wir den Zähler um 1. Analog wird der Zähler beim Entfernen eines Intervalls um 1 erniedrigt. Weiter erhält jeder Knoten  $p$  ein Feld  $p.measure$ , das das Maß der in dem Teilbaum mit der Wurzel  $p$  repräsentierten Intervalle angibt. Der Wert dieses Feldes lässt sich "lokal" berechnen, nämlich durch Betrachten des Knotens  $p$  und (evtl.) seiner Söhne:

```

p.measure =
  if p.count > 0
    then p.top - p.bottom
  else if p ist ein Blatt then 0 else p.left.measure + p.right.measure end if
  end if

```

**Satz 8.3:** Der modifizierte Segment-Baum stellt eine Intervallmenge über einem Raster der Größe  $N$  in  $O(N)$  Speicherplatz dar. Das Einfügen oder Entfernen eines Intervalls unter gleichzeitiger Korrektur aller Knotenmaße ist in  $O(\log N)$  Zeit möglich. Die Abfrage des Maßes der dargestellten Intervallmenge erfordert nur  $O(1)$  Zeit.

**Beweis:** Jeder Knoten beansprucht jetzt nur noch konstanten Speicherplatz, da er keine Intervall-Liste mehr enthält. Da der Baum  $O(N)$  Knoten hat, folgt die Speicherplatz-Schranke. Zum Einfügen oder Entfernen eines Intervalls  $i$  müssen die Knoten in  $CN(i)$ , die an dem gegabelten "Pfad" liegen, aufgesucht werden. Dieser ist zur Erinnerung noch einmal in Abbildung 8.18 gezeigt.

Das Aufsuchen dieser Knoten erfordert nur  $O(\log N)$  Zeit. Nur für Knoten entlang dem Suchpfad muss  $p.measure$  korrigiert werden. Man behandelt dazu in den rekursiven Prozeduren für das Einfügen und Entfernen eines Intervalls (Aufgabe 8.19) die Knotenmaße in *postorder*, das heißt, berechnet das korrigierte Maß eines Knotens, nachdem die Maße der Söhne bekannt sind. Schließlich ist die Abfrage des Maßes möglich durch einfachen Zugriff auf  $root.measure$ .  $\square$

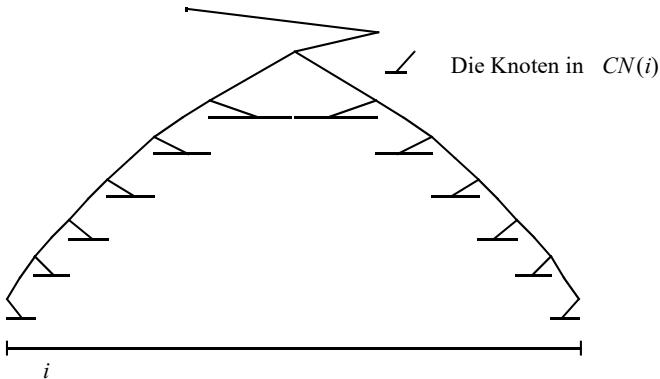


Abbildung 8.18: Gegabelter ‘‘Pfad’’ im (modifizierten) Segment-Baum

### Komplexität der Lösung des Maßproblems

**Satz 8.4:** Für eine Menge von  $n$  achsenparallelen Rechtecken in der Ebene kann das Maß mittels Plane-Sweep in  $O(n \log n)$  Zeit und  $O(n)$  Speicherplatz berechnet werden.

**Beweis:** Die Vorbereitung des Plane-Sweeps sowie der Aufbau des leeren Segment-Baums erfordert  $O(n \log n)$  Zeit. Während des Sweeps erfolgen je  $n$  Einfüge- und Lösch-Operationen von Intervallen in jeweils  $O(\log n)$  Zeit. Schließlich erfordert das  $n$ -malige Ablesen und Aufaddieren des Maßes  $O(n)$  Zeit.  $\square$

## 8.2 Divide-and-Conquer-Algorithmen für orthogonale Objekte

Beim Entwurf eines Divide-and-Conquer-Algorithmus zur Behandlung einer Menge planarer Objekte, etwa einer Rechteckmenge, ergibt sich sofort das Problem, wie man die Objektmenge aufteilt. Natürlich gibt es stets eine ‘‘triviale’’ Methode: Teile die Objektmenge  $\{o_1, \dots, o_n\}$  in zwei Mengen  $\{o_1, \dots, o_{\lfloor n/2 \rfloor}\}$  und  $\{o_{\lfloor n/2 \rfloor + 1}, \dots, o_n\}$  auf. Diese Aufteilung hilft uns aber nicht, das Problem zu lösen; es gibt keine geometrische Eigenschaft, die die Objekte der beiden Teilmengen unterscheidet. Eine bessere Idee besteht darin, eine teilende Gerade durch die Objektmenge zu legen (Abbildung 8.19).

Eine solche Gerade zerlegt aber die Objektmenge im Allgemeinen in *drei* Teilmengen: Objekte, die vollständig links von der Geraden liegen; solche, die rechts davon liegen; schließlich Objekte, die die Gerade schneiden. Diese Art der Aufteilung erweist sich als nicht sehr geeignet für den Entwurf von Algorithmen, da die Wechselwirkung zwischen

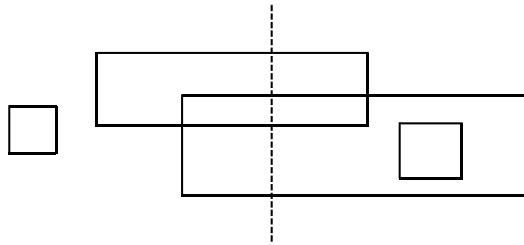


Abbildung 8.19: Teilende Gerade

Objekten, die die Gerade schneiden, und denen, die auf einer Seite liegen, schwer zu behandeln ist.

Eine wichtige Idee zur Lösung dieses Problems ist die *getrennte Darstellung* planarer geometrischer Objekte:

Stelle jedes Objekt, dessen  $x$ -Ausdehnung von 0 verschieden ist, durch sein linkes und rechtes Ende dar; behandle diese im Algorithmus als unabhängige Einheiten.

Dadurch wird das Teilen unproblematisch, da die verbleibenden Objekte jeweils durch eine  $x$ -Koordinate (und nicht ein  $x$ -Intervall) beschrieben sind. Die obige Rechteckmenge sieht dann wie im folgenden Diagramm aus (die “Haken” an den vertikalen Kanten zeigen dabei an, ob die Kante eine linke oder rechte ist):

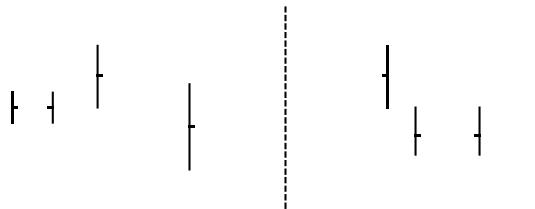


Abbildung 8.20: Objekte werden durch ihre Enden dargestellt

Um Divide-and-Conquer mit Plane-Sweep vergleichen zu können, betrachten wir zunächst noch einmal das Segmentschnitt- und das Maßproblem. Schließlich lösen wir noch ein recht komplexes Problem mit Divide-and-Conquer, das Konturproblem.

### 8.2.1 Das Segmentschnitt-Problem

Gegeben sei also eine Menge  $S$  horizontaler und vertikaler Segmente in getrennter Darstellung, das heißt, eine Menge von linken und rechten Endpunkten der horizontalen Segmente, sowie von vertikalen Segmenten. Sei *linsect* (für “line segment intersection”) der Name unseres rekursiven Algorithmus. Die ersten beiden Schritte für den Rekursionsfall, das heißt, wenn die Menge  $S$  mehr als ein Element enthält, sind klar:

*Divide:* Wähle eine  $x$ -Koordinate  $x_m$ , die  $S$  in zwei etwa gleich große Teilmenge  $S_1$  und  $S_2$  zerlegt.

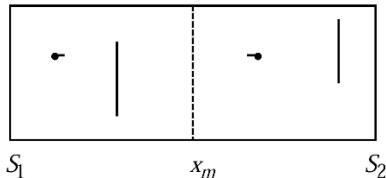


Abbildung 8.21: Divide-Schritt beim Segmentschnitt-Problem

*Conquer:* *linsect* ( $S_1$ ); *linsect* ( $S_2$ )

*Merge:* ?

An dieser Stelle müssen wir eine Annahme über das Ergebnis der beiden rekursiven Aufrufe machen. Eine solche Annahme heißt *rekursive Invariante*. In diesem Fall lautet sie:

Ein Aufruf von *linsect* ( $S_i$ ) hat nach der Beendigung alle Schnitte zwischen den in  $S_i$  repräsentierten Segmenten ausgegeben. Dabei sei ein horizontales Segment in  $S_i$  repräsentiert, falls mindestens ein Endpunkt in  $S_i$  enthalten ist.

Wenn wir diese Annahme für die Aufrufe *linsect* ( $S_1$ ) und *linsect* ( $S_2$ ) machen, so folgt, dass wir nur noch Schnitte zwischen horizontalen Segmenten in  $S_1$  und vertikalen Segmenten in  $S_2$  und umgekehrt zu berichten haben, um die rekursive Invariante für den Aufruf *linsect* ( $S$ ) zu erfüllen. Das ist also die Aufgabe des *Merge*-Schrittes.

Wir betrachten ein einzelnes horizontales Segment  $h$ , das in  $S_1$  repräsentiert ist. Es gibt folgende Fälle:

- (a) Beide Endpunkte von  $h$  liegen in  $S_1$ .

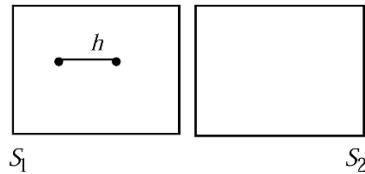


Abbildung 8.22: Beide Endpunkte von  $h$  liegen in  $S_1$

Offensichtlich schneidet  $h$  kein Segment in  $S_2$ .

- (b) Nur der rechte Endpunkt liegt in  $S_1$ .

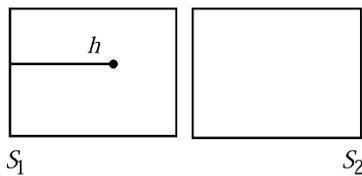


Abbildung 8.23: Nur der rechte Endpunkt liegt in  $S_1$

Auch in diesem Fall schneidet  $h$  kein Segment in  $S_2$ .

- (c) Nur der linke Endpunkt liegt in  $S_1$ .

- (c1) Der rechte Endpunkt liegt in  $S_2$ .

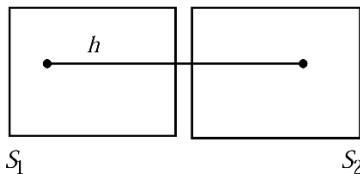


Abbildung 8.24: Linker Endpunkt in  $S_1$ , rechter in  $S_2$

Aufgrund der rekursiven Invarianten wissen wir, dass alle Schnitte zwischen  $h$  und Segmenten in  $S_2$  bereits ausgegeben worden sind, da  $h$  in  $S_2$  repräsentiert ist.

(c2) Der rechte Endpunkt liegt rechts von  $S_2$ .

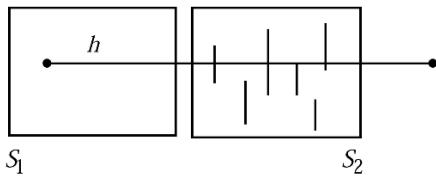


Abbildung 8.25: Linker Endpunkt in  $S_1$ , rechter rechts von  $S_2$

Das Segment  $h$  “durchquert” die gesamte Menge  $S_2$ .  $h$  schneidet alle vertikalen Segmente in  $S_2$ , deren  $y$ -Intervall seine  $y$ -Koordinate einschließt. Die Feststellung der Schnitte zwischen  $h$  und diesen Segmenten ist aber ein *eindimensionales* Problem! Das heißt, dass auch Divide-and-Conquer eine Reduktion auf ein eindimensionales Problem leistet, und zwar auf ein Mengenproblem, wie wir sehen werden.

Die zentrale Aufgabe des *Merge*-Schrittes besteht also darin, alle linken Endpunkte in  $S_1$ , deren zugehöriger “Partner” (rechter Endpunkt) weder in  $S_1$  noch in  $S_2$  liegt, zu bestimmen und mit der Menge vertikaler Segmente in  $S_2$  zu “vergleichen”. Bei diesem Vergleich interessieren nur die  $y$ -Koordinaten der Endpunkte bzw. die  $y$ -Intervalle der vertikalen Segmente, und die durchzuführende Operation (sie heiße “ $\otimes$ ”) ist die Bestimmung einer Menge von Paaren

$$C \otimes I := \{(c, i) \mid c \in C, i \in I \wedge i \text{ enthält } c\}$$

für eine Menge von Koordinaten  $C$  und eine Menge von Intervallen  $I$ . Diese Operation muss natürlich analog auch für rechte Endpunkte in  $S_2$  durchgeführt werden.

Der skizzierte Algorithmus lässt sich dann formal wie folgt beschreiben. Wir benutzen einen “Rahmenalgorithmus” *SegmentIntersectionDAC*, der die nötigen Initialisierungen durchführt und dann den eigentlichen rekursiven Algorithmus *linsect* aufruft.

**algorithm** *SegmentIntersectionDAC* ( $H, V$ )

{Eingabe ist eine Menge horizontaler Segmente  $H$  und eine Menge vertikaler Segmente  $V$ . Es gilt die Einschränkung, dass die  $x$ - und  $y$ -Koordinaten aller Segmente paarweise verschieden sind. Berechne mit Divide-and-Conquer die Menge aller sich schneidenden Paare  $(h, v)$  mit  $h \in H$  und  $v \in V$ .}

*Schritt 1:* Sei  $S = \{ (x_1, (x_1, x_2, y)) \mid (x_1, x_2, y) \in H \}$   
 $\cup \{ (x_2, (x_1, x_2, y)) \mid (x_1, x_2, y) \in H \}$   
 $\cup \{ (x, (x, y_1, y_2)) \mid (x, y_1, y_2) \in V \};$

sortiere  $S$  nach der ersten Komponente, also nach der x-Koordinate;

*Schritt 2:*  $linsect(S, LEFT, RIGHT, VERT)$

**end** *SegmentIntersectionDAC.*

Es ist interessant, dass  $S$  tatsächlich genau die gleiche Form hat wie im entsprechenden Plane-Sweep-Algorithmus. Dort wurden linke und rechte Enden von Segmenten für das Einfügen und Entfernen in die Sweep-line-Status-Struktur benötigt; hier für die getrennte Darstellung im Divide-and-Conquer. Die Parameter  $LEFT$ ,  $RIGHT$  und  $VERT$  erhalten Ausgaben von  $linsect$ , die im Rahmenalgorithmus nicht benötigt werden.

**algorithm** *linsect* ( $S, L, R, V$ )

{Eingabe ist  $S$  – eine nach  $x$ -Koordinaten sortierte Menge von Punkten, die als linke oder rechte Endpunkte identifizierbar sind, und vertikalen Segmenten.  
Ausgaben sind die folgenden Mengen:

$L$  – eine Menge von  $y$ -Koordinaten (-Objekten). Wir sprechen im Folgenden einfach von Koordinaten oder Intervallen, auch wenn diesen noch weitere Information zugeordnet ist.  $L$  enthält die  $y$ -Koordinaten aller linken Endpunkte in  $S$ , deren Partner nicht in  $S$  liegt.

$R$  – analog für rechte Endpunkte.

$V$  – eine Menge von  $y$ -Intervallen.  $V$  enthält die vertikalen Segmente in  $S$  anhand ihrer  $y$ -Intervalle.

Schließlich wird direkt ausgegeben die Menge aller sich schneidenden Paare  $(h, v)$ , wobei  $h$  ein horizontales und  $v$  ein vertikales in  $S$  repräsentiertes Segment ist.}

*Fall 1:*  $S$  enthält nur ein Element  $s$ .

- (1a)  $s = (x, (x', y))$  ist linker Endpunkt.  
 $L := \{(y, (x, x', y))\}; R := \emptyset; V := \emptyset$
- (1b)  $s = (x, (x', x, y))$  ist rechter Endpunkt.  
 $L := \emptyset; R := \{(y, (x', x, y))\}; V := \emptyset$
- (1c)  $s = (x, (x, y_1, y_2))$  ist ein vertikales Segment  
 $L := \emptyset; R := \emptyset; V := \{([y_1, y_2], (x, y_1, y_2))\}$

*Fall 2:*  $S$  enthält mehr als ein Element.

*Divide:* Wähle eine  $x$ -Koordinate  $x_m$ , die  $S$  in zwei etwa gleich große Teilmengen  $S_1$  und  $S_2$  teilt, wobei  $S_1 = \{s \in S \mid s_1 < x_m\}$  und  $S_2 = S \setminus S_1$ ;

*Conquer:* *linsect* ( $S_1, L_1, R_1, V_1$ );  
*linsect* ( $S_2, L_2, R_2, V_2$ );

*Merge:*  $LR := L_1 \cap R_2$ ;

(entspricht der Menge horizontaler Segmente, die sich im *Merge*-Schritt “treffen”)

$L := (L_1 \setminus LR) \cup L_2$ ;

$R := R_1 \cup (R_2 \setminus LR)$ ;

$V := V_1 \cup V_2$ ;

**output**  $((L_1 \setminus LR) \otimes V_2)$ ;

**output**  $((R_2 \setminus LR) \otimes V_1)$

**end** *linsect*.

Für die Eingabemenge  $S$  bildet also *linsect* unter anderem die Menge  $L$  aller linken Endpunkte, deren Partner nicht in  $S$  liegt, aus den Mengen  $L_1$  und  $L_2$ , die ihm von den rekursiven Aufrufen geliefert werden. Linke Enden ohne Partner aus  $S_2$  (repräsentiert in  $L_2$ ) sind direkt auch linke Enden ohne Partner für  $S$ . Von den linken Enden aus  $S_1$  müssen diejenigen abgezogen werden, die in  $S_2$  ihren Partner gefunden haben; entsprechende Paare von Punkten aus  $S_1$  und  $S_2$  werden vorher in  $LR$  ermittelt.

Die eigentliche Bestimmung von Paaren sich schneidender Segmente erfolgt mithilfe der oben schon beschriebenen Operation “ $\otimes$ ”. Diese Operation repräsentiert das *eindimensionale Punkteinschluss-Mengenproblem*, das lautet:

Gegeben eine Menge von Koordinaten und eine Menge von Intervallen, finde alle Paare (Koordinate, Intervall), so dass die Koordinate im Intervall liegt.

Man sieht, wie Plane-Sweep eine Reduktion eines zweidimensionalen Mengenproblems auf ein eindimensionales dynamisches Suchproblem bietet und Divide-and-Conquer eine Reduktion auf ein eindimensionales Mengenproblem. Für den obigen Algorithmus können wir nun die Operation “ $\otimes$ ” noch präziser definieren.  $C$  sei eine Menge von Paaren, deren erste Komponente (Schlüssel) eine Koordinate ist,  $I$  eine Menge von Paaren, deren Schlüssel ein Intervall ist.

$$C \otimes I := \{(h, v) \mid \exists (c, h) \in C, \exists ([c_1, c_2], v) \in I: c \in [c_1, c_2]\}$$

Der Algorithmus stützt sich auf Mengenoperationen ab. Daher müssen wir zur Analyse eine Darstellung der Mengen und eine Implementierung der Mengenoperationen festlegen. Wir repräsentieren  $S$  in einem  $x$ -sortierten Array. Der *Divide*-Schritt braucht dann nur konstante Zeit.  $L$  und  $R$  werden als  $y$ -sortierte einfach verkettete Listen dargestellt. Die Intervallmenge  $V$  wird ebenfalls als verkettete Liste dargestellt, in der die Objekte aus  $V$  nach unteren Intervallgrenzen geordnet sind. Wir wissen aus Abschnitt 4.1, dass auf dieser Darstellung alle Standard-Mengenoperationen (Vereinigung, Durchschnitt, Differenz) mittels parallelem Listendurchlauf jeweils in linearer Zeit durchführbar sind. Man kann sich überlegen, dass die Operation  $\otimes$  auf der beschriebenen Listendarstellung in  $O(|C| + |I| + |C \otimes I|)$  Zeit realisiert werden kann, das heißt, in Linearzeit plus Zeit proportional zur Anzahl der gefundenen Paare.

**Selbsttestaufgabe 8.4:** Wie kann man die Operation  $C \otimes I$  des Algorithmus *LINSECT* in  $O(|C| + |I| + |C \otimes I|)$  Zeit ausführen? □

Sei  $T(n)$  der Zeitbedarf von *linsect*, angewandt auf eine Menge  $S$  mit  $n$  Elementen. Falls  $n = 1$ , so ist offensichtlich nur konstante Zeit erforderlich, um die Aktionen des Falles 1 durchzuführen. Für  $n > 1$  lassen sich alle Aktionen des *Merge*-Schrittes in  $O(n + k')$  Zeit durchführen, wobei  $k'$  die Anzahl der in diesem *Merge*-Schritt ermittelten Schnittpaare

ist (also  $k'$  entspricht gerade  $|C \otimes I|$ ). Wenn wir die Zeiten  $k'$  getrennt betrachten, erhalten wir die aus Abschnitt 5.2 bekannte Rekursionsgleichung für Divide-and-Conquer-Algorithmen:

$$T(n) = \begin{cases} O(1) & \text{falls } n = 1 \\ O(1) + 2 \cdot T(n/2) + O(n) & \text{falls } n > 1 \end{cases}$$

Diese rekursiven Gleichungen haben gemäß Satz 5.2 die Lösung:

$$T(n) = O(n \log n)$$

Hinzu kommt der Zeitbedarf von  $O(k)$  für die Ausgabe der  $k$  sich schneidenden Paare. Der vorbereitende Algorithmus *SegmentIntersectionDAC* braucht  $O(n \log n)$  Zeit. Der Platzbedarf (Darstellung der Mengen durch Listen) ist offensichtlich linear. Es folgt:

Das Segmentschnitt-Problem kann für  $n$  Segmente mit  $k$  sich schneidenden Paaren mittels Divide-and-Conquer in  $O(n \log n + k)$  Zeit und  $O(n)$  Speicherplatz gelöst werden.

Man kann die hier beschriebene Technik auf recht einfache Art verallgemeinern, um auch das Punkteinschluss- und das Rechteckschnitt-Problem jeweils in  $O(n \log n + k)$  Zeit und  $O(n)$  Speicherplatz zu lösen. Man erhält also auch für diese beiden Probleme zeit- und speicherplatzoptimale Lösungen.

**Selbsttestaufgabe 8.5:** Wie sieht der DAC-Algorithmus für das Punkteinschluss-Problem aus? □

### 8.2.2 Das Maßproblem

Beim Versuch, das Maßproblem mit Divide-and-Conquer zu lösen, tritt eine Schwierigkeit auf, die es beim Segmentschnitt-Problem nicht gab: Man kann offenbar in einem *Merge*-Schritt keine “vernünftigen” Ergebnisse berechnen. Denn das Maß einer Menge von Rechtecken, die in zwei benachbarten, miteinander zu verschmelzenden Gebieten repräsentiert sind, kann durch ein hier nicht repräsentiertes Rechteck, das diese beiden Gebiete in x-Richtung vollständig überdeckt, ungültig werden (Abbildung 8.26).

Beim Segmentschnitt-Problem hingegen war ein gefundener Schnitt in jedem Fall gültig, unabhängig von weiteren vorhandenen Objekten. Die entscheidende Idee zur Überwindung dieser Schwierigkeit besteht darin, mittels Divide-and-Conquer *zunächst eine vollständige Beschreibung der Vereinigung aller Rechtecke zu berechnen*, die es “leicht” macht, anschließend das Maß zu berechnen.

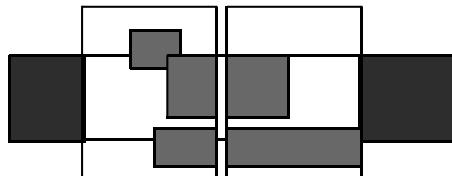


Abbildung 8.26: Merge-Schritt beim Segmentschnitt-Problem

Als Beschreibung benutzen wir eine *Streifenmenge*. Das sei eine Zerlegung des Gebietes, in dem die Rechtecke liegen, in horizontale Streifen, deren Grenzen durch die Rechtecke induziert sind, das heißt, jede horizontale Rechtekkante liegt auf einer Streifengrenze und umgekehrt.<sup>4</sup> Eine derartige Zerlegung ist in Abbildung 8.27 gezeigt.

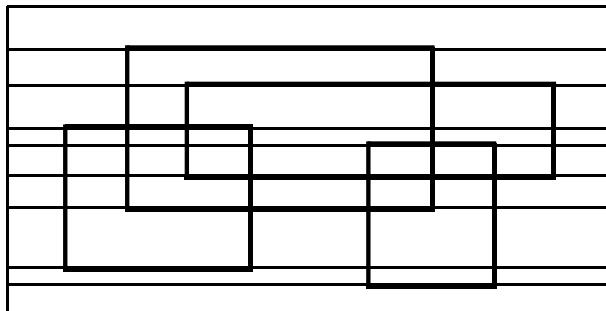


Abbildung 8.27: Streifenmenge

Eine Streifenmenge beschreibt nun nicht die Rechteckmenge an sich, sondern die *Vereinigung* dieser Rechtecke. Welche Information jedem Streifen zugeordnet ist, hängt davon ab, welche Eigenschaft der Vereinigung berechnet werden soll. Eine vollständige, präzise Beschreibung der Vereinigung erhält man, indem man jedem Streifen eine Menge disjunkter x-Intervalle zuordnet, wie in Abbildung 8.28 (a).

Eine solche Beschreibung wird benötigt, wenn man etwa aus der Streifenmenge die *Kontur* des bedeckten Gebietes berechnen will (nächster Abschnitt). Für die Berechnung des Maßes genügt es, für jeden Streifen die Gesamtlänge seiner Menge von x-Intervallen zu verwalten, genannt *x-Maß*, entsprechend der Darstellung in Abbildung 8.28 (b).

---

4. Wir nehmen zur Vereinfachung auch in diesem Abschnitt an, dass die x- und y-Koordinaten aller Rechtecke paarweise verschieden sind. Im allgemeinen Fall können auf einer Streifengrenze mehrere Rechteckkanten liegen.

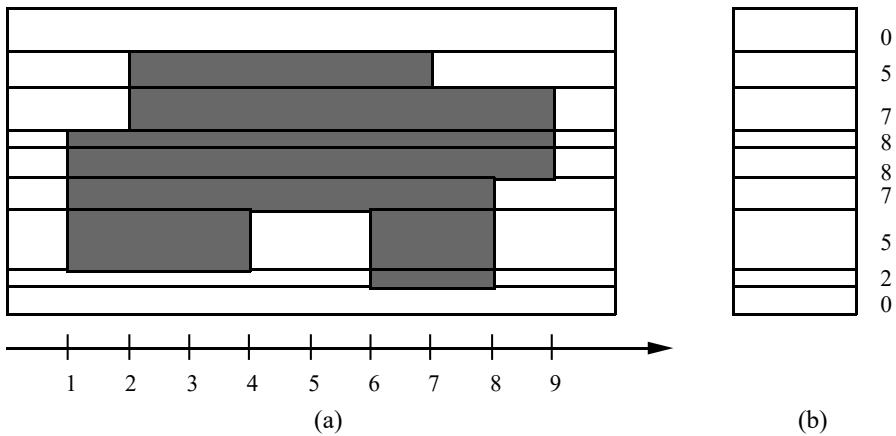
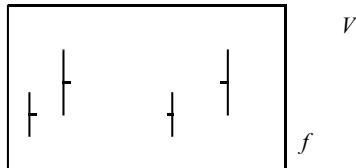


Abbildung 8.28: Streifen zugeordnete Informationen

Eine solche Darstellung macht es in der Tat leicht, das Maß der Vereinigung zu berechnen. Dazu sind einfach die Streifenmaße aufzuaddieren; jedes Streifenmaß ergibt sich als Produkt der  $y$ -Ausdehnung des Streifens mit dem eingetragenen  $x$ -Maß.

Wir überlegen nun, wie man mit Divide-and-Conquer eine solche Streifenmenge berechnen kann. Gegeben sei eine Rechteckmenge  $R$  in getrennter Darstellung, also als Menge  $V$  vertikaler linker und rechter Rechteckkanten. Zusätzlich legen wir einen ‘‘Rahmen’’  $f$  um die gesamte Menge  $V$ , also ein Rechteck, das alle Rechtecke in  $R$  einschließt. Der zu beschreibende Algorithmus heiße *stripes*; er wird angewandt auf  $V$  und  $f$ .

Abbildung 8.29: Eingabe des Algorithmus *stripes*

Die ersten beiden Schritte sind wieder klar:

- Divide:* Wähle eine  $x$ -Koordinate  $x_m$ , die  $V$  in zwei etwa gleich große Teilmengen  $V_1$  und  $V_2$  teilt. (Gleichzeitig entstehen zwei Teilrahmen  $f_1$  und  $f_2$ .)  
*Conquer:*  $\text{stripes}(V_1, f_1); \text{stripes}(V_2, f_2)$

An dieser Stelle müssen wir wissen, welches Ergebnis die rekursiven Aufrufe geliefert haben, das heißt, die *rekursive Invariante* für dieses Problem kennen. Hier benutzen wir folgende Invariante:

*stripes* ( $V_i$ ) berechnet als Aufteilung des zu  $V_i$  gehörenden Rahmens  $f_i$  eine Streifenmenge, deren Grenzen durch die Enden der Kanten in  $V_i$  induziert sind. Jedem Streifen ist das x-Maß der eingeschlossenen bedeckten Fläche zugeordnet (bedeckt bzgl. der in  $V_i$  repräsentierten Rechtecke)

Im terminierenden Zweig des Algorithmus *stripes* wird für eine einzige Rechteckkante z. B. die folgende Streifensequenz berechnet:

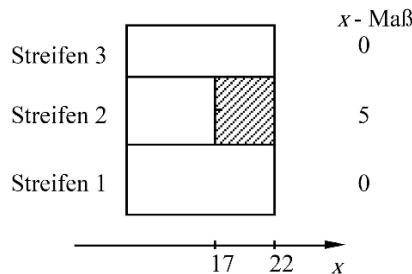


Abbildung 8.30: Ergebnis von *stripes* im terminierenden Zweig

Im *Merge*-Schritt geht es nun darum, die für  $V_1$  und  $V_2$  erzeugten Streifensequenzen zu der korrekten Streifensequenz für  $V$  zu verschmelzen. Diese Aufgabe wird in drei Teilschritten durchgeführt:

1. Streifengrenzen ausdehnen, das heißt, die y-Aufteilungen angleichen

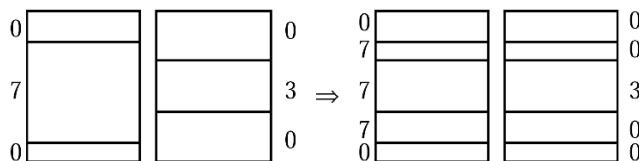


Abbildung 8.31: Ausdehnen der Streifengrenzen

Das Spalten eines Streifens durch Einziehen einer horizontalen Grenze ändert sein x-Maß nicht. Die neu entstehenden Streifen übernehmen daher einfach das x-Maß des ursprünglichen Streifens.

## 2. Gebiet bedecken

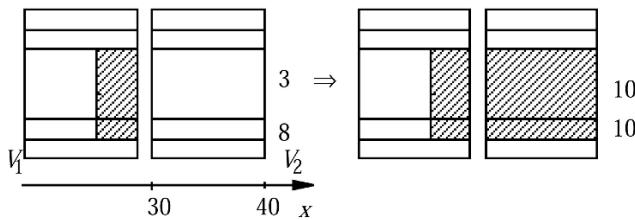


Abbildung 8.32: Bedecken des Gebietes

Wir wissen bereits vom Segmentschnitt-Algorithmus, dass nur linke Rechteckkanten in  $V_1$ , deren Partner nicht in  $V_2$  liegt, sowie rechte Kanten in  $V_2$ , deren Partner nicht in  $V_1$  liegt, betrachtet werden müssen. Sei z. B.  $e$  eine solche Kante in  $V_1$ . Alle Streifen in  $V_2$ , deren  $y$ -Intervall im  $y$ -Intervall von  $e$  enthalten ist, sind vollständig bedeckt von dem zu  $e$  gehörenden Rechteck. Daher ist ihr  $x$ -Maß zu korrigieren, und zwar auf die volle Länge ihres  $x$ -Intervalls.

## 3. Streifen aneinanderhängen

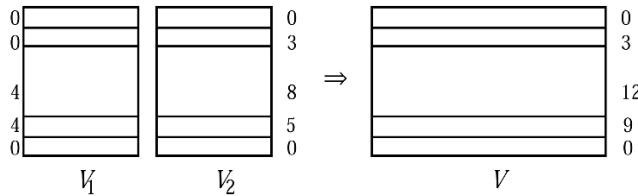


Abbildung 8.33: Aneinanderhängen der Streifen

Da die Gebiete zweier aneinandergehängerter Streifen disjunkt sind, kann man ihre  $x$ -Maße einfach addieren, um das  $x$ -Maß des resultierenden Streifens zu erhalten.

Diese Überlegungen führen zu folgender formaler Beschreibung des Algorithmus *stripes*. Zunächst wird wieder ein Rahmenalgorithmus benötigt, der die Menge vertikaler Kanten erzeugt und dann die eigentliche rekursive Prozedur *stripes* aufruft.

**algorithm** *StripesDAC* ( $R, S$ )

{Eingabe ist eine Menge von Rechtecken  $R$ . Ausgabe ist eine Streifenmenge  $S$  für  $R$ , in der jedem Streifen das x-Maß der bedeckten Fläche zugeordnet ist.}

*Schritt 1:*  $V := \{(x_1, (x_1, x_2, y_1, y_2)) | (x_1, x_2, y_1, y_2) \in R\}$

$\cup \{(x_2, (x_1, x_2, y_1, y_2)) | (x_1, x_2, y_1, y_2) \in R\};$

(die Objekte der ersten Teilmenge werden als *linke Kanten*, die der zweiten als *rechte Kanten* aufgefasst)

sortiere  $V$  nach der ersten Komponente, also nach der x-Koordinate;

*Schritt 2:* *stripes* ( $V, -\infty, +\infty, LEFT, RIGHT, POINTS, S$ )

(die Parameter für *stripes* sind unten erklärt)

**end** *StripesDAC*.

**algorithm** *stripes* ( $V, x_l, x_r, L, R, P, S$ )

{*Eingaben:*  $V$  – eine nach x-Koordinaten geordnete Menge vertikaler Rechteckkanten.

$x_l, x_r$  – linke und rechte x-Koordinaten, die alle Segmente in  $V$  einschließen (die x-Koordinaten des “Rahmens”  $f$ ).

*Ausgaben:*  $L$  – eine Menge von y-Intervallen. Enthält die y-Intervalle aller Rechtecke, deren linke Kante in  $V$  vorkommt, die rechte aber nicht.

$R$  – analog für rechte Rechteckkanten.

$P$  – eine Menge von y-Koordinaten. Enthält die y-Koordinaten aller Endpunkte von Kanten in  $V$ , sowie die Grenzwerte  $-\infty$  und  $+\infty$ .

$S$  – die von  $V$  induzierte Streifenmenge, wie oben geschildert.}

*Fall 1:*  $V$  enthält nur ein Element  $v$ .

(1a)  $v = (x, (x, x', y_1, y_2))$  ist linke Kante.

$L := \{([y_1, y_2], (x, x', y_1, y_2))\}; R := \emptyset;$

$P := \{-\infty, y_1, y_2, +\infty\};$

$S := \{([- \infty, y_1], 0), ([y_1, y_2], x_r - x), ([y_2, +\infty], 0)\}$

(1b)  $v = (x, (x', x, y_1, y_2))$  ist rechte Kante: analog.

*Fall 2:*  $V$  enthält mehr als ein Element.

*Divide:* Wähle eine x-Koordinate  $x_m$ , die  $V$  in zwei etwa gleich große

Teilmengen  $V_1$  und  $V_2$  teilt, wobei  $V_1 = \{v \in V | v_1 < x_m\}$  und  $V_2 = V \setminus V_1$ ;

*Conquer:* *stripes* ( $V_1, x_l, x_m, L_1, R_1, P_1, S_1$ );

*stripes* ( $V_2, x_m, x_r, L_2, R_2, P_2, S_2$ );

*Merge:*  $LR := L_1 \cap R_2$ ; (zusammengehörige Rechteckkanten)

```

 $L := (L_1 \setminus LR) \cup L_2;$ 
 $R := R_1 \cup (R_2 \setminus LR);$ 
 $P := P_1 \cup P_2;$ 
 $S_l := refine(S_1, P);$ 
 $S_r := refine(S_2, P);$ 
 $S_l := cover(S_l, R_2 \setminus LR, x_l, x_m);$ 
 $S_r := cover(S_r, L_1 \setminus LR, x_m, x_r);$ 
 $S := concat(S_l, S_r)$ 
```

**end stripes.**

Die Operationen *refine*, *cover* und *concat* entsprechen gerade den oben beschriebenen Schritten *Streifengrenzen angleichen*, *Gebiet bedecken* und *Streifen aneinanderhängen*. Die Operation *refine* nimmt als Operanden eine Streifenmenge  $S$  und eine Menge von y-Koordinaten  $P$  und liefert als Ergebnis eine neue Streifenmenge, in der die Streifen von  $S$  mehrfach dargestellt sind (mit korrigierten y-Intervallen), in deren Innerem y-Koordinaten aus  $P$  liegen. Die Operation *cover* nimmt eine Streifenmenge  $S$ , eine Menge von y-Intervallen  $Y$  und zwei x-Koordinaten  $x$  und  $x'$  und liefert eine korrigierte Streifenmenge, in der jeder Streifen von  $S$ , dessen y-Intervall von einem y-Intervall in  $Y$  überdeckt wird, das Maß  $x' - x$  hat; die übrigen Maße sind unverändert. Schließlich nimmt *concat* zwei Streifenmengen  $S$ ,  $S'$  und bildet eine weitere Streifenmenge; darin gibt es für je zwei zusammenpassende Streifen aus  $S$  und  $S'$  (mit gleichen y-Intervallen) einen neuen Streifen, dessen Maß die Summe der beiden Streifenmaße ist.

Man sieht wieder, wie Divide-and-Conquer eine Reduktion auf eindimensionale Mengenprobleme erreicht. *Refine* vergleicht eine Menge disjunkter y-Intervalle (*y-Partition*) mit einer Menge von y-Koordinaten, *cover* eine y-Partition mit einer Menge von y-Intervallen und *concat* zwei gleichartige y-Partitionen.

Zum Schluss müssen wir Repräsentationen für die beteiligten Mengen festlegen, die eine effiziente Ausführung aller benötigten Operationen erlauben. Das Ziel besteht darin, den *Merge*-Schritt insgesamt in Linearzeit ausführbar zu machen. Wenn wir als ‘‘Default’’ annehmen, dass alle Mengen durch y-geordnete Listen dargestellt sind (wie in *linsect*), dann sind wieder alle Standard-Mengenoperationen unproblematisch, das heißt durch parallelen Durchlauf in Linearzeit realisierbar. Das gilt auch für *refine* und *concat*, da in beiden Fällen parallele Durchläufe durch in y-Richtung total geordnete Mengen (*y-Partition* bzw. y-Koordinaten) genügen. Nicht ganz so offensichtlich ist die Realisierung von

*cover*. Hier genügt ein “eindimensionaler Sweep” in y-Richtung, bei dem folgende Aktionen durchzuführen sind (seien  $S$ ,  $Y$ ,  $x$  und  $x'$  wie oben die Parameter):

- *Initialisierung*:  $count := 0$ ; (gibt die Anzahl gerade offener Intervalle aus  $Y$  an);
- *Sweep*: beim Antreffen
  - des unteren Endes eines y-Intervalls aus  $Y$ :  $count := count + 1$ ;
  - des oberen Endes eines y-Intervalls aus  $Y$ :  $count := count - 1$ ;
  - eines Streifens aus  $S$ : falls  $count > 0$ , setze sein Maß auf  $x' - x$

Die Sweep-Status-Struktur ist hierbei ein schlichter Zähler. Wie man sieht, ist damit auch *cover* in Linearzeit realisierbar; es folgt, dass die y-Intervallmengen  $L$  und  $R$  als y-geordnete Listen repräsentiert werden müssen, in denen jedes Intervall einmal anhand des unteren und einmal anhand des oberen Endes dargestellt ist.

Da somit der *Merge*-Schritt in  $O(n)$  Zeit durchführbar ist, benötigen die Algorithmen *stripes* und *StripesDAC* insgesamt  $O(n \log n)$  Zeit, das Aufsummieren der Streifenmaße kostet nur  $O(n)$  Zeit. Der Speicherplatzbedarf ist ebenfalls linear. Es folgt:

Das Maß der Vereinigung einer Menge von  $n$  Rechtecken kann mittels Divide-and-Conquer in  $O(n \log n)$  Zeit und  $O(n)$  Speicherplatz berechnet werden.

Dieser Algorithmus ist ebenso wie der Plane-Sweep-Algorithmus (Abschnitt 8.1.3) zeit- und speicherplatzoptimal.

### 8.2.3 Das Konturproblem

Als letztes und schwierigstes Problem im Zusammenhang mit Rechteckmengen betrachten wir die Bestimmung der Kontur der Vereinigung der Rechtecke. Die *Kontur* trennt freie und bedeckte Gebiete der Ebene und besteht aus einer Menge von *Konturzyklen*, die jeweils eine Sequenz abwechselnd horizontaler und vertikaler Liniensegmente bilden (Abbildung 8.34).

Das *Konturproblem* besteht in der Berechnung dieser Menge von Konturzyklen. Alternativ kann man auch nur die Ausgabe der Menge von Segmenten, die die Kontur ausmachen, verlangen. Im Übrigen kann man die Lösung des ersten Problems auf die des zweiten zurückführen: Sei  $n$  die Größe der Rechteckmenge und  $p$  die Anzahl der Segmente in der Kontur. Falls man z. B. alle horizontalen Konturstücke kennt, so kann man aus ihnen in  $O(p)$  Zeit und Platz die vertikalen Stücke berechnen und die Konturzyklen herstellen ([Lipski und Preparata 1980], Aufgabe 8.20).

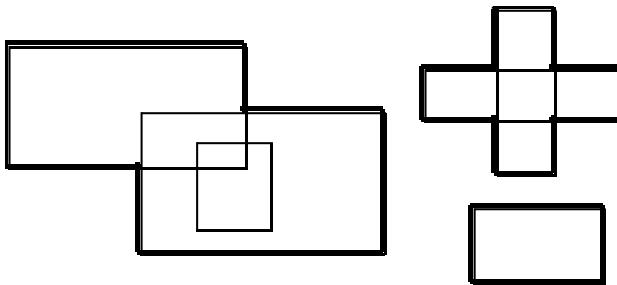


Abbildung 8.34: Kontur-Problem

Lösungen des Konturproblems haben Anwendungen im VLSI-Entwurf, in der Computer-Graphik und in speziellen Datenbankbereichen (z. B. Architekturdatenbanken).

Es gibt Plane-Sweep-Algorithmen für das Konturproblem; der zeitoptimale Plane-Sweep-Algorithmus ist aber abenteuerlich kompliziert. Er benutzt als Sweep-line-Status-Struktur eine trickreiche Modifikation des Segment-Baums. Im Gegensatz dazu ist die Divide-and-Conquer-Lösung relativ einfach zu erklären. Tatsächlich bekommen wir diese Lösung als Variante unseres Algorithmus *stripes* “fast geschenkt”.

Die Idee ist, mit *stripes* zunächst wieder eine Streifenmenge als Darstellung der Vereinigung der Rechtecke zu berechnen und aus dieser Darstellung dann die Konturstücke zu bestimmen. Wie muss der Inhalt eines Streifens gewählt werden, damit das effizient möglich ist? Wir haben im vorigen Abschnitt schon gesehen, dass eine vollständige Darstellung der Vereinigung vorliegt, wenn jedem Streifen *eine Menge disjunkter x-Intervalle* zugeordnet ist, die die bedeckten Teile des Streifens beschreibt (siehe Abbildung 8.28 (a)). Aus einer solchen Beschreibung kann man die Konturstücke folgendermaßen gewinnen:

Für jede horizontale Rechteckkante *e*:

1. Bestimme den “richtigen” benachbarten Streifen in der Streifensequenz (für die Unterkante eines Rechtecks ist es der Streifen darunter, für die Oberkante der darüber);

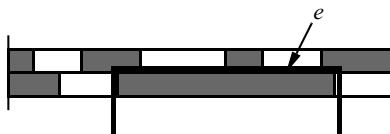


Abbildung 8.35: Benachbarte Streifen

2. Bestimme die freien x-Intervalle des Streifens innerhalb des x-Intervalls der Rechteckkante  $e$ ; diese ergeben Konturstücke von  $e$ .

Gesucht ist also eine Darstellung einer Menge disjunkter Intervalle  $D$ , die einerseits Queries dieser Art unterstützt:

- Für ein Query-Intervall  $i$ , bestimme die bezüglich  $D$  “freien” Teile von  $i$  (formal: den Schnitt von  $i$  mit dem Komplement von  $D$ ).

Eine wünschenswerte Zeitkomplexität für diese Operation wäre  $O(\log m + p_i)$ , wobei  $p_i$  die Anzahl der freien Stücke von  $i$  angibt,  $m$  die Anzahl der Intervallgrenzen im Streifen (es gilt  $m = O(n)$ ). Andererseits muss diese Darstellung im Algorithmus *stripes* effizient manipuliert werden können, das heißt, die drei Operationen des *Merge*-Schrittes unterstützen:

1. *Streifengrenzen angleichen (refine)*: Kopiere eine Intervallmenge.
2. *Gebiet bedecken (cover)*: Ersetze eine Intervallmenge durch eine Menge, die nur ein einziges Intervall enthält.
3. *Streifen aneinanderhängen (concat)*: Bilde die “Konkatenation” zweier Mengen disjunkter Intervalle.

Damit der *Merge*-Schritt weiter in linearer Zeit ausführbar ist, dürften all diese Operationen jeweils nur konstante Zeit benötigen. Das sind insgesamt ziemlich harte Anforderungen. Erstaunlicherweise existiert eine solche Datenstruktur: Eine Menge disjunkter Intervalle, also der Inhalt eines Streifens, sei dargestellt als *ein Zeiger auf einen binären Suchbaum*, in dessen Blättern die Intervallgrenzen abgespeichert sind. Jedes Blatt enthält zusätzlich Information, ob es sich um das linke oder rechte Ende eines Intervalls handelt. Jeder innere Knoten des Baumes enthält eine x-Koordinate, die zwischen den im linken bzw. rechten Teilbaum auftretenden x-Koordinaten liegt. Für eine leere Intervallmenge hat der Zeiger den Wert *nil*.

Im terminierenden Zweig des Algorithmus hat z. B. die für eine einzelne Kante konstruierte Streifenmenge folgende Darstellung:

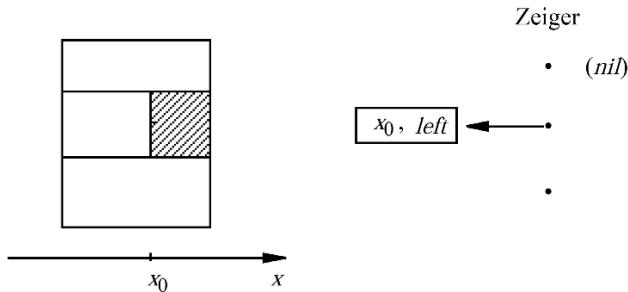


Abbildung 8.36: Streifenmenge für eine einzelne Kante

Die drei Operationen des *Merge*-Schrittes werden dann so implementiert:

1. *Ausdehnung der Streifengrenzen*:

Teile einen Streifen, kopiere seine Intervallmenge.

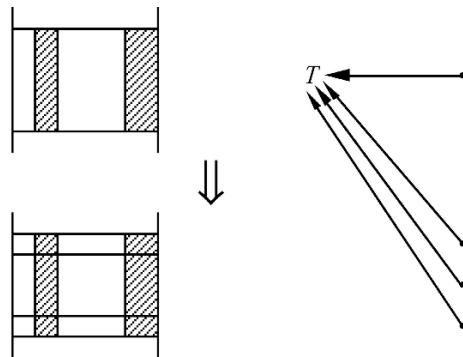


Abbildung 8.37: Ausdehnen der Streifengrenzen

Kopieren wird also durch einfaches Setzen neuer Zeiger realisiert.

2. Gebiet bedecken

Für jeden zu bedeckenden Streifen wird sein Zeiger einfach auf *nil* gesetzt, da keine Intervallgrenze in einem vollständig bedeckten Streifen liegt.

### 3. Streifen aneinanderhängen

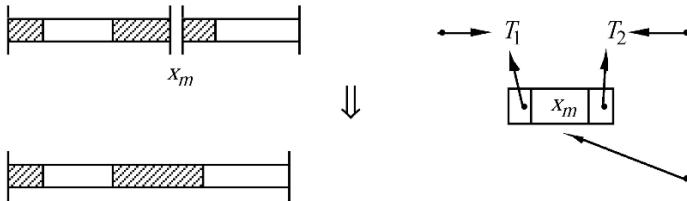


Abbildung 8.38: Aneinanderhängen der Streifen

Aus den Bäumen der beiden Ausgangsstreifen wird durch Hinzufügen eines neuen Wurzelknotens ein Baum für den konkatenierten Streifen erzeugt. Falls  $T_1$  oder  $T_2$  leer sind, so wird keine neue Wurzel erzeugt, sondern der neue Zeiger auf den nichtleeren Baum gesetzt; falls beide leer sind, wird der Zeiger auf *nil* gesetzt. So wird die Erzeugung leerer Teilbäume vermieden.

Als Ergebnis dieser Operation entsteht insgesamt eine vernetzte Struktur, da Teilbäume von vielen Streifen gemeinsam benutzt werden können. Man mache sich aber klar, dass die von einem beliebigen Streifen aus erreichbaren Knoten einen ganz normalen binären Suchbaum bilden. Konzeptionell können wir also davon ausgehen, dass jeder Streifen seinen eigenen Baum besitzt.

Man sieht, dass jede dieser Operationen nur konstante Zeit pro Streifen benötigt. Es folgt, dass der *Merge*-Schritt insgesamt lineare Zeit erfordert, woraus sich eine Gesamtzeitkomplexität von  $O(n \log n)$  für den rekursiven Algorithmus *stripes* ergibt. Wie steht es mit dem Platzbedarf? Sei  $SP(n)$  der Platzbedarf von *stripes*, angewandt auf eine Menge  $V$  mit  $n$  Elementen. Es gilt:

$$n = 1: SP(1) = O(1) \quad (1)$$

$$n > 1: SP(n) = O(n) + 2 SP(n/2) \quad (2)$$

(1) ist offensichtlich (siehe Abbildung 8.36). (2) folgt aus der Beobachtung, dass im *Merge*-Schritt  $O(n)$  Streifen erzeugt werden und für jeden Streifen im schlimmsten Fall ein neuer Wurzelknoten mit zwei Zeigern erzeugt wird, also konstanter Platz belegt wird. Dies ergibt den  $O(n)$ -Term. Die Tatsache, dass die im *Conquer*-Schritt erzeugten Strukturen nicht freigegeben, sondern weiterbenutzt werden, verursacht den  $2 SP(n/2)$ -Term. Die Lösung dieser Rekursionsgleichungen ist uns ja wohlbekannt, es gilt

$$SP(n) = O(n \log n).$$

Der Algorithmus *StripesDAC* benötigt also in diesem Fall  $O(n \log n)$  Zeit und Speicherplatz.

Abschließend wird anhand der erzeugten Streifenmenge, wie oben geschildert, die Menge der horizontalen Konturstücke berechnet. Dazu berechnet man in  $O(n)$  Zeit die horizontalen Rechteckkanten aus der gegebenen Rechteckmenge  $R$  und sortiert diese in  $O(n \log n)$  Zeit nach y-Koordinate. Es folgt ein paralleler Durchlauf durch die y-geordnete Streifenmenge und die Folge der Rechteckkanten. Für jede Kante  $e$  erfolgt eine Suche auf dem Suchbaum des “richtigen” Streifens. Diese Suche erfordert tatsächlich nur  $O(\log n + p_i)$  Zeit, wie oben gefordert; ein Streifen kann nämlich nur maximal  $O(n)$  Intervallgrenzen enthalten, und es ist aufgrund des Konstruktionsprozesses garantiert, dass der Baum balanciert ist. Alle  $p_i$ -Beiträge erzeugen Konturstücke, sind also Teil der Anzahl  $p$ , die die Gesamtgröße der Kontur bezeichnet. Insgesamt erhält man:

Für eine Menge von  $n$  Rechtecken kann die Kontur (als Menge von Konturzyklen) mittels Divide-and-Conquer in  $O(n \log n + p)$  Zeit und Speicherplatz berechnet werden, wobei  $p$  die Größe der Kontur angibt. Falls nur die Ausgabe der Menge der Kontursegmente verlangt ist, reduziert sich der Platzbedarf auf  $O(n \log n)$ .

Dieser Algorithmus ist zeitoptimal, leider aber nicht speicherplatzoptimal. Immerhin ist es bemerkenswert, dass Divide-and-Conquer überhaupt eine zeitoptimale Lösung für dieses relativ schwierige Problem liefert. Wie schon erwähnt, ist der zeit- und speicherplatzoptimale Plane-Sweep-Algorithmus viel komplizierter.

Abschließend machen wir zu Plane-Sweep- und Divide-and-Conquer-Algorithmen folgende Beobachtungen:

1. Plane-Sweep reduziert ein zweidimensionales Mengenproblem auf ein eindimensionales dynamisches Suchproblem. Diese Reduktion ist relativ einfach; die effiziente Lösung des Suchproblems erfordert aber bisweilen sehr komplexe Datenstrukturen. Bei der Verarbeitung von Rechteckmengen ist der Segmentbaum häufig ein gutes Werkzeug.
2. Divide-and-Conquer reduziert ein zweidimensionales Mengenproblem auf ein eindimensionales Mengenproblem. Die Reduktion erscheint komplexer als bei Plane-Sweep; dafür kommt aber Divide-and-Conquer oft mit einfachen Listenstrukturen aus, was auch externe Lösungen ermöglicht (im Sinne von Kapitel 9).

Die beiden Arten der Reduktion gelten übrigens auch für  $k$  anstelle von zwei Dimensionen.

### 8.3 Suchen auf Mengen orthogonaler Objekte

Ein k-dimensionales orthogonales geometrisches Objekt ist in jeder Dimension durch ein Intervall oder eine Koordinate (eindimensionaler Punkt) beschrieben. Es gibt nun einen “Baukasten” eindimensionaler Baumstrukturen, aus denen man jeweils Baumhierarchien zur Lösung spezieller k-dimensionaler Suchprobleme zusammensetzen kann. Eine solche Hierarchie besitzt dann  $k$  “Schichten”; auf jeder Schicht kann eine andere eindimensionale Struktur eingesetzt werden. Der Baukasten enthält die in Tabelle 8.1 aufgeführten Strukturen, von denen wir zwei bereits kennen.

Struktur	Dargestellte Objekte	Operationen
Segment-Baum	Intervall-Menge	Auffinden aller Intervalle, die eine Query-Koordinate enthalten
Intervall-Baum	Intervall-Menge	Auffinden aller Intervalle, die eine Query-Koordinate enthalten Auffinden aller Intervalle, die ein Query-Intervall schneiden
Range-Baum	Koordinaten-Menge	Auffinden aller Koordinaten, die in einem Query-Intervall liegen
Binärer Suchbaum (wie in 9.1.1)	Koordinaten-Menge	Auffinden aller Koordinaten, die in einem Query-Intervall liegen

Tabelle 8.1: “Baukasten” von Baumstrukturen für Suchprobleme

Offenbar erfüllen der Segment-Baum und der Intervall-Baum bzw. der Range-Baum und der binäre Suchbaum jeweils die gleiche Funktion. Wozu werden also jeweils beide Strukturen benötigt? Das liegt daran, dass man in den höheren Schichten einer Baumhierarchie nur den Segment-Baum und den Range-Baum einsetzen kann. Andererseits haben die beiden anderen Strukturen jeweils geringeren Speicherplatzbedarf, so dass man sie in der untersten Schicht bevorzugt.

Den Segment-Baum und die Variante eines binären Suchbaums haben wir bereits in Abschnitt 8.1 kennengelernt. Bevor wir uns Baumhierarchien zuwenden, betrachten wir die anderen beiden Strukturen. Sie werden, ebenso wie der Segment-Baum, gelegentlich als *semidynamische* Strukturen bezeichnet. Dem liegt folgende Unterscheidung zugrunde. Eine Datenstruktur heißt *statisch*, wenn sie nur Suchen, aber keine Updates (Einfügen, Löschen von Objekten) erlaubt. Eine solche Datenstruktur muss also zunächst zur Darstellung einer gegebenen Objektmenge vollständig konstruiert werden; danach können nur noch Suchen durchgeführt werden. Eine Datenstruktur heißt *dynamisch*, wenn sie auch Updates unterstützt. Es gibt nun noch einen Fall dazwischen: Eine Datenstruktur heißt *semidynamisch*, wenn sie nur das Einfügen und Löschen von Objekten aus einem endlichen (und relativ kleinen) Wertebereich, einem “Universum” potentiell vorhandener Objekte, gestattet. So können z. B. in den Segment-Baum nur

Intervalle über einem Raster  $\{x_0, \dots, x_N\}$  eingefügt werden. Bei einer semidynamischen Struktur hat die Größe des Universums oder Rasters gewöhnlich neben der Größe der dargestellten Objektmenge Einfluss auf die Kostenmaße, also Suchzeit, Update-Zeit und Platzbedarf. Man könnte z. B. die Bitvektor-Darstellung für Mengen aus Abschnitt 4.1 als semidynamisch klassifizieren.

### Der Range-Baum

Der Range-Baum ist ähnlich wie der Segment-Baum eine semidynamische Datenstruktur; er erlaubt die Darstellung einer Menge von Koordinaten. Der leere Baum wird über einem Raster  $X = \{x_1, \dots, x_N\}$  konstruiert, danach kann eine Objektmenge  $S \subseteq X$  repräsentiert werden. Ein *Range-Baum über  $X$*  ist ein binärer Suchbaum minimaler Höhe, dessen Blätter die Koordinaten in  $X$  enthalten. Jedem Knoten  $p$  des Baumes ist eine Liste von Koordinaten  $SUB(p)$  zugeordnet.  $SUB(p)$  enthält alle in dem Teilbaum mit der Wurzel  $p$  eingetragenen Koordinaten. Im leeren Baum sind alle Listen  $SUB(p)$  leer; eine Koordinate  $x_i$  wird in den Range-Baum eingefügt, indem sie in die Listen  $SUB(p)$  aller Knoten auf dem Pfad zu  $x_i$  eingetragen wird. Um effizientes Entfernen zu unterstützen, ist die gleiche Technik wie beim Segment-Baum einzusetzen (Aufgabe 8.2).

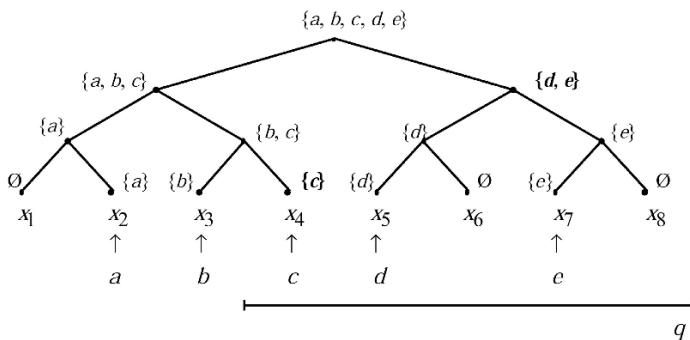


Abbildung 8.39: Range-Baum

Um alle Koordinaten in einem Query-Intervall  $q$  aufzufinden, sucht man im Range-Baum die Wurzeln aller maximal großen Teilbäume auf, die ganz in  $q$  enthalten sind, und gibt deren Listen  $SUB(p)$  als Antwort aus (diese Listen sind in Abbildung 8.39 fett gedruckt). Übrigens sind die Knoten, deren Listen auszugeben sind, genau die Knoten in  $CN(q)$ , das heißt, die Knoten, die  $q$  markieren würde, falls jedem Blatt ein Fragmentintervall zugeordnet und die Struktur dadurch als Segment-Baum definiert würde.

Da jede abgespeicherte Koordinate  $O(\log N)$  Einträge erzeugt, benötigt der Range-Baum  $O(N + n \log N)$  Speicherplatz. Indem man beim Eintragen von Koordinaten nur die Teile des Baumes aufbaut, in die tatsächlich Koordinaten eingetragen werden, lässt sich der Platzbedarf auf  $O(n \log N)$  reduzieren ( $n$  ist die Anzahl gespeicherter Koordinaten). Die Update-Zeit ist  $O(\log N)$ , die Suchzeit  $O(\log N + t)$  (wobei  $t$  die Anzahl gefundener Koordinaten angibt).

**Selbsttestaufgabe 8.6:** Modifizieren Sie den Range-Baum so, dass man effizient die Anzahl gespeicherter Koordinaten in einem Query-Intervall ermitteln kann. Wie groß ist dann der Platzbedarf?  $\square$

Die eigentlich interessante Eigenschaft des Range-Baumes ist die, dass er eine Suche in einem mehrdimensionalen Raum um eine Dimension reduziert. Wenn nämlich das Objekt, mit dem gesucht wird, in einer bestimmten Dimension  $i$  ein Intervall ist und die Objekte, auf denen gesucht wird, in der gleichen Dimension  $i$  durch eine Koordinate beschrieben sind, so reduziert der Range-Baum das Problem auf eine Suche auf  $O(\log N)$  Substrukturen. Wenn z. B. in zwei Dimensionen mit einem Rechteck auf einer Punktmenge (nach eingeschlossenen Punkten) gesucht wird, so erlaubt ein Range-Baum über x-Koordinaten die Reduktion auf eine Suche mit einem y-Intervall auf  $O(\log N)$  Substrukturen, die jeweils y-Koordinaten darstellen. Dies ist gerade die Idee der Baumhierarchien, die wir am Ende des Abschnitts 8.3 besprechen.

### Der Intervall-Baum

Auch der *Intervall-Baum* ist eine semidynamische Datenstruktur und wird über einem festen Raster  $X = \{x_0, \dots, x_N\}$  konstruiert. Es können dann Intervalle über  $X$  dargestellt werden. Für gegebenes  $X$  definieren wir eine zusätzliche Menge von Koordinaten

$$X' = \{x'_1, \dots, x'_N\} \text{ mit } x'_i = \frac{x_{i-1} + x_i}{2}$$

Die  $x'_i$  liegen also jeweils in der Mitte der durch  $[x_{i-1}, x_i]$  definierten Fragmentintervalle. Wir konstruieren nun einen binären Suchbaum minimaler Höhe über  $X'$ , das heißt, die Elemente von  $X'$  sind in inneren Knoten und Blättern dieses Baumes gespeichert. Dieser Baum dient als Gerüst, in das Intervalle über  $X$  eingetragen werden können. Dazu wird jedem Knoten  $p$  eine Intervallmenge  $STAB(p)$ <sup>5</sup> zugeordnet. Ein Intervall  $i$  wird eingetragen, indem es der Menge  $STAB$  eines bestimmten Knotens hinzugefügt wird. Dies geschieht nach folgendem Algorithmus:

---

5. engl. stab = durchstechen, aufspießen

```

insert ( $i, p$ )
{ $i = [i.bottom, i.top]$  ein Intervall,  $p$  ein Knoten}

if  $i$  ist rechts von  $p.x$  ( $p.x < i.bottom$ )
  then  $\text{insert} (i, p.right)$ 
elsif  $i$  ist links von  $p.x$  ( $p.x > i.top$ )
  then  $\text{insert} (i, p.left)$ 
else { $i$  enthält  $p.x$ }
   $STAB(p) := STAB(p) \cup \{i\}$ 
end if.

```

Das Intervall  $i$  wird also der Menge  $STAB$  des *ersten* Knotens an einem Pfad durch den Baum hinzugefügt, der  $i$  “aufspießt”. Die Intervallmenge  $STAB(p)$  hat eine interessante Eigenschaft: Für jedes Intervall in  $STAB(p)$  liegt ein Endpunkt links von  $p.x$  und ein Endpunkt rechts von  $p.x$  (Abbildung 8.40). Diese Eigenschaft von  $STAB(p)$  macht es leicht, alle Intervalle in  $STAB(p)$  zu finden, die eine Query-Koordinate  $q$  enthalten. Sei z. B.  $q < p.x$ . Dann gilt: Jedes Intervall in  $STAB(p)$ , dessen linker Endpunkt links von  $q$  liegt, enthält  $q$ . Diese Beobachtung legt folgende Strategie für das Durchsuchen von  $STAB(p)$  nahe: Stelle die linken Endpunkte aller Intervalle in  $STAB(p)$  als  $x$ -sortierte Liste dar (wie in Abbildung 8.40 mit den Pfeilen angedeutet). Durchlaufe diese Liste in aufsteigender  $x$ -Reihenfolge. Für jeden passierten Intervall-Endpunkt gib sein Intervall (als  $q$  enthaltend) aus. Sobald der erste Endpunkt rechts von  $q$  angetroffen wird, beende den Durchlauf. Kein weiteres Intervall in  $STAB(p)$  kann  $q$  enthalten. Diese Suche erfordert bei  $t$  gefundenen Intervallen nur  $O(1+t)$  Zeit. Der Fall  $q > p.x$  ist analog; die Liste wird dann vom rechten Ende her durchlaufen.

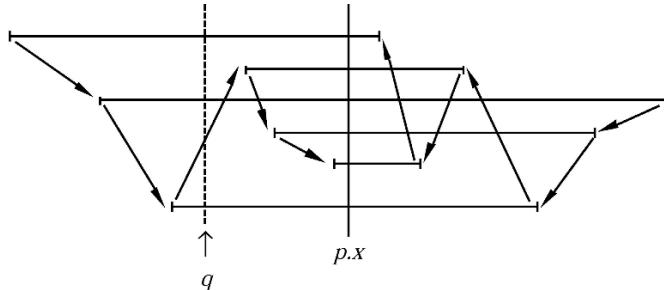


Abbildung 8.40: Intervalle in  $STAB(p)$

Falls alle Mengen  $STAB(p)$  in dieser Form als Listen organisiert sind, so kann eine Punkteinschlussuche auf dem Intervall-Baum folgendermaßen erfolgen:

```

query ( $q, p$ )
{ $q$  eine Koordinate,  $p$  ein Knoten}

if  $q < p.x$ 
then durchlaufe  $STAB(p)$  vom linken Ende her und gib gefundene Intervalle aus;
    if  $p$  ist kein Blatt then query ( $q, p.left$ ) end if
    else { $q \geq p.x$ }
        durchlaufe  $STAB(p)$  vom rechten Ende her und gib gefundene Intervalle aus;
        if  $p$  ist kein Blatt then query ( $q, p.right$ ) end if
    end if.

```

Der Suchalgorithmus folgt einem Pfad durch den Baum, an jedem Knoten entsteht für den Listendurchlauf zusätzlich ein Zeitaufwand von  $O(1 + t_i)$ . Die gesamte Suchzeit ist offensichtlich  $O(\log N + t)$ , für  $t = \sum t_i$ .

Es fehlt noch die Möglichkeit, Intervalle effizient in eine Liste  $STAB(p)$  einzufügen oder aus ihr zu entfernen. Dazu wird jeder Liste ein eigener binärer Suchbaum “aufgesetzt”, der die Listenelemente als Blätter hat. Die Koordinaten der inneren Knoten dieses Baumes können beliebig gewählt werden (aber so, dass die Suchbaum-Eigenschaft erfüllt ist). Dann ist das Einfügen und Entfernen eines Intervalls in  $STAB(p)$  in  $O(\log n)$  Zeit möglich. Trotz des Suchbaumes gibt es weiterhin Zeiger auf Anfang und Ende der Blattlisten, um effizientes Suchen zu gewährleisten. Insgesamt hat der Intervall-Baum dann z. B. die in Abbildung 8.41 gezeigte Gestalt.

Der Platzbedarf dieser Strukturen ist  $O(N + n)$ , da jedes eingetragene Intervall nur konstanten Speicherplatz benötigt. Insgesamt hat der Intervall-Baum damit folgende Eigenschaften (unter der Annahme, dass  $N = \Theta(n)$ ):

Eine Menge  $I$  von  $n$  Intervallen über  $X = \{x_1, \dots, x_N\}$  kann in einem Intervall-Baum in  $O(n)$  Speicherplatz dargestellt werden, so dass das Einfügen und Entfernen eines Intervalls in  $O(\log n)$  Zeit möglich ist. Die  $t$  Intervalle in  $I$ , die eine Query-Koordinate enthalten oder die ein Query-Intervall schneiden, können in  $O(\log n + t)$  Zeit ermittelt werden.

Um die Zeitschranke für Intervallschnitt-Suche zu garantieren, muss man allerdings noch eine Modifikation vornehmen, die hier nur angedeutet werden soll: Der Top-Level-Suchbaum über  $X'$  muss so “zusammengezogen” werden, dass er nur Knoten  $p$  mit nichtleeren Intervallisten  $STAB(p)$  enthält. Der Aufbau der Rahmenstruktur muss dazu in den Einfügealgorithmus integriert werden (ebenso der Abbau beim Löschen von Intervallen). Wenn man anstelle eines Rasters  $X = \{x_0, \dots, x_N\}$  ein Raster  $X = \{0, \dots, N\}$  verwendet, ist das ohne große Schwierigkeiten möglich, da dann die prinzipielle Struktur des Top-Level-Baumes a priori bekannt ist. Da jedes Raster der Größe  $N$  bijektiv auf das

Raster  $\{0, \dots, N\}$  abzubilden ist, können alle Intervall- und Anfrage-Koordinaten entsprechend umgerechnet werden.

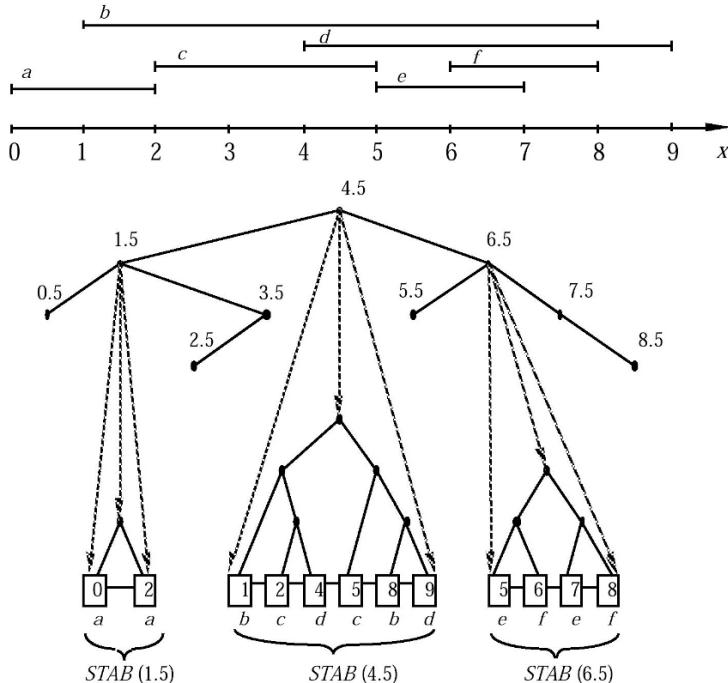


Abbildung 8.41: Intervall-Baum

Durch das Zusammenziehen des Top-Level-Suchbaumes erreicht man übrigens auch, dass der Platzbedarf  $O(n)$  wird, selbst wenn  $n$  und  $N$  unabhängig voneinander sind. Wir erhalten dann folgende Eigenschaften:

- Platzbedarf  $O(n)$
- Update-Zeit  $O(\log n + \log N)$
- Suchzeit  $O(\log N + t)$

für beide Sucharten.

Durch Einsatz des Intervall-Baums im Plane-Sweep-Algorithmus für das Rechteckschnitt-Problem (Intervallschnittsuche für jede linke Rechteckkante auf den  $y$ -Intervallen aktuell vorhandener Rechtecke) erhält man nun eine auch speicherplatzoptimale Lösung

des Rechteckschnittproblems, das heißtt, eine Lösung mit  $O(n \log n + k)$  Zeit und  $O(n)$  Platzbedarf.

### Baumhierarchien

Aus den vier beschriebenen Komponentenstrukturen kann man Hierarchien bilden, indem man jeweils die Elemente einer Knotenliste eines Segment-Baums oder Range-Baums als Baumstruktur der nächstniedrigeren Schicht organisiert. Wir erklären die Technik an zwei Beispielen.

Ein SI-Baum (Segment-Intervall-Baum) ist ein Segment-Baum, in dem jede Knotenliste als Intervall-Baum organisiert ist. Es kann z. B. der Segment-Baum über x-Koordinaten, der Intervall-Baum über y-Koordinaten angelegt sein. Da in beiden Dimensionen Intervalle dargestellt werden, speichert ein SI-Baum Rechtecke. Ein Rechteck wird in den Segment-Baum anhand seines x-Intervalls eingetragen; es erzeugt dort Einträge in  $O(\log n)$  Knotenlisten (sei  $n = \Theta(N)$ , um die beiden nicht auseinanderhalten zu müssen).

Eine solche Struktur ist in Abbildung 8.42 dargestellt. Ein SI-Baum beschreibt nun insgesamt einen rechteckigen Teilraum  $T$  der Ebene, in dem zu speichernde Rechtecke liegen können;  $T$  ist unten in Abbildung 8.42 gezeigt. Das Gebiet  $T$  wird durch den Segment-Baum in x-Richtung hierarchisch zerlegt; dem Wurzelknoten ist der gesamte Raum  $T$  zugeordnet (hier allerdings nicht gezeigt). Ein Rechteck wird anhand seiner Zerlegung in x-Intervalle auf die Knoten des Segment-Baums verteilt.

Jede Knotenliste enthält eine Menge von Rechtecken, die *alle das x-Intervall des Knotens ganz überdecken* und für die folglich nur die y-Intervalle noch interessant sind. Diese y-Intervall-Menge kann man als Intervall-Baum organisieren.

Die Struktur unterstützt das Auffinden aller Rechtecke, die einen Query-Punkt  $q = (q_x, q_y)$  enthalten: Eine Suche mit  $q_x$  identifiziert  $O(\log n)$  Knoten im Segment-Baum, wie in Abbildung 8.42 gezeigt. In jedem zugeordneten Intervall-Baum werden nun in  $O(\log n + t_i)$  Zeit die Rechtecke gefunden, deren y-Intervall  $q_y$  enthält. Die gesamte Suchzeit ist also  $O(\log^2 n + t)$ .

Da jedes Rechteck  $O(\log n)$  Einträge im Segment-Baum erzeugt, entstehen insgesamt  $O(n \log n)$  Einträge in Intervall-Bäumen. Diese werden auch in linearem Platz, also in  $O(n \log n)$  Speicherplatz dargestellt. Der Platzbedarf ist daher insgesamt  $O(n \log n)$ . Ganz allgemein kommt in diesen Baumhierarchien für jede Dimension ein Faktor  $\log n$  bei Platzbedarf, Suchzeit und Updatezeit hinzu.

Als weiteres Beispiel betrachten wir einen RI-Baum, also einen Range-Intervall-Baum. Der Range-Baum sei über x-Koordinaten, die Intervall-Bäume über y-Koordinaten ange-

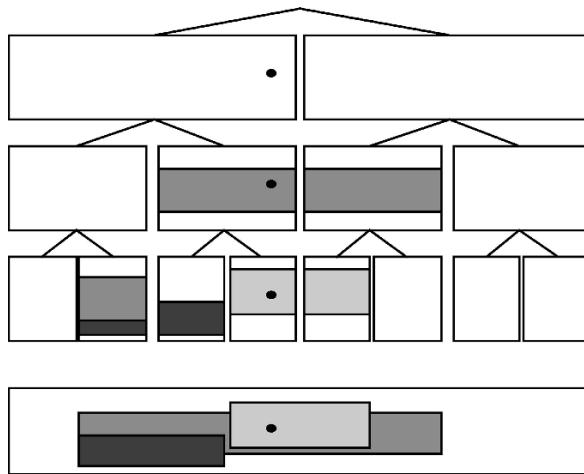


Abbildung 8.42: Segment-Intervall-Baum

legt; der RI-Baum speichert dann vertikale Segmente. Analog zum Segment-Baum kann man einen *leeren* Range-Baum so auffassen, dass jedem Knoten eine Menge von Koordinaten zugeordnet ist, nennen wir sie *Raster*, auf denen Objekte eingetragen werden können. Ein Blatt hat eine einzige feste Koordinate, das Raster eines inneren Knotens ergibt sich als Vereinigung der Raster seiner Söhne. Im zweidimensionalen Fall kann man daher jedem Knoten eines leeren RI-Baums ein Raster vertikaler Liniensegmente zuordnen, auf denen zu speichernde vertikale Segmente eingetragen werden können; diese Sicht ist in Abbildung 8.43 illustriert.

Ein RI-Baum unterstützt die Suche mit einem horizontalen Segment  $h$  nach geschnittenen vertikalen Segmenten. Die Suche auf dem Range-Baum mit dem x-Intervall von  $h$  identifiziert  $O(\log n)$  Knoten, die von  $h$  in x-Richtung vollständig überdeckt werden. Jeder Knoten stellt eine Menge vertikaler Segmente anhand ihrer y-Intervalle in einem Intervall-Baum dar, auf dem mit der y-Koordinate von  $h$  gesucht werden kann. Auch hier ist die Suchzeit insgesamt  $O(\log^2 n + t)$ , der Platzbedarf  $O(n \log n)$ .

Man könnte zur Lösung des gleichen Suchproblems auch einen SB-Baum, also einen Segment-Binär-Baum einsetzen, mit den gleichen Komplexitätsmaßen. Anstelle des SI-Baumes könnte man prinzipiell auch einen SS-Baum benutzen. Dies sollte aber wegen des höheren Platzbedarfs des Segment-Baums vermieden werden, denn ein SS-Baum braucht  $O(n \log^2 n)$  Speicherplatz. Wie schon erwähnt, sollten auf den untersten Ebenen einer Baumhierarchie stets Intervall- oder Binär-Suchbäume eingesetzt werden.

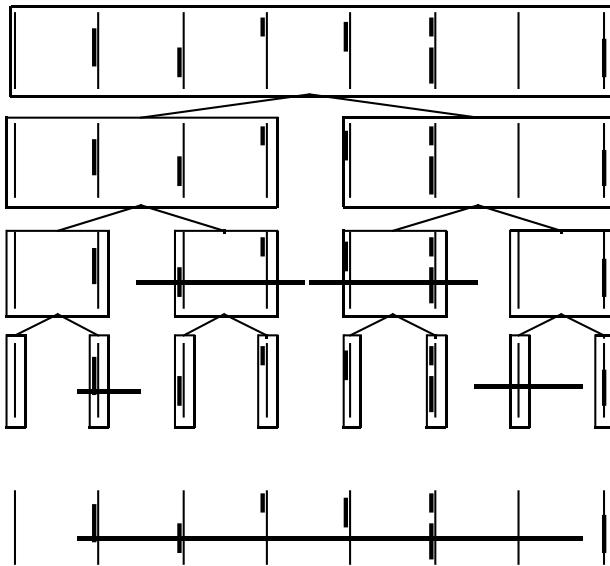


Abbildung 8.43: Range-Intervall-Baum

Es gibt trickreiche Techniken, um in solchen zweistufigen Hierarchien die Suchzeit auf  $O(\log n + t)$  zu reduzieren; die entstehenden Strukturen sind dann aber statisch, erlauben also keine Updates mehr mit vernünftigem Aufwand.

Hierarchien können durchaus Objekte in 3 oder mehr Dimensionen darstellen: Zum Beispiel ein RRB-Baum (Range-Range-Binär-Baum) speichert eine Punktmenge in drei Dimensionen in  $O(n \log^2 n)$  Speicherplatz. Die in einem Query-Quader enthaltenen Punkte können in  $O(\log^3 n + t)$  Zeit ermittelt werden.

**Selbsttestaufgabe 8.7:** Geben Sie Datenstrukturen an zur Lösung folgender Suchprobleme:

- Gegeben sei eine Menge von Quadern im dreidimensionalen Raum. Finde alle Quadern, die ein zur  $x$ -Achse paralleles Query-Liniensegment schneiden.
- Gegeben eine Menge von Punkten in der Ebene. Finde alle Punkte, die rechts von einem vertikalen Query-Segment liegen.

Wie sehen jeweils Suchzeit und Platzbedarf Ihrer Struktur aus? □

## 8.4 Plane-Sweep-Algorithmen für beliebig orientierte Objekte

Zum Schluss des Kapitels werfen wir noch einen Blick auf die Welt beliebig orientierter Objekte. Der folgende Algorithmus für das *allgemeine Segmentschnitt-Problem* ist Grundlage für viele Plane-Sweep-Algorithmen, die weniger elementare Probleme lösen.

Für eine Menge beliebig orientierter Segmente<sup>6</sup> in der Ebene sollen wiederum alle Paare sich schneidender Segmente ermittelt werden. Was geschieht während eines Plane-Sweeps durch eine solche Objektmenge?

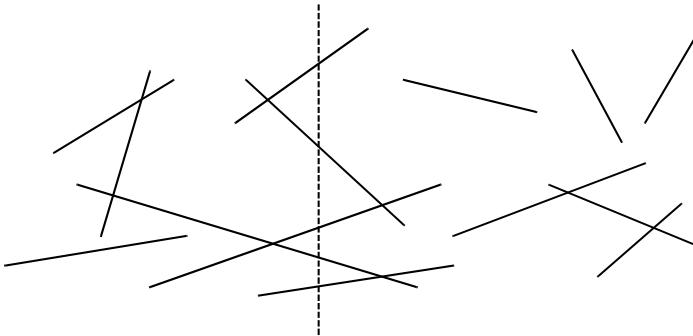


Abbildung 8.44: Beliebig orientierte Segmente

Wir machen folgende Beobachtungen:

- Zu jedem Zeitpunkt gibt es eine Menge von Segmenten, die gerade die Sweep-line schneiden, nennen wir sie *aktive* Segmente. Auf der Menge aktiver Segmente existiert eine *Ordnung*, die durch die y-Reihenfolge der Schnittpunkte mit der Sweep-line definiert ist.
- Die y-geordnete Sequenz aktiver Segmente verändert sich während des Sweeps nur bei drei Arten von Ereignissen, nämlich beim Passieren
  - des linken Endpunktes eines Segments,
  - des rechten Endpunktes eines Segments,
  - des Schnittpunktes zweier Segmente.

Um die Sequenz korrekt aufrecht zu erhalten, ist in den ersten beiden Fällen ein Element in die Sequenz einzufügen bzw. aus ihr zu entfernen, im letzten Fall sind zwei aufeinanderfolgende Elemente zu vertauschen. Sämtliche Ereignisse der drei

6. Der Einfachheit halber nehmen wir an, dass keine vertikalen Segmente in der Menge vorkommen. Es ist nicht allzu schwer, sich eine Sonderbehandlung für vertikale Segmente zu überlegen.

Arten zusammen bilden die Menge der *Haltepunkte* oder *Stationen* des Plane-Sweeps.

Die Haltepunkte erster und zweiter Art lassen sich leicht ermitteln, indem man die Endpunkte aller Segmente nach  $x$ -Koordinaten sortiert. Schwieriger ist es mit den Schnittpunkten. Diese soll der Algorithmus ja gerade berechnen! Es hilft folgende Beobachtung:

Zwei Segmente können sich nur dann schneiden, wenn sie vorher einmal in der Sequenz aktiver Segmente benachbart sind.

Diese Beobachtung führt zu folgendem Algorithmus:

1. Bilde eine Warteschlange (Queue)  $Q$  aller abzuarbeitenden Haltepunkte des Sweeps und initialisiere sie mit der nach  $x$  geordneten Menge aller Endpunkte von Segmenten. Sei  $S$  die Sequenz aktiver Segmente,  $S := \emptyset$ .
2. Entnimm  $Q$  jeweils das erste Element  $q$  und bearbeite es. Es gibt folgende Fälle:
  - a.  $q$  ist linker Endpunkt eines Segmentes  $t$ . Füge  $t$  gemäß seiner  $y$ -Koordinate in  $S$  ein. Seien  $s$  und  $u$  die benachbarten Segmente direkt unter bzw. über  $t$ . Überprüfe, ob  $t$  eines der Segmente  $s$  oder  $u$  schneidet. Füge jeden gefundenen Schnittpunkt gemäß seiner  $x$ -Koordinate in die Warteschlange  $Q$  ein. Dies ist in Abbildung 8.45 dargestellt.<sup>7</sup>

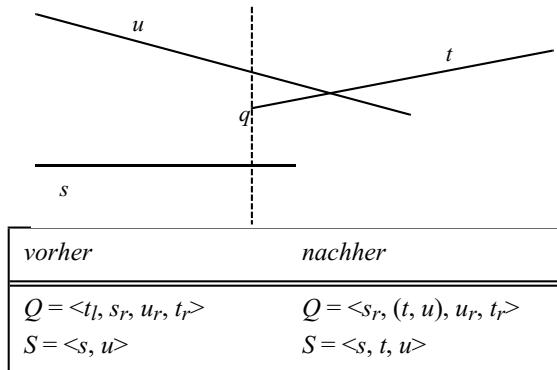


Abbildung 8.45:  $q$  ist linker Endpunkt von  $t$

---

7. Wir notieren Queue-Einträge wie folgt: Es bezeichne  $t_l$  den linken und  $t_r$  den rechten Endpunkt eines Segmentes  $t$  und ein Paar  $(t, u)$  den Schnitt zweier Segmente  $t$  und  $u$ .

- b.  $q$  ist rechter Endpunkt eines Segmentes  $t$ . Entferne  $t$  aus  $S$ . Überprüfe, ob die Segmente  $s$  und  $u$ , die unterhalb bzw. oberhalb von  $t$  lagen, sich schneiden. Trage jeden gefundenen Schnittpunkt in  $Q$  ein, falls er dort noch nicht vorhanden ist (Abbildung 8.46).

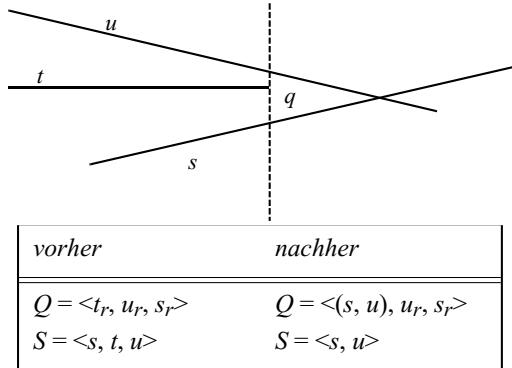


Abbildung 8.46:  $q$  ist rechter Endpunkt von  $t$

- c.  $q$  ist Schnittpunkt zweier Segmente  $t$  und  $t'$ . Gib das Paar  $(t, t')$  aus und vertausche  $t$  und  $t'$  in  $S$ . Seien wieder  $s$  und  $u$  die Nachbarssegmente unterhalb und oberhalb von  $t, t'$ . Überprüfe, ob  $s$  und  $t'$  sich schneiden und ob  $t$  und  $u$  sich schneiden. Trage jeden gefundenen Schnittpunkt in  $Q$  ein, falls er dort noch nicht vorhanden ist (Abbildung 8.47).

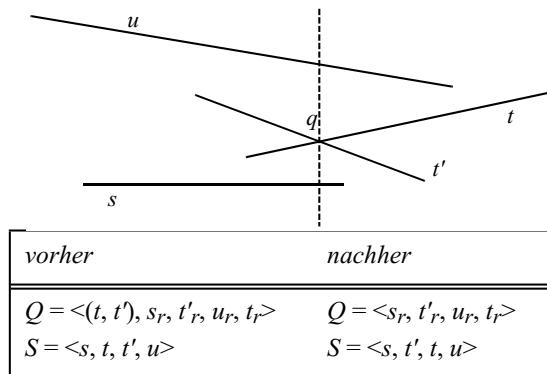


Abbildung 8.47:  $q$  ist Schnittpunkt

Mit diesem Algorithmus wird jeder Schnitt zweier Segmente entdeckt, weil jede Situation, in der zwei Segmente benachbart werden, überprüft wird. Wie steht es mit der Lauf-

zeit? Die Sweep-line-Status-Struktur  $S$  kann als balancierter Baum implementiert werden. Dazu werden in jedem Knoten des Baumes die Koeffizienten der Geradengleichung des dargestellten Segments abgespeichert. Dadurch kann in jedem Knoten an einem Haltepunkt  $x_0$  der Sweep-line die aktuell gültige  $y$ -Koordinate des Knotens ermittelt werden. Die Datenstruktur  $Q$  muss offensichtlich die Operationen

- Entnahme des Elementes mit minimaler x-Koordinate
- Einfügen eines Elementes mit beliebiger x-Koordinate

unterstützen. Das sind gerade die Operationen *deletemin* und *insert* des Datentyps *priority queue* (Abschnitt 4.3), der z. B. als Heap implementiert werden kann. Damit lässt sich jede Operation auf  $S$  oder  $Q$  in  $O(\log n)$  Zeit durchführen, für eine Segmentmenge mit  $n$  Elementen und  $k$  sich schneidenden Paaren ( $Q$  kann zwar  $\Theta(n^2)$  Elemente enthalten, aber es gilt  $O(\log n^2) = O(\log n)$ ). Der Sweep hat  $\Theta(n + k)$  Haltepunkte, daher erhalten wir folgendes Ergebnis:

Für eine Menge beliebig orientierter Segmente in der Ebene kann das Segmentschnitt-Problem mittels Plane-Sweep in  $O((n + k) \log n)$  Zeit und  $O(n + k)$  Speicherplatz gelöst werden, wobei  $n$  die Anzahl der Segmente und  $k$  die Anzahl sich schneidender Paare ist.

Den ursprünglich in dieser Form vorgeschlagenen Algorithmus kann man so modifizieren, dass der Platzbedarf auf  $O(n)$  absinkt. Die Idee dazu besteht darin, für jedes Segment jeweils nur den *nächsten* bereits bekannten Schnittpunkt in  $Q$  aufzubewahren.

Dieser Segmentschnitt-Algorithmus dient als Grundlage für viele weitere Algorithmen, die Eigenschaften von Polygonmengen in der Ebene berechnen. In diesen Algorithmen kommt es dann nicht auf die Ausgabe der Segmentschnitte an, sondern auf die korrekte Aufrechterhaltung des Zustandes der Sweep-line.

## 8.5 Weitere Aufgaben

**Aufgabe 8.8:** Geben Sie zum Segment-Baum Algorithmen

- (a) *insert* ( $i, p$ ) und
- (b) *query* ( $c, p$ )

an. Algorithmus *insert* fügt das Intervall  $i$  in den Baum mit dem Wurzelknoten  $p$  ein, *query* liefert alle Intervalle aus dem Baum mit dem Wurzelknoten  $p$ , die die Koordinate  $c$  enthalten.

**Aufgabe 8.9:** Wir betrachten einen vollständigen Segment-Baum der Höhe  $h$  mit  $N = 2^h$  Blättern.

- (a) Zeigen Sie, dass ein Intervall in einem solchen Baum maximal  $O(h)$  Einträge erzeugen kann.
- (b) Was ist die genaue Anzahl der Einträge im schlimmsten Fall?

**Aufgabe 8.10:** Wie in der vorigen Aufgabe betrachten wir einen vollständigen Segment-Baum der Höhe  $h$  mit  $N = 2^h$  Blättern. Was ist die *erwartete* Anzahl von Einträgen, die ein Intervall der Länge  $k$  erzeugt?

Dieser Baum sei über einem Raster  $X = \{0, \dots, N\}$  definiert. Um Effekte nicht betrachten zu müssen, die am Rand des Rasters entstehen, nehmen wir an, dass die Intervalllänge  $k \leq N/2$  sei und dass die linken Intervallenden gleichverteilt seien über den Koordinaten  $\{0, \dots, (N/2 - 1)\}$ .

**Aufgabe 8.11:** Gegeben sei ein Raster von Koordinaten in der Form

```
var raster: array [0 .. N] of integer
```

Schreiben Sie ein Programm, das einen leeren Segment-Baum über diesem Raster in  $O(N)$  Zeit konstruiert. Für die Darstellung des Segment-Baums können Sie den Knotentyp aus Abschnitt 8.1.2 verwenden und die Knoten explizit mit Zeigern verbinden. Es gibt aber auch die Möglichkeit, einen Array über diesem Knotentyp (ohne die Zeiger) anzulegen und die Verzeigerung implizit vorzunehmen (Abschnitt 3.5, Implementierung (b) bzw. die übliche Heap-Darstellung).

**Aufgabe 8.12:** Formulieren Sie Methoden für das Einfügen und Entfernen in einem gemäß Abschnitt 8.1.3 modifizierten Segment-Baum.

**Aufgabe 8.13:** Für eine Rechteckmenge  $R$  sei die Kontur in Form der Menge aller horizontalen Kontursegmente gegeben. Entwerfen Sie einen Algorithmus, der daraus die Kontur als Menge von Konturzyklen berechnet. Für  $n$  Rechtecke und  $p$  Konturstücke soll Ihr Algorithmus nicht mehr als  $O(n + p)$  Zeit benötigen.

Es wird vorausgesetzt, dass für die Rechteckmenge  $R$  alle vorkommenden x- und y-Koordinaten umgerechnet seien in zwei Raster  $X = \{0, \dots, N\}$  und  $Y = \{0, \dots, M\}$  mit  $N, M = O(n)$ . Das heißt, die Koordinaten der horizontalen Kontursegmente sind in dieser Form gegeben. Jeder Konturzyklus soll als verkettete Liste dargestellt werden, deren Elemente abwechselnd ein horizontales oder ein vertikales Segment darstellen.

*Hinweis:* Es wird nötig sein, die  $p$  Konturstücke in  $O(n + p)$  Zeit zu sortieren. Erinnern Sie sich daran, wie man das macht!

**Aufgabe 8.14:** Sei  $R$  eine Menge von Rechtecken,  $p$  ein Punkt in der Ebene. Die *Überdeckungszahl* von  $p$ ,  $cover(p)$ , sei definiert als die Anzahl der Rechtecke in  $R$ , die  $p$  ent-

halten. Die *Höhe* einer Menge von Rechtecken  $R$  ist die maximal auftretende Überdeckungszahl an irgendeinem Punkt der Ebene, also

$$\text{height}(R) := \max \{\text{cover}(p) \mid p \in R^2\}$$

Entwickeln Sie einen Divide-and-Conquer-Algorithmus zur Berechnung der Höhe einer Menge von Rechtecken.

**Aufgabe 8.15:** Geben Sie zum Segment-Baum Algorithmen

- (a) *insert* ( $i, p$ ) und
- (b) *query* ( $c, p$ )

an. Algorithmus *insert* fügt das Intervall  $i$  in den Baum mit dem Wurzelknoten  $p$  ein, *query* liefert alle Intervalle aus dem Baum mit dem Wurzelknoten  $p$ , die die Koordinate  $c$  enthalten.

**Aufgabe 8.16:** Wir betrachten einen vollständigen Segment-Baum der Höhe  $h$  mit  $N = 2^h$  Blättern.

- (a) Zeigen Sie, dass ein Intervall in einem solchen Baum maximal  $O(h)$  Einträge erzeugen kann.
- (b) Was ist die genaue Anzahl der Einträge im schlimmsten Fall?

**Aufgabe 8.17:** Wie in der vorigen Aufgabe betrachten wir einen vollständigen Segment-Baum der Höhe  $h$  mit  $N = 2^h$  Blättern. Was ist die *erwartete* Anzahl von Einträgen, die ein Intervall der Länge  $k$  erzeugt?

Dieser Baum sei über einem Raster  $X = \{0, \dots, N\}$  definiert. Um Effekte nicht betrachten zu müssen, die am Rand des Rasters entstehen, nehmen wir an, dass die Intervalllänge  $k \leq N/2$  sei und dass die linken Intervallenden gleichverteilt seien über den Koordinaten  $\{0, \dots, (N/2 - 1)\}$ .

**Aufgabe 8.18:** Gegeben sei ein Raster von Koordinaten in der Form

```
var raster: array [0 .. N] of integer
```

Schreiben Sie ein Programm, das einen leeren Segment-Baum über diesem Raster in  $O(N)$  Zeit konstruiert. Für die Darstellung des Segment-Baums können Sie den Knotentyp aus Abschnitt 8.1.2 verwenden und die Knoten explizit mit Zeigern verbinden. Es gibt aber auch die Möglichkeit, einen Array über diesem Knotentyp (ohne die Zeiger) anzulegen und die Verzeigerung implizit vorzunehmen (Abschnitt 3.5, Implementierung (b) bzw. die übliche Heap-Darstellung).

**Aufgabe 8.19:** Formulieren Sie Methoden für das Einfügen und Entfernen in einem gemäß Abschnitt 8.1.3 modifizierten Segment-Baum.

**Aufgabe 8.20:** Für eine Rechteckmenge  $R$  sei die Kontur in Form der Menge aller horizontalen Kontursegmente gegeben. Entwerfen Sie einen Algorithmus, der daraus die Kontur als Menge von Konturzyklen berechnet. Für  $n$  Rechtecke und  $p$  Konturstücke soll Ihr Algorithmus nicht mehr als  $O(n + p)$  Zeit benötigen.

Es wird vorausgesetzt, dass für die Rechteckmenge  $R$  alle vorkommenden x- und y-Koordinaten umgerechnet seien in zwei Raster  $X = \{0, \dots, N\}$  und  $Y = \{0, \dots, M\}$  mit  $N, M = O(n)$ . Das heißt, die Koordinaten der horizontalen Kontursegmente sind in dieser Form gegeben. Jeder Konturzyklus soll als verkettete Liste dargestellt werden, deren Elemente abwechselnd ein horizontales oder ein vertikales Segment darstellen.

*Hinweis:* Es wird nötig sein, die  $p$  Konturstücke in  $O(n + p)$  Zeit zu sortieren. Erinnern Sie sich daran, wie man das macht!

**Aufgabe 8.21:** Sei  $R$  eine Menge von Rechtecken,  $p$  ein Punkt in der Ebene. Die *Überdeckungszahl* von  $p$ ,  $\text{cover}(p)$ , sei definiert als die Anzahl der Rechtecke in  $R$ , die  $p$  enthalten. Die *Höhe* einer Menge von Rechtecken  $R$  ist die maximal auftretende Überdeckungszahl an irgendeinem Punkt der Ebene, also

$$\text{height}(R) := \max \{\text{cover}(p) \mid p \in \mathbb{R}^2\}$$

Entwickeln Sie einen Divide-and-Conquer-Algorithmus zur Berechnung der Höhe einer Menge von Rechtecken.

## 8.6 Literaturhinweise

Das Gebiet der algorithmischen Geometrie in der heute geläufigen Bedeutung wurde durch Arbeiten von Shamos [1975], insbesondere seine Dissertation [Shamos 1978] ins Leben gerufen; vorher wurde der Begriff gelegentlich in anderem Sinn benutzt. Seitdem hat das Feld einen stürmischen Aufschwung genommen. Es ist klar, dass in diesem Kapitel nur ein winziger Ausschnitt der betrachteten Probleme behandelt werden konnte. Zudem haben wir uns noch auf den Bereich der orthogonalen Objekte konzentriert; in einer “balancierten” Einführung müßten etliche andere Teilgebiete wie z. B. das Suchen in Partitionen der Ebene oder höherdimensionaler Räume, Distanzprobleme und Voronoi-Diagramme, Schnittprobleme für beliebig orientierte Objekte in k-dimensionalen Räumen, die Konstruktion konvexer Hüllen und sicher noch weitere gleichgewichtig mitbehandelt werden. Andererseits ist der Fall der orthogonalen Objekte besonders

einfach und deshalb besonders intensiv erforscht worden und die eingeführten Techniken und Datenstrukturen sind grundlegend für das ganze Gebiet.

Einen kurzen Überblick zum Gebiet der algorithmischen Geometrie bietet [Lee 1996], ein ausführlicher Überblicksartikel ist [Lee 1995]. Eine Übersicht zu betrachteten Problemen und Techniken findet man auch in [Sack und Urrutia 1996]. Einführungen in die algorithmische Geometrie werden in den Büchern von Mehlhorn [1984c], Preparata und Shamos [1985], Edelsbrunner [1987], O'Rourke [1998], Laszlo [1996], Klein [2005] und de Berg et al. [2008] gegeben. Auch das Buch von Ottmann und Widmayer [2012] enthält ein recht umfangreiches Kapitel zu geometrischen Algorithmen.

Der Segmentschnitt-Algorithmus mit Plane-Sweep stammt von Bentley und Ottmann [1979]; eine gute Einführung in die Plane-Sweep-Technik findet sich auch bei Nievergelt und Preparata [1982]. Der Rechteckschnitt-Algorithmus mit Plane-Sweep und Segment-Baum stammt von Bentley und Wood [1980]. Der Segment-Baum selbst war von Bentley [1977] vorgeschlagen worden. Die Plane-Sweep-Lösung des Maßproblems entstammt ebenfalls [Bentley 1977]. Diese Lösung lässt sich übrigens leicht auf höhere Dimensionen verallgemeinern. Um etwa das Maß einer Menge von Quadern im dreidimensionalen Raum zu ermitteln, kann man einen Sweep (mit einer Ebene anstelle einer Sweep-line) durchführen; dieser zerlegt den Raum in eine Folge von  $O(n)$  "Scheiben"; für jede Scheibe ist das Maß einer Rechteckmenge zu bestimmen, sodass man insgesamt einen Algorithmus mit einer Laufzeit von  $O(n^2 \log n)$  erhält. Dies wird in [van Leeuwen und Wood 1981] durch Einsatz eines Quad-Tree im Sweep auf eine Laufzeit von  $O(n^2)$  verbessert.

Die Divide-and-Conquer-Technik für planare Objekte mit der Idee der getrennten Darstellung stammt von Güting und Wood [1984]; dort werden Lösungen für das Segmentschnitt-Problem, das Punkteinschluss-Problem und das Rechteckschnitt-Problem beschrieben. Güting und Schilling [1987] geben eine elegantere Lösung für das Rechteckschnitt-Problem, indem nicht mehr auf Punkteinschluss und Segmentschnitt reduziert, sondern direkt mit Rechtecken gearbeitet wird. Maßproblem und Konturproblem werden mit Divide-and-Conquer in [Güting 1984b] gelöst. Eine Übersicht zur Divide-and-Conquer-Technik bietet [Güting 1985]. Es gibt eine Methode, mit der sich die meisten derartigen DAC-Algorithmen in *externe* Formen überführen lassen, die dann ähnlich ablaufen wie externes Mischartieren [Güting und Schilling 1987]. Leider funktioniert das beim Konturproblem der trickreichen Verzeigerung wegen nicht.

Das Konturproblem war zuerst von Lipski und Preparata [1980] betrachtet worden; ein Plane-Sweep mit einem modifizierten Segment-Baum brachte eine nicht-optimale Lösung. Die erste zeitoptimale Lösung mit Plane-Sweep ist in [Güting 1984a] zu finden; dort wird ein "contracted segment tree" eingesetzt.

Range-Bäume stammen, wie schon Segment-Bäume, von Bentley [1979]. Der Intervallbaum wurde von Edelsbrunner [1980, 1983] entworfen; unabhängig davon entwickelte McCreight [1980] den “tile tree”, eine äquivalente, wenn auch anders dargestellte Struktur. Baumhierarchien bestehend nur aus Range-Bäumen wurden schon von Bentley [1979] betrachtet; Anwendungen für Rechteckschnittprobleme mit Segment-Range-Bäumen finden sich z. B. bei Six und Wood [1982]. Solche Baumhierarchien wurden systematisch von Edelsbrunner [1980] und Edelsbrunner und Maurer [1981] studiert.

Eine weitere grundlegende Datenstruktur, die wir hier aus Platzgründen nicht mehr dargestellt haben, ist der Prioritätssuchbaum von McCreight [1985]. Diese Struktur erlaubt die Darstellung einer Punktmenge in zwei Dimensionen und unterstützt “1 1/2-dimensionale Suchen”, nämlich Suchen mit einem Rechteck nach enthaltenen Punkten, wobei eine der Rechteckkanten auf einer Achse des Koordinatensystems liegen muss. Die  $t$  Punkte im Halbrechteck können in  $O(\log n + t)$  Zeit ermittelt werden; die Struktur erlaubt Updates in  $O(\log n)$  Zeit.

Die Plane-Sweep-Lösung des Segmentschnitt-Problems für beliebig orientierte Segmente stammt von Bentley und Ottmann [1979]. Chazelle [1986] fand Divide-and-Conquer-Lösungen, die zwar sehr komplex sind, aber die Laufzeit von  $O((n+k)\log n)$  auf  $O(n\log^2 n + k)$  oder auch auf  $O(n\log^2 n/\log \log n + k)$  drücken. Man beachte, dass  $k$ , also die Anzahl gefundener Segmentschnitte,  $\Theta(n^2)$  sein kann und damit die Laufzeit dominieren kann. Schließlich wurde von Chazelle und Edelsbrunner [1988] sogar eine optimale Lösung mit Laufzeit  $O(n\log n + k)$  entdeckt.

Bemerkenswert ist noch eine allgemeine Technik auf der Basis dieses Segmentschnitt-Algorithmus, mit der man in vielen Fällen Algorithmen für orthogonale Objekte, also z. B. für Rechtekmengen, in Algorithmen für beliebig orientierte Objekte überführen kann, das sogenannte *Zickzack-Paradigma* [Ottmann und Widmayer 1982]. Das Verfahren eignet sich für Plane-Sweep-Algorithmen, die semidynamische Datenstrukturen, also den Segment-Baum, Range-Baum usw. als Sweep-line-Status-Struktur einsetzen. Beispielsweise kann man den Rechteckschnitt-Algorithmus damit in einen Polygonschnitt-Algorithmus überführen. Diese Technik wird auch bei Ottmann und Widmayer [2012] beschrieben.



## 9 Externes Suchen und Sortieren

Wir haben bisher stillschweigend angenommen, dass beliebig komplexe Datenstrukturen bzw. alle von einem Algorithmus benötigten Daten komplett im Hauptspeicher gehalten werden können. Für manche Anwendungen trifft diese Annahme nicht zu, vor allem aus zwei Gründen:

1. Daten sollen *persistent* sein, das heißt, die Laufzeit des Programms überdauern. Dazu sind sie z. B. auf “externem” Plattenspeicher zu halten.
2. Die zu verarbeitende Datenmenge ist schlicht zu groß, um gleichzeitig vollständig in den Hauptspeicher zu passen.

Wir sprechen von einem *externen* Algorithmus, wenn zur Verarbeitung einer Objektmenge der Größe  $n$  nur  $O(1)$  interner Speicherplatz benötigt wird (und natürlich  $\Omega(n)$  externer Speicherplatz, also Platz auf Hintergrundspeicher). Eine *externe* Datenstruktur ist vollständig auf Hintergrundspeicher dargestellt; wir sprechen dann auch von einer *Speicherstruktur*.

Das typische externe Speichermedium sind heute Magnetplatten. Beim Entwurf externer Algorithmen muss man ihre Zugriffscharakteristika beachten. Im Hauptspeicher ist das Lesen oder Schreiben von  $k$  Bytes im Allgemeinen  $k$ -mal so teuer wie das Lesen/Schreiben eines Bytes. Auf Plattenspeicher ist das Lesen/Schreiben eines Bytes nicht wesentlich billiger als z. B. das Lesen/Schreiben von 1 kBByte, da ein Großteil der Kosten (also der Zugriffszeit) auf das Positionieren des Lese/Schreibkopfes auf eine Spur der Platte und das Warten auf den Block innerhalb der Spur (während der Rotationszeit der Platte) entfällt. Als Konsequenz davon werden Daten grundsätzlich in größeren Einheiten, genannt *Blöcke*, von der Platte gelesen oder auf sie geschrieben. Typische Blockgrößen, die etwa in Betriebssystemen oder Datenbanksystemen benutzt werden, liegen zwischen 512 und 8192 Bytes (also 1/2 K bis 8 K). Aus der Sicht dieser Systeme werden Blöcke oft als *Seiten* bezeichnet. Der Zeitbedarf eines Seitenzugriffs ist relativ hoch; die CPU kann in dieser Zeit gewöhnlich viele tausend Instruktionen ausführen. Daher betrachtet man als *Kostenmaß für die Laufzeit* eines externen Algorithmus meist die *Anzahl der Seitenzugriffe*.

Da externer Speicherplatz in Form von Seiten zur Verfügung gestellt wird, misst man den *Platzbedarf* einer Speicherstruktur als *Anzahl der belegten Seiten*. Da innerhalb einer Seite aus organisatorischen Gründen im Allgemeinen nur ein Teil des angebotenen Platzes tatsächlich mit Information belegt ist, ist ein weiteres interessantes Maß für externe Speicherstrukturen ihre *Speicherplatzausnutzung*, die definiert ist als

$$\frac{\text{Anzahl benutzter Bytes}}{\text{Anzahl Seiten} * \text{Anzahl Bytes/Seite}}$$

und die gewöhnlich in % angegeben wird.

## 9.1 Externes Suchen: B-Bäume

Wir betrachten das Problem des externen Suchens:

Gegeben eine Menge von Datensätzen mit Schlüsseln aus einem geordneten Wertebereich, organisiere diese Menge so, dass ein Datensatz mit gegebenem Schlüssel effizient gefunden, eingefügt oder entfernt werden kann.

Effizient heißt nun: mit möglichst wenig Seitenzugriffen. Es geht also um eine externe Implementierung des Dictionary-Datentyps.

Eine Idee, die zu einer eleganten Lösung führt, besteht darin, *Speicherseiten als Knoten eines Suchbaums aufzufassen*. Um die Kosten für eine Suche, die der Pfadlänge entsprechen, möglichst gering zu halten, wählt man *Bäume mit hohem Verzweigungsgrad*. Wir haben ja schon in Abschnitt 3.6 gesehen, dass die minimale Höhe eines Baumes vom Grad  $d$   $O(\log_d n)$  ist; binäre Bäume mit Höhe  $O(\log_2 n)$  stellen dabei den schlechtesten Fall dar.

Ein *allgemeiner Suchbaum* (auch *Vielweg-Suchbaum* genannt) ist eine Verallgemeinerung des binären Suchbaums; eine solche Struktur ist in Abbildung 9.1 gezeigt:

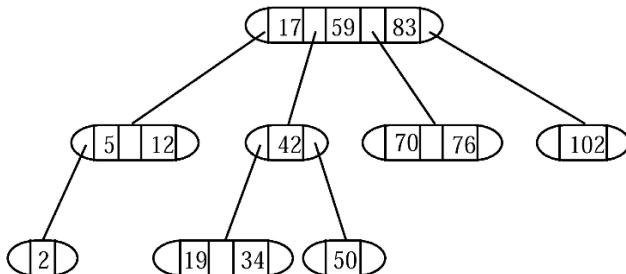


Abbildung 9.1: Vielweg-Suchbaum

**Definition 9.1:** (Vielweg-Suchbäume)

- (i) Der leere Baum ist ein Vielweg-Suchbaum mit Schlüsselmenge  $\emptyset$ .
- (ii) Seien  $T_0, \dots, T_s$  Vielweg-Suchbäume mit Schlüsselmengen  $\bar{T}_0, \dots, \bar{T}_s$  und sei  $k_1, \dots, k_s$  eine Folge von Schlüsseln, sodass gilt:

$$k_1 < k_2 < \dots < k_s.$$

Dann ist die Folge

$$T_0 \ k_1 \ T_1 \ k_2 \ T_2 \ k_3 \ \dots \ k_s \ T_s$$

ein Vielweg-Suchbaum genau dann, wenn gilt:

$$\begin{aligned} \forall x \in \bar{T}_i : \quad & k_i < x < k_{i+1} \quad \text{fnr } i = 1, \dots, s-1 \\ \forall x \in \bar{T}_0 : \quad & x < k_1 \\ \forall x \in \bar{T}_s : \quad & k_s < x \end{aligned}$$

Seine Schlüsselmenge ist  $\{k_1, \dots, k_s\} \cup \bigcup_{i=0}^s \bar{T}_i$

Der in diesem Abschnitt zu besprechende *B-Baum* ist eine spezielle Form eines Vielweg-Suchbaumes, dessen Struktur folgende Bedingungen erfüllt:

**Definition 9.2:** (B-Bäume) Ein B-Baum der Ordnung  $m$  ist ein Vielweg-Suchbaum mit folgenden Eigenschaften:

- (i) Die Anzahl der Schlüssel in jedem Knoten mit Ausnahme der Wurzel liegt zwischen  $m$  und  $2m$ . Die Wurzel enthält mindestens einen und maximal  $2m$  Schlüssel.
- (ii) Alle Pfadlängen von der Wurzel zu einem Blatt sind gleich.
- (iii) Jeder innere Knoten mit  $s$  Schlüsseln hat genau  $s+1$  Söhne (das heißt, es gibt keine leeren Teilbäume).

Ein B-Baum würde für die Schlüsselmenge des Vielweg-Suchbaumes aus Abbildung 9.1 z. B. so aussehen (Ordnung 2):

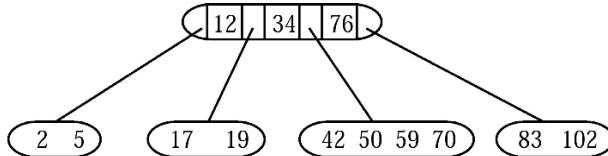
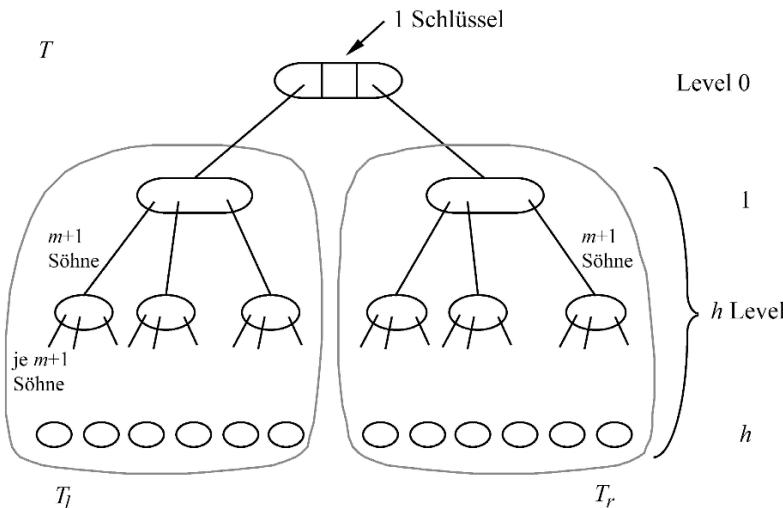


Abbildung 9.2: B-Baum zum Vielweg-Suchbaum aus Abbildung 9.1

Aus der Strukturdefinition können wir bereits eine obere Schranke für die Höhe eines B-Baumes der Ordnung  $m$  mit  $n$  Schlüsseln ableiten. Wir betrachten dazu einen minimal gefüllten B-Baum der Höhe  $h$ :

Abbildung 9.3: Minimal gefüllter B-Baum der Höhe  $h$ 

$T_l$  und  $T_r$  sind jeweils vollständige Bäume vom Grad  $(m+1)$ . Das Symbol “#” stehe für “Anzahl”.

$$\begin{aligned} \#\text{Knoten}(T_l) &= 1 + (m+1) + (m+1)^2 + \dots + (m+1)^{h-1} \\ &= \frac{(m+1)^h - 1}{(m+1) - 1} \end{aligned} \quad (\text{Grundlagen I})$$

$$\begin{aligned}\#\text{Schlüssel}(T_l) &= m \cdot \frac{(m+1)^h - 1}{m} \\ &= (m+1)^h - 1 \\ \#\text{Schlüssel}(T) &= 2 \cdot (m+1)^h - 1\end{aligned}$$

Seien nun  $n$  Schlüssel in einem Baum der Höhe  $h$  gespeichert. Es gilt

$$n \geq 2 \cdot (m+1)^h - 1$$

$$(m+1)^h \leq \frac{n+1}{2}$$

$$\begin{aligned}h &\leq \log_{(m+1)} \left( \frac{n+1}{2} \right) \\ &= O(\log_{(m+1)} n)\end{aligned}$$

Die Struktur eines Knotens, damit also auch einer Speicherseite, könnte man z. B. so festlegen:

```
type BNode = record
    used: 1..2m;
    keys: array[1..2m] of keytype;
    sons: array[0..2m] of BNodeAddress
end
```

Das Feld *used* gibt an, wieviele Schlüssel im Knoten gespeichert sind. Die Zeiger auf Söhne vom Typ *BNodeAddress* sind nun logische Seitennummern, mit denen etwa das Betriebssystem etwas anfangen kann. Für *keytype* ist angenommen, dass Schlüssel fester Länge gespeichert werden.

**Beispiel 9.3:** Nehmen wir an, dass *keytype* und *BNodeAddress* jeweils in 4 Bytes darstellbar sind und dass Seiten der Größe 1 kByte benutzt werden. Dann kann  $m = 63$  gewählt werden. Sei  $n = 10^6$ .

$$\begin{aligned}h &\leq \log_{64} 500000 \\ h_{\max} &= \lfloor \log_{64} 500000 \rfloor = 3\end{aligned}$$

Also die maximale Höhe ist 3, der Baum hat dann 4 Ebenen. Jeder Schlüssel kann mit 4 Seitenzugriffen gefunden werden.  $\square$

Der Suchalgorithmus sollte offensichtlich sein. Bei dem beschriebenen Knotentyp kann innerhalb eines Knotens binär gesucht werden. Die Kosten für eine Suche sind beschränkt durch die Höhe des Baumes.

Die Eleganz des B-Baumes liegt nun darin, dass diese Struktur auch unter Änderungsoperationen (Einfügen, Löschen) auf recht einfache Art aufrecht erhalten werden kann. Die Algorithmen für das Einfügen und Entfernen verletzen temporär die Struktur des B-Baumes, indem Knoten mit  $2m+1$  Schlüsseln entstehen (diese Verletzung heißt *Overflow*, der Knoten ist überfüllt) oder Knoten mit  $m-1$  Schlüsseln (*Underflow*, der Knoten ist unterfüllt). Es wird dann eine Overflow- oder Underflow-Behandlung eingeleitet, die jeweils die Struktur in Ordnung bringt.

### Einfügen und Löschen

Die Algorithmen für das Einfügen und Entfernen kann man so formulieren:

```
algorithm insert (root, x)
{füge Schlüssel x in den Baum mit Wurzelknoten root ein}
suche nach x im Baum mit Wurzel root;
if x nicht gefunden
then sei p das Blatt, in dem die Suche endete; füge x an der richtigen Position in p
ein;
    if p hat jetzt  $2m+1$  Schlüssel then overflow (p) end if
end if.

algorithm delete (root, x)
{entferne Schlüssel x aus dem Baum mit Wurzel root}
suche nach x im Baum mit Wurzel root;
if x wird gefunden
then if x liegt in einem inneren Knoten
        then suche x', den Nachfolger von x (den nächstgrößeren gespeicherten
            Schlüssel) im Baum (x' liegt in einem Blatt); vertausche x mit x'
        end if;
        sei p das Blatt, das x enthält; lösche x aus p;
        if p ist nicht die Wurzel
            then if p hat nun  $m-1$  Schlüssel then underflow (p) end if
            end if
        end if.
```

### Overflow

Ein Overflow eines Knotens  $p$  wird mit einer Operation  $split(p)$  behandelt, die den Knoten  $p$  mit  $2m+1$  Schlüsseln am mittleren Schlüssel  $k_{m+1}$  teilt, sodass Knoten mit Schlüsselfolgen  $k_1 \dots k_m$  und  $k_{m+2} \dots k_{2m+1}$  entstehen, die jeweils  $m$  Schlüssel enthalten.  $k_{m+1}$  wandert "nach oben", entweder in den Vaterknoten oder in einen neuen Wurzelknoten. Dadurch kann der Vaterknoten überlaufen. Diese Algorithmen lassen sich am besten grafisch anhand der von ihnen durchgeführten Baumtransformationen beschreiben.

**algorithm**  $overflow(p) = split(p)$ .

**algorithm**  $split(p)$   
 {teile den Knoten  $p$ }

**Fall 1:**  $p$  hat einen Vater  $q$ . Knoten  $p$  wird so geteilt:

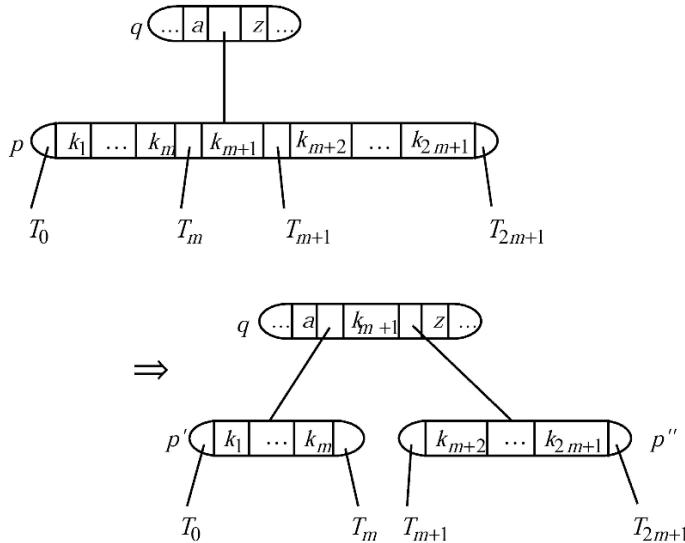


Abbildung 9.4: Aufteilen von  $p$ , wenn  $p$  nicht die Wurzel ist

**if**  $q$  hat nun  $2m+1$  Schlüssel **then**  $overflow(q)$  **end if**

**Fall 2:**  $p$  ist die Wurzel. Knoten  $p$  wird so geteilt:

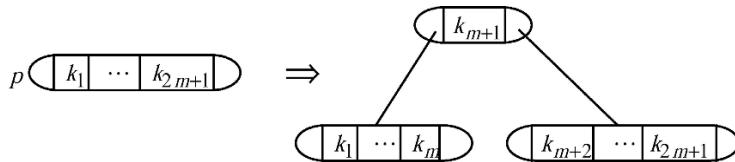


Abbildung 9.5: Aufteilen der Wurzel  $p$

**end split.**

Die Behandlung eines Overflow kann sich also von einem Blatt bis zur Wurzel des Baumes fortpflanzen.

## Underflow

Ein *Nachbar* eines Knotens sei ein direkt benachbarter Bruder. Um einen Underflow in  $p$  zu behandeln, werden der oder die Nachbarn von  $p$  betrachtet. Wenn einer der Nachbarn genügend Schlüssel hat, wird seine Schlüsselfolge mit der von  $p$  ausgeglichen (Operation *balance*), sodass beide etwa gleichviele Schlüssel haben. Andernfalls wird  $p$  mit dem Nachbarn zu einem einzigen Knoten verschmolzen (Operation *merge*).

```
algorithm underflow ( $p$ )
{behandle die Unterfüllung des Knotens  $p$ }
if  $p$  hat einen Nachbarn  $p'$  mit  $s > m$  Schlüsseln
then balance ( $p, p'$ )
else da  $p$  nicht die Wurzel sein kann, muss  $p$  einen Nachbarn mit  $m$  Schlüsseln
    haben; sei  $p'$  so ein Nachbar mit  $m$  Schlüsseln;
    merge ( $p, p'$ )
end if.
```

Die Operationen *balance* und *merge* lassen sich ebenso wie *split* am besten graphisch darstellen:

**algorithm** *balance* (*p*, *p'*)

balanciere Knoten *p* mit seinem Nachbarknoten *p'* folgendermaßen

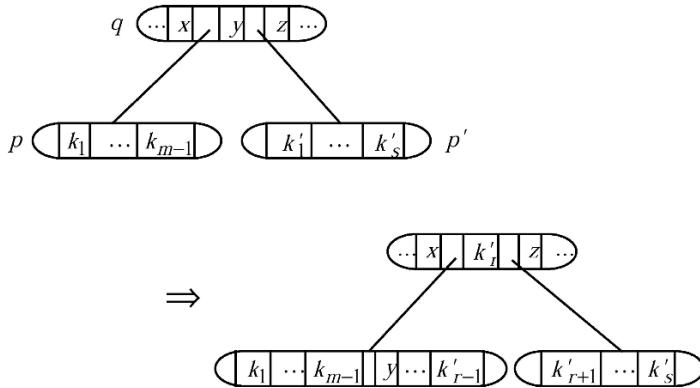


Abbildung 9.6: Balancieren der Knoten *p* und *p'*

wobei  $k'_r$  der mittlere Schlüssel der gesamten Schlüsselfolge ist, also

$$r = \left\lceil \frac{m+s}{2} \right\rceil - m$$

**end** *balance*.

Die Formel für  $r$  gilt für den in Abbildung 9.6 dargestellten Fall, dass der mittlere Schlüssel aus Knoten *p'* zu wählen ist.

**algorithm** *merge* (*p*, *p'*)

verschmelze Knoten *p* mit seinem Nachbarknoten *p'* folgendermaßen:

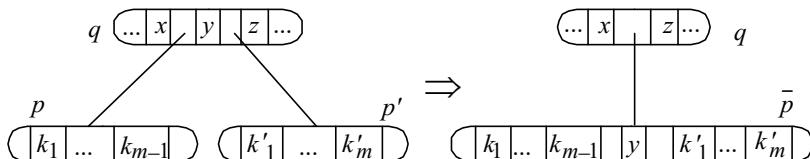


Abbildung 9.7: Verschmelzen der Knoten *p* und *p'*

**if** *q* ist nicht die Wurzel und hat  $m-1$  Schlüssel

**then** *underflow* (*q*)

**elsif** *q* ist die Wurzel und hat keinen Schlüssel mehr

**then** gib den Wurzelknoten frei und lass *root* auf  $\bar{p}$  zeigen.

**end if.**

Auch eine Löschoperation kann sich also bis zur Wurzel fortpflanzen, wenn jeweils die *Merge*-Operation durchgeführt werden muss.

**Beispiel 9.4:** Wir betrachten die Entwicklung eines B-Baumes der Ordnung 2 unter Einfüge- und Löschoperationen. Es werden jeweils die Situationen gezeigt, wenn ein Overflow (+) oder ein Underflow (-) aufgetreten ist, und die Behandlung der Strukturverletzung.

- (a) Einfügen: 50 102 34 19 5 / 76 42 2 83 / 59 70 12 17 (die Schrägstriche bezeichnen Positionen, an denen restrukturiert wird).

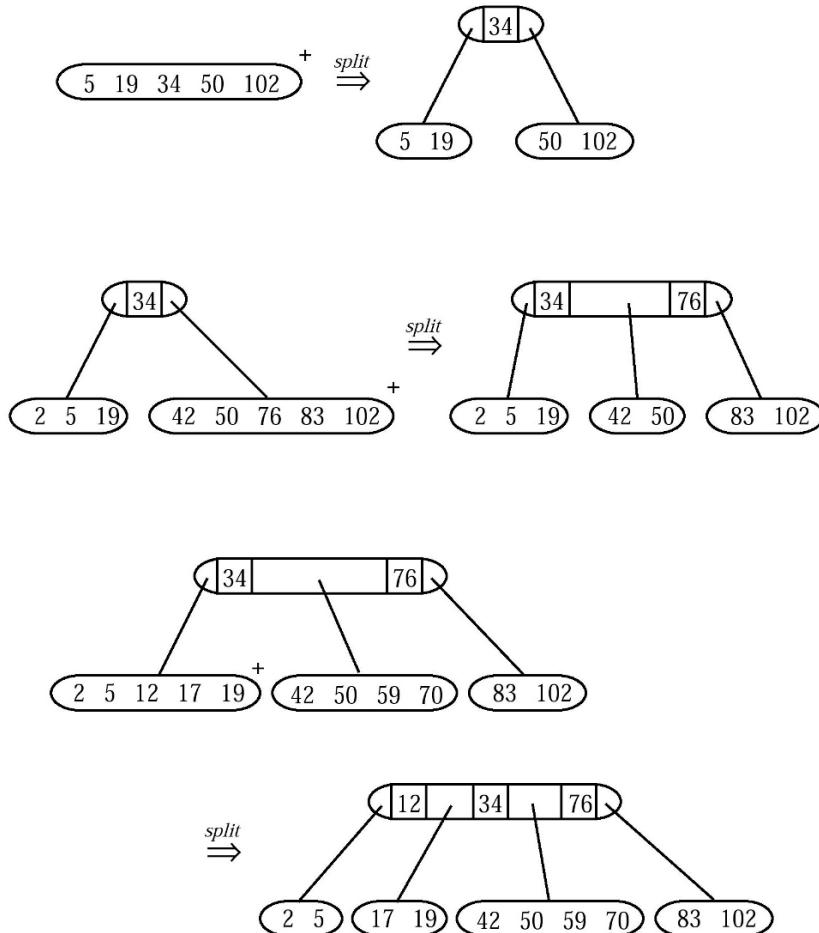


Abbildung 9.8: Aufbau des B-Baumes

(b) 83 entfernen

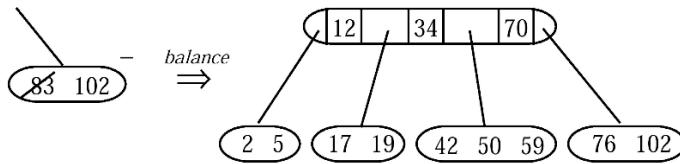


Abbildung 9.9: Entfernen des Schlüssels 83

(c) 2 entfernen

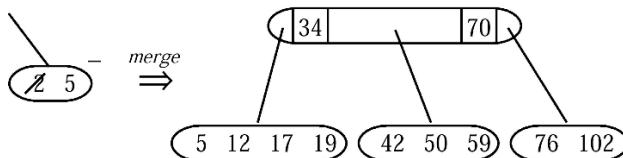


Abbildung 9.10: Entfernen des Schlüssels 2

□

Die Kosten für eine Einfüge- oder Lösch-Operation sind offensichtlich ebenfalls proportional zur Höhe des Baumes. Der B-Baum unterstützt also alle Dictionary-Operationen in  $O(\log_{(m+1)} n)$  Zeit, der Platzbedarf ist  $O(n)$ . Die Speicherplatzausnutzung ist garantiert besser als 50% (abgesehen von der Wurzel).

Beim praktischen Einsatz von B-Bäumen, etwa in Datenbanksystemen, werden meist Varianten des hier gezeigten Grundschemas verwendet. Zum Beispiel ist es üblich, in den Blättern Datensätze, in den inneren Knoten hingegen Schlüssel zu speichern; diese Schlüssel haben dann nur Wegweiserfunktion bei der Suche nach Datensätzen. Wenn als Schlüssel Zeichenketten verwendet werden, kürzt man diese soweit wie möglich ab, ohne dass die Wegweiserfunktion verlorengeht. Kriterium für Underflow und Overflow ist dann nicht mehr die Anzahl der Schlüssel (die ja nun variable Länge haben), sondern der Füllungsgrad der Seite. Man erreicht damit unter anderem, dass mehr Schlüssel auf eine Seite passen, der Verzweigungsgrad steigt und die Höhe des Baumes geringer wird.

Schließlich kann der B-Baum auch als interne balancierte Baumstruktur eingesetzt werden. In diesem Fall ist es günstig, den Verzweigungsgrad minimal zu wählen (da die Kosten für das Suchen innerhalb eines Knotens nun auch ins Gewicht fallen) und man benutzt B-Bäume der Ordnung 1. Hier hat jeder Knoten 2 oder 3 Söhne, deshalb spricht

man von *2-3-Bäumen*. Ebenso wie AVL-Bäume erlauben sie Suche, Einfügen, Entfernen in  $O(\log n)$  Zeit und  $O(n)$  Speicherplatz.

**Selbsttestaufgabe 9.1:** Nehmen wir an, die Knoten eines modifizierten B-Baumes sollten nicht nur zur Hälfte, sondern zu mindestens zwei Dritteln gefüllt sein. Wie muss man das Verfahren für das Einfügen ändern, damit dies gewährleistet wird? Welche Vorteile und Nachteile hätte ein derart veränderter B-Baum?  $\square$

## 9.2 Externes Sortieren

Das zweite zentrale Problem bei der Behandlung großer externer Datenbestände ist das Sortieren. Gegeben sei also eine gespeicherte Menge von  $n$  Datensätzen mit Schlüsseln aus einem geordneten Wertebereich. Die Sätze stehen in einem (vom Betriebssystem verwalteten) File, dessen Darstellung auf Magnetplattenspeicher wir uns als *Seitenfolge* vorstellen können. Wir nehmen der Einfachheit halber an, dass alle Datensätze feste Länge haben und dass  $b$  Sätze auf eine Seite passen.

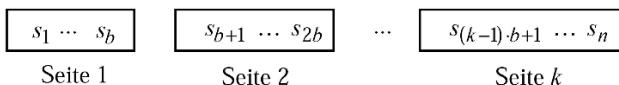


Abbildung 9.11:  $n$  Datensätze auf  $k$  Seiten

Ein sequentielles Lesen aller  $n$  Datensätze erfordert  $k = \lceil n/b \rceil$  Seitenzugriffe. Ein Lesen aller Datensätze in irgendeiner anderen Reihenfolge wird im Allgemeinen etwa  $n$  Seitenzugriffe erfordern. Da  $b$  gewöhnlich erheblich größer ist als 1 (z.B.  $b = 50$  oder  $b = 100$ ), ist man auf Sortierverfahren angewiesen, die eine Menge von Sätzen sequentiell (das heißt in Speicherungsreihenfolge) verarbeiten. Unter den höheren Sortiermethoden hat vor allem *Mergesort* diese Eigenschaft (bis zu einem gewissen Grad auch Quicksort, aber hier ist Mergesort vorzuziehen).

Ein sehr großer Datenbestand könnte auch auf Magnetband gespeichert sein. Hier ist der Zeitbedarf, um auf eine beliebige Seite zu positionieren, extrem hoch (kann im Minutenbereich liegen). Bei Magnetbändern ist daher sequentielle Verarbeitung absolut zwingend geboten. Wir betrachten im Folgenden Varianten von Mergesort.

Wir haben in Kapitel 5 Mergesort als Divide-and-Conquer-Algorithmus beschrieben, in dem die eigentliche Arbeit im Merge-Schritt geleistet wird. Im externen Fall fallen die Divide- und Conquer-Schritte weg, man beginnt direkt mit dem bottom-up Merge-Vorgang. Sortieren durch Mischen erfolgt dann in mehreren *Phasen*. In jeder Phase wird die

komplette Menge von Sätzen von zwei Eingabe-Files gelesen und auf zwei Ausgabe-Files geschrieben. Für die nächste Phase werden Eingabe- und Ausgabe-Files vertauscht. Wenn wir dem Merge-Diagramm des Kapitels 5 (Abbildung 5.1) Eingabe- und Ausgabe-Files  $f_1, f_2, g_1, g_2$  zuordnen, erhalten wir folgende Darstellung:

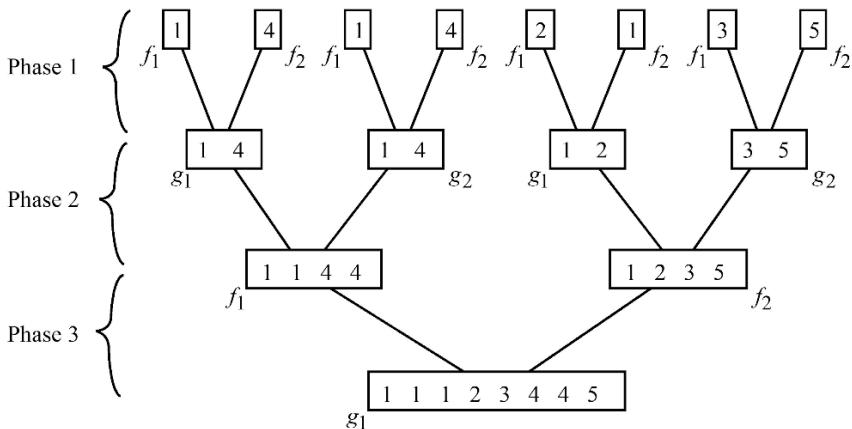


Abbildung 9.12: Externes Mergesort

Zu Beginn steht auf  $f_1$  die Folge 1 1 2 3 und auf  $f_2$  die Folge 4 4 1 5. In einem vorbereitenden Schritt ist also eine gegebene Folge von Sätzen in einem File auf zwei Files zu verteilen.

**Definition 9.5:** Eine aufsteigend geordnete Teilfolge von Sätzen innerhalb eines Files bezeichnen wir als *Lauf*.

In einer Mischphase werden nun jeweils von den beiden Eingabefiles “parallel” zwei Läufe gelesen und zu einem Ergebnislauf verschmolzen. Die so entstehenden Ergebnisläufe werden alternierend in die beiden Ausgabefiles geschrieben. Das Verschmelzen zweier Läufe entspricht dem in Abschnitt 4.1 geschilderten parallelen Durchlauf durch zwei geordnete Listen.

Der gesamte Ablauf lässt sich dann folgendermaßen skizzieren. Wir nehmen an, dass Files  $f_1, f_2, g_1$  und  $g_2$  verwendet werden und dass zu Beginn alle  $n$  Datensätze in  $g_1$  stehen; die anderen Files sind leer. Zu Anfang liest eine Prozedur *InitialRuns* (Lauf = Run) das File  $g_1$  und schreibt abwechselnd Läufe nach  $f_1$  und  $f_2$ . Im einfachsten Fall werden jeweils Läufe der Länge 1 erzeugt, das heißt, jeder Datensatz bildet seinen eigenen Lauf. *InitialRuns* gibt die Anzahl erzeugter Läufe (im einfachsten Fall also  $n$ ) in einer

Variablen  $r$  zurück. Anschließend wird der Algorithmus  $EMergeSort$  aufgerufen, der seiterseits jede Mischphase mit einer Prozedur  $Merge$  realisiert.

```

InitialRuns ( $f_1, f_2, r$ );
EMergeSort ( $f_1, f_2, g_1, g_2, r$ )
algorithm EMergeSort ( $f_1, f_2, g_1, g_2, r$ )
{sortiere durch externes Mischen mithilfe der Files  $f_1, f_2, g_1$  und  $g_2$ ; zu Anfang
stehen  $r$  Läufe in  $f_1$  und  $f_2$ }
Laufzahl :=  $r$ ;
loop
    Merge ( $f_1, f_2, g_1, g_2$ ); Laufzahl :=  $\lceil Laufzahl / 2 \rceil$ ;
    if Laufzahl = 1 then exit end if;
    Merge ( $g_1, g_2, f_1, f_2$ ); Laufzahl :=  $\lceil Laufzahl / 2 \rceil$ ;
    if Laufzahl = 1 then exit end if;
end loop.

```

Der Algorithmus  $Merge$  geht davon aus, dass die Lauf-Anzahlen seiner beiden Eingabefiles sich höchstens um 1 unterscheiden und stellt das wiederum für die beiden Ausgabefiles sicher.

```

algorithm Merge ( $f_1, f_2, g_1, g_2$ )
{verschmelze Läufe von Files  $f_1$  und  $f_2$  alternierend nach Files  $g_1$  und  $g_2$ }
var  $g$  : filename;
 $g$  :=  $g_1$ ;
while es gibt noch mindestens einen unbearbeiteten Lauf in  $f_1$  oder  $f_2$  do
    if  $f_1$  und  $f_2$  enthalten jeweils noch mindestens einen Lauf
        then verschmelze den ersten Lauf in  $f_1$  mit dem ersten Lauf in  $f_2$ , wobei der
            Ergebnislauf nach  $g$  geschrieben wird
        else schreibe den noch übrigen Lauf von  $f_1$  oder  $f_2$  nach  $g$ 
        end if;
        if  $g = g_1$  then  $g := g_2$  else  $g := g_1$  end if
    end while.

```

**Beispiel 9.6:** Für unsere B-Baum-Zahlenfolge ergibt sich folgender Ablauf:

$g_1$ : 50 102 34 19 5 76 42 2 83 59 70 12 17

InitialRuns:

$f_1$ :	50   34   5   42   83   70   17
$f_2$ :	102   19   76   2   59   12
	↑ Lauftrennsymbol

Phase 1

$g_1: 50 \ 102 | 5 \ 76 | 59 \ 83 | 17 |$   
 $g_2: 19 \ 34 | 2 \ 42 | 12 \ 70 |$

Phase 2

$f_1: 19 \ 34 \ 50 \ 102 | 12 \ 59 \ 70 \ 83 |$   
 $f_2: 2 \ 5 \ 42 \ 76 | 17 |$

Phase 3

$g_1: 2 \ 5 \ 19 \ 34 \ 42 \ 50 \ 76 \ 102 |$   
 $g_2: 12 \ 17 \ 59 \ 70 \ 83 |$

Phase 4

$f_1: 2 \ 5 \ 12 \ 17 \ 19 \ 34 \ 42 \ 50 \ 59 \ 70 \ 76 \ 83 \ 102 |$   
 $f_2:$

□

*Analyse:* In *InitialRuns* und in jeder Phase werden sämtliche Datensätze jeweils einmal sequentiell gelesen und geschrieben, mit Kosten  $O(n/b)$ . Wenn man, wie im Beispiel, mit  $n$  Läufen der Länge 1 beginnt, gibt es  $\lceil \log_2 n \rceil$  Phasen. Der Gesamtaufwand beträgt also  $O(n/b \log n)$  Seitenzugriffe. Auch der interne Zeitbedarf ist  $O(n \log n)$ . Man kann nun Mischsortierverfahren danach unterscheiden, ob zu Anfang in *InitialRuns* Läufe fester Länge (wie im Beispiel) oder Läufe variabler Länge erzeugt werden. Der erste Fall wird bisweilen *direktes Mischen*, der zweite Fall “*natürliches Mischen*” genannt. Wir werden gleich sehen, wie Läufe variabler Länge entstehen können.

### Anfangsläufe fester Länge - direktes Mischen

Externes Mischsortieren wird durchgeführt, weil nicht alle Datensätze gleichzeitig in den Hauptspeicher passen. Bisher benutzen wir im Hauptspeicher lediglich einen Pufferbereich von 4 Seiten oder evtl. 8 Seiten (eine Seite für jedes File; um Verzögerungen zu vermeiden, wenn etwa eine neue Seite eines Eingabefiles benötigt wird, wird man meist eher 2 Seiten pro File im Puffer halten). Warum soll man aber fast den gesamten Hauptspeicher ungenutzt lassen?

Deshalb wählt man in der Praxis eine *Kombination* von internem und externem Sortieren. Beim direkten Mischen sortiert man in *InitialRuns* jeweils soviele Sätze, wie in den Hauptspeicher passen, sagen wir  $m$  Sätze, mit einem internen Sortierverfahren, z. B. mit Heapsort. So werden Anfangsläufe der Länge  $m$ , also Läufe fester Länge, erzeugt. Da

nun zu Anfang nicht mehr  $n$ , sondern nur noch  $k = \lceil n/m \rceil$  Läufe erzeugt werden, kann die Anzahl der Phasen, die im Allgemeinen ja  $\lceil \log_2 k \rceil$  beträgt, dramatisch sinken.

**Beispiel 9.7:** Es seien  $10^6$  Datensätze zu sortieren, wobei jeweils  $10^4$  intern sortiert werden können. Es entstehen 100 Läufe, die in 7 Phasen sortiert werden, anstelle von 20 Phasen bei rein externem Sortieren.  $\square$

Die Anzahl der Seitenzugriffe sinkt damit auf  $O\left(\frac{n}{b} \cdot \log \frac{n}{m}\right)$ .

### Anfangsläufe variabler Länge - natürliches Mischen

Aus zwei Gründen ist man an Läufen variabler Länge interessiert:

1. In der Praxis kommt es häufig vor, dass Teile von Files bereits vorsortiert sind. Beim direkten Mischen wird das nicht ausgenutzt.
2. Durch einen Trick, genannt *Ersetzungs-Auswahl* (oder auch “Filtern durch einen Heap”) kann man beim internen Sortieren in *InitialRuns* Läufe erzeugen, deren Länge die Größe des Hauptspeichers bei weitem übersteigt! Auch hierbei ist die Länge der entstehenden Läufe variabel.

Die Idee der *Ersetzungs-Auswahl* besteht darin, dass man einen Anfangslauf aufbaut, indem man aus den im Hauptspeicher vorhandenen Sätzen jeweils den mit minimalem Schlüssel auswählt (also SelectionSort bzw. Heapsort anwendet) und ihn auf das Ausgabefile schreibt. Die entstehende Lücke wird aber sofort wieder vom Eingabefile gefüllt. Falls der Schlüssel des gerade eingelesenen Satzes größer ist als der des gerade herausgeschriebenen, so kann dieser neue Satz noch in den aktuellen (im Entstehen begriffenen) Lauf aufgenommen werden, andernfalls muss er den nächsten Lauf abwarten.

**Beispiel 9.8:** Wir erzeugen nach diesem Prinzip Anfangsläufe, indem wir die obige Schlüsselfolge durch einen Hauptspeicher filtern, der gerade 3 Sätze aufnehmen kann. Eingeklammerte Schlüssel bezeichnen Sätze, die auf den nächsten Lauf warten.

50 102 34 19 5 76 42 2 83 59 70 12 17

1	2	3	
50	102	<u>34</u>	34
<u>50</u>	102	(19)	50
(5)	<u>102</u>	(19)	102
(5)	(76)	(19)	

1. Lauf

---

5	76	19	5
42	76	<u>19</u>	19
<u>42</u>	76	(2)	42
83	<u>76</u>	(2)	76
<u>83</u>	(59)	(2)	83
(70)	(59)	(2)	

---

70	59	<u>2</u>	2
70	59	<u>12</u>	12
70	59	<u>17</u>	17
70	<u>59</u>		59
<u>70</u>			70

□

Man sieht schon am Beispiel, dass Läufe entstehen, die länger sind als 3, was der Größe des Hauptspeichers entspricht. Es ist auch klar, dass eine bereits aufsteigend sortierte Folge im Eingabefile komplett in den aktuellen Lauf übernommen wird, falls ihr erster Satz übernommen wird (man kann das ansatzweise beim Aufbau des 3. Laufes beobachten). Andernfalls wird sie spätestens im nächsten Lauf komplett untergebracht sein.

Man kann zeigen, dass die erwartete Länge der Läufe bei einer zufälligen Schlüsselverteilung

$$2 \cdot m \quad (m \text{ die Größe des Hauptspeichers})$$

beträgt [Knuth 1998]. Falls Vorsortierungen vorhanden sind, kann die tatsächliche Länge noch erheblich größer werden. Man realisiert die Idee der Ersetzungs-Auswahl, indem man innerhalb eines Arrays vorne einen Heap-Bereich zur Bildung des aktuellen Laufs und hinten eine Liste von Datensätzen, die “auf den nächsten Lauf warten”, unterhält.

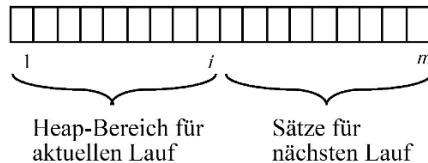


Abbildung 9.13: Array-Aufteilung für Ersetzungs-Auswahl

Jeder Einzelschritt bei der Bildung des aktuellen Laufs sieht so aus:

```

entnimm dem Heap den Datensatz  $s$  mit minimalem Schlüssel  $k_{min}$  und schreibe
ihn auf das Ausgabefile;
lies vom Eingabefile den Datensatz  $s'$  mit Schlüssel  $k$ ;
if  $k_{min} \leq k$ 
then lass den Datensatz  $s'$  in den Heap einsinken
else reduziere den Heap-Bereich auf  $1..i-1$  und schreibe den Datensatz  $s'$  auf die
Position  $i$ 
end if

```

Der gesamte Algorithmus zur Bildung von Anfangsläufen hat dann folgende Struktur:

```

lies die ersten  $m$  Sätze vom Eingabefile in den Array;
loop
    1. konstruiere einen Heap im Bereich  $1..m$ ; beginne einen neuen Lauf;
    2. repeat füge dem aktuellen Lauf (wie oben beschrieben) einen Satz hinzu
       until der Heap-Bereich ist leer
end loop

```

Wir haben dabei die Abbruchbedingung für die äußere Schleife ignoriert. Sobald vom Eingabefile keine Sätze mehr nachkommen, verarbeitet man noch in leicht modifizierter Form die restlichen Sätze im Array.

### Vielweg-Mischen

Als letzten Aspekt des externen Sortierens betrachten wir das *Vielweg-Mischen*: Man kann anstelle von je zwei Eingabe- und Ausgabefiles auch  $k$  Eingabe- und  $k$  Ausgabefiles benutzen. In einer Phase werden wiederholt je  $k$  Läufe zu einem Lauf verschmolzen; die entstehenden Läufe werden zyklisch auf die Ausgabefiles verteilt. Intern muss man dabei in jedem Schritt aus den  $k$  ersten Schlüsseln der Eingabefiles den minimalen Schlüssel bestimmen; bei großem  $k$  kann das wiederum effizient mithilfe eines Heaps geschehen.

*Analyse:* In jeder Phase wird nun die Anzahl vorhandener Läufe durch  $k$  dividiert; wenn zu Anfang  $r$  Läufe vorhanden sind, ergeben sich  $\lceil \log_k r \rceil$  Phasen, gegenüber  $\lceil \log_2 r \rceil$  beim binären Mischen. Die Anzahl externer Seitenzugriffe wird also noch einmal erheblich reduziert. Der interne Zeitbedarf ist  $O(n \cdot \log_2 k)$  pro Phase, also insgesamt

$$O(n \cdot \underbrace{\log_2 k \cdot \log_k r}_{\log_2 r}) = O(n \cdot \log_2 r)$$

ohne die Zeit für das Erzeugen der Anfangsläufe, ändert sich also nicht gegenüber dem binären Mischen.

Man kann  $k$  allerdings nicht beliebig groß machen: Einerseits benötigt jedes File im Hauptspeicher einen entsprechenden Pufferbereich. Außerdem kann es weitere Restriktionen, etwa vom Betriebssystem, für die Gesamtzahl gleichzeitig geöffneter Files geben.

**Selbsttestaufgabe 9.2:** Geben Sie die Läufe an, die beim Sortieren der Eingabefolge

10, 3, 44, 32, 44, 12, 33, 12, 12, 65, 78, 165, 4, 9, 7, 33, 87, 2, 9

durch Vielwegmischen ( $k = 4$ ) entstehen. Vergleichen Sie diese Läufe mit denen, die beim Sortieren derselben Folge mit binärem Mischen und Ersetzungsauswahl mit einem Speicher der Größe 3 entstehen.  $\square$

### 9.3 Weitere Aufgaben

**Aufgabe 9.3:** Binäre Bäume können auch für die Darstellung von Listen benutzt werden, in denen man direkten Zugriff auf Elemente anhand ihrer *Position* haben will. Die Idee ist, ein Feld *rank* einzuführen, in dem für jeden Knoten die Anzahl der Knoten in seinem linken Unterbaum plus eins notiert wird. Die Elemente werden dabei so gespeichert, dass ein Inorder-Durchlauf die Liste in ihrer ursprünglichen Reihenfolge liefert.

- (a) Geben Sie einen Algorithmus zum Auffinden eines Elementes mit gegebener Listenposition an.
- (b) Ist diese Idee auf B-Bäume übertragbar? Wenn ja, wie?
- (c) Warum ist es günstiger, *rank* zu speichern, als direkt die Listenposition als Schlüssel zu verwenden?

**Aufgabe 9.4:** In der Betrachtung des Mischartierens haben wir ignoriert, dass bei der Verarbeitung mit Magnetbändern auch für das Zurückspulen der Bänder Zeit benötigt wird. Falls die Bänder sowohl vorwärts als auch rückwärts gelesen werden können, kann

man diese Totzeiten vermeiden, indem die Mischphasen die Bänder abwechselnd vor- und rückwärts verarbeiten.

- (a) Geben Sie einen Algorithmus für den derart modifizierten Mergesort an.
- (b) Wenn nummerische Daten zu sortieren sind, werden die Enden der Läufe häufig durch einen “Scheindatensatz” markiert, nämlich durch  $+\infty$ . Was muss im Fall des modifizierten Algorithmus statt dessen getan werden?

**Aufgabe 9.5:** Zeigen Sie, dass jedes externe Sortierverfahren, das nur ein Band als externen Speicher benutzt, bzw. nur eine Speichereinheit mit ausschließlich sequentiellem Zugriff, im schlimmsten Fall  $\Omega(n^2)$  Zeit benötigt, um  $n$  Datensätze zu sortieren. Nehmen Sie dafür an, dass sich die Zugriffskosten aus Positionierungs-, Schreib- und Lesekosten zusammensetzen. Die Kosten für einen einzelnen Schreib- oder Lesezugriff seien konstant, die Kosten für Positionierung proportional zur auf dem Band zurückgelegten Entfernung.

**Aufgabe 9.6:** Gibt es einen Algorithmus, der eine gegebene Eingabefolge in  $O(n \log n)$  Laufzeit invertiert, wenn drei Bänder zur Verfügung stehen, die nur vorwärts gelesen/beschrieben werden können? Wenn ja, geben Sie ihn an. Wenn nein, so belegen Sie, warum es einen solchen Algorithmus nicht gibt.

**Aufgabe 9.7:** Eine Modifikation von  $k$ -Wege-Mergesort, die dem Standard-Algorithmus bei sehr großen Eingabefolgen überlegen ist, wendet folgendes Verfahren an (nehmen wir an, es ständen  $k=6$  Bänder  $T_1, \dots, T_6$  zur Verfügung): Zunächst erfolgt eine Verteilung der Eingabefolge in Läufe der Länge 1 auf  $T_1$  bis  $T_5$ . Diese Verteilung unterliegt bestimmten Gesetzen, die hier nicht besprochen werden sollen. In der ersten Phase werden solange  $T_1$  bis  $T_5$  mittels 5-Wege-Mischen zu Läufen der Länge 5 verschmolzen, die auf  $T_6$  ausgegeben werden, bis  $T_5$  leer ist. Anschließend werden aus  $T_1$  bis  $T_4$  durch 4-Wege-Mischen Läufe der Länge 4 auf  $T_5$  erzeugt, dann erfolgt 3-Wege-Mischen auf  $T_4$ , 2-Wege-Mischen auf  $T_3$ . In der zweiten Phase erfolgt 5-Wege-Mischen auf  $T_2$ , dann 4-Wege-Mischen auf das zuletzt frei gewordene Band usw. Dieses Verfahren heißt Kaskaden-Mergesort.

Geben Sie die Länge und Anzahl der Läufe auf jedem Band nach jedem Durchlauf an für einen Kaskaden-Mergesort auf  $k=4$  Bändern und der Anfangsverteilung 70-56-31-0, das heißt 70 Elemente auf  $T_1$ , 56 Elemente auf  $T_2$ , 31 Elemente auf  $T_3$  und kein Element auf  $T_4$ .

## 9.4 Literaturhinweise

B-Bäume stammen von Bayer und McCreight [1972]; eine ältere File-Organisationsmethode, die bereits gewisse Ähnlichkeiten aufweist, ist die ISAM-Technik (index sequential access method) [Ghosh und Senko 1969]. Erwartungswerte für die Speicherplatzausnutzung eines B-Baumes wurden von Nakamura und Mizzogushi [1978] berechnet (vgl. auch Yao [1985]). Es ergibt sich unabhängig von der Ordnung des B-Baumes eine Speicherplatzausnutzung von  $\ln 2 \approx 69\%$ , wenn  $n$  zufällig gewählte Schlüssel in einen anfangs leeren B-Baum eingefügt werden. Die Speicherplatzausnutzung kann erhöht werden, wie in Aufgabe 9.1 angedeutet; derart "verdichtete" Bäume wurden von Culik *et al.* [1981] untersucht. Wie am Ende des Abschnitts 9.1 erwähnt, bilden Datenbanksysteme das Haupteinsatzfeld für B-Bäume. Dafür wurden verschiedene Varianten entwickelt, etwa mit dem Ziel, effizienten sequentiellen Zugriff zu erreichen und Schlüssel, die Zeichenketten sind, zu komprimieren [Wedekind 1974, Bayer und Unterauer 1977, Küspert 1983, Wagner 1973]. Derartige Techniken werden in [Lockemann und Schmidt 1987] diskutiert; ein Überblicksartikel zu B-Bäumen und ihren Varianten ist [Comer 1979].

Die wesentliche Quelle auch für externes Sortieren ist das Buch von Knuth [1998]; dort wird eine Fülle weiterer Varianten und Strategien diskutiert und analysiert, z. B. auch das Kaskaden-Mischsortieren (Aufgabe 9.7) und eine ähnliche Technik, das *Mehrphasen-Mischsortieren* (polyphase merging), eine Modifikation des Vielweg-Mischens, die mit  $k+1$  anstelle von  $2k$  Files auskommt (siehe auch [Shell 1971]). Als Alternative zum Mischen kann man übrigens auch Quicksort als externes Sortierverfahren einsetzen [Six und Wegner 1981]. Neuere Untersuchungen zum externen Sortieren versuchen, die Möglichkeiten moderner Rechnertechnik, z. B. sehr große interne Speicher [Salzberg 1989] oder Parallelität [Salzberg *et al.* 1990], [Barve, Grove und Vitter 1997] auszunutzen.

Ein ausgezeichneter Überblicksartikel zu externen Algorithmen und Datenstrukturen ist [Vitter 2001].

## Mathematische Grundlagen

### I Einige Summenformeln

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

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

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

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

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

#### Beweis:

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

Die Technik bei diesem Beweis ist die Ausnutzung der Assoziativität:

$$\begin{aligned} S_{n+1} &= (a_1 + \dots + a_n) + a_{n+1} \\ &= a_1 + (a_2 + \dots + a_{n+1}) \end{aligned}$$

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

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

□

## II Einige Grundlagen der Wahrscheinlichkeitsrechnung und Kombinatorik

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

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

- (2) Eine *Zufallsvariable*  $X$  ist eine Abbildung  $X: \Omega \rightarrow D$  (mit  $D = \mathbb{N}$  oder  $D = \mathbb{R}$ ); sie ordnet also jedem Ereignis in  $\Omega$  einen Zahlenwert zu.  
(3) Der *Erwartungswert* (=Durchschnitt, Mittelwert) einer Zufallsvariablen  $X$  ist

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

**Beispiel:** Der “durchschnittliche” Wert (also der Erwartungswert) beim Würfeln mit zwei Würfeln ist

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

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

$$\begin{array}{lll} (1,1) & (1,2) & (6,6) \\ & (2,1) & \end{array}$$

□

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

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

- (5) Notation:  $n^k := n \cdot (n-1) \cdot \dots \cdot (n-k+1)$

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

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

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

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

### III Umgang mit Binomialkoeffizienten

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

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

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

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

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

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

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

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

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

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

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

**Beispiel:**

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

□

## IV Harmonische Zahlen

$$(1) \quad H_n := 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \sum_{k=1}^n \frac{1}{k} \quad , \text{ n } \geq 0, \text{ heißt die n-te harmonische Zahl}$$

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

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

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

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

## V Umwandlung von Rekursion in eine Summe

Gegeben: Rekursionsgleichung der Form

$$a_n T_n = b_n T_{n-1} + c_n \quad n \geq 1$$

Multipliziere beide Seiten mit schlau gewähltem *Summierungsfaktor*  $s_n$

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

Wähle  $s_n$  so, dass

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

Dann gilt nämlich

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

also

$$U_n = U_{n-1} + s_n c_n$$

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

und

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

Wie findet man  $s_n$ ? Kein Problem:

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

Also

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

$s_1$  kann beliebig  $\neq 0$  gewählt werden

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

## VI Fibonacci-Zahlen

(1) Die Fibonacci-Zahlen sind so definiert:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad n > 1$$

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

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

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

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

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

bzw.

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

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

## Lösungen zu den Selbsttestaufgaben

### Aufgabe 1.1

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

### Aufgabe 1.2

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

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

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

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

### Aufgabe 1.3

*Additionsregel:*

Um zu zeigen, dass

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

suchen wir  $n_0 \in \mathbb{N}$ ,  $c \in \mathbb{R}$  mit  $c > 0$ , so dass für alle  $n \geq n_0$

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

Aufgrund der Annahmen aus der Aufgabenstellung haben wir

$$\begin{aligned} n_1 &\in \mathbb{N}, c_1 \in \mathbb{R}, \text{ so dass } \forall n \geq n_1: T_1(n) \leq c_1 f(n), \\ n_2 &\in \mathbb{N}, c_2 \in \mathbb{R}, \text{ so dass } \forall n \geq n_2: T_2(n) \leq c_2 g(n). \end{aligned} \quad (*)$$

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

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

*Multiplikationsregel:*

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

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

### Aufgabe 1.4

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

### Aufgabe 1.5

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

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

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

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

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

**Aufgabe 1.6****functions**

(a)	<i>reset</i>	= 0
	<i>inc(n)</i>	= $n + 1$
	<i>dec(n)</i>	= $\begin{cases} 0 & \text{falls } n = 0 \\ n - 1 & \text{sonst} \end{cases}$
(b)	<i>reset</i>	= 0
	<i>inc(n)</i>	= $n + 1$
	<i>dec(n)</i>	= $n - 1$
(c)	<i>reset</i>	= 0
	<i>inc(n)</i>	= $\begin{cases} n + 1 & \text{falls } n < p \\ 0 & \text{falls } n = p \end{cases}$
	<i>dec(n)</i>	= $\begin{cases} n - 1 & \text{falls } n > 0 \\ p & \text{falls } n = 0 \end{cases}$

**axioms**

<i>dec(inc(n))</i>	= $n$
<i>inc(dec(n))</i>	= $n$

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

**Aufgabe 1.7**

(a)

**algebra puzzle15**

<b>sorts</b>	<i>tile, position, board, bool</i>		
<b>ops</b>	<i>init:</i>	$tile^{16}$	$\rightarrow board$
	<i>move:</i>	$board \times tile$	$\rightarrow board$
	<i>solved:</i>	$board$	$\rightarrow bool$
	<i>pos:</i>	$board \times tile$	$\rightarrow position$

(b)

**sets**

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

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

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

(c)

**functions**

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

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

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

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

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

**end puzzle15.**

### Aufgabe 1.8

```

adt int
  sorts   int
  ops    0:           → int
            succ:      int       → int
            pred:      int       → int
            +:          int × int → int
            -:          int × int → int
            *:          int × int → int
  axs
            pred(succ(x))      = x
            succ(pred(x))       = x
            x + 0               = x
            x + succ(y)         = succ(x + y)
            x + pred(y)         = pred(x + y)
            x - 0               = x
            x - succ(y)         = pred(x - y)
            x - pred(y)         = succ(x - y)
            x * 0               = 0
            x * succ(y)         = (x * y) + x
            x * pred(y)         = (x * y) - x
end int.

```

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

**Aufgabe 3.1**

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

**Aufgabe 3.2**

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

```

    q.succ = s;
}
return this;
}

```

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

### Aufgabe 3.3

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

```

import Stack;

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

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

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

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

```

```

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

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

```

### Aufgabe 3.4

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

(1) Blatt  $x$ :

$$\begin{aligned} B((\emptyset, x, \emptyset)) &= 1, I((\emptyset, x, \emptyset)) = 0 \\ \Rightarrow \Delta((\emptyset, x, \emptyset)) &= 1 \end{aligned}$$

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

$$\begin{aligned} B((T_1, x, T_2)) &= B(T_1) + B(T_2), \\ I((T_1, x, T_2)) &= I(T_1) + I(T_2) + 1 \\ \Rightarrow \Delta((T_1, x, T_2)) &= B(T_1) + B(T_2) - (I(T_1) + I(T_2) + 1) \\ &= \Delta(T_1) + \Delta(T_2) - 1 \end{aligned}$$

Nun können wir durch Induktion über die Anzahl  $m$  der gesamten Blätter im Baum zeigen, dass  $\Delta$  immer 1 liefert:

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

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

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

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

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

womit die Behauptung bewiesen ist.  $\square$

### Aufgabe 3.5

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

### Aufgabe 4.1

Die Methode *inclusion* durchläuft beide Mengen solange, bis in einer von beiden ein Element auftritt, das in der anderen Menge nicht vorkommt, danach wird getestet, ob die Menge, die dieses Element enthält, eine Obermenge der anderen ist. Ist dies nicht der Fall, so liegt keine Inklusion vor, andernfalls ist die Menge mit dem zusätzlichen Element die Obermenge. Dies lässt sich sehr leicht rekursiv formulieren, wenn wir annehmen, dass beide Mengen aufsteigend sortiert vorliegen:

```
public static List inclusion(List l1, List l2)
{
    if(l1.isEmpty()) return l2;
    if(l2.isEmpty()) return l1;
    if(l1.first().isEqual(l2.first()))
        return inclusion(l1.rest(), l2.rest());
    if((l1.first().isLess(l2.first())) && (included(l2, l1.rest())))
        return l1;
    if(l2.first().isLess(l1.first()) && (included(l1, l2.rest())))
        return l2;
    return null;
}
```

```
private static boolean included(List l1, List l2)
{
    if(l2.isEmpty())
        if(l1.isEmpty()) return true;
        else return false;
    else
        if(l1.isEmpty()) return true;
        else {
            if(l1.first().isEqual(l2.first()))
                return included(l1.rest(), l2.rest());
            if(l2.first().isLess(l1.first()))
                return included(l1, l2.rest());
            return false;
        }
}
```

### Aufgabe 4.2

Die Pfeile in der Tabelle beschreiben die Verschiebung eines Elementes beim Auftreten von Kollisionen.

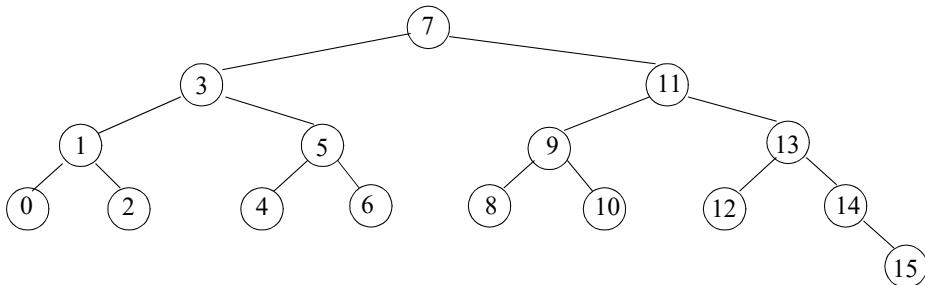
0	November
1	April
2	März
3	
4	
5	Dezember
6	Mai
7	September
8	Januar
9	Juli
10	
11	Juni
12	August
13	Februar
14	
15	
16	Oktober



### Aufgabe 4.3

Offensichtlich tritt nur der Fall (a1) aus Abschnitt 3.2.4 auf, da der Baum nur im äußerst rechten Ast wächst, solange nicht balanciert werden muss.

Nach dem Einfügen der Elemente 2, 4, 5, 6, 8, 9, 10, 11, 12, 13 und 14 muss jeweils rebalanciert werden. Es ergibt sich der folgende Baum:



### Aufgabe 4.4

Wieder implementieren wir die Operation *insert* zunächst als Methode der Klasse *Node*. Anders als beim einfachen binären Suchbaum ist der Rückgabewert hier die Wurzel des Baumes nach dem Einfügen, die beim AVL-Baum nicht konstant bleibt. In der folgenden Implementierung verwenden wir einige Hilfsmethoden:

```
private static int max(int i1, int i2)
```

liefert den größeren der beiden Integer-Parameter zurück,

```
private static void setHeight(Node n)
```

setzt das *height*-Attribut des Knotens *n* auf  $1 + (\text{Maximum der } height\text{-Attribute der Söhne von } n)$ , und

```
private int balance()
```

liefert den Balance-Wert eines Knotens.

Auf dieser Basis implementieren wir die Methode *insert* der Klasse *Node* wie folgt:

```
public Node insert(Elem x)
{
    Node t1, t2;
    if(x.AreEqual(key)) return this;
    else {
        if(x.isLess(key))
            if(left == null) {
                left = new Node(null, x, null);
                height = 2;
                return this;
            }
        else {
            left = left.insert(x);
            setHeight(this);
        }
    else
        if(right == null) {
            right = new Node(null, x, null);
            height = 2;
            return this;
        }
    else {
        right = right.insert(x);
        setHeight(this);
    }
}

if(balance() == -2)           /* rechter Teilbaum gewachsen */
    if(right.balance() == -1){          /* Fall a.1 */
        t1 = right; right = t1.left; t1.left = this;
        setHeight(t1.left); setHeight(t1);
        return t1;
    }
}
```

```

        else {                                     /* Fall a.2 */
            t1 = right; t2 = t1.left;
            t1.left = t2.right; t2.right = t1;
            right = t2.left; t2.left = this;
            setHeight(t2.left); setHeight(t2.right); setHeight(t2);
            return t2;
        }

        if(balance() == 2)                      /* linker Teilbaum gewachsen */
            if(left.balance() == 1)              /* symmetrisch a.1 */
            {
                t1 = left; left = t1.right;
                t1.right = this;
                setHeight(t1.right); setHeight(t1);
                return t1;
            }
            else                                /* symmetrisch a.2 */
            {
                t1 = left; t2 = t1.right;
                t1.right = t2.left; t2.left = t1;
                left = t2.right; t2.right = this;
                setHeight(t2.left); setHeight(t2.right); setHeight(t2);
                return t2;
            }
        /* Keine Verletzung der Balance: */
        setHeight(this);
        return this;
    }
}

```

Die Methode insert der Klasse *Tree* lautet damit:

```

public void insert(Elem x)
{
    if(isempty()) root = new Node(null, x, null);
    else root = root.insert(x);
}

```

### Aufgabe 4.5

Da in Java Arrays nicht mit Position 1 beginnen, sondern 0 der Index des ersten Elements ist, passen wir die Implementierung entsprechend an: Der linke Sohn eines Knotens mit dem Index  $n$  befindet sich nun an Position  $2n + 1$ , der rechte an der Position  $2n + 2$ , und der Index des Vaters errechnet sich zu  $(n - 1) \text{ div } 2$ .

Man muss sich in der Heap-Datenstruktur die erste freie Position merken, da man sonst das Einfügen nicht in logarithmischer Zeit realisieren kann. Dazu dient das Attribut *firstFree*. Für einen leeren Heap hat *firstFree* natürlich den Wert 0.

```
public class Heap
{
    Elem[] field;
    int firstFree;

    public Heap(Elem[] f, int numOfElems) // möglicher Konstruktor
    {
        field = f;
        firstFree = numOfElems;
    }

    public void insert(Elem e)
    {
        Elem a;
        if(firstFree > field.length - 1) // Fehlerbehandlung: Überlauf;
        else
        {
            int i = firstFree++;
            field[i] = e;
            int j = (i - 1) / 2;
            while(i > 0 && field[i].IsLess(field[j]))
            {
                a = field[j]; field[j] = field[i]; field[i] = a;
                i = j; j = (i - 1) / 2;
            }
        }
    }
}
```

```
Elem deletemin()
{
    int i, j;
    Elem a, result;
    boolean cont = true;
    if(firstFree == 0)
    {
        /* Fehlerbehandlung: leerer Heap */
        return null;
    }
    else
    {
        result = field[0];
        field[0] = field[--firstFree];
        i = 0;
        while(2 * i + 1 < firstFree && cont)
            /* mind. 1 Sohn existiert und noch kein Abbruch */
        {
            if (2 * i + 2 == firstFree) /* nur linker Sohn vorhanden */
                j = 2 * i + 1;
            else
                if(field[2 * i + 1].IsLess(field[2 * i + 2]))
                    j = 2 * i + 1;
                else
                    j = 2 * i + 2;

            /* nun ist j der Sohn, mit dem ggf. zu vertauschen ist */
            if(field[j].IsLess(field[i]))
            {
                a = field[i];
                field[i] = field[j];
                field[j] = a;
                i = j;
            }
            else
                cont = false;
        }
    }
}
```

```

        return result;
    }
}

```

### Aufgabe 5.1

Die Methode besteht aus zwei ineinander verschachtelten Schleifen. Die äußere dieser Schleifen läuft über alle Array-Elemente, während die innere Schleife jeweils das aktuelle Element des Arrays solange “aufsteigen” lässt, bis ein Element gefunden wird, das größer ist.

```

public static void bubbleSort(Elem[] S)
{
    for(int i = 1; i < S.length; i++)
        for(int j = S.length - 1; j >= i; j--)
            if(S[j].isLess(S[j - 1]))
                swap(S, j - 1, j);
}

```

Da die äußere Schleife  $n$ -mal ( $n$ : Größe des Arrays), die innere im Mittel  $n/2$ -mal durchlaufen wird, ist die Gesamlaufzeit offensichtlich  $O(n^2)$ .

Graduelle Verbesserungen können z. B. eingefügt werden, indem man sich merkt, ob im letzten Durchlauf überhaupt noch Vertauschungen vorgenommen worden sind und den Lauf abbricht, falls dies nicht der Fall ist.

### Aufgabe 5.2

```

public static void insertionSort(Elem[] S)
{
    Elem r;
    int left, right, m;
    for(int i = 1; i < S.length; i++)
    {
        r = S[i];
        left = 0; right = i - 1;

```

```
while(left <= right)
{
    m = (left + right) / 2;
    if(r.isLess(S[m]))
        right = m - 1;
    else
        left = m + 1;
}
for(int j = i - 1; j >= left; j--)
    S[j + 1] = S[j];
S[left] = r;
}
```

Die Verbesserung ist deshalb nur geringfügig, weil durch sie nur die Anzahl der Zugriffe zum Auffinden der richtigen Position des neuen Elementes herabgesetzt wird. Dabei handelt es sich nur um Lesezugriffe. Die Anzahl der vorgenommenen Verschiebungen von Elementen, d. h. von Schreibzugriffen, lässt sich dadurch nicht herabsetzen.

### Aufgabe 5.3

Im Folgenden gibt jede Zeile den Ausschnitt des Arrays an, der von der jeweiligen Prozedurinkarnation sortiert wird. Das zum Teilen gewählte Element für den nächsten Schritt wird unterstrichen dargestellt.

```
17 24 3 1 12 7 9 5 2 0 24 42 49 46 11
17 11 3 1 12 7 9 5 2 0
0 11 3 1 12 7 9 5 2
0 2 3 1 5 7 9
0 1
0
1
3 2 5 7 9
2
3 5 7 9
3
5 7 9
5
7 9
7
```

9  
12 11  
11  
12  
17  
24 42 49 46 24  
24 24  
49 46 42  
42 46  
42  
46  
49

Zum Beispiel bei der Auswahl von 42 als Trennelement kann man erkennen, dass das minimale Element nicht zur Teilung gewählt werden darf, weil der Algorithmus sonst nicht terminiert.

### Aufgabe 5.4

Das Problem liegt darin, dass bei der Teilung des Arrays im ungünstigsten Fall das teilende Element immer so gewählt wird, dass der zu sortierende Array in ein einzelnes Element und einen weiteren Array zerfällt, der nur um eins kleiner ist als der ursprüngliche. Dieser Fall war in Aufgabe 5.3 mehrfach zu beobachten. Auf diese Weise kann der Stack, d. h. die Anzahl von rekursiven Aufrufen, proportional zur Länge der Eingabe werden.

Die Stacklänge kann auf den Logarithmus der Eingabelänge beschränkt werden, indem die rekursive Version von Quicksort in eine iterative Methode umgeformt wird, in der jeweils die kleinere der entstandenen Partitionen *direkt*, d. h. ohne rekursiven Aufruf, weiterzerlegt wird. Der noch erforderliche rekursive Aufruf für den größeren Teil wird auf einem Stack gespeichert. Nach dem Sortieren des kleineren Teils wird der gemerkte Aufruf für den größeren Teil vom Stack genommen und fortgefahrene. Dieser Stack kann maximal  $O(\log n)$  Länge haben.

### Aufgabe 5.5

Sei  $s_n$  die Teilsummenfolge dieser Reihe, d. h.  $s_n$  ist die Summe ihrer ersten  $n$  Glieder.

$$s_n = \sum_{i=0}^n \frac{i}{2^i}$$

Dann ist

$$(*) \quad s_{n+1} = s_n + \frac{n+1}{2^{n+1}}$$

und mit  $s_0 = 0$

$$\begin{aligned} s_{n+1} &= 0 + \sum_{i=0}^{n+1} \frac{i}{2^i} \\ &= \sum_{i=0}^n \frac{i+1}{2^{i+1}} \\ &= \sum_{i=0}^n \frac{i}{2^{i+1}} + \sum_{i=0}^n \frac{1}{2^{i+1}} \\ &= \frac{1}{2} \left( \sum_{i=0}^n \frac{i}{2^i} + \sum_{i=0}^n \frac{1}{2^i} \right) \\ &= \frac{1}{2} \left( s_n + \sum_{i=0}^n \frac{1}{2^i} \right) \end{aligned}$$

Der verbleibende Summenterm kann gemäß der Formeln für geometrische Reihen direkt berechnet werden. Damit ergibt sich:

$$s_{n+1} = \frac{1}{2} \left( s_n + 2 - \frac{1}{2^n} \right)$$

und mit (\*)

$$\begin{aligned} s_{n+1} &= s_n + \frac{n+1}{2^{n+1}} = \frac{1}{2} \left( s_n + 2 - \frac{1}{2^n} \right) \\ \frac{1}{2} s_n &= 1 - \frac{1+n+1}{2^{n+1}} \end{aligned}$$

und also

$$s_n = 2 - \frac{n+2}{2^n}$$

Mit

$$\lim \frac{n+2}{2^n} \text{ für } n \rightarrow \infty$$

ergibt sich also

$$\lim s_n = 2 \text{ für } n \rightarrow \infty$$

und damit die zu beweisende Gleichung

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$$

### Aufgabe 5.6

In einem solchen Fall bietet es sich an, zunächst in einem sequentiellen Durchlauf die unsortierten Werte zu isolieren, so dass ein (großes) sortiertes und ein (kleines) unsortiertes Feld vorliegt. Das unsortierte Feld kann dann mit einem schnellen internen Verfahren, z. B. Quicksort, sortiert werden und anschließend in einem Merge-Lauf mit den bereits sortierten Werten verschmolzen werden. Wenn man die Laufzeit des internen Algorithmus vernachlässigt, weil nur wenige unsortierte Werte vorliegen, ist die Laufzeit eines solchen Verfahrens offensichtlich  $O(n)$ .

### Aufgabe 6.1

(a)

	a	b	c	d	e	f
a	$\infty$	3	$\infty$	4	$\infty$	5
b	$\infty$	$\infty$	1	$\infty$	$\infty$	1
c	$\infty$	$\infty$	$\infty$	2	$\infty$	$\infty$
d	$\infty$	3	$\infty$	$\infty$	$\infty$	$\infty$
e	$\infty$	$\infty$	$\infty$	3	$\infty$	2
f	$\infty$	$\infty$	$\infty$	2	$\infty$	$\infty$

(b)

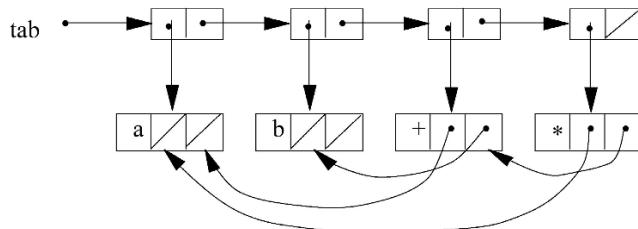
- a → (b, 3) → (d, 4) → (f, 5)
- b → (c, 1) → (f, 1)
- c → (d, 2)
- d → (b, 3)
- e → (d, 3) → (f, 2)
- f → (d, 2)

**Aufgabe 6.2**

Als Datenstruktur für eine Symboltabelle benutzen wir folgenden Typ:

```
type table = list(tree);
```

Damit sind alle Knoten des DAGs durch eine lineare Liste verbunden. Für den Ausdruck  $a * (a + b)$  ergibt sich z. B. am Schluss folgende Struktur:



**algorithm** *DAG* (*t*: Baum)

**input:** Ein Operatorbaum mit Operatoren + und \* sowie Operanden aus der Menge {a, ..., z}

**output:** Ein DAG, der keinen Teilausdruck doppelt enthält.

**method:**

```

if t enthält Operator op then
    l := DAG(left(t));
    r := DAG(right(t));
    return mknnode(op, l, r)
else (* t enthält Operanden x *)
    return mkleaf(x)
end if
end DAG.
```

Dabei werden folgende Hilfsfunktionen verwendet:

*mknoden(op, left, right) = (op: Operator, left, right: Bäume)*

**if** es existiert ein Eintrag in der Symboltabelle, der auf einen DAG-Knoten zeigt,  
der *op*, *left* und *right* enthält

**then** liefere einen Zeiger auf diesen Knoten zurück

**else** erzeuge einen solchen Knoten, trage ein entsprechendes Element in der  
Symboltabelle ein und liefere einen Verweis auf den Knoten zurück

**end if**

*mkleaf(entry) = (entry: Operand)*

**if** es existiert ein Eintrag in der Symboltabelle, der auf einen DAG-Knoten zeigt,  
der *entry* enthält

**then** liefere einen Zeiger auf diesen Knoten zurück

**else** erzeuge einen solchen Knoten, trage ein entsprechendes Element in der  
Symboltabelle ein und liefere einen Verweis auf den Knoten zurück

**end if**

Die Zeitkomplexität ist  $O(n^2)$ , falls  $n$  die Anzahl der Knoten im Baum ist: *DAG* selbst durchläuft alle Knoten genau einmal. Die Suche, ob ein Knoten schon vorhanden ist, dauert dann jeweils  $O(n)$ .

### Aufgabe 7.1

Der Array *father* enthält nach dem Aufruf der Methode *Dijkstra* den Baum der kürzesten Wege (rote Kanten), wobei *father[i]* den Vaterknoten des Knotens *i* in diesem Baum bezeichnet. Zu Beginn wird für alle Knoten der Vater "0" angenommen (Voraussetzung: der Graph ist zusammenhängend). Die gelben Knoten ergeben sich implizit: Ein Knoten *i* ist gelb, wenn *dist[i] ≠ ∞* und *green[i] = false*. Für  $\infty$  setzen wir in der Implementierung *Float.MAX\_VALUE* ein.

```
public static void Dijkstra(float[][] cost, int[] father)
{
    int n = cost.length; /* Zusicherung: cost ist quadratisch und
                           father hat die Länge n */
    float[] dist = new float[n];
    boolean[] green = new boolean[n];
    int w;
    float minDist;
```

```
for(int i = 1; i < n; i++) // dist, green und father initialisieren
{
    dist[i] = cost[0][i];
    father[i] = 0;
    green[i] = false;
}

green[0] = true;
for(int i = 0; i < n - 1; i++) /* finde Knoten w mit minimalem
                                Abstand minDist */
{
    minDist = Float.MAX_VALUE;
    w = 0;
    for(int j = 1; j < n; j++)
        if(dist[j] < minDist && !green[j])
        {
            minDist = dist[j];
            w = j;
        }
    if(w > 0) green[w] = true;
    else return; /* Alle j mit dist[j] ≠∞ sind grün, d.h. GELB ist
                   leer. Falls i < n - 1, ist der Graph nicht zusammenhängend. */

    for(int j = 1; j < n; j++) /* ggf. neuen (kürzeren) Weg
                                eintragen */
    {
        if(dist[j] > dist[w] + cost[w][j])
        {
            dist[j] = dist[w] + cost[w][j];
            father[j] = w;
        }
    }
}
}
```

**Aufgabe 7.2**

- (a) Es genügt ein Gegenbeispiel. Sei  $G$  folgender Graph:



Dann ist  $A_{1,1} = -\infty$ . Bei *Floyd* enthält  $A[1,1]$  aber -2.

- (b) Am Ende des Algorithmus von Floyd müssen die falschen Werte korrigiert werden. Es werden also folgende Zeilen hinten angefügt:

```

 $N = \{i \mid A_{i,i} < 0\};$ 
for each  $i \in N$  do
    for  $j := 1$  to  $n$  do
        if  $A_{i,j} < \infty$  then  $A_{i,j} := -\infty$  end if;
        if  $A_{j,i} < \infty$  then  $A_{j,i} := -\infty$  end if;
    end for;
    for each  $k \in V \setminus N$  do
        for each  $j \in V \setminus N$  do
            if  $A_{k,i} + A_{i,j} < \infty$  then  $A_{k,j} := -\infty$  end if;
        end for;
    end for;
end for;

```

**Aufgabe 7.3**

```

public static void path(int[][] P, int i, int j)
{
    int k = P[i][j];
    if(k != -1)
    {
        path(P, i, k);
        System.out.println(k);
        path(P, k, j);
    }
}

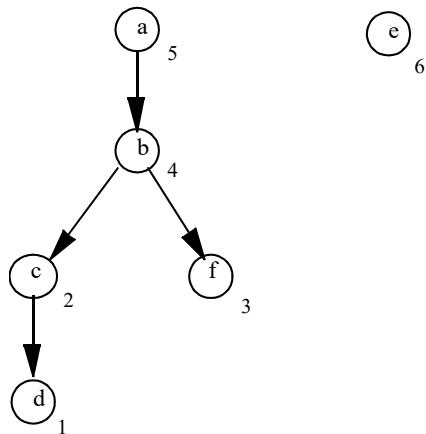
```

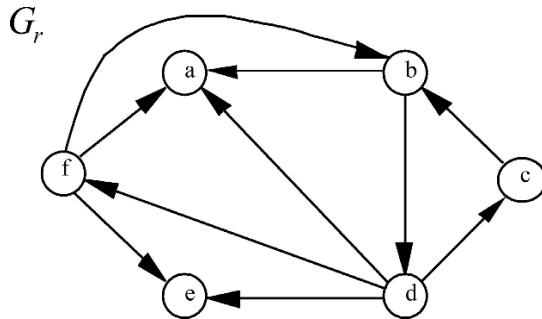
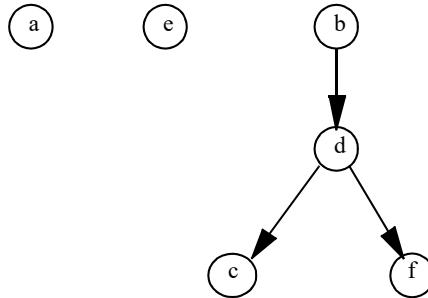
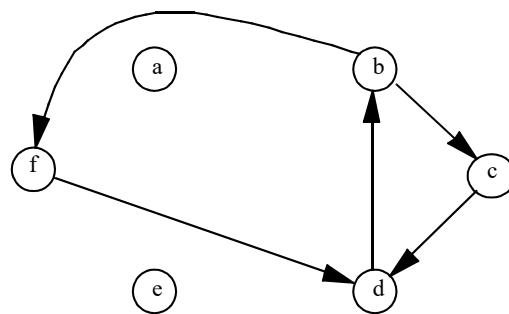
**Aufgabe 7.4**

```
public static void Warshall(boolean[][] A, boolean[][] C)
{
    int n = A.length; /* Zusicherung: A und C sind quadratisch und
                       gleich groß */
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            A[i][j] = C[i][j];
    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                if(!A[i][j]) A[i][j] = A[i][k] && A[k][j];
}
```

**Aufgabe 7.5**

## 1. Depth-First-Spannbäume und Nummerierung



2. Inverser Graph  $G_r$ 3. Spannbäume für  $G_r$ 4. Starke Komponenten von  $G$ 

**Aufgabe 7.6**

Sei  $|V| = n$  und  $|E| = e$ . Der Algorithmus hat bei Adjazenzlistendarstellung eine Komplexität von  $O(n+e)$ .

Für den Aufbau der Spannbäume über die Tiefensuche wird  $O(n+e)$  Zeit benötigt, da *dfs* für jeden Knoten einmal aufgerufen wird und alle Kanten einmal betrachtet werden. Bei der Konstruktion des inversen Graphen müssen ebenfalls alle Kanten einmal betrachtet werden, was in  $O(e)$  Zeit möglich ist, so dass sich insgesamt ein Zeitbedarf von  $O(n+e)$  ergibt.

**Aufgabe 7.7**

Für den Test kann man den Tiefendurchlauf verwenden, der so modifiziert wird, dass er abbricht, sobald er auf einen bereits als besucht markierten Knoten trifft, und dann ausgibt, dass es einen Zyklus gibt. Wenn er andererseits alle Knoten genau einmal besucht hat, gibt es keinen Zyklus.

Dieser Algorithmus hat eine Zeitkomplexität von  $O(n)$ , da mit jedem Schritt entweder ein noch nicht besuchter Knoten gefunden oder bei einem bereits besuchten abgebrochen wird.

**Aufgabe 8.1**

Im nachfolgenden Algorithmus, der eine Modifikation des im Text vorgestellten Algorithmus darstellt, sind die gegenüber dem ursprünglichen Algorithmus geänderten oder hinzugefügten Stellen durch Fettdruck hervorgehoben. Die wesentliche Änderung ist die Sortierung von  $S$ , so dass jetzt bei gleichen  $x$ -Koordinaten von Endpunkten oder vertikalen Segmenten diese gemäß der 2. bzw. 3. Priorität geordnet werden. Diese Sortierung stellt sicher, dass auch tatsächlich alle Schnitte gefunden werden können (auch die Randfälle).

**Algorithmus Segmentschnitt**

**Input:** Eine Menge  $L = H \cup V$  horizontaler und vertikaler Segmente.

**Output:**  $\{ (s, s') \mid s \in L, s' \in L \wedge s \text{ schneidet } s' \}$

**Method**

*Schritt 1:* Sei  $H'$  die Menge linker und rechter Endpunkte von Segmenten in  $H$ , d. h. jedes Segment ist je einmal für jeden Anfangs- und Endpunkt aufgeführt.

**Sei  $V'$  die Menge unterer Endpunkte von Segmenten in  $V$ , d. h. vertikale Elemente werden im Folgenden nach ihren unteren Endpunkten sortiert.**

Sei  $S = H \cup V$ .

**Sortiere  $S$  nach**

- **$x$ -Koordinaten (1. Priorität),**
- **linker Endpunkt / vertikales Segment / rechter Endpunkt (2. Priorität) und**
- **$y$ -Koordinaten(3. Priorität).**

**Schritt 2:** Sei  $Y$  eine Menge von  $y$ -Koordinaten.  $Y \leftarrow \emptyset$ . Durchlaufe  $S$ . Das gerade erreichte Objekt ist:

- (a) linker Endpunkt eines horizontalen Segments  $h = (x_1, x_2, y)$ :  
**Teste, ob  $h$  ein horizontales Segment schneidet:**  
**Finde alle  $y' \in Y$  mit  $y' = y$  und gib die Paare  $(h, h')$  aus, wobei  $h' = (x_1', x_2', y')$ .**  
 $Y \leftarrow Y \cup \{y\}$  (füge  $y$  in  $Y$  ein)
- (b) rechter Endpunkt eines horizontalen Segments  $h = (x_1, x_2, y)$ :  
 $Y \leftarrow Y \setminus \{y\}$  (entferne  $y$  aus  $Y$ )
- (c) ein vertikales Segment  $v = (x, y_1, y_2)$ :  
**Durchlaufe die nächsten Elemente  $s \in S$ , solange  $s$  vertikales Segment ist und  $v$  schneidet. Gib alle so gefundenen Paare  $(v, s)$  aus (Überlappungen von vertikalen Segmenten).**  
 $A \leftarrow \{y \in Y \mid y \in [y_1, y_2]\}$   
(finde alle  $y$ -Koordinaten im  $y$ -Intervall von  $v$ )  
Gib alle Paare  $(u, v)$  mit  $u, v \in A$  aus.

**end** Segmentschnitt.

Die Datenstruktur für  $Y$  soll wieder eine Variante des AVL-Baums sein. Hier können jetzt aber bis zu  $c$  identische  $y$ -Koordinaten auftreten, die alle in den Baum eingetragen werden müssen. Da dieser nach  $y$  sortiert ist, kann man sie jedoch so nicht mehr unterscheiden. Daher wird der Baum so modifiziert, dass an den Blättern verkettete Listen hängen, die die jeweils identischen Koordinaten aufnehmen. Dann sind die notwendigen Operationen folgendermaßen durchzuführen:

- (a) Das Einfügen kann durch Anhängen der neuen Koordinate vor die jeweilige Liste durchgeführt werden.
- (b) Beim Entfernen muss das entsprechende Element erst in der Liste gesucht werden.
- (c) Bei der Ausgabe (möglicherweise eines Identifikators) müssen alle betroffenen Listen vollständig durchlaufen werden.

Das Einfügen ist damit noch immer in  $O(\log n)$  Zeit möglich. Das Entfernen braucht im Prinzip  $O(\log n + c)$  Zeit, da wir angenommen haben, dass  $c$  eine kleine Konstante ist, also  $O(\log n)$  Zeit. Das vollständige Durchlaufen der Listen bei der Ausgabe schadet nichts, da alle besuchten Elemente zum Ergebnis gehören. Insgesamt hat der modifizierte Algorithmus also die gleiche Laufzeit wie der ursprüngliche.

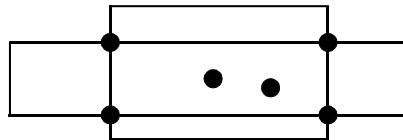
### Aufgabe 8.2

Das Problem beim Löschen eines Intervalls  $i$  aus dem Segment-Baum besteht im Auffinden des Intervall-Namens in den Listen der Knoten aus  $CN(i)$ . Eine sequentielle Suche würde im worst case eine Laufzeit von  $O(n)$  pro Liste – also insgesamt  $O(n \log n)$  – haben.

Daher wird eine zusätzliche Tabelle (z. B. als AVL-Baum mit maximal  $n$  Einträgen) verwaltet, deren Einträge für jedes aktuell gespeicherte Intervall aus einer verketteten Liste bestehen, deren Elemente direkt auf die Knoten in den oben erwähnten Listen von  $CN(i)$  zeigen. Ein Intervallname kann in dieser Tabelle in  $O(\log n)$  Zeit gefunden werden. Daher braucht das Löschen insgesamt auch nur  $O(\log n)$  Zeit.

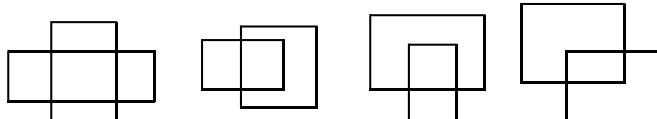
### Aufgabe 8.3

Bei dem beschriebenen Algorithmus kann der Schnitt zwischen zwei Rechtecken bis zu sechsmal gefunden (und berichtet) werden:



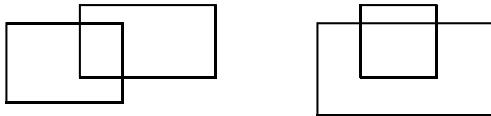
Die Schnitte zweier Rechtecke können nach 4 Fällen unterschieden werden:

- (a) Kein linker unterer Eckpunkt eines Rechtecks liegt im anderen Rechteck.



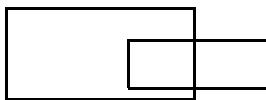
Dann schneidet genau eine linke Kante eine untere Kante.

- (b) Der linke untere Eckpunkt des einen Rechtecks liegt im anderen Rechteck, der linke obere Eckpunkt aber nicht.



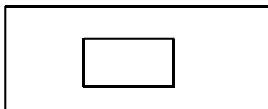
Dann schneidet eine linke Kante eine obere Kante.

- (c) Der linke untere und der linke obere Eckpunkt eines Rechtecks liegen in dem anderen.



Dann schneidet eine rechte Kante eine untere Kante.

- (d) Die Rechtecke umschließen sich.



Dann schneiden sich keine der Kanten.

Mehrfachausgaben können durch einfache Tests bei Segmentschnitt und Punkteinschluss unterdrückt werden:

Punkteinschluss:

Test, ob für die beiden Rechtecke Fall (d) vorliegt. Nur dann erfolgt die Ausgabe, und zwar nur für den Punkt des inneren Rechtecks.

Segmentschnitt:

Fall	Ausgabe nur für Schnitt
(a)	linke / untere Kante
(b)	linke / obere Kante
(c)	rechte / untere Kante

Es ergibt sich weder ein zusätzlicher Platzbedarf noch eine asymptotisch schlechtere Laufzeit.

**Aufgabe 8.4**

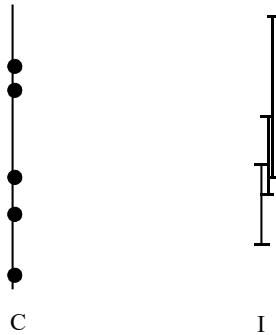
Die interessierenden Anweisungen in LINSECT sind:

- (a) output  $((L_1 \setminus LR) \otimes V_2)$ ;
- (b) output  $((R_2 \setminus LR) \otimes V_1)$

$C := L_1 \setminus LR$  bzw.  $C := R_2 \setminus LR$  sind nach  $y$ -Koordinaten sortierte verkettete Listen horizontaler Segmente. Anzahl:  $|C|$ .

$I := V_1$  bzw.  $I := V_2$  sind nach unterer Intervallgrenze ( $y_1$ ) sortierte verkettete Listen vertikaler Segmente. Anzahl:  $|I|$ .

Die Berechnung wird dann folgendermaßen durchgeführt:  $C$  und  $I$  werden parallel durchlaufen. Für jedes Intervall  $v = [y_1, y_2]$  aus  $I$  wird der (Haupt-) Durchlauf in dieser Liste angehalten. Von der aktuellen Position in  $C$  (die zu  $y_1$  gehört) wird ein *Ausgabedurchlauf* durchgeführt, der ein Paar  $(h, v)$  für jeden gefundenen Punkt  $h = (x', y')$  aus  $C$  ausgibt. Dieser Durchlauf endet, sobald eine  $y$ -Koordinate  $y' > y_2$  gefunden wird. Danach wird der Hauptdurchlauf mit dem nächsten  $v$  aus  $I$  fortgesetzt.



Lösungsskizze:

```

 $h := \text{first}(C);$ 
 $v := \text{first}(I);$ 
while ( $h \neq \text{nil}$  and  $v \neq \text{nil}$ ) do
    if  $h \uparrow .y < v \uparrow .y_1$  then
         $h := \text{next}(C, h)$ 
    elsif  $h \uparrow .y > v \uparrow .y_2$  then
         $v := \text{next}(I, v)$ 
    else  $h' := h;$ 

```

```

while  $v \uparrow.y_1 \leq h' \uparrow.y \leq v \uparrow.y_2$  do
    output( $h'$ ,  $v$ );
     $h' := next(C, h')$ ;
end while;
     $v := next(I, v)$ 
end if
end while

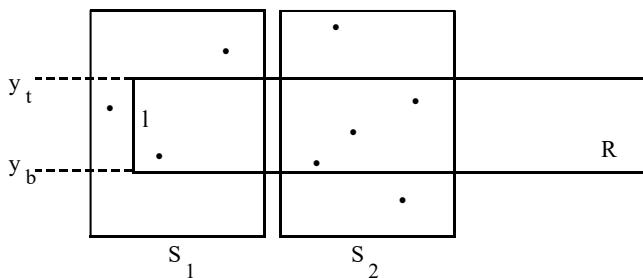
```

Die in der Aufgabenstellung geforderte Laufzeit ist offensichtlich.

### Aufgabe 8.5

Getrennte Repräsentation liefert – angewandt auf die gegebene Menge – eine Menge von linken und rechten Rechteckkanten sowie von Punkten, die nach der  $x$ -Koordinate sortiert wird. POINT\_ENCL wird auf die entstehende Menge angewendet.

POINT\_ENCL teilt eine Menge  $S$  in zwei Teilmengen  $S_1$  und  $S_2$  auf und berechnet rekursiv  $y$ -sortierte Intervallmengen  $L_i$  (die  $y$ -Projektionen der linken Rechteckkanten),  $R_i$  (die  $y$ -Projektionen der rechten Kanten) und eine Menge von  $y$ -Koordinaten  $P_i$  ( $y$ -Projektionen von Punkten) aus  $S_i$ . Der Merge-Schritt, der  $L$ ,  $R$  und  $P$  aus  $L_i$ ,  $R_i$  und  $P_i$  ( $i = 1,2$ ) berechnet, ist wiederum der etwas komplexere Schritt. Dabei wird von folgender Beobachtung Gebrauch gemacht:



Falls eine linke vertikale Kante  $l \in S_1$  eines Rechtecks  $R = (x_l, x_r, y_b, y_t)$  keinen Partner in  $S_2$  hat, durchqueren die untere und die obere Kante von  $R$   $S_2$  vollständig. Das heißt aber, dass alle Punkte aus  $P_2$  innerhalb des Intervalls  $[y_b, y_t]$  in  $R$  liegen.  $[y_b, y_t]$  ist gerade die  $y$ -Projektion von  $l$  in  $L_i$ .

Die Operation  $P \otimes I$  entspricht derjenigen aus Abschnitt 8.2. Damit kommt man zu folgendem Algorithmus, der eine einfache Anpassung des LINSECT-Algorithmus darstellt.

**Algorithmus** DAC-Punkteinschluss**Input:**  $P$  - eine Menge von Punkten $R$  - eine Menge von Rechtecken**Output:** Die Menge aller Paare  $(p, r)$  mit  $p \in P, r \in R$  und  $p$  liegt in  $r$ .**Method:**

*Schritt 1:* Sei  $R'$  die Menge der von  $R$  definierten linken und rechten Rechteckkanten.

Sei  $S = R' \cup P$ , sortiert nach der  $x$ -Koordinate.

*Schritt 2:* POINT\_ENCL(S, LEFT, RIGHT, POINTS)

**end** DAC-Punkteinschluss.

Die rekursive Invariante lautet für POINT\_ENCL:

Bei der Beendigung hat  $POINT\_ENCL(S, \dots)$  alle Punkteinschlüsse innerhalb von  $S$  ausgegeben, d.h alle Paare  $(p, r)$ , wobei  $p \in P \cap S, r \in R$  und  $r$  ist in  $S$  durch seine linke oder rechte Kante repräsentiert.

**Algorithmus** POINT\_ENCL( $S, L, R, P$ )**Input:**  $S$  - eine nach  $x$ -Koordinate sortierte Menge von Rechteckkanten und Punkten.**Output:**  $L$  - eine Menge von  $y$ -Koordinaten.  $L$  enthält die  $y$ -Projektion aller linken Rechteckkanten in  $S$ , deren Partner nicht in  $S$  liegt. $R$  - analog für rechte Rechteckkanten. $P$  - eine Menge von Punkten.  $P$  enthält die  $y$ -Projektion von  $S \cap P$ .

Die Menge aller Paare  $(p, r)$ , wobei  $p$  ein Punkt und  $r$  ein in  $S$  repräsentiertes Rechteck ist.

**Method:**

*Fall 1:*  $S$  enthält nur ein Element  $s$ .

(1a)  $s = (x, y_1, y_2)$  ist linke Rechteckkante.

$L \leftarrow \{[y_1, y_2]\}; R \leftarrow \emptyset; P \leftarrow \emptyset$

(1b)  $s = (x, y_1, y_2)$  ist rechte Rechteckkante.

$L \leftarrow \emptyset; R \leftarrow \{[y_1, y_2]\}; P \leftarrow \emptyset$

(1c)  $s = (x, y)$  ist ein Punkt

$L \leftarrow \emptyset; R \leftarrow \emptyset; P \leftarrow \{y\}$

*Fall 2:*  $S$  enthält mehr als ein Element.

*Divide:* Wähle eine  $x$ -Koordinate  $x_m$ , die  $S$  in zwei etwa gleich große Teilmengen  $S_1$  und  $S_2$  teilt.

*Conquer:*  $POINT\_ENCL(S_1, L_1, R_1, P_1);$   
 $POINT\_ENCL(S_2, L_2, R_2, P_2)$

*Merge:*  $LR \leftarrow L_1 \cap R_2;$

$L \leftarrow (L_1 \setminus LR) \cup L_2;$   
 $R \leftarrow R_1 \cup (R_2 \setminus LR);$   
 $P \leftarrow P_1 \cup P_2;$

*output*  $(P_2 \otimes (L_1 \setminus LR));$   
*output*  $(P_1 \otimes (R_2 \setminus LR))$

**end** *POINT\_ENCL.*

### Aufgabe 8.6

Statt einer Liste  $SUB(p)$  wird jedem Knoten  $p$  des Baumes nur eine Zahl  $|SUB(p)|$  zugeordnet, d. h., es werden keine Koordinatenlisten mehr gespeichert. Diese Zahl lässt sich beim Einfügen und Löschen leicht aufrecht erhalten. Der Platzbedarf ist dann  $O(N)$ .

### Aufgabe 8.7

(a) Ein Quader sei definiert durch  $q = (x_1, x_2, y_1, y_2, z_1, z_2)$ , ein Query-Segment sei definiert durch  $h = (x_1', x_2', y, z)$ .

Die erforderlichen Suchvorgänge sind:

- $y \in [y_1, y_2]$ ,
- $z \in [z_1, z_2]$  und
- $[x_1', x_2']$  schneidet  $[x_1, x_2]$ .

Eine mögliche Hierarchie dafür ist:

1. Level: *Segment-Baum* für  $y$  und
2. Level: *Segment-Baum* für  $z$  und
3. Level: *Intervall-Baum* für  $x$

Die Suchzeit ist  $O(\log^3 n + t)$ , der Platzbedarf  $O(n \log^2 n)$ .

(b) Ein Punkt sei definiert durch  $p = (x, y)$ , ein vertikales Query-Segment sei definiert durch  $v = (x', y_1, y_2)$ .

Die erforderlichen Suchvorgänge sind:

- $y \in [y_1, y_2]$ .und
- $x > x'$

Eine mögliche Hierarchie hierfür ist:

1. Level: *Segment-Baum* für  $y$
2. Level: *binärer Baum* für  $x$

Die Suchzeit ist  $O(\log^2 n + t)$ , der Platzbedarf  $O(n \log n)$ .

### Aufgabe 9.1

Beim Einfügen darf Teilen von inneren Knoten erst dann vorgenommen werden, wenn der zu teilende Knoten einen vollständig gefüllten Bruder hat, dann können diese beiden Knoten auf drei neue verteilt werden, die jeweils mindestens  $\frac{4}{3} m$  Einträge haben. Bevor dieser Zustand eintritt, muss zwischen benachbarten Knoten ausgeglichen werden.

Vorteilhaft ist die bessere Speicherplatzausnutzung, die nun bei mindestens 66% liegt. Nachteilig ist allerdings, dass man dafür häufigeres Umverteilen der Knoteneinträge in Kauf nehmen muss.

### Aufgabe 9.2

Der Inhalt eines Files ist jeweils in einer Spalte dargestellt. Das Ende eines Runs ist durch    markiert.

Anfangsaufteilung:

10	3	44	32
<u>44</u>	<u>12</u>	<u>33</u>	<u>12</u>
<u>12</u>	<u>65</u>	<u>78</u>	<u>165</u>
<u>4</u>	<u>9</u>	<u>7</u>	<u>33</u>
<u>87</u>	<u>2</u>	<u>9</u>	<u>  </u>

Nach dem ersten Durchlauf:

3	12	12	4
10	12	65	7

32	33	78	9
44	44	165	33
—	—	—	—
2			
9			
87			

Nach dem zweiten Durchlauf:

3	2		
4	9		
7	87		
9			
10			
12			
12			
12			
32			
33			
33			
44			
44			
65			
78			
165			

Der dritte Durchlauf liefert die vollständige Sortierung. Im Gegensatz dazu benötigt normales binäres Mischen - wie erwartet - einen Durchlauf mehr. Wendet man jedoch die Technik der Ersetzungsauswahl an, so werden zusätzlich zur Erzeugung der Anfangsrums nur zwei Durchläufe benötigt, da die folgenden Anfangsrums entstehen:

3	12		
10	12		
32	12		
44	33		
44	65		
—	78		
—	165		

$$\begin{array}{r} 4 & 2 \\ 7 & 9 \\ 9 & - \\ 33 & \\ 87 & \\ \hline \end{array}$$

Bei größerem  $k$  und längeren Eingaben entstehen aber dennoch deutliche Effizienzunterschiede zwischen diesen beiden Verfahren.

## Literatur

- Abraham, I., D. Delling, A.V. Goldberg und R.F. Werneck [2011]. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. 10th Intl. Symposium on Experimental Algorithms, 230-241.
- Abraham, I., D. Delling, A. Fiat, A.V. Goldberg und R.F. Werneck [2012]. HLDB: Location-Based Services in Databases. SIGSPATIAL Intl. Conference on Advances in Geographic Information Systems (SIGSPATIAL GIS), 339-348.
- 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*, 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.
- Bast, H., S. Funke, D. Matijevic, P. Sanders und D. Schultes [2007]. In Transit to Constant Time Shortest Path Queries. Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments, ALENEX, 46-59.
- Bauer, F. L. und H. Wössner [1984]. Algorithmische Sprache und Programmierung. 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, Manuscript.

- 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.
- Cormen, T.H., C.E. Leiserson, R.L. Rivest, und C. Stein [2010]. Algorithmen - Eine Einführung. 3. Aufl., Oldenbourg-Verlag, München.
- 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.
- de Berg, M., O. Cheong, M. van Kreveld, und M. Overmars [2008]. Computational Geometry: Algorithms and Applications. 3rd Ed., Springer-Verlag, Berlin.
- 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.
- Evans, B.J. und D. Flanagan [2014]. Java in a Nutshell, Sixth 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.
- Geisberger, R. [2008]. Contraction Hierarchies: Faster and Simpler Routing in Road Networks. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe. [http://algo2.iti.kit.edu/documents/routeplanning/geisberger\\_dipl.pdf](http://algo2.iti.kit.edu/documents/routeplanning/geisberger_dipl.pdf).
- Geisberger, R., P. Sanders, D. Schultes und C. Vetter [2012]. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science* 46, 388-404.
- Geisberger, R., P. Sanders und D. Schultes [2008]. Contraction Hierarchies Source Code. <http://algo2.iti.kit.edu/routeplanning.php>
- 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.
- Güting, R.H. [1984a]. An Optimal Contour Algorithm for Iso-Oriented Rectangles. *Journal of Algorithms* 5, 303-326.
- Güting, R.H. [1984b]. Optimal Divide-and-Conquer to Compute Measure and Contour for a Set of Iso-Rectangles. *Acta Informatica* 21, 271-291.
- Güting, R.H. [1985]. Divide-and-Conquer in Planar Geometry. *International Journal of Computer Mathematics* 18, 247-263.
- Güting, R.H., und D. Wood [1984]. Finding Rectangle Intersections by Divide-and-Conquer. *IEEE Transactions on Computers* C-33, 671-675.
- Güting, 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 [2007]. Fundamentals of Data Structures in C. 2nd Ed., Silicon Press, Summit, NJ.
- 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. [2012]. Graphs, Networks and Algorithms. Fourth Ed., 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. [2005]. Algorithmische Geometrie: Grundlagen, Methoden, Anwendungen. 2. Aufl., Springer-Verlag, Berlin..
- 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. und T. Stark [2009]. Handbuch der Java-Programmierung: Standard Edition Version 6. 6. Aufl, Addison-Wesley Longman Verlag, 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.
- Madkour, A., W.G. Aref, F.U. Rehman, M.A. Rahman und S. Basalamah [2017]. A Survey of Shortest-Path Algorithms. Computing Research Repository (CoRR), May 2017, <http://arxiv.org/abs/1705.02044v1>.
- 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. [2003]. Softwaretechnik mit Ada 95: Entwicklung großer Systeme. 2. Aufl., Vieweg+Teubner Verlag, Wiesbaden.
- Nakamura, T., und T. Mizogushi [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 [2011]. Programmierung mit PASCAL. 8. Aufl., Vieweg+Teubner-Verlag, Wiesbaden.
- Ottmann, T., und P. Widmayer [2012]. Algorithmen und Datenstrukturen. 5. Aufl., Spektrum Akademischer Verlag, Heidelberg.

- 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.
- Saake, G., und K.U. Sattler [2013]. Algorithmen und Datenstrukturen. Eine Einführung mit Java. 5. Aufl., dpunkt.verlag, Heidelberg.
- 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.
- Samet, H., J. Sankaranarayanan und H. Alborzi [2008]. Scalable Network Distance Browsing in Spatial Databases. Proceedings of the ACM SIGMOD Intl. Conference on Management of Data, 43-54.
- Schiedermeier, R. [2010]. Programmieren mit Java. Eine methodische Einführung. 2. Aufl., 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. [2009]. Data Structures and Problem Solving Using Java. 4. Aufl., 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. [2010]. Introduction to Graph Theory. 5th 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. [1996]. Algorithmen und Datenstrukturen mit Modula-2. 5. Aufl., Teubner-Verlag, Stuttgart.
- Wirth, N. [2000]. Algorithmen und Datenstrukturen. Pascal-Version. 5. Aufl., Teubner-Verlag, Stuttgart.
- Wood, D. [1993]. Data Structures, Algorithms, and Performance. Addison-Wesley Publishing Co., Reading, Massachusetts.
- 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

$\Omega$ -Notation 20

## Numerisch

2-3-Baum 320

## A

$A^*$ , Algorithmus 225, 247

Abbildung 91, 234

Abkömmling 95

abstrakter Datentyp 1, 2, 35, 38

Abstraktionsebene 1

Abwärtsgraph 230

Abwärtskante 229

Ada 61

addcomp 158

adjazent 202

Adjazenzlisten 206, 223

Adjazenzmatrix 204, 218

ADT 35

Aggregation 39

Akkumulator 7

aktives Segment 299

Aktivierungs-Record 86

Algebra 1, 2, 23

algebraische Spezifikation 38

algebraischer Entscheidungsbaum 199

algorithmische Geometrie 249

Algorithmus 1, 33

all pairs shortest path-Problem 220

allgemeiner Baum 101

allgemeiner Suchbaum 310

allgemeines Sortierverfahren 190

amortisierte Laufzeit 160

Analyse 2

Analyse von Algorithmen 38

ancestor 95

append 64, 82

Äquivalenzrelation 156

Array 39

Assemblersprache 7

atomarer Datentyp 41

Aufwärtsgraph 230

Aufwärtskante 228

Aufzählungstyp 59

Ausgangsgrad 202

average case 10

AVL-Baum 109, 141, 184, 234, 257

Axiom 24, 35

## B

Backward 231

Backward-Label 232

Bag 152

balance 317

balancierter Suchbaum 141

Baum 92

Baum beschränkter Balance 166

Baumhierarchie 290, 296

Baumsortieren 184

B-Baum 311

Behälter 115, 194

beliebig orientierte Segmente 299

beliebig orientiertes Objekt 252

best case 10

bester Fall 10

Betriebssystem 52

bidijkstra 233

bidijkstra, Algorithmus 233

Bidirektonaler Algorithmus von Dijkstra  
232

binäre Suche 17

binärer Suchbaum 109, 129, 290

Bitvektor-Darstellung 110

Blatt 94

Block 309  
 bol 66  
 Bottom-Up-Heapsort 188  
 breadth-first 240  
 breadth-first-Spannbaum 209  
 breadth-first-traversal 209  
 Breitendurchlauf 207, 209  
 Bruder 95  
 BubbleSort 173  
 Bucket 194  
 bucket 115  
 BucketSort 195

**C**

Clever Quicksort 183  
 CN(i) 263, 269  
 computational geometry 249  
 concat 64, 82  
 contracted segment tree 306  
 Contraction Hierarchy 226

**D**

DAC-Algorithmus 177  
 DAG 211  
 Datenobjekt 28  
 Datenspeicher 7  
 Datenstruktur 1, 2, 22, 35  
 Datentyp 1, 23, 35  
 Definitionsmodul 29  
 degenerierter binärer Suchbaum 135  
 delete 66, 113, 133, 314  
 deletemin 153, 155  
 denotationelle Spezifikation 38  
 depth-first 240  
 depth-first-Spannbaum 209  
 depth-first-traversal 208  
 dequeue 89  
 Dereferenzierung 50  
 descendant 95  
 Dictionary 109, 113, 234  
 difference 110

Dijkstra, Algorithmus von 213, 220, 225  
 directed acyclic graph 211  
 direktes Auswählen 170  
 direktes Mischen 323  
 dispose 52  
 Distanzproblem 305  
 Divide-and-Conquer 173, 253, 289  
 Divide-and-Conquer-Algorithmus 270  
 Divisionsmethode 128  
 domain 91  
 Doppel-Hashing 127  
 Doppelrotation 143  
 Duplikat 63  
 dynamisch 290

**E**

eindimensionales Punkteinschluß-Mengenproblem 276  
 einfacher Pfad 203  
 Eingangsgrad 202  
 Einheitskosten 7  
 Elementaroperation 6, 7  
 EMergeSort 322  
 empty 64, 109  
 enqueue 89  
 Entscheidungsbaum 191  
 enumerate 109, 110  
 eol 66  
 Ersetzungs-Auswahl 324  
 Erwartungswert 332  
 Euler-Konstante 335  
 Expansion eines Graphen 207  
 exponentiell 15  
 extern 169  
 externe Datenstruktur 309  
 externer Algorithmus 309  
 externes Verfahren 169  
 Exzentrizität 245

**F**

Feld 42

Fibonacci-Zahlen 150, 337  
FIFO 89  
find 66, 157, 158, 161  
findx 182  
first 64, 82  
Floyd, Algorithmus von 220, 225  
Forward 230  
Forward-Label 232  
Fragmentintervall 292  
freier Baum 240  
front 65, 89  
Funktion 1, 3, 23

## G

Garbage Collection 52  
Geburtstagsparadoxon 117  
gegabelter Pfad 263, 269  
gerichteter azyklischer Graph 211  
gerichteter Graph 201, 202  
geschlossenes Hashing 116, 118  
Gesetz 24  
getrennte Darstellung 271  
Gewicht eines Baumes 165  
gewichtsbalancierter Baum 166  
Gleichverteilung 10  
Grad eines Baumes 102  
Grad eines Knotens 102, 202  
Graph 201

## H

Halbrechteck 307  
Haltepunkt 300  
harmonische Zahl 125, 335  
Hashfunktion 115, 128  
Hashing 109  
Haufen 153  
Heap 153  
Heapsort 153, 173, 184  
heterogene Algebra 23, 35  
Heuristikfunktion 247  
Höhe eines Baumes 95

Hub Labeling 232

## I

ideales Hashing 120  
in situ 169, 199  
index sequential access method 329  
Indextyp 42  
Infix-Notation 99  
InitialRuns 321, 323  
innerer Knoten 95  
inorder 97  
Inorder-Durchlauf 184  
insert 66, 110, 113, 132, 153, 154, 314  
InsertionSort 170, 173  
Instruktion 7  
intern 169  
internes Verfahren 169  
intersection 110  
Intervall-Baum 290, 292  
Intervallschnitt-Suche 294  
inverse Adjazenzliste 206  
inzident 202  
ISAM-Technik 329  
isempty 64  
isomorph 25

## K

kanonisch bedeckte Knoten 263  
Kante 93, 202  
Kaskaden-Mergesort 328  
key 98  
key-Komponente 169, 184  
Klammerstruktur 84, 93  
Knoten 93, 202  
Knotenliste 262  
Knotenmarkierung 129  
Kollision 116  
Komplexität der Eingabe 6, 21  
Komplexität des Problems 20  
Komplexitätsklasse 9, 20  
Komponente 157, 240

Königsberger Brückenproblem 246  
 Königsberger Brückenproblem 211  
 konstant 15  
 Kontraktion 227  
 Kontraktionshierarchie 226, 227  
 Kontur 278, 284  
 Konturproblem 284  
 Konturzyklus 284  
 konvexe Hülle 252, 305  
 Korrektheit 5  
 Kostenmaß 7, 309  
 Kostenmatrix 222  
 Kruskal, Algorithmus von 242  
 Kürzeste-Wege-Baum 234

**L**

Label 232  
 Länge eines Pfades 95  
 last 65  
 Lauf 321  
 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 271, 275  
 Liste 64  
 Liste im Array 78  
 Listenkopf 107  
 Listenschwanz 107  
 logarithmisch 15  
 logarithmisches Kostenmaß 7

**M**

maketree 98  
 mapping 91  
 markierte Adjazenzmatrix 205

markierter Graph 204  
 Maschinenmodell 38  
 Maschinensprache 7  
 Maßproblem 266  
 Mehrphasen-Mischsortieren 329  
 mehrsortige Algebra 23, 35  
 member 113, 114, 131  
 member-Problem 251  
 Menge 109  
 Mengenoperation 61  
 Merge 322  
 merge 157, 158, 159, 161, 316, 317  
 MergeSort 174  
 Mergesort 173, 320  
 minimaler Spannbaum 240  
 Mittel-Quadrat-Methode 128  
 Modell 25, 35  
 modifizierter Segment-Baum 268, 269  
 Modul 1, 2  
 monomorph 25, 35  
 Multimenge 152  
 Multiset 152

**N**

Nachbar 316  
 Nachfahr 95  
 Nachfolger 58  
 natürliches Mischen 323  
 nearest-neighbour-Problem 253  
 next 66  
 nil 49

**O**

offene Adressierung, 119  
 offenes Hashing 116, 117  
 offset 48  
 O-Notation 11, 12, 15, 19, 21  
 OpenStreetMap 231  
 Operandenstack 84  
 Operation 22, 34  
 Operationssymbol 23, 34

Operatorenstack 84  
optimal 20  
Ordnung 63, 110  
orthogonal 252  
Overflow 314, 315, 316  
overflow 116

**P**

partiell geordneter Baum 153  
Partition 156  
partition 158  
PASCAL 61  
Permutation 169, 192  
persistent 309  
Pfad 95, 202  
Pfadexpansion 235  
Pfadkompression 162  
Phase 320  
Plane-Sweep 253, 289  
Plattenspeicher 309  
Platzbedarf 2, 6  
Pointer 49  
Polygonschnitt-Algorithmus 307  
polymorph 25, 35  
Polynom 106  
polynomiell 15  
polyphase merging 329  
pop 82  
Postfix-Notation 99  
postorder 97  
pqueue 153  
Präfix-Notation 99  
pred 58  
preorder 97  
preprocessing 213  
previous 66  
Primärkollision 127  
Priorität 152  
Prioritätssuchbaum 307  
Priority Queue 109, 152, 219, 234  
priority queue 302  
Probe 120  
Problem 33

Programmspeicher 7  
Programmzähler 7  
Prozedur 1, 2  
Prozedurinkarnation 86  
Punkteinschluß-Problem 259, 260  
Punkteinschlußsuche 293  
push 82

**Q**

quadratisch 15  
Quadratisches Sondieren 126  
Queue 89  
QuickSort 174, 177  
Quicksort 173, 178, 182, 320, 329

**R**

Radixsort 196  
Radixsortieren 196  
RAM 7, 33, 38  
Random-Access-Maschine 7  
range 91  
Range-Baum 290, 291  
Range-Intervall-Baum 296  
Range-Range-Binär-Baum 298  
rank 327  
rappend 89  
Raster 292  
rationaler Entscheidungsbaum 199  
real RAM 7, 38  
Rebalancieren 142  
Rebalancieroperation 141  
Rechteckeinschluß-Problem 259  
Rechteckschnitt-Problem 259, 295  
Record 39, 47, 61  
Register 7  
Registermaschine 38  
rehashing 119  
Reheap 188  
Reihung 42  
Rekursionsgleichung 15, 175, 176, 335  
rekursive Invariante 272, 280

rekursive Struktur 97  
 Repräsentation 39  
 rest 64, 82  
 retrieve 66  
 reverse 231  
 RI-Baum 296  
 right 98  
 Ring 105  
 Rotation 142  
 RRB-Baum 298  
 Rückwärtssuche 232

**S**

SB-Baum 297  
 schlimmster Fall 10  
 Schlüssel 115  
 Schlüsseltransformation 115  
 Schlüsselvergleichs-Sortieralg. 193  
 Schlüsselvergleichs-Verfahren 190  
 Schlüsselwert 169  
 schrittweise Verfeinerung 35  
 Segment-Baum 262, 268, 290  
 Segment-Binär-Baum 297  
 SegmentIntersectionDAC 274  
 SegmentIntersectionPS 256  
 Segment-Intervall-Baum 296  
 Segmentschnitt-Problem 254, 299  
 Seite 309  
 Seitenfolge 320  
 Seitenzugriff 309  
 Sekundärkollision 127  
 SelectionSort 170  
 Selektion 42, 48  
 Selektor 48  
 Semantik 23  
 semidynamisch 290  
 separate chaining 118  
 Sequenz 63  
 Set 60  
 set 110  
 Shellsort 197  
 SI-Baum 296  
 Signatur 23, 34, 35

single source shortest path-Problem 213  
 Sorte 23, 34  
 Sortieralgorithmus 169  
 Sortierproblem 169, 190  
 Spannbaum 240  
 spannender Wald 209  
 Speicherplatzausnutzung 309, 329  
 Speicherstruktur 309  
 Speicherzelle 7  
 Spezifikation 1  
 Spezifikation als abstrakter Datentyp 24  
 Spezifikation als Algebra 23  
 split 315  
 SS-Baum 297  
 STAB (p) 292  
 stabil 170  
 Stack 82  
 stack 82  
 Stackebene 84  
 Standard-Heapsort 184  
 Stapel 82  
 stark verbunden 203  
 stark verbundene Komponente 203  
 starke Komponente 203, 235  
 Station 300  
 statisch 290  
 Stirling'sche Formel 193  
 Streifenmenge 278  
 stripes 282  
 StripesDAC 282  
 StrongComponents 236  
 Strukturinvariante 141  
 SUB (p) 291  
 succ 58  
 Suchen 109  
 Suchproblem 251  
 Sweep-Event-Struktur 256  
 Sweepline 254  
 Sweepline-Status-Struktur 255

**T**

tail 107  
 Teilbaum 94, 95

Teilgraph 203  
Teilheap 184  
Tiefe eines Knotens 95  
Tiefendurchlauf 207, 208  
tile tree 307  
top 82  
Trägermenge 23  
transitive Hülle 235  
tree 98  
Turingmaschine 7, 33  
Türme von Hanoi 87  
Typ 1, 39  
Typkonstruktor 68  
Typsystem 40

**U**

Überlauf 116  
unabhängig 127  
Underflow 314, 316  
underflow 316  
ungerichteter Graph 201, 239  
uniformes Hashing 120  
union 110  
universale Algebra 23  
Universum 110  
Unterbereichstyp 59  
untere Schranke 20

**V**

Vater 94

verbunden 240  
Vielweg-Mischen 326, 329  
Vielweg-Suchbaum 310, 311  
vollständiger binärer Baum 96, 100  
von Neumann, John 198  
Vorfahr 95  
Vorgänger 58  
Voronoi-Diagramm 305  
Vorwärtssuche 232

**W**

Wahrscheinlichkeitsraum 332  
Wald 102  
Warshalls Algorithmus 235  
Warteschlange 90, 152  
worst case 10  
Wörterbuch 113  
Wurzel 94, 207  
Wurzelgraph 207

**Z**

Zeiger 49  
Zeigerstruktur 97  
Zentrum 245  
Zickzack-Paradigma 307  
Zufallsvariable 332  
Zugriffscharakteristika 309  
zusammenhängend 252  
zyklische Liste 105  
Zyklus 203, 239