Dieses Buch soll als Begleitmaterial für eine Vorlesung über Algorithmen und Datenstrukturen an Fachhochschulen dienen. Es wendet sich an Leser mit guten Kenntnissen in Programmierung und führt in die Problembereiche des Sortierens, Suchens und grundlegende Techniken der Optimierung ein. Der Autor ist Professor für Informatik an der Fachhochschule Hamburg.

ISBN 3-89639-172-0
€ 15,80

NORDAKADEMIE
012230

Inf & Ing

Vorlesungen zum Informatik- und Ingenieurstudium

Band 5

Bernd Owsnicki-Klewe

Algorithmen und Datenstrukturen

4. Auflage 2002

EDV 55 Ows

Wißner

Z02/03/3892
NAK 21382

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Owsnicki-Klewe, Bernd:
Algorithmen und Datenstrukturen / Bernd Owsnicki-Klewe. –
3., veränd. Aufl. – Augsburg : Wißner, 1999
(Inf & Ing ; Bd. 5)
ISBN 3-89639-172-0

ISBN 3-89639-172-0

© Wißner-Verlag, Augsburg 1999
4. Auflage 2002

www.wissner.com

Das Werk und seine Teile sind urheberrechtlich geschützt. Jede Verwertung in anderen als den gesetzlich zugelassenen Fällen bedarf deshalb der vorherigen schriftlichen Einwilligung des Verlages.

Der Inhalt des vorliegenden Bandes liegt in der Verantwortung des Autors.

Inhaltsverzeichnis

1 Grundlegende Begriffe **11**

1.1 Der Begriff des „Algorithmus“ 11

1.2 Pseudocode 12

1.3 Fehlerbehandlung 13

1.4 Abstrakte Datentypen 13

1.4.1 Schnittstelle eines ADT 14

1.4.2 Vor- und Nachbedingungen 14

1.4.3 Invariante eines ADT 15

1.4.4 Implementation 16

1.4.5 Notation von ADTs 16

1.4.6 Beispiele 17

2 Komplexität von Algorithmen **21**

2.1 Asymptotische Analyse 22

2.1.1 Die O-Notation 22

2.1.2 Die Ω-Notation 24

2.1.3 Beispiel 25

2.1.4 Die Θ-Notation 26

2.2 Häufig auftretende Ressourcenanforderungen 26

2.3 Analyse rekursiver Algorithmen 28

2.3.1 Raten und Beweisen 29

2.4 Zusammenfassung 31

3 Elementare Datenstrukturen **33**
3.1 Lineare und kreisförmige Strukturen . . . 33
3.1.1 Einfach und doppelt verkettete Listen . . . 33
3.1.2 Ringlisten . . . 36
3.1.3 Darstellung höherer Strukturen . . . 37
3.2 Graphen und Bäume . . . 38
3.2.1 Grundlegende Begriffe . . . 38
3.2.2 Bäume und ihre Darstellung . . . 39
3.2.3 Darstellung von Graphen . . . 44

4 Internes Sortieren **47**
4.1 Elementare Sortierverfahren . . . 48
4.1.1 Sortieren durch Auswählen I: Selection Sort . . . 48
4.1.2 Sortieren durch Einfügen: Insertion Sort . . . 49
4.2 Sortieren durch Auswählen II: Heapsort . . . 52
4.2.1 Teilweise sortierte Strukturen . . . 52
4.2.2 Heaps . . . 54
4.2.3 Heaps als Implementation der Prioritäts-Warteschlange . . . 54
4.2.4 Auf dem Weg zu Heapsort . . . 58
4.2.5 Heapsort . . . 59
4.2.6 Zusammenfassung . . . 62
4.3 Sortieren durch Zerlegen: Quicksort . . . 63
4.3.1 Die Idee . . . 63
4.3.2 Partitionierung . . . 64
4.3.3 Komplexität . . . 68
4.3.4 Wahl des Medians . . . 70
4.3.5 Quicksort auf kleinen Arrays . . . 72
4.4 Vergleich . . . 73

5 Externes Sortieren **75**
5.1 Sortieren durch Mischen . . . 75
5.1.1 Bänder . . . 76
5.1.2 Die Idee des Mischens . . . 76
5.1.3 Der ADT „triple-tape“ . . . 79
5.1.4 Komplexitätsbetrachtungen . . . 83
5.2 Verbesserungen . . . 83

6 Hashing **87**
6.1 Einführung . . . 87
6.2 Das Szenario . . . 88
6.2.1 Beispiele für Hash-Funktionen . . . 88
6.3 Kollisionsbehandlung . . . 90
6.3.1 Verkettung . . . 90
6.3.2 Sondieren . . . 94
6.4 Schlußbemerkungen . . . 100

7 Suchen in Baumstrukturen **101**
7.1 Sortierte Binärbäume . . . 101
7.1.1 Elementare Operationen auf sortierten Binärbäumen . . . 103
7.1.2 Der Ordnungs-Nachfolger . . . 105
7.1.3 Löschen aus einem sortierten Binärbaum . . . 107
7.2 Rot/Schwarz Bäume . . . 110
7.2.1 Definition und elementare Eigenschaften . . . 110
7.2.2 Rotation . . . 112
7.2.3 Einfügen in einen RS-Baum . . . 114
7.2.4 Löschen aus einem RS-Baum . . . 119
7.3 B-Bäume . . . 126
7.3.1 Benutzung von B-Bäumen . . . 127
7.3.2 Struktur von B-Bäumen . . . 129
7.3.3 Suchen in einem B-Baum . . . 131
7.3.4 Einfügen in einen B-Baum . . . 132
7.3.5 Löschen aus einem B-Baum . . . 138
7.3.6 Komplexität . . . 148

8 Suchen in Graphen **151**
8.1 Tiefensuche . . . 151
8.2 Breitensuche . . . 159
8.3 Kürzeste Wege I: Breitensuche . . . 161

9 Greedy-Optimierung **163**
9.1 Allgemeiner Aufbau von Greedy-Algorithmen . . . 165
9.2 Minimalgerüste . . . 166
9.2.1 Datenstrukturen für disjunkte Mengen . . . 166
9.2.2 Der Kruskal-Algorithmus . . . 168
9.3 Scheduling-Probleme . . . 170
9.4 Datenkompression . . . 177
9.4.1 Aufbau des Codebaums . . . 179
9.4.2 Komplexität . . . 185
9.4.3 Komprimierung . . . 185
9.4.4 Expansion . . . 186
9.5 Kürzeste Wege II: Single-Source Shortest Path . . . 187
9.5.1 Implementation des Dijkstra-Algorithmus . . . 187
9.5.2 Beispiel . . . 192
9.5.3 Komplexität . . . 193
9.6 Weitere greedy lösbare Problem . . . 193

10 Dynamische Programmierung **195**
10.1 Kürzeste Wege III: All-Pair Shortest Path . . . 195
10.1.1 Erste Überlegungen . . . 195
10.1.2 Der Floyd-Warshall Algorithmus . . . 197
10.2 Der 0-1 Rucksack mit Dynamischer Programmierung . . . 201
10.3 Parsing kontextfreier Sprachen . . . 204
10.3.1 Die Chomsky-Normalform . . . 204
10.3.2 Der DP-Ansatz . . . 205
10.3.3 Beispiel . . . 207
10.3.4 Komplexität . . . 209
10.4 Schlußbemerkungen . . . 210

11 Branch and Bound **211**
11.1 Der 0-1 Rucksack . . . 211
11.2 Schlußbemerkungen . . . 217

Vorbemerkungen

Dieses Buch ist als Skript zur Vorlesung „Algorithmen und Datenstrukturen“ an der Fachhochschule Hamburg entstanden. Ziel ist es dabei gewesen, Einblicke in typische Problemstellungen und wichtige Klassen von Algorithmen zu liefern und eine tiefere Vorstellung zu ermöglichen, was die Qualität eines Algorithmus ausmacht und wann bestimmte Algorithmen einsetzbar sind.

Bei vielen algorithmischen Problemen gibt es anerkannte Standardlösungen, die ein Informatiker kennen sollte. Diese Lösungen zu vermitteln, ist die erste Hauptaufgabe dieses Buchs. Auf der anderen Seite ist für den Einsatz der vorgestellten Techniken, insbesondere aus dem Bereich der Optimierung, einiges an Phantasie erforderlich, um zu „sehen“, daß ein bestimmtes Verfahren für ein gegebenes Problem günstig einsetzbar ist. So werden gerade für diesen Bereich mehrere Beispiele angeführt, die verschiedene Probleme lösen. Ein allgemeines „Kochrezept“ dafür, *wann* man welches Verfahren anwenden sollte, kann hier natürlich nicht geliefert werden. Die Darstellung ist gegliedert in vier Teile:

Grundlegende Methoden: (Kapitel 1 – 3) Schwerpunkt hier sind Verfahren zur Spezifikation, Notation und Analyse von Algorithmen. In diesem Teil wird der Begriff des *Abstrakten Datentyps* (ATD) vorgestellt, der zur Notation von Algorithmen verwendete *Pseudocode* wird erläutert und es werden die wichtigsten Begriffe der *Komplexitätsanalyse* von Algorithmen dargestellt.

Für das Verständnis der in den nächsten Teilen vorgestellten Algorithmen ist es notwendig, sich in *elementaren Datenstrukturen* einigermaßen auszukennen. Obwohl sich dieses Skript nicht an Anfänger in der Progammierung wendet sondern Grundkenntnisse in einer höheren Programmiersprache und eine gewisse Erfahrung in der Programmierung voraussetzt, werden die wichtigsten elementaren Datenstrukturen hier noch einmal — wenn auch sehr kurz — vorgestellt.

Zu den Datenstrukturen, die hier im Vordergrund stehen, zählen in erster Linie *Graphen* und — als Spezialfall — *Bäume*. Beide Strukturen lassen sich in verschiedenen Problemen wiederfinden.

Sortieren: (Kapitel 4 – 5) Sortieren ist eine der wesentlichen Aufgaben, die ein Programm zu erledigen hat. Im kommerziellen Bereich sind Dateien nach gewissen Schlüsseln zu sortieren, Meßdaten im naturwissenschaftlichen Bereich, usw. Der Sinn des Sortierens kann einerseits darin bestehen, den Zugriff auf die Daten zu beschleunigen oder eine sequentielle Verarbeitung in einer vorgegebenen Reihenfolge zu ermöglichen.

Bei Sortierverfahren wird man zunächst grob zwei Klassen unterscheiden, die sich in der Zugriffsdisziplin auf die einzelnen Elemente unterscheiden:

Interne Verfahren: Sie lassen einen wahlfreien Zugriff auf jedes beliebige Element der zu sortierenden Struktur zu. Das typische Beispiel für solche Strukturen wird durch *Arrays* geliefert. Interne Sortierverfahren gehören zur Ausrüstung eines jeden Programmierers und sind häufig (aber nicht immer) in Bibliotheken vorhanden.

Hier werden wir zwei Unterklassen von Verfahren kennenlernen. Einerseits die „elementaren“ Verfahren, die einfach zu verstehen und einfach zu programmieren sind. Andererseits gibt es die „höheren“ Verfahren, die etwas aufwendiger zu implementieren aber effizienter als die elementaren Verfahren sind.

Externe Verfahren: Sie sind geeignet zum Sortieren sequentieller Dateien, auf die nur ein beschränkter Zugriff — auf den laufenden Datensatz — möglich ist.

Interessanterweise wird sich zeigen, daß die externen Verfahren die gleiche Komplexität aufweisen, wie die guten internen. Sie eignen sich also auch zur Sortierung verketteter Strukturen, bei denen ähnliche Einschränkungen wie für Dateien gelten.

Die Komplexität der dargestellten Verfahren wird hier einen großen Raum einnehmen, insbesondere bei den einfachen Verfahren sind die Komplexitätsbetrachtungen auch im mittleren Fall einigermaßen übersichtlich.

Suchen: (Kapitel 6 – 8) Beim Suchen handelt es sich i.w. darum, ein gewisses Datenobjekt anhand einer Identifikation, eines *Schlüssels*, in einer geeigneten Datenstruktur wiederzufinden, bzw. anzuzeigen, daß ein solches Datenobjekt nicht vorhanden ist.

Damit sind solche Verfahren primär als Implementation des Abstrakten Datentyps „Dictionary“ (1.4.6) geeignet. Für die Praxis bedeutet das, daß neben den Operationen zum Einfügen und Suchen eines Datenobjekts auch Operationen zum *Löschen* vorgesehen werden müssen.

Natürlich gibt es sehr einfache Datenstrukturen, wie das Array, in dem diese Operationen einfach zu implementieren sind. Die einzig praktische Erleichterung bei solchen Operationen kann man dadurch erreichen, daß der Array-Inhalt *sortiert* ist, was man mit den Techniken aus Kapitel 4 immer hinbekommen kann (Stichwort: Binärsuche). Diese Techniken werden wir hier nicht diskutieren.

Aufwendiger zu lösen wird das Suchproblem, wenn wir vorab *keine* Kenntnisse über die Anzahl der zu speichernden Datenobjekte haben, Arrays als Datenstruktur also ausfallen. Hier bieten sich zwei große Klassen von Verfahren an:

Streuverfahren: Hier wird man unter dem Schlagwort „Hashing“ versuchen, den Bereich der potentiell erlaubten Schlüssel auf einen kleinen Adreßbereich abzubilden, in der Hoffnung, daß nicht alle Schlüssel tatsächlich auftreten. Natürlich kann man sich darauf nicht verlassen, wird also Maßnahmen für den Fall treffen müssen, daß es doch zu Konflikten kommt.

Diese Verfahren erlauben ebenfalls eine meist recht einfache Analyse ihres Verhaltens, sind vergleichsweise einfach zu implementieren und liefern, wenn man keine groben Fehler macht, ganz brauchbare Ergebnisse in der Praxis.

Baumsuchverfahren: Eine der naheliegenden Datenstrukturen zur Speicherung von identifizierbaren Datenobjekten sind Bäume. Um einen effizienten Zugriff auf die Schlüssel zu bekommen, müssen solche Bäume so organisiert sein, daß man in jedem Knoten ein für alle Mal die Entscheidung darüber treffen kann, welchen Nachfolger man als nächsten untersuchen muß, um das gesuchte Objekt zu finden.

Der einfachste Fall wird dabei durch die *sortierten Binärbäume* geliefert, bei denen lediglich das Löschen eines Schlüssels einen Augenblick des Nachdenkens erfordert. Leider haben diese Bäume theoretische (und manchmal leider auch praktische) Nachteile, die man dadurch ausräumt, daß man zu *höhenbalancierten* Bäumen übergeht, die hier durch die *Rot/Schwarz-Bäume* vertreten werden.

Das Problem allerdings, sehr große Schlüsselmengen (10^6 und mehr) effizient zu verwalten, läßt sich nur noch theoretisch mit solchen Strukturen lösen. Solche Datenmengen treten typischerweise in indiziert zugreifbaren Dateisystemen, vornehmlich unterhalb von Datenbanken auf. Dafür gibt es eingeführte und durchgängig verwendete Techniken, von denen die bekannteste — die Verwendung von *B-Bäumem* — hier ausführlich dargestellt wird.

Optimieren: (Kapitel 9 – 11) Optimieren bedeutet ganz abstrakt, aus einer Menge von Lösungen eines gegebenen Problems, eine (oder mehrere) auszuwählen, die unter einer gewissen Bedingung optimal ist. Interessant ist die Geschichte natürlich nur, wenn es *sehr viele* mögliche Lösungen gibt, so daß es keinen Sinn macht, einfach alle durchzuprobieren, um die beste zu finden.

Gebiete wie die angewandte Mathematik oder Operations Research liefern Methoden, für eine große Klasse von Problemen optimale Lösungen zu finden, die bekannteste Methode ist vielleicht die der „Linearen Optimierung“, s. [Maas, 1993].

Darum soll es hier allerdings nicht gehen, wir sind interessiert an Lösungsverfahren, die eher algorithmischer Natur sind und für auch recht abgelegene Probleme eingesetzt werden können:

Greedy-Algorithmen: Solche Algorithmen treffen Entscheidungen über Aspekte der Lösung durch Verfolgen *lokal optimaler* Teillösungen. Sie entsprechen dem intuitiven Gefühl über Optimierung, sind einfach zu implementieren, reichen allerdings nicht immer aus, um *global optimale* Lösungen zu konstruieren.

Dynamische Programmierung: Hier wird mit etwas mehr theoretischem Aufwand unter gewissen Bedingungen eine global optimale Lösung garantiert. Diese Klasse von Verfahren verbessert einfache rekursive Optimierungsverfahren dadurch, daß Teillösungen tabellarisch festgehalten und zur Konstruktion größerer Lösungen herangezogen werden. Die entstehenden Algorithmen sind oft sehr einfach aufgebaut und deshalb ausgesprochen stabil gegen Programmierfehler.

Branch and Bound: Das ist ein letzter Notanker, wenn auch mit Dynamischer Programmierung kein Erfolg zu erzielen ist.

Prinzipiell versucht diese Technik, durch heuristische Steuerung des Suchprozesses aus einer sehr großen Menge von Lösungskandidaten praktisch nur „wenige“ anzugucken, um aus ihnen die optimale zu bestimmen.

Wie bereits angedeutet, stellt dieses Skript gewisse Anforderungen an den Leser, die in erster Linie in soliden Grundkenntnissen in Programmierung bestehen. Auch etwas Mathematik sollte zum Rüstzeug gehören; in erster Linie Vertrautheit mit Summationen

und Produkten, etwas Matrizenrechnung und einigen Grundkenntnissen in Wahrscheinlichkeitsrechnung. Alles andere wird hier bereitgestellt.

Grundlage dieses Skripts sind in erster Linie folgende Bücher:

[Cormen *et al.*, 1990] ist sehr modern, vollständig und leicht zu lesen. Es ist Grundlage der Kapitel 2, 6, 7, 8 und 10. [Brassard und Bratley, 1988] behandelt viele Themen unter etwas anderen Gesichtspunkten. Darauf basieren i.w. die Kapitel 9 und 11. [Mehlhorn, 1988] ist etwas mathematischer orientiert. Es liegt der Darstellung der Kapitel 6 und 4 zugrunde.

Als zusätzliche Literatur würde ich empfehlen:

[Sedgewick, 1992] enthält neben alternativen Darstellungen auch weitgehend ausformulierte Programme. [Horowitz und Sahni, 1978] deckt viele andere interessante Gebiete der Algorithmenlehre ab. [Knuth, 1968], [Knuth, 1969], [Knuth, 1973], die „Klassiker" auf dem Gebiet der Algorithmen. [Maas, 1993] stellt die benötigten graphentheoretischen Hilfsmittel zur Verfügung.

Viele Personen haben zu diesem Skript beigetragen. Mein ehemaliger Kollege U. Breymann (jetzt Hochschule Bremen) hat mir seine gesamten Unterlagen und Beispielprogramme überlassen, die beim Aufbau dieses Skripts eine sehr große Hilfe waren. Meine Kollegen G. Pfeiffer und M. Böhm haben sehr wesentliche Hinweise zur Strukturierung des Materials gegeben, die hier teilweise verwirklicht worden sind. Mein Kollege C. Maas hat die letzte Version sehr sorgfältig und kritisch gelesen und viel für den Aufbau des Materials getan. Herr A. Jahnke hat viele Fehler gefunden und wichtige Anmerkungen gemacht, die mich lange in Atem gehalten haben. Meine Frau Yvonne hat mir bei mathematischen Problemen sehr geholfen und darauf geachtet, daß ich mich einigermaßen präzise ausdrücke. Ihnen allen gilt mein besonderer Dank!

Die Studenten der FH Hamburg, insbesondere die „Versuchskaninchen" des Jahrgangs 1990, die als erste mit diesem Material konfrontiert worden sind, haben in der Zwischenzeit noch einige gut versteckte Fehler gefunden, die nun hoffentlich beseitigt sind. Die verbliebenen Fehler gehen voll auf mein Konto.

Ein Wort noch! Algorithmenentwurf ist eine der interessantesten Aufgaben für einen Informatiker. In der Praxis wird man mit interessanten Algorithmen allerdings nicht so oft konfrontiert, wie dieses Buch glauben machen will. Wenn es aber gelingt, wenigstens bei einigen Lesern so etwas wie den Blick für die Logik und Eleganz wichtiger Algorithmen zu wecken, so bin ich schon zufrieden.

Hamburg, Januar 1994
B. Owsnicki-Klewe

Kapitel 1

Grundlegende Begriffe

Zur Arbeit mit diesem Skript sind einige einleitenden Bemerkungen notwendig, die sich auf Begriffe, Notationen und theoretischen Hintergrund beziehen.

1.1 Der Begriff des „Algorithmus"

Der Begriff ist abgeleitet aus dem Namen Abu Ja'far Mohammed ibn Musa al Khowarizmi (persischer Mathematiker, 780–850 n. Chr.). Auf ihn geht die Idee zurück, mathematische Probleme systematisch dadurch zu lösen, daß man eine bestimmte Folge von Anweisungen „blind" ausführt. Für die Informatik hat der Algorithmus die Bedeutung einer Vorschrift zur Lösung eines Problems mit Hilfe eines Computers: Die Vorschrift bildet einen Eingangsdatenbereich auf einen Ausgangsdatenbereich ab.

Zu nennen sind die Anforderungen an ein Verfahren, das den Namen „Algorithmus" verdient:

Eindeutigkeit: Jeder elementare Schritt ist unmißverständlich beschrieben und läßt keine Wahlmöglichkeit offen.

Endlichkeit: Die Beschreibung des Algorithmus hat endliche Länge (durch Schleifen o.Ä. kann das ablaufende Programm allerdings durchaus unendlich lange brauchen, wenn der Algorithmus fehlerhaft ist).

Terminiertheit: Genau das soll *nicht* passieren. Ein aktuelles Programm, das diesen Algorithmus implementiert, soll in endlicher Zeit beendet sein.

Reproduzierbarkeit: Der Algorithmus soll unabhängig von einer speziellen Implementation sein. Verschiedene (korrekte) Implementationen sollen dasselbe Ergebnis liefern.

Für unsere Zwecke sollte man aber auch den Begriff der **Datenstruktur** charakterisieren. Wir verstehen darunter eine Organisation von Daten basierend auf einfachen Datentypen, die

1. eine problemgemäße Abbildung der Daten in eine formale rechnergemäße Struktur liefert, und die

2. geeignet ist zur Bearbeitung mit einem bestimmten Algorithmus.

Das bedeutet, daß Algorithmen und Datenstrukturen zusammengehören. Ein Algorithmus benötigt in der Praxis eine von mehreren geeigneten Datenstrukturen, um optimal laufen zu können (z.B. ist eine verkettete Liste nicht geeignet für die binäre Suche). Wir gehen hier davon aus, daß die Implementation elementarer Datenstrukturen (etwa so etwas wie einer verketteten Liste) i.w. ein „technisches" Problem darstellt, daß der Leser also in der Lage ist, eine solche Struktur in der Programmiersprache seiner Wahl zu implementieren.

Das gleiche gilt ein wenig abgemildert dann auch für die Algorithmen selbst. Wie wir wissen, gehört zu einer lauffähigen Version eines Algorithmus noch ein beträchtliches Maß an „syntaktischem Grünzeug", das der Compiler und das Laufzeitsystem zusätzlich verlangen. Wenn ich das alles bei der Vorstellung des Verfahrens explizit hinschreiben wollte, kämen wir hier nicht sehr weit. Außerdem muß man schon sehr gut programmieren können, damit im Endprodukt der Kern des Algorithmus noch zu erkennen ist.

Man muß sich also einen gewissen Kompromiß zwischen Vollständigkeit der Notation und Klarheit des Verfahrens finden. Ein geeignetes Mittel dazu ist natürlich die *Abstraktion* in ihrer bekanntesten Ausprägung, dem *abstrakten Datentyp* (ATD). Man wird also versuchen, all das, was beim Studium des Algorithmus stören könnte, in einer abstrakten Schnittstelle zu verstecken, deren Implementation immer noch schwierig genug sein dürfte. Aber auch hierzu wird man Hinweise im Text finden. Ein wichtiges Hilfsmittel zur kompakten Notation von Algorithmen ist der sogenannte *Pseudocode*.

1.2 Pseudocode

Es handelt sich beim Pseudocode um eine von fast allen syntaktischen Feinheiten bereinigte Programmiersprache, die lediglich den algorithmischen Kern des Verfahrens notiert. Der hier verwendete Pseudocode lehnt sich eng an `Pascal` und teilweise an `C` an. Folgende Strukturelemente sind im Pseudocode vorgesehen:

Prozeduren: Ein benanntes Codestück, das *parametrisiert* werden und von verschiedenen Stellen aus aufgerufen werden kann. Eine Prozedur liefert *keinen* Wert und kann *Seiteneffekte* haben.

Funktionen: Entsprechen i.w. Prozeduren, liefern aber einen Wert. Es ist schlechter Stil, in einer Funktion einen Seiteneffekt ausführen zu lassen.

Variablen: Stellen Speicherstellen dar, deren Wert verändert werden kann. Es gibt drei Typen von Variablen im Pseudocode:

1. *Einfache* Variablen enthalten einen einzigen Wert ihres *Basistyps*.
2. *Indizierte* Variablen stellen eine n-dimensionale Struktur dar ($n = 1$: Vektor, $n = 2$: Matrix), auf die mit n *Indizes* zugegriffen wird,
3. *Strukturierte* Variablen bestehen aus Daten verschiedener Typen.

Bezeichner: Sie bezeichnen *Variablen*, *Parameter*, *Prozeduren* und *Funktionen*.

1. Bezeichner für *einfache* Variablen sind Namen.
2. Bezeichner für *indizierte* Variablen sind Namen, bei denen durch einen oder mehrere *Indexausdrücke* — eingeschlossen zwischen „`[`" und „`]`" — auf die einzelnen Elemente zugegriffen wird.

3. Bezeichner für *strukturierte* Variablen sind Namen, bei denen durch einen Punkt „`.`" auf die einzelnen Komponenten zugegriffen werden kann.

Parameter: Steuern das Verhalten von Prozeduren und Funktionen. Standardmäßig wird „call by value" als Übergabemechanismus vorausgesetzt, ein *Resultatsparameter* („call by reference") wird durch das vorangestellte Schlüsselwort `var` gekennzeichnet.

Kontrollstrukturen: Neben der *Iteration* benutzen wir lediglich

1. den `if`-Vergleich mit optionalem `else`-Zweig,
2. die `while`- und
3. die `repeat`-Schleife.

Weiterhin gehört dazu, daß man vor allen Dingen explizite Typdeklarationen so weit wie möglich fortläßt, weil das sowieso implementationsabhängig ist. Weiterhin ersetzen wir Schachtelungen mittels `begin` und `end` durch entsprechende Einrückungen, Kommentare werden mittels „`{`" und „`}`" eingeklammert. Die Rückgabe von Funktionsergebnissen geschieht nicht mehr durch eine Zuweisung auf den Funktionsnamen sondern einfacher durch das Schlüsselwort `return`, ähnlich wie in C. Dies sollte es ermöglichen, den Algorithmus mit vergleichsweise wenig Aufwand in die gewünschte Zielsprache zu übertragen.

1.3 Fehlerbehandlung

Selbst wenn ein Algorithmus gemäß seiner Spezifikation korrekt ist, so gibt es mehr als genug Fälle, in denen eine konkrete Implementation dennoch abstürzt, weil irgendwelche Daten doch nicht den Anforderungen genügen. Jeder Mensch schützt sich, das Programm und die Umwelt vor solchen Unglücken durch eine ordentliche Fehlerbehandlung.

Wie jeder weiß, macht eine solche Fehlerbehandlung in der Praxis einen recht großen Teil des fertigen Programmcodes aus und der algorithmische Kern ist dann nur unter Schwierigkeiten zu erkennen.

Genau das soll hier natürlich nicht passieren Aus diesem Grund wird in (fast) allen Algorithmen auf eine explizite Fehlerbehandlung *verzichtet*! Jeder, der den einen oder anderen Algorithmus implementiert, ist aufgefordert, eine geeignete Fehlerbehandlung selbst vorzusehen.

1.4 Abstrakte Datentypen

Wie bekannt, bilden abstrakte Datentypen (ADTs) ein wichtiges Mittel, um Algorithmen bzw. Programme von Implementationsdetails zu verschonen. Der Sinn von ADTs besteht somit in praktischer Hinsicht darin, daß die Einzelheiten der Implementation im eigentlichen Algorithmus nicht mehr auftreten und den Leser verwirren. Leser sind leichter zu verwirren als Computer! Also werden Programme für menschliche Leser geschrieben. Daß sie dann auch auf einem Rechner laufen, ist eine Frage der Technik. Außerdem kann man sich eine eventuell effizientere Implementation vorbehalten, wenn man darauf stößt.

In der Tat werden wir an einigen Stellen darauf zu sprechen kommen, inwieweit eine bestimmte Implementation (etwa einer „priority queue“ als Liste oder als Heap) die Gesamteffizienz eines Algorithmus beeinflußt, der einen ADT verwendet. Das kann manchmal sehr drastisch werden!

Gewöhnlich handelt es sich bei einem ADT um einen zusammengesetzten Datentyp, der also selbst wieder Teil-*Elemente* enthält (Stack, Heap, ...). Um den Unterschied zwischen der Ausprägung des ADT und den darin enthaltenen Elementen klar zu machen, werden wir immer von einem *Objekt* sprechen, wenn wir eine solche Ausprägung eines ADT meinen, die enthaltenen Elemente heißen dann auch *Elemente*. Man beachte, daß es beileibe nicht ausgeschlossen ist, Elemente selbst wieder als Objekte aufzufassen, wenn sie diesen Namen verdienen. Natürlich sind Stacks denkbar, deren Elemente Prioritäts-Warteschlangen sind, die alle Matrizen von Zahlen enthalten. Das Problem bleibt dann, wie man so etwas in einer Sprache mit starker Typenbindung einigermaßen vernünftig implementiert.

1.4.1 Schnittstelle eines ADT

Zu einem ADT gehört in erster Linie seine *operationale Spezifikation*, also eine Sammlung formaler Beschreibungen davon, was mit diesem ADT gemacht werden kann. Solche Spezifikationen treten in Form von Prozeduren oder Funktionen in Erscheinung, die sich auf ein Objekt des jeweiligen Datentyps beziehen. Die Sammlung dieser Operationen nennen wir kurz die *Schnittstelle* (engl. interface) des ADT. Solche Operationen fallen gewöhnlich in fünf Klassen:

1. *Initialisierung:* Es wird ein Objekt des Datentyps aufgebaut, das bestimmte Eigenschaften besitzt (i.a. diejenige, in geeigneter Weise „leer“ zu sein).
2. *Nicht-destruktive Operationen:* Dabei wird der Zustand des Objekts nicht verändert; es werden i.a. nur Datenelemente angeguckt und/oder das Objekt nach irgendwelchen Kriterien getestet.
3. *Destruktive Operationen:* Hierbei wird der Inhalt (abstrakter „der innere Zustand”) des Objekts verändert, indem etwa Elemente hinzugefügt oder gelöscht werden.
4. *Funktionen:* Hierbei macht man aus einem oder mehreren Objekten in geeigneter Weise ein neues. Beispiel dafür ist so etwas wie die Vereinigung zweier Mengen oder die Inversion einer Matrix.
5. *Zerstörung:* Nach Beendigung der Arbeit kann es notwendig sein, Speicher freizugeben, den die Implementation des ADT belegt hat.

1.4.2 Vor- und Nachbedingungen

Zu jeder solchen Operation spezifizieren wir zwei Bedingungen:

1. Die *Vorbedingung* (engl. precondition): Sie gibt an, welche Eigenschaften das Objekt erfüllen muß, damit die entsprechende Operation fehlerfrei anwendbar ist. Der „Benutzer“ des ADT hat vor Benutzung der Operation sicherzustellen, daß die Vorbedingung eingehalten ist. Implementatoren sind angehalten, sich nicht hundertprozentig darauf zu verlassen!
2. Die *Nachbedingung* (engl. postcondition): Hier wird spezifiziert, welche Eigenschaft das Objekt *nach* Beendigung der entsprechenden Operation aufweist und was das eventuelle Resultat der Operation ist.

Vor- und Nachbedingungen können manchmal ziemlich kompliziert ausfallen, man denke etwa daran, wie man einigermaßen formal beschreibt, daß ein Array *sortiert* ist. Wir werden deshalb manchmal im Text eine formale Beschreibung des angestrebten Zustands angeben und uns in den Bedingungen auf so etwas beziehen.

Zustände eines ADT

Komplexe Datenstrukturen, die mit solchen Operationen manipuliert werden, lassen sich oft sehr gut dadurch beschreiben, daß man den „Zustand“ der Datenstruktur vor Ausführung der Operation mit dem Zustand nach Beendigung der Operation vergleicht. Einen solchen Anfangszustand einer Datenstruktur D notieren wir mit D_{pre}, den angestrebten Endzustand mit D_{post}. Schematisch sieht eine solche Transformation so aus:

$$D_{pre} \longrightarrow \text{Operation} \longrightarrow D_{post}$$

Oft werden wir Vor- oder Nachbedingung durch eine Angabe spezifizieren, die solche Ausdrücke enthält.

1.4.3 Invariante eines ADT

Natürlich verlangt man von einem ADT, der zu irgendetwas nütze sein soll, bestimmte Eigenschaften, die *immer* gelten sollen. Triviales Beispiel: Ein *sortiertes* Array weist eine bestimmte Eigenschaft (Sortierung) auf seinen Elementen auf. Verschwindet diese Eigenschaft, so hört das Objekt auf, ein sortiertes Array zu sein.

Diese Eigenschaft eines ADT nennen wir die **Invariante** des ADT, weil sie durch nichts und niemanden geändert werden darf.

Natürlich ist es kaum zu realisieren, die Invariante zu jedem beliebigen Zeitpunkt der Lebensdauer des Objekts zu garantieren! Viele Algorithmen modifizieren zunächst das gegebene Objekt so, daß es die Invariante verletzt wird, basteln sie dann aber *sofort* wieder hin. Beispiele sind *Heaps* (4.2.2) oder *Rot/Schwarz-Bäume* (7.2), bei denen genau diese Strategie angewendet wird.

Wir verlangen also, daß die Invariante nach jeder Operation auf dem ADT wieder gilt, *wenn sie vor der Operation gegolten hat*. Dieser Nebensatz ist ganz wichtig, weil sich einige Algorithmen darauf verlassen, daß die Invariante gilt und nur unter dieser Bedingung die Invariante auch wieder hinbekommen.

In vielen Fällen ist die Invariante ziemlich kompliziert. Sie kann z.B. durch eine ziemlich umfangreiche *Definition* gegeben werden, auf die wir uns dann bei der Formulierung der Invariante beziehen.

1.4.4 Implementation

Durch seine Struktur ist ein ADT weitgehend unabhängig von einer konkreten Implementation. In der Tat gibt es meistens mehrere Möglichkeiten, die Funktionalität eines gegebenen ADT mit bekannten Techniken zu implementieren. In den meisten Fällen werden wir uns allerdings mit konkreten Implementationen befassen, um zu formulierbaren Algorithmen zu kommen.

1.4.5 Notation von ADTs

Natürlich wird die Notation eines ADTs eine recht aufwendige Sache, wenn man sich überlegt, was man alles braucht:

1. den Namen des ADT,
2. seine Schnittstelle,
3. seine Invariante, sowie
4. eventuell Aussagen über eine konkrete Implementation.

Hier ein schematisches Beispiel dafür, wie die Sache hier ablaufen soll. Einen allgemeinen Eindruck bekommt man durch eine simple Graphik, die den ADT im Überblick darstellt. Im allgemeinen wird hier nichts über eine eventuelle Implementation ausgesagt (Abb. 1.1).

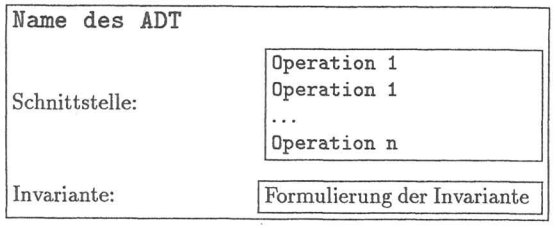


Abbildung 1.1: Graphik eines ADT ohne Implementation

Soll über eine ganz bestimmte Implementation eines ADT gesprochen werden, so zeigt die Graphik die charakteristischen Teile dieser Implementation im unteren Teil (Abb. 1.2).

Die einzelnen Operationen der Schnittstelle werden gewöhnlich separat notiert:

1. `op-1(...)`
   Precondition: `Formulierung der Vorbedingung`
   Postcondition: `Formulierung der Nachbedingung`
   Erläuternder Text.
2. ...

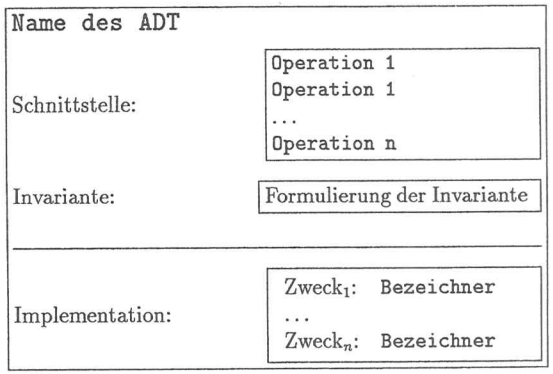
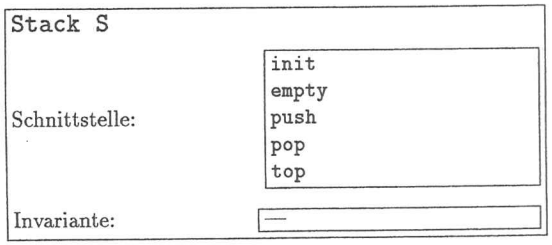


Abbildung 1.2: Graphik eines ADT mit Implementation

1.4.6 Beispiele

Hier sollen einige wichtige ADTs kurz vorgestellt werden, die wir im Laufe unserer weiteren Arbeit noch benötigen werden.

Stack: Ein Stack (Keller, Stapel, LIFO-Liste) besteht aus einer linearen Liste $S = (a_1, a_2, a_3, \ldots, a_n)$. Diese Elemente heißen Atome. Atome werden nur am Listenanfang eingefügt oder entnommen (*Last In – First Out*).



Schnittstelle:

1. `init(S)`
   Precondition: ---
   Postcondition: $S_{post} = \emptyset$
2. `empty(S)`
   Precondition: ---
   Postcondition: ---
   Resultat: $S = \emptyset$.
3. `push(S,x)`
   Precondition: ---
   Postcondition: $S_{post} = S_{pre} + \{x\}$

Das Atom `x` wird „oben" auf dem Keller gespeichert.

4. `pop(S)`
   Precondition: `not empty(S)`
   Postcondition: S_{post} = S_{pre} - `Oberstes Element`
   Resultat: Oberstes Element von `S`. Dieses Element wird vom Stack entfernt.
5. `top(S)`
   Precondition: `not empty(S)`
   Postcondition: ---
   Resultat: Oberstes Element von `S`. Das Element verbleibt im Stack.

Implementationsmöglichkeiten: Arrays, verkettete Listen.

Queue: Eine Queue (Warteschlange, FIFO-Liste) besteht aus einer linearen Liste $S = (a_1, a_2, a_3, \ldots, a_n)$. Diese Elemente heißen Atome. Atome werden nur am Listenende eingefügt aber am Listenanfang entnommen (*First In – First Out*).

`Queue Q`	

Schnittstelle:	`init`
	`empty`
	`enqueue`
	`dequeue`
	`front`

Invariante:	—

Schnittstelle:

1. `init(Q)`
   Precondition: ---
   Postcondition: $Q_{post} = \emptyset$
2. `empty(Q)`
   Precondition: ---
   Postcondition: ---
   Resultat: $Q = \emptyset$.
3. `enqueue(Q,x)`
   Precondition: ---
   Postcondition: $Q_{post} = Q_{pre} + \{x\}$
   Das Atom `x` wird „hinten" in der Schlange gespeichert.
4. `dequeue(Q)`
   Precondition: `not empty(Q)`
   Postcondition: $Q_{post} = Q_{pre}$ - `Vorderstes Element`
   Resultat: Vorderstes Element von `Q`. Dieses Element wird aus der Queue entfernt.
5. `front(Q)`
   Precondition: `not empty(Q)`
   Postcondition: ---
   Resultat: Vorderstes Element von `Q`. Das Element verbleibt in der Queue.

Implementationsmöglichkeiten: Arrays, verkettete Listen.

Prority Queue: (Prioritätsgesteuerte Warteschlange, kurz: PQ) Eine Queue, bei der jedem Element der Warteschlange eine *Priorität* zugeordnet wird. Die Entnahme eines Elements geschieht nach dessen Priorität (größte oder kleinste Priorität). Hier als Beispiel: Größte Priorität.

`Priority Queue Q`	

Schnittstelle:	`init`
	`empty`
	`insert`
	`remove-largest`
	`largest`

Invariante:	—

Schnittstelle:

1. `init(Q)`
   Precondition: ---
   Postcondition: $Q_{post} = \emptyset$
2. `empty(Q)`
   Precondition: ---
   Postcondition: ---
   Resultat: $Q = \emptyset$.
3. `insert(Q,x)`
   Precondition: ---
   Postcondition: $Q_{post} = Q_{pre} + \{x\}$
   Das Atom `x` wird in die PQ Schlange gespeichert.
4. `remove-largest(Q)`
   Precondition: `not empty(Q)`
   Postcondition: $Q_{post} = Q_{pre}$ - `Größtes Element`
   Resultat: Größtes Element von `Q`. Dieses Element wird aus der Queue entfernt.
5. `largest(Q)`
   Precondition: `not empty(Q)`
   Postcondition: ---
   Resultat: Größtes Element von `Q`. Das Element verbleibt in der Queue.

Implementationsmöglichkeiten: Arrays, verkettete Listen, Heaps (4.2.2).

Aufgabe: *Definieren Sie den ADT* **Priority Queue** *so, daß er das* kleinste *Element liefert.*

Dictionary: Liefert eine einfache tabellenartige Speicherungsstruktur, die *inhaltsorientiert* arbeitet. Es ist also kein Index notwendig; jedes Element wird über seinen Inhalt (oder einen Teil davon) identifiziert.

Dictionary D	
Schnittstelle:	init empty insert member delete
Invariante:	—

Schnittstelle:

1. `init(D)`
   Precondition: ---
   Postcondition: $D_{post} = \emptyset$

2. `empty(D)`
   Precondition: ---
   Postcondition: ---
   Resultat: $D = \emptyset$.

3. `member(D,x)`
   Precondition: ---
   Postcondition: ---
   Resultat: $x \in D$(als Wahrheitswert).

4. `insert(D,x)`
   Precondition: ---
   Postcondition: $D_{post} = D_{pre} + \{x\}$
   Das Atom x wird in das Dictionary gespeichert.

5. `delete(D,x)`
   Precondition: `member(D,x)`
   Postcondition: $Q_{post} = Q_{pre} - \{x\}$
   Das Atom x wird aus dem Dictionary entfernt.

Kapitel 2

Komplexität von Algorithmen

Bei der Betrachtung von Algorithmen werden wir ganz entscheidend mit der Frage ihrer *Effizienz* konfrontiert. Die *algorithmische Komplexitätstheorie* behandelt Methoden und Notationen, mit denen man den Aufwand eines Verfahrens formal beschreiben kann. Als Analyse von Algorithmen bezeichnen wir die Untersuchung von Speicherplatzbedarf und Rechenzeit in Abhängigkeit von Anzahl und Art der Eingangsdaten mit dem Ziel der Effizienzverbesserung.

Speicherplatzbedarf und benötigte Rechenzeit werden auch als *Komplexität* bezeichnet, wobei meistens der Schwerpunkt auf der Rechenzeit liegt. Der Aufwand für irgendein Verfahren wird (bis auf wenige angenehme Ausnahmen) von der Größe des Eingangsdatenbereichs, z.B. der Anzahl der zu sortierenden Elemente n, abhängen.

Die Aufgabe der Algorithmenanalyse besteht darin, *ohne* ein aktuelles Programm, *ohne* Rechner und *ohne* Stoppuhr Aussagen über die zu erwartende Effizienz des Algorithmus zu machen. Das ist in vielen Fällen nicht ganz einfach, weil nicht-triviale Algorithmen „je nach Lage der Dinge" ganz unterschiedliches Laufzeitverhalten zeigen. So unterscheidet man normalerweise drei Fälle:

Der beste Fall (best case): Das ist diejenige Struktur der Eingabedaten, für die der Algorithmus am effektivsten ist. So ist z.B. für einige Sortierverfahren eine bereits sortierte Sequenz der beste Fall (Bubblesort), für andere eine genau umgekehrt sortierte Sequenz (Heapsort).

Weil beste Fälle ziemlich selten auftreten, werden wir uns um ihre Behandlung nicht sehr häufig kümmern.

Der schlechteste Fall (worst case): Diejenige Eingabestruktur, für die der Algorithmus am wenigsten effektiv ist. Sucht man z.B. ein Element in einem Array, so ist der schlechteste Fall immer der, daß das Element gar nicht drin ist.

In gewisser Weise ist dies der interessanteste Fall für uns.

Der mittlere Fall (average case): Die zu erwartende Effizienz bei beliebiger Strukturierung der Eingabedaten.

Dieser Fall ist in der Praxis ausgesprochen interessant, hat aber den gewichtigen Nachteil, daß seine Behandlung mathematisch ausgesprochen kompliziert ist: Man

braucht dazu in erster Linie Verfahren der Wahrscheinlichkeitsrechnung. Grundsätzliche Überlegungen stehen in [Cormen *et al.*, 1990] und speziell in [Kemp, 1984]. Wir werden uns nicht so oft mit diesem Problem befassen müssen (s. z.B. 6.3.1 und 4).

Natürlich haben z.B. Programmiersprache, Compiler, Betriebssystem und verwendeter Rechner einen Einfluß auf die effektive Laufzeit des fertigen Programms, aber mit einer einfachen Überlegung erkennt man doch, daß es u.U. andere Kriterien für die Effizienz eines Algorithmus gibt: Betrachten wir zur Illustration zwei Code-Segmente:

```
for i := 1 to n do
    for j := 1 to n do
        s := s + 1
```

```
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to n do
            s := s + 1
```

Nehmen wir an, das linke Fragment liefe auf einem uralten Sinclair Z80, der 10^3 Operationen „`s := s + 1`“ in der Sekunde ausführen kann und nehmen wir ferner an, das rechte auf einer CRAY XMP-4, die 10^7 davon in der Sekunde schafft. Kurzes Nachrechnen ergibt, daß für $n = 10^4$ beide Programme die gleiche Zeit brauchen und daß für größere n das Programm auf dem Sinclair schneller ist. Unter dem Gesichtspunkt des Verhaltens *bei wachsender Größe der Eingabe* ist der linke Algorithmus also effizienter als der rechte.

Dieser Gesichtspunkt wird durch durch die sogenannte **asymptotische Analyse** präzisiert. Man sucht dabei für eine gegebene *Ressourcenfunktion* nach *Vergleichsfunktionen* und vergleicht beide im Hinblick darauf, wie sie sich bei wachsendem Argument verhalten.

2.1 Asymptotische Analyse

Wir haben es bei den Objekten unserer Bemühungen grundsätzlich mit Funktionen zu tun, die die *Ressourcenanforderungen* eines Algorithmus in Abhängigkeit von der Anzahl der Eingaben messen. Im allgemeinen wird es sich also um Abbildungen

$$f : \mathbb{N}^+ \longrightarrow \mathbb{R}$$

handeln, die wir als *Ressourcenfunktionen* bezeichnen. Sinnvollerweise sind solche Ressourcenfunktionen *monoton steigend* [1], sonst hieße das ja, daß die Anforderungen eines Algorithmus bei steigender Anzahl von Eingaben abnehmen!

2.1.1 Die O-Notation

Versuchen wir zunächst, Vergleichsfunktionen zu einer Ressourcenfunktion $g(n)$ zu finden, die sich bei steigendem n „schlechter“ als g verhalten:

[1] ...nicht unbedingt „strikt“

Definition 2.1 *Für eine gegebene Ressourcenfunktion $g(n)$ ist die Funktionsmenge $O(g(n))$ definiert als*

$$O(g(n)) \stackrel{\text{def}}{=} \{f \mid \exists c > 0, n_0 > 0 \colon \forall n \geq n_0 \colon f(n) \leq c \cdot g(n)\}.$$

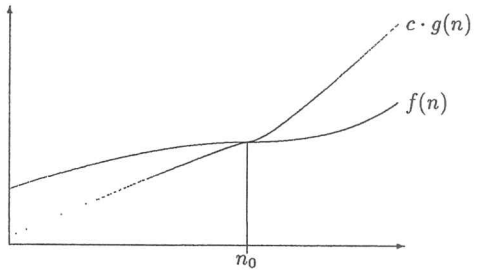


Abbildung 2.1: $f(n) \in O(g(n))$

Abb. 2.1 zeigt diese Definition schematisch! Umgangssprachlich sind die Funktionen in $O(g(n))$ die, die bei wachsendem n irgendwann nicht oberhalb von $c \cdot g(n)$ verlaufen. $g(n)$ ist für diese Funktionen also so etwas wie eine *obere Schranke*, wenn man von einem konstanten Faktor (z.B. durch einen schnelleren Rechner) absieht. Für einen Algorithmus mit der Laufzeit $f(n)$ bedeutet $f(n) \in O(g(n))$, daß er *höchstens* — also im schlechtesten Fall — $g(n)$ Schritte benötigt (abgesehen von einem konstanten Faktor).

Es hat sich in der Literatur eingebürgert, anstelle von $f \in O(g)$ einfach $f = O(g)$ zu schreiben. Gesprochen wird der Ausdruck $O(g)$ als „groß-Oh von g“ oder (im englischen) „big-Oh of g“.

Aussage 2.1 *Es gelten folgende einfachen Eigenschaften:*

1. $f(n) = O(f(n))$.
2. $c \cdot f(n) = O(f(n))$.
3. *Wenn* $f(n) = O(g(n))$ *und* $g(n) = O(h(n))$, *dann* $f(n) = O(h(n))$.

Beispiele

Wir zeigen zuerst: Für $a, b \in \mathbb{R}$, $a, b > 0 : an + b = O(n^2)$.

Beweis: Setzen wir $c = a + b$, so haben wir nur zu zeigen, daß ab einem $n_0 > 0$ gilt $an + b < cn^2 = (a + b)n^2 = an^2 + bn^2$. Das gilt aber offenbar schon für alle $n \geq n_0 = 1$ und damit ist alles gezeigt. Entsprechend folgt $n^k = O(n^{k+m})$, falls $m > 0$.

Aufgabe: *Beweisen Sie das.*

Etwas arg schlampig, aber durchaus üblich, ist der Ausdruck $O(1)$. Dies suggeriert eine Funktion $f(n) = 1$, also eine konstante Funktion. Intuitiv gesprochen ist jede Funktion in

$O(1)$ entweder *monoton fallend* oder *konstant*. Da wir aber nur monoton steigende Ressourcenfunktionen betrachten, bleiben nur die konstanten Funktionen übrig. Algorithmen mit einer Laufzeit von $O(1)$ laufen also immer gleich lang, egal was man eingibt.

Aufwendiger zu beweisen sind folgende Aussagen (s. [Reischuk, 1990]):

1. $n^k = O(r^n)$, für $k \in \mathbb{N}, 1 < r \in \mathbb{R}$. Das bedeutet lediglich, daß ein Polynom beliebiges Grads jede beliebige Exponentialfunktion als obere Schranke hat.
2. $\log_r n = O(n^k)$, für $k \in \mathbb{N}, 1 < r \in \mathbb{R}$. Das ist lediglich der vorige Sachverhalt einmal logarithmiert.
3. $\sqrt{n} = O(n)$

Definieren wir

1. $(f+g)\colon \mathbb{N}^+ \to \mathbb{R}, \quad (f+g)(n) = f(n) + g(n)$
2. $(f \cdot g)\colon \mathbb{N}^+ \to \mathbb{R}, \quad (f \cdot g)(n) = f(n) \cdot g(n)$
3. $\max(f,g)\colon \mathbb{N}^+ \to \mathbb{R}, \quad \max(f,g)(n) = \max(f(n) + g(n)),$
4. $O(f) + O(g) = \{f^* + g^* \,|\, f^* \in O(f) \text{ und } g^* \in O(g)\}$
5. $O(f) \cdot O(g) = \{f^* \cdot g^* \,|\, f^* \in O(f) \text{ und } g^* \in O(g)\}$

so gelten

1. $O(f) + O(g) = O(f+g)$ [Reischuk, 1990]
2. $O(f) \cdot O(g) = O(f \cdot g)$ [Reischuk, 1990]
3. $(f+g) = O(\max(f,g))$ [Brassard und Bratley, 1988]

Diese Aussage erlaubt es, einen einfachen O-Ausdruck durch Maximumsbildung zu errechnen:

$$\begin{aligned} n^3 + 3n^2 + n + 8 &= n^3 + (3n^2 + (n+8)) \\ &= O(\max(n^3, \max(3n^2, \max(n, 8)))) \\ &= O(n^3) \end{aligned}$$

Damit kommt man zu folgendem „Kochrezept“ für die Konstruktion eines O-Ausdrucks für eine gegebene Funktion f :

> Für eine Summe entferne man konstante Faktoren und errechne dann das Maximum der einzelnen Summanden nach der Folge
>
> $$1 < \log n < \sqrt{n} < n < n^2 < \ldots < n^k < n^{k+1} < \ldots c^n.$$
>
> Für ein Produkt bilde man das Produkt der O-Ausdrücke der einzelnen Faktoren und fasse die zu einem zusammen.

2.1.2 Die Ω-Notation

Das Spiegelbild zur O-Notation (für obere Schranken) ist die Ω-Notation, die eine untere Schranke liefert:

Definition 2.2 *Für eine gegebene Ressourcenfunktion $g(n)$ ist die Funktionsmenge $\Omega(g(n))$ definiert als*

$$\Omega(g(n)) \stackrel{\text{def}}{=} \{f \,|\, \exists c > 0, n_0 > 0\colon \forall n \geq n_0\colon c \cdot g(n) \leq f(n)\}.$$

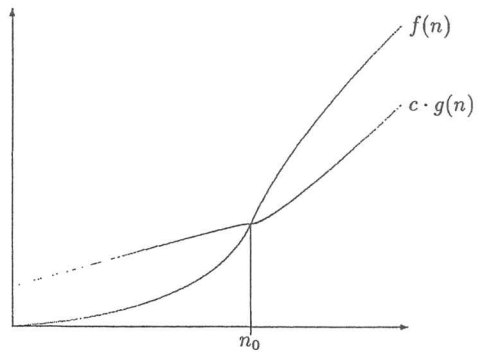


Abbildung 2.2: $f(n) \in \Omega(g(n))$

Für einen Algorithmus mit der Laufzeit $f(n)$ bedeutet $f(n) = \Omega(g(n))$[2], daß er *mindestens* — also im besten Fall — $g(n)$ Schritte benötigt (abgesehen von einem konstanten Faktor).

Es gilt die leicht einzusehende Beziehung:

$$f(n) = O(g(n)) \quad \Leftrightarrow \quad g(n) = \Omega(f(n)).$$

Alle weiteren wichtigen Eigenschaften der O-Notation gelten (u.U. umgedreht) auch für die Ω-Notation.

2.1.3 Beispiel

Nehmen wir nun noch einmal die beiden Programme von oben unter die Lupe:

```
for i := 1 to n do
    for j := 1 to n do
        s := s + 1
```

```
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to n do
            s := s + 1
```

Da `s := s + 1` offenbar $O(1)$ ist, kann man für das linke Fragment einige obere Schranken bestimmen: Z.B. $O(2^n)$, $O(n^9)$ und auch $O(n^2)$. Natürlich sind solche Schranken wie $O(2^n)$ ausgesprochen langweilig: Die uns interessierende Schranke ist natürlich $O(n^2)$.

[2] Dies ist eine Fußnote und *kein* Exponent! Wir benutzen hier natürlich wieder die Schreibweise $f(n) = \Omega(g(n))$ anstelle von $f(n) \in \Omega(g(n))$.

Eine Aussage „die Laufzeit ist $O(f(n))$" bedeutet ab sofort, daß $f(n)$ die *niedrigste* obere Schranke der Laufzeit des Algorithmus darstellt.

Eine Aussage „die Laufzeit ist $\Omega(f(n))$" bedeutet ab sofort, daß $f(n)$ die *höchste* untere Schranke der Laufzeit des Algorithmus darstellt.

Es ergibt sich für die Laufzeit des linken Fragments eine obere Schranke von $O(n^2)$ und eine untere Schranke von $\Omega(n^2)$.

Aufgabe: *Und für das rechte Fragment?*

Solche Codepassagen mit gleicher oberer und unterer Schranke kommen sehr häufig vor. Sie entstehen grundsätzlich beim Einsatz von `for`-Schleifen. In einem solchen Fall ist diese Vergleichsfunktion eine sehr präzise Beschreibung des Laufzeitverhaltens und verdient eine eigene Notation.

2.1.4 Die Θ-Notation

Definition 2.3 *Für eine gegebene Ressourcenfunktion $g(n)$ ist die Funktionsmenge $\Theta(g(n))$ definiert als*

$$\Theta(g(n)) \stackrel{\text{def}}{=} \{f \mid \exists c_1, c_2 > 0, n_0 > 0\colon \forall n \geq n_0\colon c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}.$$

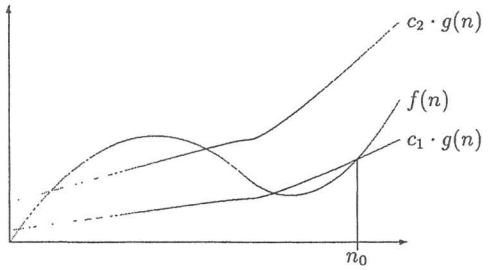


Abbildung 2.3: $f(n) \in \Theta(g(n))$

Θ wird auch als *asymptotisch enges* Maß bezeichnet. Unsere beiden Codefragmente von oben sind also $\Theta(n^2)$ bzw. $\Theta(n^3)$.

2.2 Häufig auftretende Ressourcenanforderungen

- $O(1)$ (konstant). Ein Algorithmus ohne Schleifen, die von der Anzahl der Eingabedaten abhängen (Laufzeit) oder der nur einfache Variablen oder solche mit konstanter Größe verwendet (Speicherbedarf).

- $O(\log n)$ (logarithmisch, unabhängig von der Basis). Ein Algorithmus, der (z.B. durch fortgesetzte Halbierung der Eingabe) nur logarithmisch viele Elemente anguckt (Laufzeit). Logarithmischer Speicherbedarf kann durch entsprechende Rekursion (auf dem Laufzeitkeller) zustandekommen.
- $O(n)$ (linear). Ein Algorithmus, der jedes Eingabeelement einmal oder öfter anguckt (Laufzeit) und einmal oder öfter speichert (Speicherbedarf).
- $O(n \log n)$ (überlogarithmisch). Entsteht häufig durch Unterteilung in kleinere Unterprobleme (Quicksort, s. 4.3).
- $O(n^2)$ (quadratisch). Typisch für paarweise Verarbeitung, z.B. einfache Sortierverfahren („jeder mit jedem").
- $O(n^3)$ (kubisch). Einige Verfahren zum Lösen von Gleichungssystemen, einige Optimierungs- und Parsingverfahren (s. 10.1, 10.3).
- $O(2^n)$ (exponentiell). Klassische Beispiele: Aufzählen der Potenzmenge einer Menge mit n Elementen, Aufzählen der möglichen Belegungen von n booleschen Variablen mit den beiden Wahrheitswerten.
- $O(n!)$ (faktoriell). Beispiel: Aufzählen aller Permutationen von n Elementen.

Man kann sich eine kleine Faustregel ins Gedächtnis einprägen: *Mit einer Komplexität unterhalb von $O(n)$, also etwa $O(1)$ oder $O(\log n)$, kann man kaum etwas nicht-triviales anstellen.*

Der Grund besteht darin, daß ein Algorithmus von solcher Komplexität nicht einmal alle seine Eingabedaten angucken kann. Dafür braucht er ja schon n Schritte. Entweder tut der Algorithmus irgendetwas fürchterlich uninteressantes — irgendein Element aus einem Array lesen ist $O(1)$ — oder seine Eingabedaten sind schon sehr hoch strukturiert — Binärsuche in einem sortierten Array ist $O(\log n)$ — und man fragt sich, wie teuer es war, diese Struktur aufzubauen.

Im allgemeinen bezeichnet man alle Algorithmen, deren Komplexität nach oben durch ein Polynom beschränkt ist, also $O(\pi(n))$ mit einem Polynom π, als *handhabbar* (engl. tractable), während etwa ein $O(2^n)$-Algorithmus als *nicht handhabbar* (engl. intractable) gilt. Man mag darüber streiten, ob ein Algorithmus mit der Komplexität $O(n^{93})$ als „handhabbar" bezeichnet werden kann. Interessanterweise sind allerdings solche Polynome in der Praxis recht selten — oder fällt Ihnen ein Verfahren mit dieser Komplexität ein? Erfahrungsgemäß gilt für polynomial beschränkte Algorithmen, daß der Grad des Polynoms meistens ≤ 3 ist.

Andererseits ist ein $O(n^2)$ Verfahren manchmal auch schon nicht mehr praktikabel, wenn es sich um große n handelt. Typische Beispiele sind eben die bekannten Sortierprobleme, bei denen der Schritt vom quadratischen zum überlogarithmischen Verfahren ganz entscheidend sein kann, wenn man große Mengen sortieren will.

Einen kleinen Überblick über das Wachstum bestimmter wichtiger Komplexitätsfunktionen gibt die folgende Tabelle.

n	$\log_2 n$	$n \log_2 n$	n^2	2^n	$n!$
1	0.0	0.0	1	2	1
5	2.32	11.6	25	32	120
10	3.32	33.2	100	1024	3628800
50	5.64	282.2	2500	$1 \cdot 10^{15}$	$3 \cdot 10^{64}$
100	6.64	664.4	10000	$1 \cdot 10^{30}$	$9 \cdot 10^{157}$
500	8.97	4482.9	250000	$3 \cdot 10^{150}$	$1 \cdot 10^{1134}$
1000	9.97	9965.8	1000000	$1 \cdot 10^{301}$	$1 \cdot 10^{2567}$
5000	12.29	61438	$2.5 \cdot 10^7$	$1 \cdot 10^{1505}$	$4 \cdot 10^{16325}$
10000	13.29	132877	$1 \cdot 10^8$	$2 \cdot 10^{3010}$	$3 \cdot 10^{35659}$

Aufgabe: *Entwickeln Sie eine Idee, mit der Sie den angenäherten Wert* $10000! = 3 \cdot 10^{35659}$ *auf einem normalen Computer bestimmen können. Für Besitzer eines Mathematikbuchs: Gucken Sie im Stichwortverzeichnis unter dem Begriff „Stirlingsche Formel" nach! Wenn Ihr Buch diesen Begriff nicht kennt, werfen Sie es weg.*

Man könnte verzweifeln, wenn man nur das Wachstum in der $n \log_2 n$-Spalte anguckt! Anscheinend sind sowieso nur die logarithmischen Verfahren in der Praxis brauchbar, vielleicht noch die linearen. Alles „darüber" scheint schrecklich zu sein. Aber das läßt sich leider nicht immer realisieren. Meistens sind wir schon zufrieden, wenn wir es mit einem quadratischen Algorithmus zu tun haben.

2.3 Analyse rekursiver Algorithmen

Im iterativen Fall ist die Komplexitätsanalyse einigermaßen einfach; schlimmer wird die Geschichte, wenn man sich Algorithmen anguckt, die *rekursiv* definiert sind. Wie nicht anders zu erwarten, steht man dann vor dem Problem, auch die Laufzeitfunktion rekursiv aufgetischt zu bekommen und daraus eine geschlossene Formel ableiten zu müssen.

Nehmen wir ein (vielleicht) bekanntes Beispiel: *Die Türme von Hanoi*:

> Gegeben sind n Scheiben abnehmender Größe, die auf alle übereinander so auf einem Platz p_1 liegen, daß auf jeder Scheibe immer eine kleinere liegt (so pyramidenförmig). Die Aufgabe besteht darin, diesen Turm auf einen Platz p_2 zu versetzen, wobei
>
> 1. immer nur die oberste Scheibe eines Turms bewegt werden darf und
> 2. auf jede Scheibe immer eine kleinere gelegt werden muß.
>
> Dazu steht ein „Hilfsplatz" p_3 zur Verfügung, auf dem ebenfalls Scheiben abgelegt werden dürfen.

Ohne lange zu philosophieren, hier ein rekursiver Algorithmus, der das tut:

Algorithmus 2.1 *Lösung des Problems der Türme von Hanoi.* `n` *ist die Anzahl der zu versetzenden Scheiben, die auf dem Platz* `p1` *stehen.* `p2` *ist der Platz, auf den die Scheiben sollen,* `p3` *ist der Hilfsplatz.*

Precondition: `Alle Scheiben auf p1`
Postcondition: `Alle Scheiben auf p2`

```
procedure hanoi(n,p1,p2,p3)
  if n = 1 then                          {Eine Scheibe: Trivial!}
     "Setze Scheibe von p1 nach p2"
  else
     {Versetze alle bis auf die oberste}
     {von p1 nach p3 ueber p2}
     hanoi(n-1,p1,p3,p2)

     "Setze Scheibe von p1 nach p2"

     {Versetze alle eben versetzten zurueck}
     {von p3 nach p2 ueber p1}
     hanoi(n-1,p3,p2,p1)
```

Nehmen wir mal an, die Operation „`Setze Scheibe von p1 nach p2`" sei von der Komplexität $O(1)$ und fragen nach der Laufzeit in Abhängigkeit von n, mal geschrieben als $T(n)$.

Wir wissen zunächst aus dem Code: $T(1) = 1$, denn in diesem Fall wird nur eine Scheibe verschoben. Und sonst? Ist das Programm rekursiv, so schreiben wir auch die Laufzeit rekursiv:

$$T(n) = \begin{cases} & T(n-1) & \texttt{hanoi(n-1,p1,p3,p2)} \\ + & 1 & \texttt{Setze Scheibe von p1 nach p2} \\ + & T(n-1) & \texttt{hanoi(n-1,p3,p2,p1)} \end{cases}$$

Nun brauchen wir eine geschlossene Formel für $T(n)$, wenn wir an seine Komplexität in O, Ω oder Θ herankommen wollen und das sieht nicht so einfach aus. Wir schlagen hier eine praktikable Möglichkeit zur Lösung dieses Problems vor:

„Raten und Beweisen".

2.3.1 Raten und Beweisen

Diese Methode führt im Beispiel von Algorithmus 2.1 sehr schnell zum Ergebnis. Zunächst rät man eine geschlossene Formel für $T(n)$ und beweist sie dann. Raten kann man am besten durch genaues Angucken der Rekursionsformel:

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T(n-1) + 1 & \text{sonst} \end{cases} \tag{2.1}$$

Setzen wir mal einige Schritte ein:

$$\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2(2T(n-2)+1)+1 \\
&= 4T(n-2)+3 \\
&= 4(2T(n-3)+1)+3 \\
&= 8T(n-3)+7 \\
&= \ldots \\
&= 2^kT(n-k)+2^k-1 \qquad \text{(die Vermutung!)} \\
&= 2^{n-1}T(n-(n-1))+2^{n-1}-1 \\
&= 2^{n-1}T(1)+2^{n-1}-1 \\
&= 2^{n-1}\cdot 1+2^{n-1}-1 \\
&= 2^n-1
\end{aligned}$$

Das war also das Raten. Raten ist nicht schlecht, aber Beweisen gibt Sicherheit:

Aussage 2.2 *Die Rekursionsgleichung 2.1 ist äquivalent zu*

$$T(n) = 2^n - 1.$$

Beweis: (durch vollständige Induktion über n).

1. *Induktionsverankerung:* $T(1) = 2^1 - 1 = 2 - 1 = 1$ □
2. *Induktionsannahme:* $T(n) = 2^n - 1$.
3. *Induktionsschritt:*

$$\begin{aligned}
T(n+1) &= 2T(n)+1 && \text{Definition} \\
&= 2(2^n-1)+1 && \text{Induktionsannahme} \\
&= 2^{n+1}-2+1 \\
&= 2^{n+1}-1 \ \square
\end{aligned}$$

Insgesamt bekommen für die Laufzeit des Hanoi-Algorithmus das unangenehme $\Theta(2^{n+1})$. Raten wir mal für folgendes Programm-Schema:

```
procedure proc1(n)
  if n = 1 then
     {Irgendwas mit O(1)}
  else
     proc1(n/2)
     {Irgendwas mit O(1)}
     proc1(n/2)
```

Offenbar bekommen wir als Rekursionsgleichung

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T(n/2)+1 & \text{sonst} \end{cases} \tag{2.2}$$

Hier geht es nun nicht mehr so weiter wie eben: Wir ertrinken in Divisionen durch 2. Aber eine kleine Umformung führt ans Ziel.

> Für Rekursionsgleichungen wie (2.2), bei denen die Rekursion durch eine Division durch 2 bestimmt ist, setze man $n = 2^k$ und versuche es damit.

Aufgabe: *Zeigen Sie, daß für Gleichung (2.2) gilt:* $T(2^k) = 2^{k+1} - 1$.

Unsere Vermutung besteht also darin, daß allgemein $T(n) = 2n - 1 = \Theta(n)$. Noch zwei zum Testen:

```
procedure proc2(n)
  if n = 1 then
     {Irgendwas mit O(1)}
  else
     proc(n/2)
     {Irgendwas mit O(n)}
     proc(n/2)
```

```
procedure proc3(n)
  if n = 1 then
     {Irgendwas mit O(1)}
  else
     {Irgendwas mit O(n)}
     proc(n/2)
```

Aufgabe:

1. *Stellen Sie für beide Algorithmen die Rekursionsgleichungen auf.*
2. *Zeigen Sie, daß für der linke Algorithmus eine Laufzeit von* $\Theta(n \log n)$ *hat.*
3. *Geben Sie für den rechten Algorithmus seine Laufzeit in* Θ*-Notation an.*
4. *Geben Sie eine Prozedur an, die der Rekursionsgleichung*

$$T(n) = \begin{cases} n & \text{falls } n = 1 \\ 2T(n/2)+n & \text{sonst} \end{cases}$$

genügt und lösen Sie die Gleichung auf.

2.4 Zusammenfassung

Die asymptotische Analyse abstrahiert weitgehend von vielen praktisch wichtigen Aspekten des Ressourcenbedarfs von Algorithmen. Sie ist für die Praxis lediglich bedingt geeignet, wie man sehr deutlich in 4.3.5 und 4.4 sehen kann.

Natürlich kann ein asymptotisch schlechterer Algorithmus in bestimmten Fällen einem besseren vorzuziehen sein, weil die ignorierten multiplikativen Konstanten und Terme niedriger Ordnung gerade bei kleinen Anzahlen ziemlich ins Gewicht fallen können. Die

asymptotische Analyse liefert Aussagen über das Verhältnis zwischen Algorithmen lediglich für große Anzahlen von Eingabedaten und ist in der Praxis immer einer genaueren Inspektion zu unterwerfen.

Aufgabe: *Gegeben ein Algorithmus mit dem Laufzeitverhalten* $T_1(n) = 2^n + 2$ *und einer mit* $T_2(n) = 2x^3 + 2x^2 + 3x + 7$. *Finden Sie eine Zahl* n_0 *derart, daß für weniger als* n_0 *Eingabedaten der erste vorzuziehen ist.*

Die Verwendung der asyptotischen Notation hat den niedlichen Effekt, daß es bei den Logarithmen, die recht häufig bei der Komplexitätsanalyse von Algorithmen auftreten, keinen Sinn macht, die Basis mit anzugeben.

Aufgabe: *Warum?*

Wir werden dementsprechend im Normalfall die Basis eventuell auftretender Logarithmen fortlassen, obwohl es sich normalerweise immer um die Basis 2 handeln wird. Studiert man die entsprechenden Algorithmen, z.B. Quicksort (4.3) oder externe Sortierverfahren (5.1), so kann man — mit einiger Phantasie — Ideen entwickeln, mit denen man intuitiv einleuchtende Verbesserungen realisieren kann, die aus der Basis 2 eine Basis von 3 oder größer machen kann.

Asymptotisch bringen solche „Verbesserungen" gar nichts, wie wir nun wissen. Algorithmisch führen sie gewöhnlich zu erheblichen Komplizierungen, die man nur unter massivem Druck in Kauf nehmen sollte. Verbesserungen, die nicht asymptotisch bemerkbar sind, sollte man nur dann vornehmen, wenn ein fest vorgegebenes Problem nicht mit den verfügbaren Ressourcen gelöst werden kann. In allen anderen Fällen ist ein einfacher und übersichtlicher Algorithmus vorzuziehen.

Die hier vorgestellten Techniken der asymptotischen Analyse stellen nur einen kleinen Ausschnitt aus dem theoretischen Repertoire dar. Für eine ausführlichere Behandlung sei auf [Cormen *et al.*, 1990], [Reischuk, 1990] und [Graham *et al.*, 1994] verwiesen.

Kapitel 3

Elementare Datenstrukturen

Hier wollen wir zunächst die wichtigsten Begriffe definieren, die wir für unsere Zwecke benötigen. Weiter werden wir elementaren Datenstrukturen kennenlernen, mit denen man an die Implementation solcher Objekte herangehen kann. Nochmal sei darauf hingewiesen, daß es durchaus möglich ist, ein und dieselbe Struktur mit verschiedenen Techniken darzustellen, und daß es sehr wohl auf den angepeilten Algorithmus ankommt, wofür man sich letzlich entscheidet.

Wir unterscheiden hier zwischen

- *linearen* bzw. *kreisförmigen* Datenstrukturen, die Bewegungen in eine oder zwei Richtungen zulassen und
- *Graphen* und (als spezielle Graphen) *Bäume*, bei denen an jedem Punkt eine Entscheidung über die nächste Bewegungsrichtung erforderlich ist.

3.1 Lineare und kreisförmige Strukturen

Die einfachsten Datenstukturen, die einem Softwerker zur Verfügung stehen, sind kettenförmige Gebilde, die i.w. eine sequentielle Verarbeitung der Elemente zulassen.

3.1.1 Einfach und doppelt verkettete Listen

Das allereinfachste Beispiel ist die *verkettete Liste*, dargestellt in Abb. 3.1.

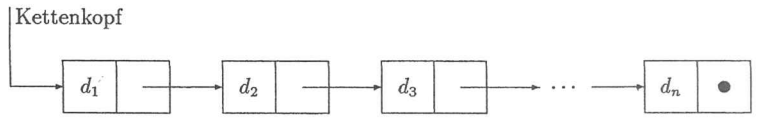


Abbildung 3.1: Eine einfach verkettete Liste

Jedes Element einer solchen verketteten Liste enthält also Daten d_i, die von der Anwendung gebraucht werden und einen Zeiger `next`, der lediglich zur Listenorganisation dient

und auf den Nachfolger des Knotens zeigt. Der ausgezeichnete Zeigerwert `nil` sagt aus, daß dort kein Nachfolger mehr steht, der Knoten also der letzte in der Kette ist.

Der *Kettenkopf* ist ein separater Zeiger, der für jede Kette benötigt wird und auf das erste Element der Kette zeigt. Ist der Kettenkopf `nil`, so enthält die Kette kein einziges Element.

Die verkettete Liste unterstützt die Operationen:

1. `insert`: Das Einfügen eines Elements
2. `look-up`: Das Suchen eines Elements
3. `nth`: Das n-te Element in der Kette
4. `delete`: Das Löschen eines Elements

Hier ganz schnell Pseudocode für diese vier Operationen in ihrer einfachsten Form:

Algorithmus 3.1 *Einfügen, Suchen, direkter Zugriff und Löschen in einer einfach verketteten Liste mit dem Kettenkopf* `head`.

```
procedure chained-list-insert(var head; datum)

  {Fuegt "datum" am Anfang der verketteten Liste ein}

  next(datum) := head
  head        := datum

{-----------------------------------------------------------------}

function chained-list-lookup(head; datum)

  {Liefert den Zeiger auf den Knoten mit datum,
     falls so ein Knoten in der Liste steht.
   Liefert nil, wenn so ein Knoten nicht existiert.}

  current := head
  while (current <> nil) and (datum(current) <> datum) do
     current := next(current)

  return current

{-----------------------------------------------------------------}

function chained-list-nth(head; n)

  {Liefert das n-te Element der verketteten Liste
   Liefert nil, wenn die Kette weniger als n Elemente enthaelt}

  current := head
  count   := 0
```



```
  while (current <> nil) and (count < n) do
     count   := count + 1
     current := next(current)

  return current

{-----------------------------------------------------------------}

procedure chained-list-delete(head; datum)

  {Loescht den Knoten mit "datum" aus der Liste}

  if head <> nil then
     if datum(head) = datum then
        head := next(head)
     else
        current := head

        {"datum" suchen}
        while (current <> nil) and (datum(current) <> datum) do
           prev    := current
           current := next(current)

        {Falls gefunden, ausklinken}
        if current <> nil then
           next(prev) := next(current)
```

Die Komplexität der Operationen ist sehr einfach zu bestimmen. Wenn in der Kette n Elemente gespeichert sind, bekommen wir:

- `insert`: $\Theta(1)$
- `look-up`: $O(n)$
- `delete`: $O(n)$

Hat man auf den Daten einen *Schlüssel* definiert, der als Suchbegriff fungieren soll, und existiert zwischen den Schlüsseln eine *Ordnung* „$<$“, so kann man — auf Kosten etwas höheren Aufwands beim Einfügen — die Reihenfolge der Elemente in der Kette aufsteigend sortiert halten und beim erfolglosen Suchen schneller abbrechen. Man muß ja nicht bis ganz zum Ende der Kette laufen sondern nur bis zu dem Punkt, an dem ein Element mit einem größeren Schlüssel auftritt. Danach kann der gesuchte Schlüssel ja nicht mehr kommen.

Hier als Beispiel nur das Einfügen:

Algorithmus 3.2 *Einfügen des Elements* `key` *in eine verkettete Liste. Zwischen den Schlüsseln der Elemente sei eine Ordnung „$<$“ definiert:*

Precondition: ---
Postcondition: `Kette enthält key`

```
procedure chained-list-insert(var head; key)
  if head = nil then          {Kette ist leer}
     head := key
  else
     if key(head) >= key      {Vorne einfuegen}
        next(key) := head
        head      := key
     else                     {Dahinter einfuegen}
        current := head
        while (current <> nil) and (key(current) < key) do
           prev    := current
           current := next(current)

        next(key)  := current
        next(prev) := key
```

Aufgabe: *Geben Sie Pseudocode für die anderen Operationen an, in denen diese Eigenschaft ein Rolle spielt.*

Man kann natürlich neben dem `next`-Zeiger noch einen umgekehrt laufenden `pred`-Zeiger verwenden, mit dem man von einem Knoten auch auf seinen Vorgänger zugreifen kann, s. Abb. 3.2.

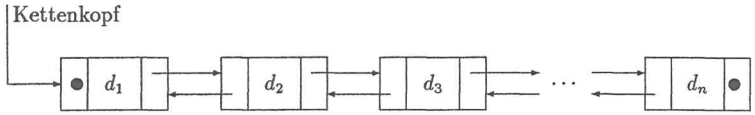


Abbildung 3.2: Eine doppelt verkettete Liste

Aufgabe: *Geben Sie Pseudocode für das Einfügen, Suchen und Löschen in einer doppelt verketteten Liste an.*

3.1.2 Ringlisten

Ringlisten (manchmal auch Kreislisten genannt) entstehen aus doppelt verketteten Listen dadurch, daß man die beiden zwangsläufig auftretenden `nil`-Zeiger so umbiegt, daß sich beim Durchlaufen ein Kreis ergibt, s. Abb. 3.3.

Ringe unterstützen z.B. den Zugriff auf das letzte Element, ohne daß man einen separaten Zeiger dafür beötigt. Kennt man die maximale Anzahl der Elemente eines Rings `max` von Anfang an, so kann man es auch in ein entsprechend dimensioniertes Array `ring[0:max-1]` abbilden. Die Zugriffe auf `pred` und `next` lassen sich dann einfach durch Subtraktion bzw. Addition von 1 modulo `max` realisieren.

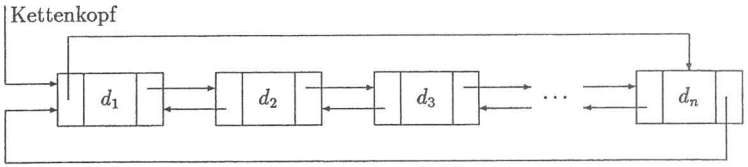


Abbildung 3.3: Eine Ringliste

3.1.3 Darstellung höherer Strukturen

Verkettete Strukturen eignen sich ganz vorzüglich, um wichtige „höhere" Datenstrukturen zu realisieren. Stacks und Queues sind dabei die einfachsten.

Stacks

Für die Realisierung von Stacks verwenden wir die oben angegebene Prozedur `chained-list-insert` zur Realisierung von `push`, die ein Element vorne in die Kette einfügt. Damit liegen die wichtigsten anderen Funktionen für Stacks schon fest:

- `empty`: Wird realisiert durch den Test `head = nil`
- `top`: Liefert das erste Element der Kette zurück
- `pop`: Entfernt das erste Element der Kette.

`top` und `pop` sollten einfach zu realisieren sein!

Queues

Hier muß man sowohl am Anfang als auch am Ende der Kette operieren. Sinnvollerweise wird man für das Kettenende einen eigenen Zeiger vorsehen und mit einer doppelten Verkettung arbeiten, damit man beim Entfernen am Kettenende den Vorgänger leichter findet. Es ergeben sich dann die Operationen:

- `enqueue`: Wie `push` beim Stack.
- `empty`: Wie beim Stack.
- `front`: Liefert das letzte Element der Kette.
- `dequeue`: Entfernt das letzte Element der Kette.

Deques

Ein Deque („double ended queue", manchmal auch „Deck" geschrieben und auch so ausgesprochen) ist eine Datenstruktur, die anschaulich einem Kartenstapel gleicht. Man kann sowohl oben (am Anfang der Kette) als auch unten (am Ende der Kette) Elemente einfügen, angucken und entfernen. Sie lassen sich natürlich dadurch realisieren, daß man die Operationen für Stacks und Queues miteinander vereinigt. Deshalb hier kein weiterer Kommentar dazu.

Priority Queues

Auch Priority Queues bilden kein großes Problem. Sie unterscheiden sich von „normalen" Queues lediglich dadurch, daß anstelle des Einfügens am Kettenanfang (Algorithmus 3.1) die Version verwendet wird, die die Ordnung auf den Schlüsseln verwendet (Algorithmus 3.2). Alle anderen Operationen bleiben unberührt.

Komplexität

Die Komplexität der einzelnen Operationen in der Realisierung durch verkettete Listen ist nach den vorigen Bemerkungen einfach zu bestimmen. Tabelle 3.1 zeigt alles noch einmal.

	Test auf leer	Einfügen	Inspzieren	Löschen
Stack	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Queue	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Deque	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Priority Queue	$\Theta(1)$	$O(n)$	$\Theta(1)$	$\Theta(1)$

Tabelle 3.1: Komplexität von Operationen auf höheren Datenstrukturen bei Realisierung durch verkettete Listen

Lediglich die Priority Queue fällt mit ihrem linearen Aufwand für das Einfügen ein wenig aus dem Rahmen. Wir werden allerdings später (in 4.2.3) eine andere Implementation kennenlernen, die andere Charakteristika aufweist.

3.2 Graphen und Bäume

Allgemeine Graphen und eingeschränkte Varianten davon sind wohl die universalen Hilfsmittel zur Darstellung (und Veranschaulichung) wichtiger Tatbestände. Hier nur kurz die wichtigsten Definitionen; näheres findet man z.B. in [Maas, 1993].

3.2.1 Grundlegende Begriffe

Anschaulich gesprochen ist ein *Graph* ein Gebilde aus *Ecken* und *Kanten*. Ecken sind durch Kanten miteinander verbunden, wobei man solchen Kanten eine *Richtung* zuschreiben kann — „eine Kante von a nach b" — oder auch nicht — „eine Kante zwischen a und b".

Definition 3.1 *Ein* **Graph** $G = (V, E)$ *besteht aus einer (hier endlichen) Menge* $V = \{v_1, v_2, \ldots, v_m\}$ *von* Ecken *(engl.* vertices, *sing.* vertex[1]*) und einer (hier ebenfalls endlichen) Menge* $E = \{e_1, e_2, \ldots, e_n\}$ *von* Kanten *(engl.* edges*).*

[1]Daher die Bezeichnungen V und v_i. E für die Eckenmenge findet man im deutschen Sprachraum auch manchmal, führt aber zu Verwechslungen mit der Menge E, die wir hier auch brauchen. Man muß dann immer wissen, ob der Autor das deutsche E (Ecke) oder das englische E (edge) meint. Hier meinen wir das englische!

Eine Kante e_k *ist von der Form* $e_k = (v_i, v_j)$, $v_i, v_j \in V$.

Zu Graphen gibt es einige Begriffe, die wir noch kurz anführen wollen:

Definition 3.2 *Zwei verschiedene Ecken* v_i *und* v_j *heißten* **verbunden**, *wenn es Kanten* $e_{k_1} = (v_i, v_{l_1}), e_{k_2} = (v_{l_1}, v_{l_2}), \ldots, e_{k_m} = (v_{l_{m+1}}, v_j)$ *gibt, also eine von* v_i *nach* v_j *führende ununterbrochene Folge von Kanten.*

Ein Graph heißt **zusammenhängend**, *wenn je zwei verschiedene Ecken des Graphens verbunden sind.*

Eine Kantenfolge $e_{k_1} = (v_i, v_{l_1}), e_{k_2} = (v_{l_1}, v_{l_2}), \ldots, e_{k_m} = (v_{l_{m+1}}, v_i)$, *die also die Ecke* v_i *mit sich selbst verbindet, heißt* **Zyklus** *(Kreis, Schleife, engl. cycle).*

Ein Graph heißt **zyklenfrei** *(kreisfrei, engl. acyclic), wenn er keinen Zyklus enthält.*

Die angesprochene Frage der Richtung einer Kante wirft definitorisch einige kleine Probleme auf. Der Grund liegt darin, daß man einer Kante $e_k = (v_i, v_j)$ dann die Ecke v_i als *Startecke*, die andere v_j als *Endecke* zuordnen muß und kein „richtig" formales Kriterium findet, das einigermaßen anschaulich ist. Natürlich kann man grundsätzlich von gerichteten Kanten ausgehen und einen Graphen als *ungerichtet* bezeichnen, wenn es zu jeder Kante $e_k = (v_i, v_j)$ auch die umgekehrte Kante $e'_k = (v_j, v_i)$ gibt.

Aufgabe: *Machen Sie sich bitte klar, daß ein Graph als Eckenmenge* V *mit einer Relation* E *auf* V *aufgefaßt werden kann. Ungerichtet wäre er dann, wenn* E symmetrisch *ist.*

Ganz wichtig für viele Anwendungen sind DAGs: *Ein* **DAG** *(directed acyclic graph) ist ein gerichteter zyklenfreier Graph.*

3.2.2 Bäume und ihre Darstellung

Bäume stellen in gewisser Weise die wichtigste nicht-lineare Datenstruktur der Informatik dar. Man findet sie in allen möglichen Anwendungen, u.U. auch dort, wo man sie kaum vermutet. Sie treten auf als

- Darstellung teilsortierter Strukturen (4.2.1, Kapitel 7),
- Darstellung von Entscheidungsmöglichkeiten (Kapitel 11),
- interne Darstellung für die syntaktische Struktur von Sprache, z.B. im Compilerbau (10.3).

Definition 3.3 *Ein zusammenhängender, kreisloser Graph heißt* **Baum**.

Das entspricht dem informatischen Verständnis eines Baums eigentlich nicht so ganz. Wir wollen lieber so etwas wie eine „Wurzel", „Vorgänger" und „Nachfolger" haben. Was die Informatik mit einem „Baum" meint ist das, was die Graphentheoretiker einen „Wurzelbaum" (engl. rooted tree) nennen. Ein Blick auf Abb. 3.4 sollte den Unterschied verständlicher machen.

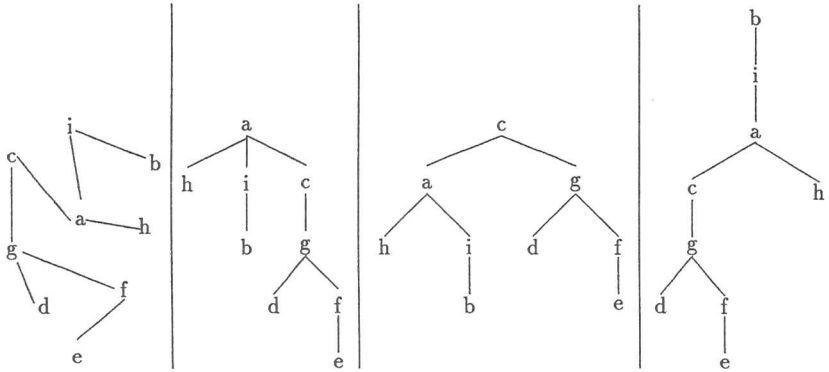


Abbildung 3.4: Ein Baum und verschiedene dazugehörende Wurzelbäume

Dort sieht man links einen Baum nach Def. 3.3. Ein Wurzelbaum weist genau dieselbe Struktur auf, es ist lediglich eine Ecke zur „Wurzel" bestimmt worden. Man kann sich das so vorstellen, als nehme man eine Ecke und ziehe damit den Baum aus der Zeichenebene heraus. Was man dann in der Hand hält, ist der zum gegebenen Baum gehörende Wurzelbaum mit der gewählten Ecke als Wurzel.

> Wenn wir hier von einem „Baum" reden, meinen wir *ab sofort* immer einen „Wurzelbaum"!
>
> Weiterhin sprechen wir im Zusammenhang mit Bäumen immer von „Knoten" anstatt von „Ecken".

Nun macht es auch Sinn, von „Vorgängern" und „Nachfolgern" zu sprechen. Die Wurzel besitzt keine Vorgänger, diejenigen Knoten, die keinen Nachfolger haben, nennen wir „Blätter". Zu jedem Knoten `k` eines Baums notieren wir den *Vorgänger* von `k` mit `parent(k)`.

Bisher hingen die Nachfolger eines Knotens in beliebiger Reihenfolge drunter, was den Effekt hat, daß bei gegebener Wurzel sogar mehrere Wurzelbäume existieren. Legt man nun auf den Nachfolgern eines Knotens eine Ordnung fest, so spricht man von einem *geordneten Baum*. Beispiele für geordnete Bäume sind die in Kürze zur Debatte stehenden Binärbäume (7.1) und die allgemeinen B-Bäume (7.3).

Prinzipiell kann ein Knoten in einem Baum beliebig viele Nachfolger haben. Wir bezeichnen für einen Knoten die Anzahl seiner Nachfolger als *Grad* des Knotens. Daraus kann man eine Definition ableiten:

Definition 3.4 *Ein Baum, dessen Knoten alle einen Grad $\leq n$, für ein gegebenes n haben, heißt* **n-ärer Baum** *(Baum vom Grad n, engl. n-ary tree).*

Ganz wichtig sind die binären Bäume:

Definition 3.5 *Ein geordneter Baum vom Grad 2 heißt* **binärer Baum** *(Binärbaum, engl. binary tree).*

Für Binärbäume kann man die Nachfolger eines Knotens ganz einprägsame Namen finden:

- `left(k)` ist der linke (erste) Nachfolger von `k`.
- `right(k)` ist der rechte (zweite) Nachfolger von `k`.

Höhenbalancierte Bäume

Für die Beschreibung quantitativer Eigenschaften von Bäumen spielt die „Höhe" eine entscheidende Rolle.

Definition 3.6 *Für einen Baum T ist die* **Höhe** *(engl. height) $h(T)$ definiert als die größte Anzahl von Kanten zwischen der Wurzel und einem Blatt.*

In Abb. 3.4 sind die Höhen der drei Bäume 4, 3 und 6. Wir werden später sehen, daß algorithmisch von zwei Bäumen der mit der kleineren Höhe vorzuziehen ist. Natürlich hängt die Höhe eines n-ären Baums nicht zuletzt von der Anzahl der Knoten ab. Wichtig ist folgende Aussage für Binärbäume:

Aussage 3.1 *Für einen Binärbaum T mit n Knoten gilt*

$$\lfloor \log_2 n \rfloor \leq h(T) \leq n - 1.$$

Man wird nun nach einer Charakterisierung von Binärbäumen suchen, die minimale Höhe haben, s. Abb. 3.5, links.

Abbildung 3.5: Ein voller und ein vollständiger Binärbaum

Definition 3.7 *Ein Binärbaum heißt* **voll**, *wenn alle Blätter auf derselben Ebene liegen.*

Volle Binärbäume sind selten, denn es gibt zu jeder Höhe i.w. nur einen einzigen.

Aussage 3.2 *Ein voller Binärbaum der Höhe h hat $2^{h+1} - 1$ Knoten.*

Für Bäume, die zufälligerweise *nicht* über eine solche Anzahl von Knoten verfügen, gibt es eine andere Bezeichnung:

Definition 3.8 *Ein Binärbaum heißt* **vollständig**, *wenn*

1. *jede Ebene bis auf die unterste vollständig besetzt ist und*
2. *auf der untersten Ebene höchstens Knoten ganz rechts fehlen.*

In Abb. 3.5 sehen wir rechts einen vollständigen Binärbaum.

Realisierung vollständiger Bäume durch Arrays

Vollständige Binärbäume lassen sich ganz einfach in ein Array abbilden, s. Abb. 3.6.

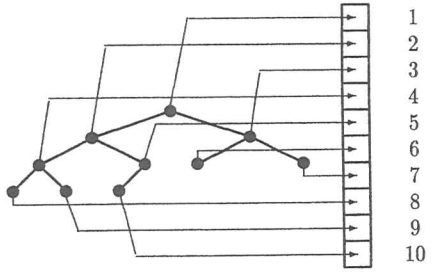


Abbildung 3.6: Abbildung eines vollständigen Binärbaums in ein Array

Die Idee ist eigentlich ganz einfach:

- Die Wurzel bekommt den Index 1.
- Die Knoten auf der zweiten Ebene die Indizes 2 und 3.
- Auf der dritten Ebene sind die Indizes 4, 5, 6 und 7 dran.
- Auf Ebene n sind es die Indizes 2^{n-1} bis $2^n - 1$.

Der Binärbaum *muß* allerdings vollständig sein, wie man an Abb. 3.7 erkennt. Dort entstehen durch die fehlenden Knoten Lücken in der Belegung des Arrays.

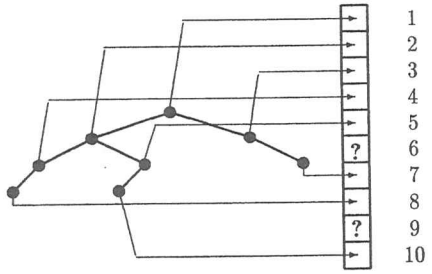


Abbildung 3.7: Ein Binärbaum, der kein Array darstellt

Nehmen wir dazu ein solches Array `a[1:n]` und einen Index `k` mit $1 \leq k \leq n$. Die drei Basisoperationen für Binärbäume lassen sich dann ganz einfach formulieren:

- `parent(k)  = k div 2`
- `left(k)  = k*2`
- `right(k)  = k*2+1`

Baumtraversierungen

Hat man einen allgemeinen Baum gegeben, so kann man, ausgehend von der Wurzel, mit einfachen Verfahren so durch den Baum marschieren, daß alle Knoten des Baums einmal besucht werden. Das einfachste Verfahren beruht auf der Idee, einen Knoten zu „betreten" und — wenn er kein Blatt ist — dann rekursiv alle seine Nachfolger zu betreten.

In dieser Form wird der Baum so traversiert, daß versucht wird, möglichst schnell „nach unten" in Richtung der Blätter zu gelangen. Man spricht dann von „Tiefensuche" oder „Tiefentraversierung" (engl. depth-first search) des Baums. Normalerweise wird man auf einem Knoten irgendetwas mit seinem Inhalt anstellen und wenn es nur darum geht, seine Bezeichnung auszugeben. Es stellt sich dann die Frage, *wann* man das tut: *Vor* dem Betreten der Nachfolger oder *nach* dem Betreten der Nachfolger. Je nach dem, wie man sich entscheidet, erhält man verschiedene Reihenfolgen, in denen die Aktionen auf den Knoten ausgeführt werden.

Wird die Aktion *vor* dem Betreten der Nachfolger ausgeführt, so spricht man von einer *Präordnungs-Traversierung* (engl. preorder traversal) andernfalls von *Postordnungs-Traversierung* (engl. postorder traversal). Hier Pseudocode für beide Formen:

Algorithmus 3.3 *Präordnungs- (links) und Postordnungs-Traversierung (rechts) des Knotens* `node`.

Precondition: ---
Postcondition: ---

```
procedure preorder(node)
  {tue irgendetwas mit "node"}
  for n in succ(node) do
     preorder(n)
```

```
procedure postorder(node)
  for n in succ(node) do
     postorder(n)
  {tue irgendetwas mit "node"}
```

Um den Effekt solcher Traversierungen besser zu sehen, betrachten wir die Bäume aus Abb. 3.4, wobei wir der Einfachheit halber annehmen wollen, daß es sich um *geordnete Bäume* handelt, so daß die Nachfolger eines Knotens von links nach rechts angeordnet sind. Nimmt man als Aktion lediglich an, der Name des Knotens würde ausgegeben, so bekommen wir für den ersten Wurzelbaum aus Abb. 3.4:

Präordnung: a h i b c g d f e
Postordnung: h b i d e f g c a

Aufgabe: *Sie haben es sicher erwartet: Geben Sie die entsprechenden Knotenfolgen für die beiden anderen Wurzelbäume aus Abb. 3.4 an.*

Für Binärbäume gibt es natürlich noch eine dritte Möglichkeit der Traversierung: Man führt die Aktion durch, *nachdem* man den linken Nachfolger betreten hat aber *bevor* man den rechten betritt. In diesem Fall spricht man von einer „Inordnungs-Traversierung" (engl. inorder traversal).

Algorithmus 3.4 *Inordnungs-Traversierung des Knotens* `node` *in einem sortierten Binärbaum.*

Precondition: ---
Postcondition: ---

```
procedure inorder(node)
  if {"node" hat einen linken Nachfolger left(node)}
    inorder(left(node))

  {tue irgendetwas mit "node"}

  if {"node" hat einen rechten Nachfolger right(node)}
    inorder(right(node))
```

Wie das Schicksal es will, ist der zweite Wurzelbaum aus Abb. 3.4 ein Binärbaum und es ergibt sich folgende Inordnungs-Traversierung:

h a b i c d g e f

3.2.3 Darstellung von Graphen

Um allgemeine gerichtete und ungerichtete Graphen darzustellen, stehen uns prinzipiell zwei Möglichkeiten offen. Prinzipiell gehen wir in einem Algorithmus davon aus, daß wir innerhalb unserer Algorithmen entweder mit einer Ecke oder mit einer Kante reden. Von einer Ecke möchten wir z.B. wissen:

1. Welche Kanten enden in dieser Ecke?
2. Welche anderen Ecken sind von hier aus erreichbar?

Von einer Kante wüßten wir gern, welches die beiden Ecken sind, die zu dieser Kante gehören. Dies sind Fragen, die wir an allgemeine ungerichtete Graphen stellen können.

Bei gerichteten Graphen kann und muß man präziser fragen, weil die Richtung der Kanten eine Rolle spielt.

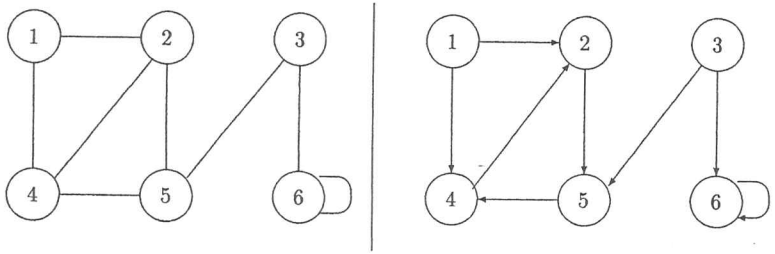


Abbildung 3.8: Ungerichteter und gerichteter Graph

In Abbildung 3.8 sehen wir zwei Varianten eines Graphen: Links ungerichtet und rechts gerichtet. Man beachte bitte, daß es sich bei dem gerichteten Exemplar *nicht* um einen DAG handelt!

Das Problem der Darstellung in einem Rechner können wir in verschiedenen Formen lösen. Die beiden wichtigsten wollen wir hier für beide Beispiele angeben: Die *Adjazenzliste* und die *Adjazenzmatrix*.

Darstellung durch eine Adjazenzliste

Die Darstellung eines ungerichteten Graphen mit der Eckenmenge V und der Kantenmenge E besteht darin, daß man eine Liste `Adj` (im statischen Fall als Array) aller Ecken anlegt. Von einem solchen Eintrag `Adj`$[v_i]$ der Ecke v_i aus führen Zeiger auf alle Ecken, die mit v_i durch eine Kante verbunden sind. So eine Struktur kann man als einfach verkettete Liste oder als Ringliste implementieren. Für den gerichteten Fall nehmen wir nur die Ecken in die Struktur auf, die über eine entsprechend gerichtete Kante mit v_i verbunden sind.

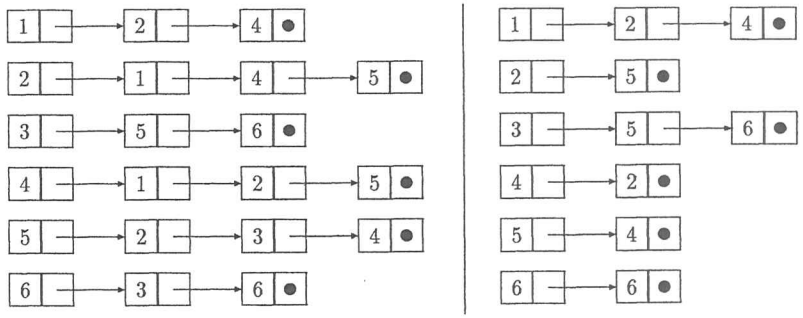


Abbildung 3.9: Darstellung durch Adjazenzlisten

Der linke Teil von Abbildung 3.9 zeigt die Struktur für die ungerichtete, der rechte Teil die für die gerichtete Variante.

Der Speicherbedarf einer Adjazenzliste für einen Graphen mit der Eckenmenge V und der Kantenmenge E ist ganz human: $O(|V|+|E|)$. Adjazenzlisten sind ganz gut dazu geeignet, zu einer Ecke hintereinander die entsprechenden Folgeecken zu bekommen. Problematisch ist hierbei lediglich eine Frage, ob eine bestimmte Kante (v_i, v_j) im Graphen enthalten ist, was manchmal ganz interessant sein kann. Hier besteht keine andere Möglichkeit, als den Eintrag `Adj`$[v_i]$ — also die Liste der benachbarten Ecken — linear durchzutesten, ob v_j darin vorkommt.

Die Darstellung *gewichteter Graphen* in Form einer Adjazenzliste ist auch nicht so schwer. Ein gewichteter Graph ist ja mit einer Abbildung w der Kantenmenge in die reellen Zahlen ausgestattet. Der Wert $w(v_i, v_j)$ — also das Gewicht der Kante zwischen v_i und v_j — wird im Auftreten der Ecke v_j innerhalb des Eintrags `Adj`$[v_i]$ vermerkt. Generell eignet sich diese Darstellung für die Speicherung aller möglicher Varianten, weil man beliebige Informationen in die entsprechenden Knoten packen kann.

Als ADT vereinbaren wir noch, daß wir die Kette der benachbarten Ecken am Eintrag in `Adj` durch Aufruf des Nachfolgers „`succ`" abklappern können. In einer höheren Programmiersprache wäre so etwas durch einen Zeiger, etwa mit Namen `next`, zu realisieren.

Darstellung durch eine Adjazenzmatrix

Eine andere nützliche Darstellung bedient sich der *Adjazenzmatrix* a_{ij}, die den Wert 1 aufweist, wenn (v_i, v_j) eine Kante des Graphen ist. Hier vereinbaren wir für ungerichtete Graphen, daß $a_{ij} = a_{ji}$, d.h. die Adjazenzmatrix eines ungerichteten Graphen ist symmetrisch zur Hauptdiagonalen. Der Aufwand zur Prüfung, ob eine Kante (v_i, v_j) im Graphen vorhanden ist, ist bei der Matrix-Darstellung äußerst gering: Man guckt nur das Element a_{ij} an.

Abb. 3.10 zeigt die Darstellung der beiden Graphen aus Abb. 3.8.

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Abbildung 3.10: Darstellung durch Adjazenzmatrizen

Der Speicheraufwand für diese Matrix beträgt natürlich konstant $\Theta(|V|^2)$ Elemente unabhängig von der Anzahl der Kanten! Unter Umständen ist das ziemlich recht ungünstig, wenn der Graph *dünn* ist, also vergleichsweise wenig Kanten besitzt.

Auch hier ist es recht einfach, Gewichte von Kanten abzuspeichern, indem man sie in die Matrix anstelle der 0 bzw. 1 einträgt.

Kapitel 4

Internes Sortieren

Die Klasse der *internen Sortierverfahren* umfaßt generell Algorithmen, die grundsätzlich eine Array-Datenstruktur, sagen wir mal `a`, als Grundlage hat. Wir gehen dabei davon aus, daß an jeder Arrayposition zwischen 1 und einer gegebenen Zahl `n` ein Element gespeichert ist, das einen *Schlüssel* `key` enthält, der als *Sortierkriterium* dient. „Sortieren" bedeutet nun spezieller, daß nach Ablauf des Algorithmus folgende Bedingung gilt:

$$\forall\, \mathtt{i,j} \in \{1,\ldots,\mathtt{n}\} : \mathtt{i} \le \mathtt{j} \;\Rightarrow\; \mathtt{key(a[i])} \le \mathtt{key(a[j])}. \tag{4.1}$$

Das entspricht einer *aufsteigenden* Sortierung, für eine *absteigende* Sortierung muß rechts vom Pfeil „$\Rightarrow$" statt „$\le$" dann „$\ge$" stehen. Formal schwerer zu beschreiben, aber hier natürlich stillschweigend vorausgesetzt, ist die Forderung, daß das Array `a` nach der Sortierung *dieselben* Elemente enthalten soll wie vorher.

Noch etwas allgemeiner stellt sich das Problem, *nur einen gewissen Teil* des Arrays zwischen den Grenzen `start` und `end` einschließlich sortiert zu bekommen.

Natürlich muß gelten

$$1 \le \mathtt{start} \le \mathtt{end} \le \mathtt{n} \tag{4.2}$$

Für $\mathtt{start} = 1$ und $\mathtt{end} = \mathtt{n}$ ergibt sich dann aus Bedingung 4.1 die Bedingung 4.3.

$$\forall\, \mathtt{i,j} \in \{\mathtt{start},\ldots,\mathtt{end}\} : \mathtt{i} \le \mathtt{j} \;\Rightarrow\; \mathtt{key(a[i])} \le \mathtt{key(a[j])}. \tag{4.3}$$

Bedingung 4.3 ist so grundlegend, daß wir sie grundsätzlich als Nachbedingung für die behandelten Algorithmen verwenden werden. Bedingung 4.2 wird gegebenenfalls als Vorbedingung fungieren.

Auch beim internen Sortieren gibt es eine Unterteilung in weitere Klassen. Die allgemeinste Klasse, die wir hier auch ausschließlich behandeln wollen, besteht darin, Schlüssel verschiedener Elemente miteinander zu vergleichen und gegebenenfalls Elemente des Arrays an eine andere Stelle zu bewegen. Man kann dann von „Sortieren durch Vergleichen und Bewegen" sprechen. Diese Klasse macht als einzige keine Annahmen über Schlüsselwerte, was von anderen Algorithmen ausgenutzt werden kann, Beispiele dafür sind Algorithmen

wie „Counting Sort“, „Radix Sort“ oder „Bucket Sort“[1], die z.B. in [Cormen *et al.*, 1990] dargestellt werden.

Die hier behandelten Algorithmen teilen sich wiederum in zwei Klassen:

1. „Elementare“ Sortierverfahren: Das sind sehr einfache Verfahren, die man normalerweise „aus dem Kopf“ implementieren kann, wenn das Prinzip klar ist. Das Problem ist, daß sie (asymptotisch) nicht sehr effizient sind.
2. „Höhere“ Sortierverfahren: Diese Verfahren sind effizienter als die vorigen, erfordern aber einiges an Gedankenschmalz, wenn man sie entwickelt. Sie sind von der Codierung her aufwendiger und somit inhärent anfälliger für Programmierfehler.

Für die Darstellung der folgenden Verfahren gehen wir von etwas vereinfachten Gegebenheiten aus. Natürlich brauchen wir für jeden Array-Eintrag einen Schlüssel, an den wir nach unseren Konventionen mittels `key(a[i])` herankämen. Üblicherweise wird man allerdings den Array-Eintrag und seinen Schlüssel miteinander identifizieren, d.h. wir sortieren immer die Einträge `a[i]` selber. Alle Verfahren arbeiten natürlich auch für `records`, da muß man bei den Vergleichen natürlich immer auf den Schlüssel zugreifen.

Eine Basisoperation, die andauernd verwendet wird ist das *Vertauschen* von zwei Werten `x` und `y`, hier einfach geschrieben

```
swap(x,y)
```

Die Implementation von `swap` ist offensichtlich.

4.1 Elementare Sortierverfahren

Wir nehmen uns hier zwei der bekanntesten elementaren Verfahren kurz vor, die eigentlich beide zum Standard-Repertoire eines Programmierers gehören[2]:

1. Sortieren durch Auswählen: „Selection Sort“
2. Sortieren durch Einfügen: „Insertion Sort“

Wir werden allerdings demnächst sehen, daß *Selection Sort* nur das einfachste Beispiel der Klasse „Sortieren durch Auswählen“ ist. Aus diesem Grund gibt's diesen Titel zweimal.

4.1.1 Sortieren durch Auswählen I: Selection Sort

Die prinzipielle Idee ist einfach:

1. Man suche das kleinste Element zwischen den Positionen `i` = 1 und `n` und tausche es nach vorne auf Position 1.

[1] Allein die Namen machen schon neugierig! Es ist gerade bei diesen Verfahren schwierig, eine vernünftige deutsche Übersetzung zu finden („Eimer-Sortieren“???)

[2] Ich weigere mich, das seltsamerweise so unheimlich beliebte „Bubblesort“ zur Kenntnis zu nehmen! Es ist kontra-intuitiv und ineffizient.

2. Dann suche man das kleinste Element zwischen den Positionen `i` = 2 und `n` und tausche es auf Position 2.
3. Das mache man, bis man mit `i` auf `n` − 1 angekommen ist.

Algorithmus 4.1 *Sortieren des Array* `a[1:n]` *zwischen* `start` *und* `end` *einschließlich durch fortgesetztes Auswählen des kleinsten Elements.*

Precondition: `Gleichung 4.2`
Postcondition: `Gleichung 4.3`

```
procedure selection-sort(var a; start,end)

  for i := start to end-1 do
     {Suche kleinstes Element zwischen i und end}
     minindex := i
     minvalue := a[i]

     {Suche kleinstes Element zwischen i+1 und end}
     for j := i+1 to end do
        if a[j] < minvalue then
           minvalue := a[j]
           minindex := j

     {Auf minindex steht es: Nach vorn tauschen}
     swap(a[i],a[minindex])
```

Die Laufzeitkomplexität ist einfach zu bestimmen, wenn wir kurz `start` = 1 und `end` = `n` annehmen. Für Teilbereiche davon setze man `n` = `end` - `start` + 1:

$$\text{Vergleiche:} \quad \underbrace{\sum_{i=1}^{n-1} \overbrace{(n-i)}^{\text{innere Schleife}}}_{\text{äußere Schleife}} = \frac{n(n-1)}{2}$$

$$\text{Bewegungen:} \quad 3(n-1)$$

Nix mit O oder Θ, Selection Sort braucht *immer* dieselbe Anzahl von Vergleichen bzw. Bewegungen. Er hat keinen besten Fall und auch keinen schlechtesten, alle Fälle sind mittlere Fälle. Er ist derjenige Sortieralgorithmus, der immer das gleiche tut und geeignet für Leute, die immer wissen wollen, worauf sie sich einlassen!

4.1.2 Sortieren durch Einfügen: Insertion Sort

Hier geht's ein wenig anders zu. Zu jedem Zeitpunkt kann man sich das Array in zwei Teile zerlegbar denken (s. Abb. 4.1):

1. Ein Anfangsstück `a[start:i-1]`, das bereits in sich sortiert ist.

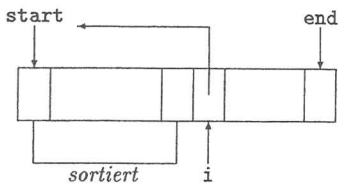


Abbildung 4.1: Zerlegung des Arrays

2. Ein Endstück `a[i:end]`, das noch nicht betrachtet wurde.

Die Aufgabe besteht nun darin, das Anfangsstück sukzessive zu vergrößern, bis es den gesamten zu sortierenden Bereich umfaßt. Um es um eine Stelle zu verlängern, braucht man nur das Element `a[i]` nach links zu verschieben, bis es an der passenden Stelle steht, wobei man die Elemente aus dem Anfangsstück, die größer als `a[i]` sind, nach rechts schiebt.

Algorithmus 4.2 *Sortieren des Array* `a[1:n]` *zwischen* `start` *und* `end` *einschließlich durch Einfügen der Elemente an der richtigen Stelle.*

Precondition: Gleichung 4.2
Postcondition: Gleichung 4.3

```
procedure insertion-sort(var a; start,end)

  {Die Grenze i hochschieben}
  for i := start+1 to end do
     j := i - 1      {Rechtes Ende des sortierten Bereichs}
     t := a[i]       {Zu verschiebender Wert}

     {a[i] nach links schieben}
     while (j >= start) and (a[j] > t) do
        a[j+1] := a[j]      {Muss rechts von t stehen}
        j      := j - 1     {Suchbereich verkleinern}

     {assert: (j < start) or (a[j] <= t)}
     {Also t auf a[j+1] speichern}
     a[j+1] := t
```

Wie hoch ist der Aufwand für dieses Verfahren? Die Analyse ist nun schon etwas anspruchsvoller als im vorigen Fall. Wieder mal setzen wir zur rechnerischen Erleichterung `start` = 1 und `end` = `n`.

Man erkennt schnell, daß Insertion Sort tatsächlich einen besten Fall hat: Ein bereits sortiertes Array. Es gilt dann ja vor dem Eintritt in die `while`-Schleife `a[j]` = `a[i-1]` $\leq$ `a[i]` und das heißt, daß die Schleife überhaupt nicht ausgeführt wird. Dieser Umstand wird mit einem einzigen Schlüsselvergleich festgestellt. Was bleibt, ist das — eigentlich sinnlose — Speichern von `t` auf `a[j+1]`, das man natürlich noch unterbinden kann.

Somit gilt für den besten Fall von Insertion Sort:

- Anzahl der Vergleiche: $n-1$
- Anzahl der Bewegungen: 0

Entsprechend stellt ein umgekehrt sortiertes Array den schlechtesten Fall dar. Jedes Element `a[i]` muß durch das gesamte sortierte Anfangsstück bis nach ganz vorne „durchgereicht" werden. Wieviele Vergleiche braucht man für eine Durchreich-Operation? Richtig: `i - 1` sind es. Zusammen mit der äußeren Schleife:

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = n(n-1)/2 = (n^2-n)/2.$$

Für die Anzahl der Bewegungen gilt fast dasselbe, es kommt nach der `while`-Schleife lediglich noch eine Speicherung auf `a[j+1]` dazu, also $n(n-1)/2+n-1 = (n^2+n-1)/2$. Zusammenfassend für den schlechtesten Fall:

- Anzahl der Vergleiche: $(n^2-n)/2 = \Theta(n^2)$
- Anzahl der Bewegungen: $(n^2+n-1)/2 = \Theta(n^2)$

Dem mittleren Fall nähern wir uns ganz vorsichtig! Zunächst die Vergleiche: Die Anzahl der Vergleiche, die beim Runterschieben von `a[i]` in der `while`-Schleife ausgeführt werden, liegt irgendwo zwischen 1 und $i-1$, das hatten wir schon festgestellt, und der Mittelwert ist dann $i/2$. Summiert man wieder die äußere Schleife rein, kommt man auf die Anzahl der Vergleiche

$$\sum_{i=2}^{n} i/2 = 1/2\cdot\sum_{i=2}^{n} i = 1/2\cdot\sum_{i=1}^{n} i \;-1 = 1/2\cdot n(n+1)/2-1 = (n^2+n)/4-1.$$

Für die Anzahl der Bewegungen machen wir eine kleine Annahme, die uns aus dem gröbsten heraushält: Die letzte Zuweisung in der äußeren Schleife `a[j+1] := t` soll in jedem Fall ausgeführt werden. Dann allerdings ist die Geschichte wieder einigermaßen einfach: Innerhalb der äußeren Schleife werden zwischen 0 und $i-1$ Transporte vorgenommen, also im Mittel $(i-1)/2$. Summiert über die äußere Schleife bekommen wir also

$$\sum_{i=2}^{n}(i-1)/2 = 1/2\cdot\sum_{i=2}^{n}(i-1) = 1/2\cdot\sum_{i=1}^{n-1} i = 1/2\cdot(n-1)n/2 = (n^2-n)/4.$$

Dazu kommen noch die $n-1$ Transporte am Schleifenende, die sich zum Gesamtwert von $(n^2+3n-4)/4$ addieren. Im Zusammenhang für den mittleren Fall

- Anzahl der Vergleiche: $(n^2+n)/4-1 = \Theta(n^2)$
- Anzahl der Bewegungen: $(n^2-n)/4 = \Theta(n^2)$

Zusammenfassend zeigt sich, daß Insertion Sort von einer Laufzeitkomplexität von $O(n^2)$ ist und im mittleren Fall etwa doppelt so schnell wie im schlechtesten ist. Er braucht ungefähr gleichviel Bewegungen wie Vergleiche und gemessen an Selection Sort ungefähr halb soviel Vergleiche. Selection Sort hingegen ist unschlagbar, was die Anzahl der Bewegungen angeht.

4.2 Sortieren durch Auswählen II: Heapsort

Heapsort (zuerst dargestellt in [Williams, 1964]) stellt vielleicht den elegantesten und ungewöhnlichsten aller gängigen Sortieralgorithmen dar. Prinzipiell ist es ein Auswahlverfahren, wie wir am Ende sehen werden, beruht aber auf einer sehr hübschen Idee, die sich zunächst damit befaßt, etwas mehr über Ordnung und Unordnung von Daten in Erfahrung zu bringen.

4.2.1 Teilweise sortierte Strukturen

Die Idee, die hinter den folgenden steht, ist eigentlich einfach: Man betrachtet nicht mehr unsortierte Strukturen sondern *teilweise sortierte* Strukturen, die dann sukzessive zu sortierten Strukturen „umgebastelt" werden.

Zur Illustration einmal eine (absteigend) sortierte Struktur, wobei wir den Pfeil „$\longrightarrow$" benutzen, um darzustellen, daß ein Element größer ist als ein anderes (Abb. 4.2).

$$10 \longrightarrow 7 \longrightarrow 5 \longrightarrow 4 \longrightarrow 3 \longrightarrow 1$$

Abbildung 4.2: Eine sortierte Struktur

In einem solchen Pfeildiagramm stellt sich eine sortierte Struktur also als einfache Kette dar. Bringen wir diese Struktur einmal (etwas) durcheinander, so daß sie so aussieht:

$$8, 5, 4, 6, 7, 1, 3 \tag{4.4}$$

Mit einer kleinen Überlegung gelingt es, auch hierfür eine geeignete Darstellung zu finden (Abb. 4.3).

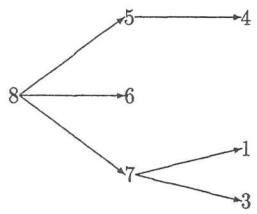


Abbildung 4.3: Eine teilweise sortierte Struktur

Hier sind die einzelnen Elemente nach folgendem Verfahren angeordnet, wobei wir *immer nur* zwei Elemente miteinander vergleichen:

1. Das erste Element (8) steht ganz links an der Wurzel. *Aktueller Knoten: 8.*
2. $5 < 8$: 5 wird unterhalb dieses Knotens (8) angebracht. *Aktueller Knoten: 5.*
3. $4 < 5$: 4 wird unterhalb dieses Knotens (5) angebracht. *Aktueller Knoten: 4.*
4. $6 > 4$: 6 kann also *nicht* unter der 4 angebracht werden. Unter der 5 darüber paßt es auch nicht, aber unter die 8. *Aktueller Knoten: 8.*

5. $7 < 8$: 7 geht unter die 8. *Aktueller Knoten: 7.*
6. $1 < 7$: 1 geht runter die 7. *Aktueller Knoten: 1.*
7. $3 > 1$: 3 paßt nicht darunter; aber unter der 7 geht es.

Eigenschaften teilweise sortierter Strukturen

Man erkennt, daß man mit diesem Verfahren ganz merkwürdige Gebilde erzeugen kann. Insbesondere klappt die ganze Geschichte nur so schön, wenn das erste Element der Folge auch das größte ist. Ist das *nicht* der Fall, so muß man mit einem *nicht-zusammenhängenden* Graphen zurechtkommen. Da wir das nicht wollen, fordern wir die

> **Allgemeine Baumeigenschaft:** Es gibt *genau einen* Knoten, der keinen Vorgänger hat.

Die Tatsache, daß der Graph keine Zyklen enthält, folgt natürlich aus der Bedeutung von „$\longrightarrow$." Es ist folgendes klar:

Aussage 4.1 *Die Darstellung einer unsortierten Folge erfüllt genau dann die allgemeine Baumeigenschaft, wenn das erste Element der Folge das größte ist.*

Nun ist nicht jeder Baum Darstellung einer teilweise sortierten Folge (s. Abb. 4.4).

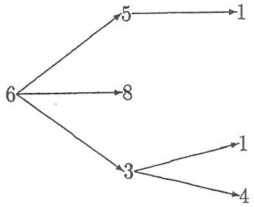


Abbildung 4.4: Ein Baum, der keine teilweise sortierte Struktur darstellt

Schaut man Abb. 4.3 etwas genauer an, so erkennt man eine Eigenschaft, mit der man *lokal* erkennen kann, ob ein gegebener Baum als solche Darstellung einer teilweise sortierten Folge ist:

> **Lokale Teilordnungseigenschaft:** Jeder Knoten eines solchen Baums ist größer (bzw. größer oder gleich) als seine Nachfolger.

Das Problem dabei, die lokale Teilordnungseigenschaft zu prüfen, liegt natürlich darin, daß wir für jeden Knoten *alle* Nachfolger angucken und vergleichen müssen. Mit etwas Pech kann bei unserem Verfahren eine Situation entstehen, daß bei einer Folge von n Elementen ein Knoten $O(n)$ Nachfolger hat, so daß diese Prüfung bereits lineare Zeit in Anspruch nehmen kann.

Es liegt also nahe, die maximale Anzahl von Nachfolgern durch eine Konstante c zu beschränken. Die naheliegende Wahl ist dabei $c = 2$:

> **Binärbaumeigenschaft:** Jeder Knoten eines solchen Baums besitzt *höchstens* zwei Nachfolger.

4.2.2 Heaps

Ein Heap ist eine Datenstruktur, die eine teilweise sortierte Struktur unter beiden Sichtweisen — als Binärbaum und als Array — darstellt.

Definition 4.1 *Ein Binärbaum ist ein* **max-Heap**, *wenn*

1. *er vollständig ist (vgl. Def. 3.8) und*
2. *jeder Knoten größer oder gleich seinen Nachfolgern ist.*

Da wir ja wissen, daß ein vollständiger Binärbaum zu einem Array äquivalent ist (vergl. 3.2.2), hat diese Definition natürlich ein Spiegelbild für die Sichtweise als Array:

Definition 4.2 *Ein Array* `a[1:n]` *ist ein* **max-Heap** *wenn* `a[parent(k)]` $\geq$ `a[k]` *(also* `a[k div 2]` $\geq$ `a[k]`*) für* $1 < k \leq n$ *gilt.*

Der Begriff *max-Heap* rührt daher, daß dabei das größte Element immer an der Wurzel (auf Indexposition 1) steht. Natürlich gibt es auch so etwas wie einen **min-Heap**, der dadurch entsteht, daß man statt „$\geq$" einfach „$\leq$" in die entsprechenden Definitionen einsetzt.

Natürlich muß das Array für diese Definition nicht unbedingt auf der Indexposition 1 beginnen; jeder Teil eines Arrays entspricht einem vollständigen Binärbaum und so kann jeder solche Teil auch einem *max*-Heap entsprechen.

Definition 4.3 *Ein Array* `a[1:n]` *ist ein* **max-Heap ab Position** `p`, $1 \leq p \leq n$, *wenn* `a[parent(k)]` $\geq$ `a[k]` *(also* `a[k div 2]` $\geq$ `a[k]`*) für* $p \leq k$ `div` $2 < k \leq n$ *gilt.*

Wozu sind Heaps eigentlich zu gebrauchen außer zur Darstellung teilweise sortierter Strukturen?

4.2.3 Heaps als Implementation der Prioritäts-Warteschlange

Die Prioritäts-Warteschlange (priority queue) ist bekanntlich ein ADT, der eine verallgemeinerte Warteschlange (queue) darstellt, s. 1.4.6.

In der Tat kann man die Operationen auf einer PQ sehr schön auf einem max-Heap implementieren. Man braucht noch lediglich einen Elementzähler `nelem`, der dem Array zugeordnet ist und die Anzahl der aktuell besetzten Elemente anzeigt. Abb. 4.5 zeigt den ADT `Priority Queue` im Zusammenhang.

Aufgabe: *Geben Sie Pseudocode für* `init` *und* `empty` *an.*

`Priority Queue`	

Schnittstelle:	`init`
	`empty`
	`insert`
	`remove-largest`

Invariante:	Def. 4.2
Implementation:	Speicher: `heap[1:n]` Zähler: `nelem`

Abbildung 4.5: Der ADT `Priority Queue`

Einfügen in einen Heap

Nun geht es zunächst darum, das *Einfügen* in einen max-Heap zu implementieren. Dazu sind einige Vorüberlegungen notwendig, die wir ganz vorsichtig anstellen.

Das erste, was man sich überlegen wird, ist eine generelle Strategie für diese — und auch für jede weitere — Heap-Operation. Beim Einfügen gehen wir dabei recht einfach vor:

1. Das einzufügende Element `x` wird zunächst auf die Position `nelem` + 1 gestellt. Das macht Sinn, weil sich die Anzahl der Elemente um eins erhöhen wird.

   „Baumseitig" bedeutet das, daß `x` als neues Blatt der untersten Ebene *rechts* neben die anderen gehängt wird.
2. Danach wird die *lokale Ordnungseigenschaft* wieder hergestellt. Die anderen Eigenschaften sind ja sicherlich durch den vorigen Schritt *nicht* verletzt worden.

Abb. 4.6 zeigt die ganze Idee des Ablaufs im zweiten Schritt[3].

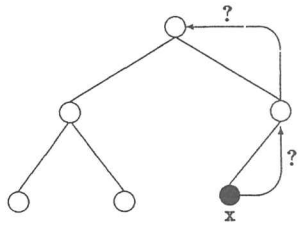


Abbildung 4.6: Einfügen in einem Heap

Einfügen von `x` an der Stelle `nelem` + 1 fügt `x` also als ganz rechtes Blatt ein. Ob die lokale Ordnungseigenschaft dadurch verletzt worden ist, kann man durch Vergleich mit dem Vorgänger feststellen, den „Bruder" brauchen wir dafür nicht.

[3] Den ersten zu erläutern, ist vielleicht ein bißchen albern.

Sollte die lokale Ordnungseigenschaft gelten, so sind wir schon fertig, ansonsten schieben wir den Vorgänger nach unten auf die Position von x und gucken weiter nach oben.

Die Geschichte bricht natürlich ab, wenn der Vorgänger oberhalb der Wurzel liegt oder (arrayseitig), wenn der Index des Vorgängers 0 ist.

Wir geben hier den Algorithmus für einen *max*-Heap an, der für einen *min*-Heap sieht (fast) genauso aus.

Algorithmus 4.3 *Einfügen des Elements* x *in den max-Heap* Q.

Precondition: Def. 4.1
Postcondition: $Q_{post} = Q_{pre} + \{x\}$

```
procedure insert(var Q; x)
  {x an Position nelem + 1 einsetzen}
  Q.nelem          := Q.nelem + 1
  Q.heap[Q.nelem]  := x

  succ             := Q.nelem        {Laufender Knoten}
  pred             := succ div 2     {Sein Vorgaenger}

  {Hochklettern bis oben aus dem Heap raus oder
                bis Vorgaenger >= x}
  while pred > 0 and Q.heap[pred] < x do
     Q.heap[succ]  := Q.heap[pred]    {Vorgaengerwert runter}
     succ          := pred            {Laufenden Knoten hoch}
     pred          := pred div 2

  {Hier muss x hin!}
  Q.heap[succ] := x
```

Entfernen des größten Elements aus einem Heap

Das Entfernen des größten (aus einem *max*-Heap) bzw. des kleinsten (aus einem *min*-Heap) Elements ist i.w. genau spiegelbildlich zum Einfügen. Hier gehen wir einfach so vor, daß wir zunächst das Wurzelelement als Ergebnis der Operation zur Seite nehmen und dann das Element k auf Position nelem — also das letzte — an die Wurzel (auf Position 1) schieben.

Das ist sinnvoll, denn das Array wird nun um eine Stelle kürzer und die Wurzel ist der beste Platz, um das „überflüssige" Element zu parken (Abb. 4.7, links).

Nun steht k oben an der Wurzel, und das kann natürlich falsch sein. Es muß also wohl *runter*! Um zu entscheiden, wohin es muß, haben wir allerdings die beiden Nachfolger (sofern die existieren, natürlich) in Betracht zu ziehen. Also in jedem Schritt

1. den linken (sofern der existiert) und den rechten Nachfolger (sofern der existiert) miteinander vergleichen,
2. den größeren mit k vergleichen

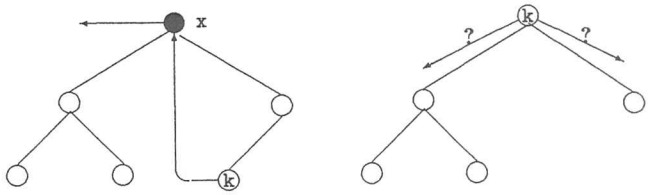


Abbildung 4.7: Entfernen des Wurzelelements aus einem Heap

- ist k kleiner, den größeren auf die alte Position von k schieben, k runterkicken und weiter,
- ist k nicht kleiner, sind wir fertig.

Algorithmus 4.4 *Entfernen des größten Elements aus dem max-Heap* Q, *das auf* x *zurückgegeben wird.*

Precondition: not empty(Q_{pre})
Postcondition: $Q_{post} = Q_{pre} - \{x\}$

```
procedure remove-largest(var Q; var x)
  x          := Q.heap[1]          {Das ist das Ergebnis}
  Q.heap[1]  := Q.heap[Q.nelem]    {Letztes Element an die Wurzel}
  Q.nelem    := Q.nelem - 1        {Ein Element weniger im Heap}
  k          := Q.heap[1]          {Das muss u.U. runter}
  pred       := 1                  {Da steht es jetzt}
  lim        := Q.nelem div 2
  continue   := true               {Solange k noch nach unten muss}

  while pred <= lim and continue do
     {Linker Nachfolger existiert bestimmt}
     succ := 2*pred

     {Test, ob rechter auch existiert und echt groesser ist}
     if succ < Q.nelem and                          {existiert}
        Q.heap[succ + 1] > Q.heap[succ] then        {ist groesser}
        succ := succ + 1                            {dann der rechte}

     {assert: succ ist Index des groesseren Nachfolgers von pred}
     {Muss der Nachfolger hoch?}
     if k < Q.heap[succ] then
        Q.heap[pred] := Q.heap[succ]      {Ja! Nachfolger hoch}
        pred         := succ              {Position von k runter}
     else
        continue     := false             {Nein! Fertig}

  Q.heap[pred] := k
```

Komplexität

Das ist recht einfach! Studiert man die Algorithmen für das Einfügen in und Entfernen aus einem Heap[4], so erkennt man, daß der Aufwand für beide Operationen $O(h)$ ist, wenn h die Höhe des Heaps ist.

Frage also: Wie hoch ist ein Heap mit n Elementen?

Aussage 4.2 *Enthält ein Heap n Elemente, so gilt für seine Höhe h — also die Anzahl der Kanten auf dem längsten Weg von der Wurzel zu einem Blatt:*

$$h = \lfloor \log_2 n \rfloor.$$

Anders ausgedrückt:

> Die Operationen `insert` und `remove-largest` (bzw. `remove-smallest`) sind von der Komplexität $O(\log n)$.

4.2.4 Auf dem Weg zu Heapsort

Hat man diese beiden Operationen, so kann man natürlich einen sehr eleganten, aber etwas problematischen Sortieralgorithmus entwickeln:

> Man benötigt zusätzlich zu dem zu sortierenden Array `a[1:n]` eine Priority Queue `Q`, die durch einen *min*-Heap implementiert ist und mindestens `n` Elemente aufnehmen kann.
>
> Die Elemente von `a` werden nun sukzessive in `Q` eingefüllt und dann wieder entnommen und zurückgespeichert.

Weil kein Mensch so sortiert, hier kein Algorithmus, sondern nur etwas Pseudocode:

```
procedure heapsort-extern(var a; n)
  {Q ist eine lokale PQ mit min-Heap}

  {Elemente von a in die PQ packen ...}
  for i := 1 to n do
     insert(Q,a[i])

  {... und sortiert wieder rausholen}
  for i := 1 to n do
     remove-smallest(Q,a[i])
```

[4]...oder einfacher: Studiert man die beiden Abbildungen 4.6 und 4.7!

Die Komplexität ist ganz erfreulich. Wir brauchen lediglich für die Einfügephase

$$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

Schritte, was tatsächlich $O(n \log n)$ ist, wie man mit Mühe nachrechnet. Dasselbe gilt für die Entnahmephase, so daß wir für das ganze eine Laufzeit von $O(n \log n)$ bekommen, allerdings auf Kosten eines zusätzlichen Platzbedarfs von $O(n)$. Das zu umgehen, ist die Idee von Heapsort!

4.2.5 Heapsort

Die grundlegende Idee hinter Heapsort besteht „einfach" darin, den externen Heap einzusparen, indem das zu sortierende Array *selbst* als Heap (hier interessanterweise als *max*-Heap) organisiert wird.

Hat man das getan, so ist die Entnahmephase relativ einfach, wie man an folgender Grobversion sieht:

```
procedure heapsort(var a; n)

  {Verwandle a in einen max-Heap}

  {Groesstes Element aus a[1] entnehmen}
  {max-Heap restaurieren}
  for i := n downto 1 do
     remove-largest(a,x)
     a[i] := x
```

Die einzige Frage, die man sich stellt, ist die, wie in aller Welt man ein unsortiertes Array *intern* in einen max-Heap verwandeln kann. Glücklicherweise können wir das i.w. schon!

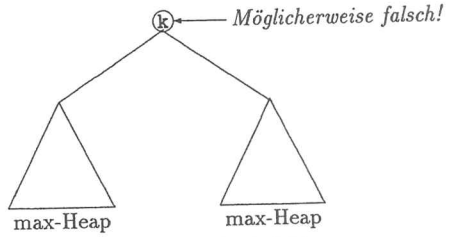
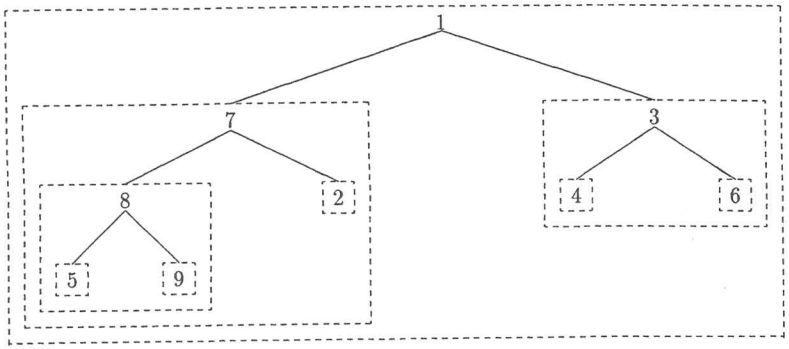


Abbildung 4.8: Ein „Beinahe"-Heap

Die Idee zeigt sich in Abb. 4.8. Dort ist schematisch die Situation dargestellt, die sich vor der Reorganisationsphase des Heap in `remove-largest` (Algorithmus 4.4) bietet. In dieser

Situation besteht die Konstruktion aus einer Wurzel, an der links und rechts perfekte *max*-Heaps hängen; lediglich die Wurzel selbst verletzt u.U. die lokale Ordnungseigenschaft. Genau das haben wir dann repariert.

Nun gibt's in auch einem unsortierten Array jede Menge *max*-Heaps, wie man in Abb. 4.9 sieht.



1,7,3,8,2,4,6,5,9

Abbildung 4.9: Heaps und andere Binärbäume in einem unsortierten Array

Wir finden dort die Darstellung der unsortierten Folge 1,7,3,8,2,4,6,5,9 als vollständigen Binärbaum. Darin enthalten sind weitere vollständige (Teil-) Bäume, die eingekästelt sind. Und siehe da: Wir finden 5 (in Worten „fünf") *max*-Heaps!

Natürlich ist jedes Blatt ein *max*-Heap: Die Blätter 5, 9, 2, 4 und 6 sind perfekt. Betrachtet man unter dieser Voraussetzung die *Vorgänger* dieser Blätter, so sind das offenbar Wurzeln solcher „Beinahe"-Heaps wie in Abb. 4.9. Der Baum mit der Wurzel 8 z.B. zeigt genau dieses Problem. Kann man den in einen Heap verwandeln, könnte man auf der nächsten Ebene schon an den Baum mit der Wurzel 7 herangehen usw.

Wiederherstellen der Heap-Eigenschaft

Da diese Bäume „mitten im" Array liegen, verwandeln wir die Reorganisationsphase in eine allgemeine Prozedur, die in irgendeinem nicht-leeren (!) Teil eines Arrays, etwa zwischen den Positionen `root` (der Wurzelposition) und `max` (der Position des letzten Blatts), die Heap-Eigenschaft wiederherstellt, wenn es sich um einen solchen „Beinahe"-Heap handelt.

Algorithmus 4.5 *Wiederherstellen der* max-*Heap Eigenschaft in einem Array* `a[1:n]` *zwischen den Positionen* `root` *und* `max` *einschließlich.*

Precondition: $1 \leq$ `root` $\leq$ `max` $\leq$ `n`, `a[root:max] ist ein Beinahe-Heap`

Postcondition: `a[root:max] ist ein max-Heap`


```
procedure heap-adjust(var a; root,max)
  k        := a[root]      {Element an der Wurzel}
  pred     := root
  lim      := max div 2
  continue := true

  {Genau wie in remove-largest}
  while pred <= lim and continue do
     succ := 2*pred

     if succ < max and
        a[succ + 1] > a[succ] then
        succ := succ + 1

     if k < a[succ] then
        a[pred]  := a[succ]
        pred     := succ
     else
        continue := false

  a[pred] := k
```

Aufbauphase des Heaps

Die Aufbauphase des Heaps besteht nun einfach darin, das Array *von unten* in einen *max*-Heap zu verwandeln, s. Abb. 4.10.

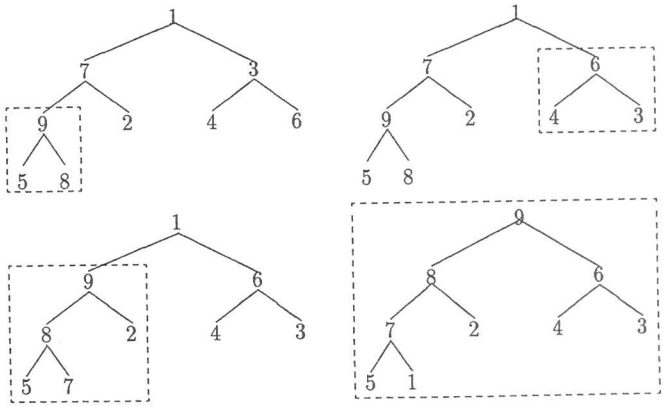


Abbildung 4.10: Aufbauphase des Heaps

Links oben sieht man das Ergebnis nach Reorganisation des Baums mit der Wurzel 8 aus Abb. 4.9. Im nächsten Schritt werden wir — eine Ebene drüber — den mit der Wurzel 3 anfassen (rechts oben). Nun ist der Baum mit der Wurzel 7 dran und das Ergebnis sieht

man links unten. Damit sind die beiden Nachfolger der Wurzel 1 richtige *max*-Heaps und zum guten Schluß werden wir das gesamte Array reorganisieren (rechts unten).

Das einzige, was man sich halt überlegen muß, ist die Frage, bei welchem Knoten (welcher Indexposition) diese Geschichte zu beginnen hat. Bei einer Elementzahl von n steht die erste fragwürdige Wurzel auf Position $\lfloor n/2 \rfloor$.

```
{Aufbauphase des Heaps}
for i := n div 2 downto 1 do
   heap-adjust(a,i,n)
```

Abbauphase des Heaps

Nun haben wir einen *max-Heap* vorliegen und können ihn nun rückwarts wieder abbauen und dabei die endgültige Sortierung herstellen. Was man weiß, ist die Tatsache, daß das größte Element des Arrays nun auf Position 1 steht und auf Position n muß. Das Element auf Position n hingegen stellen wir einfach auf Position 1 und reorganisieren den entstehenden „Beinahe"-Heap, allerdings nur bis zur Position $n-1$. Dann steht das zweitgrößte Element vorne, es geht an Position $n-1$, das Element dort nach vorne, reorganisieren bis $n-2$, usw.

Algorithmus 4.6 Heapsort: *Sortieren des Arrays* `a[1:n]`,

Precondition: ---
Postcondition: `Gleichung 4.1`

```
procedure heapsort(var a; n)
   {Aufbauphase des Heaps}
   for i := n div 2 downto 1 do
      heap-adjust(a,i,n)

   {Abbauphase des Heaps}
   for i := n downto 2 do
      h    := a[1]
      a[1] := a[i]
      a[i] := h
      heap-adjust(a,1,i-1)
```

4.2.6 Zusammenfassung

Heapsort ist ein ausgesprochen seltsamer Sortieralgorithmus, anders kann man es kaum ausdrücken. In völligem Gegensatz zur Anschauung besteht seine erste Aufgabe darin, das größte Element des Array *ganz nach vorne* zu bringen! Der Effekt dabei ist, daß die Aufbauphase am schnellsten abläuft, wenn das Array ursprünglich umgekehrt sortiert ist.

Aufgabe: *Machen Sie sich bitte klar, daß ein absteigend sortiertes Array ein* max-*Heap ist.*

Machen Sie sich bitte ferner klar, daß die Aufbauphase in diesem Fall von der Komplexität $O(n)$ ist.

Die Komplexität von Heapsort ist einfach zu bestimmen, wenn man erkennt, daß sowohl die Aufbau- als auch die Abbauphase im schlechtesten Fall $O(n \log n)$ Schritte benötigen, beides zusammen also auch.

Das eigentlich schöne daran ist, daß Heapsort (wenigstens asymptotisch) keinen schlechtesten Fall hat; wie immer auch das Array anfangs ausschaut, nach der $O(n \log n)$ Aufbauphase ist die Sache in jedem Fall in Ordnung.

Ja, Heapsort hat einen besten Fall, in der er $O(n)$ Schritte braucht!

Aufgabe: *Charakterisieren Sie den besten Fall für Heapsort. Er ist allerdings in der Praxis ziemlich irrelevant!*

4.3 Sortieren durch Zerlegen: Quicksort

Quicksort [Hoare, 1962] stellt den heute am besten untersuchten Sortieralgorithmus dar. Quicksort gilt heute als das praktisch effizienteste interne Sortierverfahren — viele haben sich bemüht, ihn zu verbessern, ohne großen Erfolg. Er hat lediglich zwei kleine Nachteile:

1. Sein schlechtester Fall ist *sehr* schlecht.
2. Quicksort ist in seiner einfachsten Form rekursiv.

Zunächst die Idee!

4.3.1 Die Idee

Abb. 4.11 zeigt den Ablauf, wie Quicksort an ein unsortiertes Array herangeht.

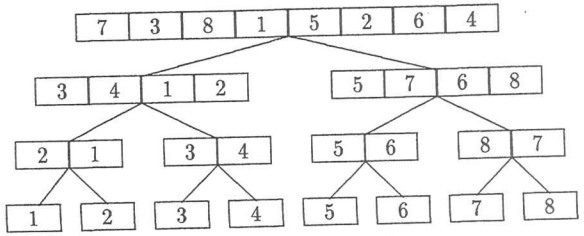


Abbildung 4.11: Ablauf von Quicksort

Das Prinzip — hier idealisiert dargestellt — besteht darin, ein gegebenes unsortiertes Array in zwei möglichst gleichgroße Teile zu zerlegen, von denen der linke die „kleinen" und der rechte die „großen" Schlüssel enthält. Diese beiden Teile können dann unabhängig voneinander rekursiv sortiert werden, bis man zu Teilen kommt, die nur noch ein Element enthalten.

Das einzige Problem, das wir lösen müssen, besteht anscheinend darin, in jedem Schritt das gegebene Array so zu arrangieren, daß die Möglichkeit zur Zerlegung besteht. Das wird einige Vergleiche und Vertauschungen brauchen, das ist klar. Der Effekt dabei ist natürlich, daß wir später keine Mühe mehr damit haben, die sortierten Teile wieder miteinander zu verbinden.

Wir brauchen also lediglich eine einigermaßen handfeste Definition, wie eine solche Zerlegung des Arrays `a[start:end]` aussehen soll. Da scheiden sich die Geister ein wenig; wir wollen folgendes erreichen:

> Gegeben das Array `a[start:end]`. Gesucht sind zwei Indizes `l` (für „links") und `r` mit folgenden Eigenschaften:
>
> 1. $\texttt{start} \leq \texttt{l} \leq \texttt{r} \leq \texttt{end}$.
> 2. Alle Elemente aus dem Bereich `a[start:l]` sind $\leq$ allen Elementen aus `a[r:end]`.
> 3. Alle Elemente aus dem Bereich `a[l:r]` stehen bereits an der richtigen Stelle.

Hat man solche Werte `l` und `r` gefunden und das Array so arrangiert, reicht es dann aus, die Teile `a[start:l]` und `a[r:end]` separat zu sortieren. Wir sprechen bei diesem Teil des Algorithmus von der **Partitionierung** (der „Zerlegung") des Arrays.

Wir haben es also mit einem strukturell sehr einfachen Algorithmus zu tun, bei dem lediglich im Detail ein wenig „Fummelei" nötig ist:

```
procedure quicksort(var a; start, end)
  if start < end then

     {Arrangiere a[start:end] so, wie oben beschrieben}
     {Besetze "l" und "r" entsprechend}

     {Dann kann man diese Teile getrennt sortieren}
     quicksort(a,start,l)
     quicksort(a,r,end)
```

4.3.2 Partitionierung

In diesem Abschnitt wird also die gesamte Arbeit geleistet[5]. Jeder, der zum ersten Mal mit Quicksort konfrontiert wird und eine Definition für Partitionierung wie oben vorgesetzt bekommt, knobelt erstmal herum, wie man das machen könnte, findet aber meistens nur Lösungen, die recht uneffizient sind. Man muß nur auf die zugrundeliegende Idee kommen. Wir möchten nämlich erreichen, daß diese Partitionierung in *linearer Zeit* abläuft.

Dazu modifiziern wir die Anforderungen an die Partitionierung ein wenig:

[5] Physiker weghören! Ich weiß, daß man „Arbeit" nicht „leisten" kann.

> Gegeben `a[start:end]` und ein Schlüsselwert `median`, der irgendwo zwischen dem kleinsten und dem größten Schlüssel in `a[start:end]` liegen muß.
>
> Es ist `a[start:end]` so zu arangieren, daß es zwei Indizes `l` und `r` gibt, daß folgende Bedingungen erfüllt sind:
>
> 1. $\texttt{start} \leq \texttt{l} \leq \texttt{r} \leq \texttt{end}$.
> 2. Alle `a[i]` mit $\texttt{start} \leq \texttt{i} < \texttt{r}$ sind $\leq$ `median`,
> 3. Alle `a[i]` mit $\texttt{l} < \texttt{i} \leq \texttt{end}$ sind $\geq$ `median`.

Illustriert wird die ganze Geschichte durch Abb. 4.12.

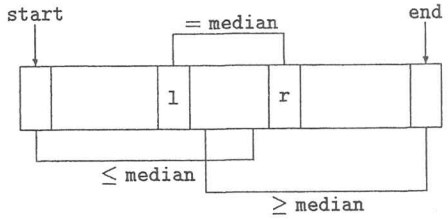


Abbildung 4.12: Situation nach der Partitionierung

Dort sieht man auch, daß alle Werte *zwischen* `l` und `r` gleich `median` sein müssen. Weiterhin erkennt man auch, daß in der Tat jedes Element im Bereich `a[start:l]` kleiner oder gleich jedem Element aus dem Bereich `a[r:end]` ist und daß die Elemente dazwischen auch an der richtigen Stelle stehen.

Diese Anforderungen an die Partitionierung kann man nun mit allen bekannten Entwurfsmethoden relativ einfach herstellen. Wir werden zunächst einmal einen Algorithmus entwerfen, der eine etwas schwächere Bedingung herstellt:

1. $\texttt{l} \leq \texttt{r}$.
2. Alle `a[i]` mit $\texttt{start} \leq \texttt{i} < \texttt{r}$ sind $\leq$ `median`,
3. Alle `a[i]` mit $\texttt{l} < \texttt{i} \leq \texttt{end}$ sind $\geq$ `median`.

Sie unterscheidet sich nur im ersten Teil von der angestrebten; daß auch das schärfere Original gilt, müssen wir uns später noch klarmachen. Um diese Bedingung herzustellen, gehen wir wie folgt vor:

1. Wir initialisieren `l` und `r` so, daß nur die *erste* Bedingung nicht gilt.
2. Die zweite und dritte erklären wir zur *Schleifeninvariante* für eine Schleife, die zustätzlich die erste Bedingung erzwingt.

Also: Gegeben den Median `median`, sowie `start` und `end`, suchen wir nach einer Besetzung von `l` und `r`, die den beiden letzten Bedingungen genügt. Einen Moment nachdenken ....

```
r := start
l := end

{assert: alle a[i] mit start <= i < r sind <= median
         alle a[i] mit l < i <= end sind >= median}
```

Aufgabe: *Warum gilt nach dieser Initialisierung die Zusicherung?*

Nun entwerfen wir mal grob die Schleife:

```
{inv: alle a[i] mit start <= i < r sind <= median
      alle a[i] mit l < i <= end sind >= median}

repeat
   {Irgendetwas, das l und r aufeinander zubewegt
    und unsere Invariante erhaelt}

   {inv: alle a[i] mit start <= i < r sind <= median
         alle a[i] mit l < i <= end sind >= median}
until r > l

{assert: alle a[i] mit start <= i < r sind <= median
         alle a[i] mit l < i <= end sind >= median
         l <= r}
```

Und in der Schleife? Zunächst schieben wir `r` so weit nach rechts und `l` so weit nach links, wie es die Invariante zuläßt:

```
while a[r] < median do r := r + 1
{assert: alle a[i] mit start <= i < r sind <= median
         a[r] >= median}

while a[l] > median do l := l - 1
{assert: alle a[i] mit l < i <= end sind >= median
         a[l] <= median}
```

Nun weiß keiner, wo `l` und `r` am Ende stehen. Mit Pech ist immer noch $r \leq l$. In diesem Fall sieht die Geschichte so aus, wie in Abb. 4.13 dargestellt ist.

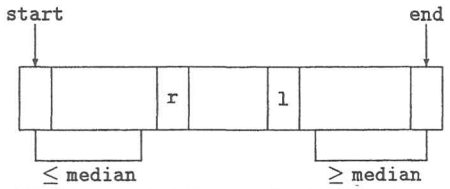


Abbildung 4.13: Mitten in der Partitionierung

Da wir nun wissen, daß `a[r]` $\geq$ `median` und `a[l]` $\leq$ `median` sind, liegt es nahe, die beiden Werte `a[r]` und `a[l]` zu vertauschen. Das hat aber den Effekt, daß unsere Invariante verschwindet!


```
if r <= l then
   swap(a[l],a[r])
   {assert: a[r] <= median <= a[l]
            alle a[i] mit start <= i <= r sind <= median
            alle a[i] mit l <= i <= end sind >= median}
```

Hier ist nun in der ersten Zusicherung aus `i < r` das neue `i <= r` und in der zweiten aus `l < i` ein `l <= i` geworden, was es uns erlaubt, endlich einen Schritt weiterzukommen:

```
if r <= l then
   swap(a[l],a[r])

   r := r + 1
   l := l - 1
   {inv: alle a[i] mit start <= i < r sind <= median
         alle a[i] mit l < i <= end sind >= median}
```

Fertig! Unsere Schleife erhält die angegebene Invariante *und* sie terminiert auch.

Aufgabe: *Warum? Die* `while`*-Schleifen müssen* `l` *und* `r` *doch gar nicht bewegen.*

Bis auf die Frage, wie man einen geeigneten Median herankommt, ist Quicksort nun fertig.

Algorithmus 4.7 *Sortieren des Array* `a[1:n]` *zwischen* `start` *und* `end` *einschließlich durch Partitionierung und rekursives Sortieren (Quicksort).*

Precondition: Gleichung 4.2
Postcondition: Gleichung 4.3

```
procedure quicksort(var a; start, end)
  if start < end then
     median := "Ein geeigneter Wert"

     {Partitionierung}
     r := start
     l := end

     repeat

        {l und r aufeinander zu bewegen}
        while a[r] < median do r := r + 1
        while a[l] > median do l := l - 1

        {ggf. Vertauschen und Marken weiterbewegen}
        if r <= l then
           swap(a[l],a[r])
```

```
            r := r + 1
            l := l - 1

        until r > l

{Rekursives Sortieren beider Teile}
        quicksort(a,start,l)
        quicksort(a,r,end)
```

4.3.3 Komplexität

Wie bei einer rekursiven Prozedur üblich, gehen wir das Problem der Komplexität durch das Aufstellen und Lösen von Rekursionsgleichungen heran, vergl. 2.3. Wir setzen für alle Operationen n = `end - start + 1`, also der Anzahl der zu sortierenden Elemente. Das erste ist einfach:

Aussage 4.3 *Der Partitionierungsteil ist von der Komplexität* $\Theta(n)$.

Klar: Jedes Element zwischen `start` und `end` wird einmal angeguckt. Damit hängt nun die Komplexität von Quicksort von der Größe der beiden entstehenden Partitionen `a[start:l]` und `a[r:end]` ab.

Der beste Fall

Der beste Fall entsteht, wenn die beiden Partitionen gleich groß sind, denn wird die Arbeit bestmöglich aufgeteilt. Unter dieser Annahme bekommen wir folgende Rekursionsgleichung heraus:

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T(n/2) + \Theta(n) & \text{sonst} \end{cases} \tag{4.5}$$

Ohne lange zu rechnen, stellen wir einfach fest, daß Gleichung (4.5) die Lösung $T(n) = \Theta(n \log n)$ hat. Die Komplexität im besten Fall liegt also in derselben Größenordnung wie die von Heapsort.

Der schlechteste Fall

Am schrecklichsten arbeitet Quicksort unter folgender Annahme: Der gesamte Partitionierungsaufwand führt letztlich dazu, daß eine Partition *genau ein* Element enthält. Dann enthält die andere natürlich $n - 1$ Elemente und wir kommen zu folgender Rekursionsgleichung:

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ T(n-1) + \Theta(n) & \text{sonst} \end{cases} \tag{4.6}$$

Offenbar kommt bei der Auflösung von Gleichung 4.6 eine Summe der Form

$$T(n) = \Theta(n) + \Theta(n-1) + \cdots + 1$$

heraus, was $\Theta(n^2)$ ergibt.[6] Im schlechtesten Fall hat Quicksort also quadratische Laufzeit.

Der schlechteste Fall ist selten

Das Problem bei Quicksort liegt also offenbar darin, daß der schlechteste Fall das ganze Verfahren in die Klasse der „niederen" Verfahren katapultiert. Damit wäre Quicksort eigentlich aus dem Spiel, hätte es nicht einige Eigenschaften, die das ganze etwas relativieren. So stellen wir uns schnell folgende Frage:

> Angenommen, es gäbe eine Konstante $c > 0$ so, daß Quicksort bei jeder Partitionierung von n Elementen eine Partition von cn Elementen — und damit eine andere von $(1-c)n$ Elementen — bildet. Was wäre der Effekt für die Laufzeit?

Die Antwort ist beruhigend:

Aussage 4.4 *Unter der obigen Annahme ist die Laufzeit von Quicksort* $\Theta(n \log n)$.

Abb. 4.14 verdeutlicht, warum das so ist.

Im ersten Schritt (oben an der Wurzel) haben wir eine Partitionierung mit n[7] Schritten. Dazu kommen dann die beiden Unteraufgaben, die jeweils cn und $(1-c)n$ Schritte in der Partitionierung ausführen (erste Ebene). So geht das immer weiter bis irgendwann Knoten entstehen, bei denen nur noch ein Element übriggeblieben ist. Ein solcher Knoten hat natürlich keine Nachfolger mehr.

Summiert man nun für jede Ebene die Anzahl der Schritte (horizontal) zusammen, so ergibt sich für eine vollbesetzte Ebene immer n, wie man sich bald klarmacht. Für eine nicht voll besetzte Ebene ist die Summe immer $< n$. Das ist ganz rechts in Abb. 4.14 angedeutet.

Man muß nun eigentlich nur noch wissen, *wie tief* dieser Baum werden kann. Das ist auch ganz einfach:

Aussage 4.5 *Unter der zusätzlichen Annahme* $c \leq 1/2$ *hat der Baum eine Tiefe von* $\lceil \log_{1/c} n \rceil$.

[6] ...man denke an die Summenformel $\sum_{i=1}^{n} i = n(n+1)/2$

[7] Wieder kein Exponent, sondern wieder eine Fußnote! Wir lassen das mal mit dem ewigen Θ, sonst wird es doch arg kompliziert.

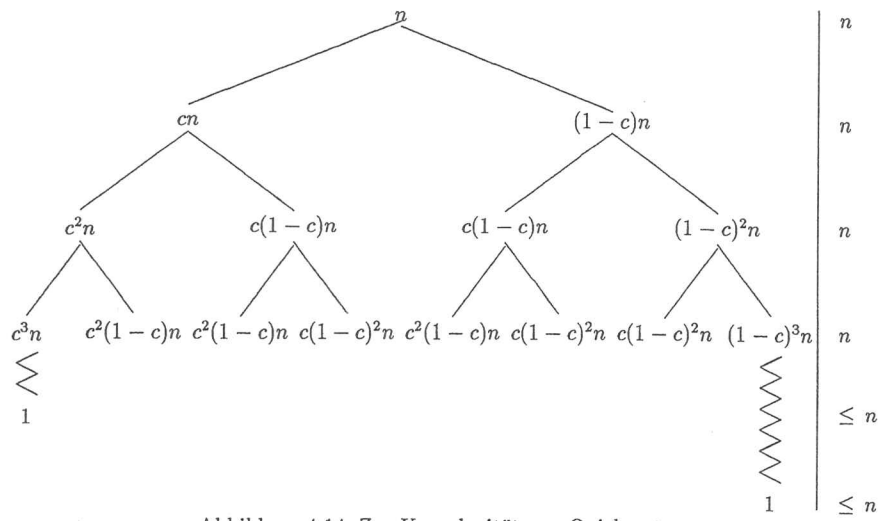


Abbildung 4.14: Zur Komplexität von Quicksort

Aufgabe: *Warum ist das so? Warum darf man $c \leq 1/2$ wählen?*

Damit summiert sich aber die Gesamtkomplexität auch für diesen Fall zu höchstens $n \log_{1/c} n$ auf, ist also $O(n \log n)$. Es gilt sogar, daß die Laufzeit *bei proportionaler Partitionierung* $\Theta(n \log n)$ ist, aber das ist nicht so furchtbar interessant.

Wovon hängt denn nun ab, wie gut die Partitionierung bei Quicksort ist? Auch klar: Von der Wahl des Median. Das sollte uns nun einen Moment beschäftigen.

4.3.4 Wahl des Medians

Wir wissen nun, was wir von einem gescheiten Median verlangen müssen:

1. Er soll schnell — in $O(1)$ Schritten — zu finden sein, sonst geht uns die Rekursionsgleichung 4.5 kaputt.
2. Er soll auf jeden Fall wenigstens eine proportionale Partitionierung (am besten mit $c = 1/2$) liefern.

Daß das eigentlich ein bißchen viel verlangt ist, ist klar und viele Leute haben sich um Ideen bemüht, wie man fertigbringen kann. Hier einige Vorschläge.

Feste Mediane

Am einfachsten ist es natürlich, ein Element aus dem Bereich `a[start:end]` mit einem festen Index zu wählen und sich dabei darauf zu verlassen, daß der schlechteste Fall nicht eintritt. Einfach geht

- die Wahl des *ersten* Elements `a[start]`,
- die Wahl des *letzten* Elements `a[end]`,
- die Wahl des *mittleren* Elements `a[(start+end) div 2]`.

In der Praxis sind diese Möglichkeiten schon ausreichend, um das Auftreten des schlechtesten Falls ziemlich zuverlässig zu unterbinden. Es ist ausgesprochen unwahrscheinlich, daß bei einem dieser Verfahren *immer* eine unproportionale Partition herauskommt, aber sicher ist ds natürlich nicht.

Bester von 3

Eine sehr hübsche Methode besteht darin, bei der Wahl des Medians die drei Indexpositionen `start`, `end` und `(start+end) div 2` anzugucken und den mittleren von den dreien zu nehmen. Wenn man die drei nun schon mal in der Hand hat, kann man sie auch mit einigen Vertauschungen so anordnen, daß der kleinste auf Positition `start`, der größte auf `end` und der eigentliche Median auf `(start+end) div 2` zu stehen kommt.

Das hat den Effekt, daß man bei der Initialisierung bereits mit `l := start+1` und `r := end-1` loslegen kann (Nachdenken!) und der Median schon in der Gegend steht, in die er später auch ungefähr hinrutscht. Dieses Verfahren liefert einen ziemlich verknubbelten Code, in der Praxis Laufzeitverbesserungen um 5 – 10% und wird im englischen „best-of-three median“ genannt. Hier nur der entscheidende Teil der Verfahrens:

```
if start < end then
   {Linkes, mittleres und rechtes Element bestimmen}

   middle := (start + end) div 2
   xl     := a[start]            {Element links}
   xm     := a[middle]           {Element in der Mitte}
   xr     := a[end]              {Element rechts}

   {Mittleres der drei nach a[middle]
    das kleinste nach a[start]
    das groesste nach a[end]}

   if xl < xm then
      if xr < xm then
         if xl < xr then
            swap(a[right],a[middle])
         else
            swap(a[left],a[middle])
   else
      if xr > xm then
         if xl < xr then
            swap(a[left,middle])
         else
            swap(a[right],a[middle])
```

```
{Median und Startzeiger bestimmen}

median := a[middle]
r      := start + 1
l      := end - 1
```

Der Rest läuft wie im Original (Algorithmus 4.7) ab.

Zufällige Mediane

Die letzte Idee bei der Medianwahl besteht darin, die Wahrscheinlichkeit für das Auftreten des schlechtesten Falls dadurch zu minimieren, daß man den Median völlig regellos auswählt, um eventuelle ungünstige Strukturen in den Eingabedaten zu kompensieren.

Die meisten kommerziellen Programmiersprachen enthalten in ihrer Laufzeitbibliothek einen *Pseudozufallszahlen-Generator*, der (ideal) gleichverteilte Zufallszahlen aus einem vorgegebenen Zahlenbereich erzeugt.

Nehmen wir an, ein Aufruf

```
a := random(n)
```

liefert eine zufällige Zahl $0 \leq \mathtt{a} < \mathtt{n}$ (das tun so ziemlich alle Zufallszahlengeneratoren), so wäre ein brauchbarer Median

```
median := a[start + random(end-start)]
```

4.3.5 Quicksort auf kleinen Arrays

So gut Quicksort auch ist (wir sehen das im nächsten Abschnitt 4.4), seine ganzen Qualitäten kann er, wie auch Heapsort, erst bei großen Datenmengen zeigen. Bei kleinen Arrays sind beide Algorithmen hoffnungslos überdimensioniert: Man schießt mit Kanonen auf Spatzen!

In der Tat sind für kleine Bereiche Selection Sort bzw. Insertion Sort grundsätzlich vorzuziehen, es ist lediglich nicht ganz klar, was „klein“ denn nun bedeutet.

Das hat Konsequenzen nicht nur für den Sortierer sondern auch für den inneren Aufbau von Quicksort selbst. Durch das ewige Partitionieren entstehen nämlich irgendwann mal solche kleinen Bereiche, denen man dann besser mit einem niederen Verfahren an den Kragen geht. Der genaue Wert, ab dem auf das andere Verfahren gewechselt werden soll, und auch die Frage nach dem dann anzuwendenden Verfahren sind nicht leicht zu beantworten und man sollte — wenn es wirklich hart auf hart geht — mit einigen Varianten experimentieren, bis man die geeignete gefunden hat.

Allgemein werden wir eine Konstante `limit` brauchen, die uns sagt, wann das Verfahren zu wechseln ist. Nach [Sedgewick, 1992] tut ein beliebiger Wert zwischen 5 und 25 gute Dienste. Bei der Verwendung von Insertion Sort sähe der Kopf von Quicksort dann so aus:


```
procedure quicksort(var a; start, end)
  if (end - start) <= limit then
     insertion-sort(a,start,end)
  else
      {Nun das eigentliche Quicksort}
```

Das bringt in der Praxis eine weitere Verringerung um ca. 10%. Man kann Quicksort tatsächlich noch weiter beschleunigen, wenn nun auch mit weniger drastischen Effekten. In [Sedgewick, 1992] steht noch einige Ideen[8], aber es soll nun genug sein.

4.4 Vergleich

Hier nun im Überblick einige Daten, die mit den vier hier behandelten Verfahren gewonnen wurden. Man kann schon einige Schlüsse daraus ziehen. Die Daten lagen in einer zufälligen Reihenfolge vor, es gibt also hier keine Beispiele für sortierte oder umgekehrt sortierte Eingabefolgen.

n gibt natürlich die Anzahl der zu sortierenden Elemente an. Der Wert $n = 5$ ist hier nur aufgeführt, um die Behauptung, Quicksort und Heapsort seien auf kleinen Arrays nicht so effizient, zu stützen.

Bei allen Werten außer $n = 5$ wurde in Quicksort Insertion Sort mit `limit` $= 7$ eingesetzt. Bei $n = 5$ wurde natürlich bis zum bitteren Ende Quicksort benutzt. Der Median wurde mit dem „best-of-three“ Verfahren bestimmt.

Die Zeiten wurden auf einem PC 486-33 genommen und natürlich sind Zeiten < 0.1 sec. ausgesprochen unzuverlässig und mit Vorsicht zu genießen.

Tabelle 4.1 zeigt die wichtigstenErgebnisse im Zusammenhang.

Man erkennt das, was man i.w. erwartet hat. Bei großem n fangen Heapsort und Quicksort so langsam an warmzulaufen, Selection Sort und Insertion Sort sind hier gar nicht mehr aufgeführt. Interessant ist, daß Quicksort grundsätzlich um etwa den Faktor 2 besser abschneidet als Heapsort, was man sich als Regel ganz gut merken kann.

Interessant ist die Tatsache, daß das unscheinbare Selection Sort grundsätzlich *die wenigsten Bewegungen* ausführt. Das ist nützlich, wenn man vor der Situation steht, daß ein Schlüsselvergleich relativ schnell geht, aber eine Bewegung viel Zeit kostet, etwa weil der Schlüssel zu einem großen Record gehört, der jedesmal komplett kopiert werden muß. Solche Anwendungen können die Grenze der Anwendbarkeit von Selection Sort ziemlich nach oben drücken.

In der Laufzeit tun sich Selection Sort und Insertion Sort *hier* nicht viel. Diese Messungen wurden noch mal auf einer VAX-8300 ausgeführt und es wurden Laufzeitunterschiede *zugunsten* von Selection Sort gefunden, die ganz gewaltig waren! Offenbar liebt es die VAX überhaupt nicht, Daten von einer Stelle des Hauptspeichers an eine andere zu verschieben, was u.U. mit der virtuellen Speicherverwaltung zusammenhängt.

[8] Kein Wunder! Sedgewick hat über Quicksort promoviert.

n	Selection Sort		Insertion Sort		Heapsort		Quicksort	
5	Beweg.:	12	Beweg.:	6	Beweg.:	23	Beweg.:	24
	Vergl.:	10	Vergl.:	2	Vergl.:	12	Vergl.:	26
	Zeit.:	0.00	Zeit.:	0.00	Zeit.:	0.00	Zeit.:	0.00
500	Beweg.:	1479	Beweg.:	62976	Beweg.:	5776	Beweg.:	3432
	Vergl.:	124750	Vergl.:	62487	Vergl.:	7424	Vergl.:	2612
	Zeit.:	0.11	Zeit.:	0.16	Zeit.:	0.00	Zeit.:	0.00
1000	Beweg.:	2997	Beweg.:	246534	Beweg.:	12554	Beweg.:	7325
	Vergl.:	499500	Vergl.:	245535	Vergl.:	16878	Vergl.:	6672
	Zeit.:	0.66	Zeit.:	0.66	Zeit.:	0.05	Zeit.:	0.05
4000	Beweg.:	11997	Beweg.:	4018449	Beweg.:	58151	Beweg.:	37600
	Vergl.:	7998000	Vergl.:	4014450	Vergl.:	83296	Vergl.:	25236
	Zeit.:	10.32	Zeit.:	10.16	Zeit.:	0.22	Zeit.:	0.11
16000	—		—		Beweg.:	264770	Beweg.:	187267
	—		—		Vergl.:	397162	Vergl.:	107470
	—		—		Zeit.:	1.37	Zeit.:	0.61
32000	—		—		Beweg.:	561042	Beweg.:	420077
	—		—		Vergl.:	857712	Vergl.:	194942
	—		—		Zeit.:	2.85	Zeit.:	1.15
64000	—		—		Beweg.:	1186421	Beweg.:	927213
	—		—		Vergl.:	1843480	Vergl.:	415727
	—		—		Zeit.:	6.21	Zeit.:	2.63

Tabelle 4.1: Quantitativer Vergleich der internen Sortierverfahren

Kapitel 5

Externes Sortieren

Alle Verfahren des vorigen Kapitels — von *Selection Sort* bis *Quicksort* — waren *interne* Verfahren, Verfahren also, die auf einer geeigenten hauptspeicher-internen Datenstruktur, vornehmlich dem *Array*, ausgeführt wurden.

Andere Probleme ergeben sich natürlich, wenn die zu sortierenden Daten auf einer Datei vorliegen. Natürlich kann man versuchen, dieses Problem dadurch zu lösen, daß man die gesamten Daten in ein Array einliest, intern sortiert und dann wieder zurückschreibt. So geht z.B. der MS-DOS `SORT`-Befehl vor. Der resultierende Nachteil liegt auf der Hand: Die Anzahl der zu sortierenden Elemente ist durch die Größe des zur Verfügung stehenden Hauptspeichers begrenzt. Das führt etwa bei der Verwendung des MS-DOS `SORT` dazu, daß man nur vergleichsweise kleine Dateien sortieren kann. Das Problem des externen Sortierens besteht demnach darin, daß man nach Verfahren suchen muß, die unter zwei wesentlichen Einschränkungen vorgehen:

1. *Speicherbedarf* $O(1)$: Es wird also gefordert, daß der intern benötigte Speicher *nicht mehr* von der Anzahl der Eingabewerte auf der Datei abhängt.
2. *Eingeschränkte Zugriffsmöglichkeit auf die zu sortierenden Elemente:* Bei einer normalen sequentiellen Datei ist ein Zugriff ja bestenfalls auf das aktuelle Eingabedatum möglich. Man kann nicht beliebig zurück oder nach vorne „fahren", um ein Datenelement zu finden.

Insbesondere die zweite Einschränkung führt dazu, daß man sich nach vollkommen anderen Verfahren umgucken muß, als für die interne Sortierung. Das Zauberwort für diese Verfahren heißt *Mischen*.

5.1 Sortieren durch Mischen

Die Idee des Mischens bedarf einer gewissen Erläuterung. Normalerweise versteht man unter „Mischen" eine Operation, die eine geordnete Struktur — etwa ein Kartenspiel — so weit wie möglich in Unordnung bringt. Das ist an sich ein algorithmisch hochinteressantes Problem, hier aber nicht gemeint. Gemeint ist vielmehr folgendes:

Man hat z.B. zwei sortierte Dateien mit gleicher Struktur vorliegen und steht vor der Aufgabe, daraus eine einzige *sortierte Datei zu machen, die die Elemente beider Ausgangsdateien enthält.*

Schematisch:

Datei 1: 1 3 5 7 9 ↘
Zieldatei: 1 2 3 4 5 6 7 8 9 10
Datei 2: 2 4 6 8 10 ↗

In der Tat sind die beiden Eingangsdateien in sich sortiert. Der Mischvorgang erzeugt nun eine neue wiederum in sich sortierte Datei, die alle Elemente der beiden Eingangsdateien enthält. Das ist auch i.w. die grundlegende Idee des externen Sortierens. Vorher — wie immer — Abstraktion!

5.1.1 Bänder

Für die Zwecke des Pseudocodes definieren wir uns — in Anlehnung an `Pascal`-Files — einen ADT `tape` (Band), der auf folgende Schnittstelle hört:

1. `read(f,x)` besetzt die Variable x mit dem nächsten Wert aus dem file f.
2. `write(f,x)` schreibt den Wert der Variablen x als nächstes auf den file f.
3. `eof(f)` ist wahr, wenn auf f kein weiterer Wert mehr zur Verfügung steht.

Außerdem sei verabredet, daß `tape`s immer mit dem Buchstaben „`t`" beginnen.

5.1.2 Die Idee des Mischens

Nun kann man einen sehr einsichtigen Misch-Algorithmus `tape-merge` für solche Bänder entwerfen. Er wird in mehreren Phasen ablaufen, die jeweils aus zwei Schritten bestehen:

1. „Verteilen" des Inhalts von einem Band `t` auf zwei andere `t1` und `t2` (`distribute`).
2. „Mischen" der beiden Bänder `t1` und `t2` zurück auf das erste `t` (`tape-merge`).

Schematisch zeigt das Abb. 5.1.

Wir brauchen also *drei* Bänder, was dem Verfahren seinen Namen gibt: *3-Band Mischen* bzw. *3-Band Sortieren*. Natürlich läßt sich das ganze auf mehr als drei Bänder generalisieren, aber asymptotisch ergibt sich keine Verbesserung.

Die Idee, die hinter dem 3-Band-Sortieren steht, ist schnell erklärt. Nehmen wir an, daß wir als erstes einen Verteilungslauf vom Eingabeband `t` auf die beiden anderen Hilfsbänder vornehmen. Schematisch:

t: 1 7 3 2 8 6 4 5 → distribute ↗ t1: 1 3 8 4
↘ t2: 7 2 6 5

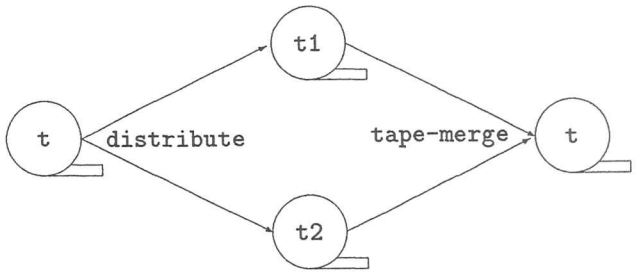


Abbildung 5.1: `distribute` und `tape-merge` schematisch

Mischt man beide Bänder nun zurück nach `t`, entsteht folgendes:

t1: 1 3 8 4
↘ tape-merge → t: 1 7 2 3 6 8 4 5
↗ t2: 7 2 6 5

Betrachtet man nun den Inhalt des Bands `t`, so stellt man fest, daß es nun etwas „sortierter" geworden ist: Es besteht nun aus in sich sortierten *Zweierblöcken*.

Um die nächste Phase klarzumachen, nehmen wir einen konzeptionell gewagten Schritt vor: Wir betrachten als nächstes *zwei hintereinanderliegende* Werte als Einheit, die durch das Verteilen nicht mehr getrennt werden dürfen[1] und machen dann eine neue Verteilung:

t: [1 7] [2 3] [6 8] [4 5] → distribute ↗ t1: [1 7] [6 8]
↘ t2: [2 3] [4 5]

Nun brauchen wir nur noch eine kleine entsprechende Modifikation des Mischens, die nicht mehr die beiden Kopfelemente der Eingabebänder anguckt, sondern in der Lage ist, aus zwei jeweils sortierten Zweierfolgen eine sortierte *Viererfolge* zu konstruieren:

t1: [1 7] [6 8]
↘ tape-merge → t: [1 2 3 7] [4 5 6 8]
↗ t2: [2 3] [4 5]

Den Rest sieht man nun ganz schnell. Aus den beiden Viererfolgen machen wir durch Verteilen und Mischen eine sortierte *Achterfolge*. In diesem Fall ist die ganze Angelegenheit dann auch schon erledigt:

[1] Deshalb enthält unser Beispiel auch 8 Werte, also eine Zweierpotenz! Die Komplikationen kommen immer noch früh genug.

```
                                          t1: [1 2 3 7]
t: [1 2 3 7] [4 5 6 8]   → distribute ↗
                                          t2: [4 5 6 8]

t1: [1 2 3 7]
                  ↘
                    tape-merge →  t: [1 2 3 4 5 6 7 8]
t2: [4 5 6 8]
                  ↗
```

Dieses Verfahren heißt **Geradeaus-Mischen** (engl. straight merge) und stellt das einfachste externe Sortierverfahren dar. Die Komplexität sieht ganz passabel aus: `distribute` und `tape-merge` haben in der Tat die Komplexität $\Theta(n)$, wie wir später sehen werden, und werden $\Theta(\log n)$ mal aufgerufen, so daß sich eine Gesamtkomplexität von $\Theta(n \log n)$ ergibt, also prinzipiell optimal.

Bei der Implementation dieses Verfahrens stellen sich sofort zwei Probleme:

Problem 1: Wie erkennt man, daß mit der Sortierung fertig ist? Anders gefragt: Wieviele `distribute`/`tape-merge`-Läufe muß man ausführen?

Problem 2: Erfahrungsgemäß haben die wenigsten Bänder eine Länge, die eine Zweierpotenz ist. Unser Verfahren sollte allerdings auch mit so etwas fertig werden. Wie äußert sich die Tatsache, daß wir keine Zweierpotenz vor uns haben?

Das erste Problem läßt sich relativ schnell lösen, wenn man eine Vorstellung davon hat, *wieviele* Elemente das zu sortierende Band enthält. Das kann man aber beim ersten `distribute` feststellen. Wenn sich dabei eine Anzahl n von Elementen auf dem Band ergibt, so muß das letzte `tape-merge` mit Folgen der Länge $\lceil n/2 \rceil$ vorgenommen werden. Damit hat man bereits den Kern dieses Verfahrens vor sich:

```
procedure straight-merge(t)

  seqlen := 1

  {1. Aufteilung. Dabei Laenge bestimmen}
  distribute(t,t1,t2,seqlen)
  lim := ceiling(n/2)          {Groesste Folgenlaenge}

  repeat                       {Mischen/Teilen bis max. Laenge}
     tape-merge(t1,t2,t,seqlen)
     seqlen := 2*seqlen        {Laenge verdoppeln}
     distribute(t,t1,t2,seqlen)
  until seqlen > lim

  tape-merge(t1,t2,t,seqlen)  {Letzes Mischen. Fertig}
```

Das zweite Problem mit den Nicht-Zweierpotenzen hat es ein wenig in sich, weil gegen Ende der Sortierungen die Folgen immer länger werden und die beiden Hilfsbänder u.U. sehr unterschiedlich belegt sein können.

Aufgabe: *Machen Sie sich klar, daß bei einer Länge $n = 2^k + 1$ bei der letzen Phase auf einem Band 2^k Elemente stehen und auf dem anderen gerade eins.*

Wenn man nun die Prozeduren `distribute` und `tape-merge` mit den normalen Operationen implementiert, gelangt man zu einen Code, der ausgesprochen undurchsichtig ist. Ich hab' das selbst mal versucht und war ausgesprochen unglücklich mit dem Ergebnis!

5.1.3 Der ADT „triple-tape"

Also macht man das anders! Man definiert sich einen ADT `triple-tape` (3-Band). Wir stellen uns so ein Objekt vor als Zusammenfassung von drei Bändern, die auf die Namen `main`, `aux1` und `aux2` hören. `main` entspricht dabei unserem `t`, die beiden `aux`-Bänder den Bändern `t1` und `t2`. Jedes der drei Bänder ist mit einem Puffer ausgestattet, der es erlaubt, *genau ein* Datenelement des Bands zu speichern. Zusätzlich besitzt jedes der drei Bänder ein *Begrenzungsregister* namens `limit`, das gesetzt und abgefragt werden kann. Die Bedeutung der drei hauptsächlichen Operationen `getrec`, `putrec` und `moverec` entnehme man der Abbildung 5.2.

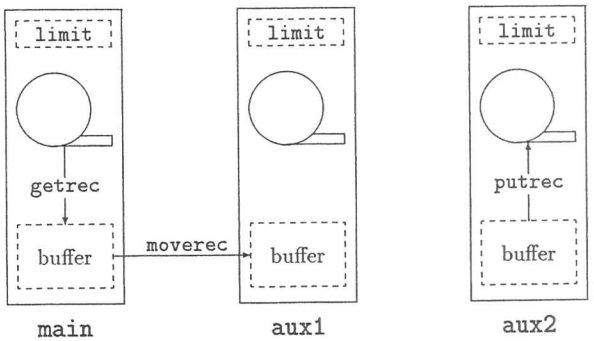


Abbildung 5.2: Der ADT `triple-tape`

Arbeiten wir zunächst an seiner Schnittstelle, wobei wir annehmen, daß `t` für ein solches 3-Band steht. Es gibt drei Gruppen von solchen Operationen:

Gruppe 1: Verbindungen zur Außenwelt. Hier werden die wesentlichen Operationen definiert, die die Funktionen eines Bands darstellen.

1. `connect(t,x,string)`, `x` $\in$ {`main`, `aux1`, `aux2`}. Damit wird die Verbindung eines Bands zu einem physikalischen file mit dem Name `string` hergestellt. Das macht man eigentlich grundsätzlich so, weil hier oft sehr maschinenabhängige Operationen vorgenommen werden müssen, mit denen man sich im Text nicht andauernd herumärgern will.
2. `toread(t,x)` bzw. `towrite(t,x)`, `x` $\in$ {`main`, `aux1`, `aux2`}. Setzt das entsprechende Band an den Anfang und öffnet es zum Lesen bzw. zum Schreiben.
3. `endfile(t,x)`, `x` $\in$ {`main`, `aux1`, `aux2`} ist `true`, wenn auf dem entsprechenden Band kein Element zur Verfügung steht.

4. `getrec(t,x)`, `x` $\in$ {`main`, `aux1`, `aux2`}, transportiert ein Element vom Band in den zugehörigen Pufferbereich.
5. `putrec(t,x)`, `x` $\in$ {`main`, `aux1`, `aux2`}, transportiert ein Element vom Pufferbereich auf das entsprechende Band.
6. `moverec(t,x,y)`, `x,y` $\in$ {`main`, `aux1`, `aux2`}, transportiert das Element im Puffer von `x` in den Puffer von `y`.

Gruppe 2: Operationen auf dem `limit`-Register. Dieses Register dient zur abstrakten Beschränkung der Leseoperationen eines Bands auf Folgen bestimmter Länge. Dieses Register ist zu einem bestimmten Zeitpunkt mit einer Zahl l besetzt. Solange von diesem Band weniger als l Elemente gelesen worden sind, geschieht nichts. Erst wenn das l-te Element eingelesen wurde, wird das Band in einen Zustand namens `atlimit` gesetzt, der von außen abgefragt werden kann.

1. `setlimit(t,x,n)`, `x` $\in$ {`main`, `aux1`, `aux2`}, besetzt das Begrenzungregister des Bands `x` mit dem Wert `n`.
2. `atlimit(t,x)`, `x` $\in$ {`main`, `aux1`, `aux2`} (das Ergebnis ist ein Wahrheitswert). Es wird angezeigt, ob das Band `x` im `atlimit`-Zustand ist, d.h. es wurde der l-te Element bereits eingelesen.
3. `expandlimit(t,x,delta)`, `x` $\in$ {`main`, `aux1`, `aux2`} erhöht den Wert des `limit`-Registers von `x` um den Wert `delta`.

Gruppe 3: Sonstiges.

1. `other(x)`, `x` $\in$ {`aux1`, `aux2`} liefert das jeweilig andere Hilfsband zu `x`, also `other(aux1) = aux2` und `other(aux2) = aux1`.
2. `buff(t,x)`, `x` $\in$ {`main`, `aux1`, `aux2`} liefert das aktuelle Element im Puffer von Band `x`.

Mit diesen Operationen läßt sich eine neue Operation schnell definieren. `copyrec` kopiert einen Satz vom Band `source` auf das Band `dest`, es sein denn, auf `source` ist nichts mehr zu lesen. In diesem Fall geschieht nichts. Der Vorteil ist, daß man diese Prozedur aufrufen kann, sooft man will. Wenn `source` abgearbeitet ist, geschieht halt nichts. Man kann dann im rufenden Programm blind zählen, ohne sich um das Verhalten des `source`-Bands kümmern zu müssen:

```
procedure copyrec(t,source,dest)
  getrec(t,source)
  if not endfile(t,source) then
     moverec(t,source,dest)
     putrec(t,dest)
```

`writerec` schreibt das Pufferelement des `source`-Bands auf das `dest`-Band:

```
procedure writerec(t,source,dest)
  moverec(t,source,dest)
  putrec(t,dest)
```

Wenn man diese beiden Operationen einmal fertig hat, ist die Formulierung von `distribute` fast schon trivial.

Algorithmus 5.1 *Verteilen des Inhalts des* `main`*-Bands auf die beiden* `aux`*-Bänder, wobei Folgen der Länge n nicht unterbrochen werden.*

Precondition: `main enthält wenigstens ein Element`
Postcondition: `Inhalt von main steht auf den aux-Bändern`

```
procedure distribute(t,n)
  toread(t,main)          {main wird gelesen}
  towrite(t,aux1)         {aux1 wird beschrieben}
  towrite(t,aux2)         {aux2 ebenfalls}
  dest := aux1            {Zielband ist anfangs aux1}

  repeat
     for i := 1 to n do   {n Elemente kopieren}
        copyrec(t,main,dest)
     dest := other(dest)  {Zielband wechseln}
  until endfile(t,main)
```

`tape-merge` ist nun auch nicht mehr sehr kompliziert. Man baut das ganze so auf, daß die wesentliche Arbeit in einer Funktion `min-tape` geleistet wird, die entscheidet, von welchem der beiden `aux`-Bänder das nächste Element auf das `main`-Band zu kopieren ist. Die Vorbedingung für `min-tape` besteht lediglich darin, daß *nicht beide* `aux`-Bänder am Ende stehen. Eins allein ist in Ordnung.

In der Funktion `min-tape` werden folgende vier Fälle unterschieden:

1. *Eins der Bänder steht am Ende:* Das andere ist das Ergebnis
2. *Beide Bänder stehen auf* `atlimit`: Beide `limit`-Register werden um n weitergesetzt und danach das `min-tape` bestimmt (z.B. rekursiv!).
3. *Genau eins der Bänder steht auf* `atlimit`: Das andere ist das Ergebnis.
4. *Nichts davon ist der Fall:* Lediglich in diesem Fall werden die beiden Puffer-Elemente der `aux`-Bänder miteinander verglichen und dasjenige mit dem kleinsten Element ist das Ergebnis.

Algorithmus 5.2 *Bestimmen des* `aux`*-Bands, von dem das nächste Element beim Mischen genommen werden muß.*

Precondition: `not (endfile(t,aux1) and endfile(t,aux2))`
Postcondition: `---`

```
function min-tape(t,n)
  if endfile(t,aux1) then {Fall 1: Steht eins der Baender am Ende?}
     return aux2
  else
     if endfile(t,aux2) then
        return aux1
```

```
else                    {Fall 2: Stehen beide Baender atlimit?}
   if atlimit(t,aux1) and atlimit(t,aux2) then
      expandlimit(t,aux1,n)
      expandlimit(t,aux2,n)
      return min-tape(t,n)
   else                 {Fall 3: Eins der Baender atlimit?}
      if atlimit(t,aux1) then
         return aux2
      else
         if atlimit(t,aux2) then
            return aux1
         else           {Fall 4: Pufferelemente Vergleichen!}
            if buff(t,aux1) <= buff(t,aux2) then
               return aux1
            else
               return aux2
```

Nun ist alles ganz einfach! Hier ist nun die eigentliche tape-merge Prozedur!

Algorithmus 5.3 *Mischen des Inhalts der beiden* aux-*Bänder auf das* main-*Band. Es werden Folgen der Länge* n *zusammengemischt.*

Precondition: Beide aux-Bänder nicht leer
Postcondition: Inhalt der aux-Bänder steht auf main

```
procedure tape-merge(t,n)
  toread(t,aux1)            {aux-Baender zum Lesen oeffnen}
  toread(t,aux2)

  setlimit(t,aux1,n)        {limit-Register auf Folgenlaenge}
  setlimit(t,aux2,n)

  towrite(t,main)           {Auf das main-Band wird geschrieben}

  getrec(t,aux1)            {Die ersten beiden Elemente lesen}
  getrec(t,aux2)

  repeat
     source := min-tape(t,n)        {Quellband bestimmen}
     writerec(t,source,main)        {source ---> main}
     if not endfile(t,source) then
        getrec(t,source)            {Element nachlesen}
  until endfile(t,aux1) and endfile(t,aux2)
```

Nun müssen wir nur noch die Prozedur straight-merge an unseren ADT anpassen. Das ist eine Kleinigkeit!

Algorithmus 5.4 *Externes Sortieren durch Geradeaus-Mischen.*

Precondition: main-Band nicht leer
Postcondition: Inhalt des main-Bands ist sortiert

```
procedure straight-merge(t)

  seqlen := 1
  distribute(t,seqlen)        {1. Aufteilung. Dabei n bestimmen}
  lim := ceiling(n/2)         {Groesste Folgenlaenge}

  repeat                      {Mischen/Teilen bis max. Laenge}
     tape-merge(t,seqlen)
     seqlen := 2*seqlen       {Laenge verdoppeln}
     distribute(t,seqlen)
  until seqlen > lim

  tape-merge(t,seqlen)        {Letzes Mischen. Fertig}
```

5.1.4 Komplexitätsbetrachtungen

Man muß nun klären, wie teuer Geradeaus-Mischen eigentlich ist. Was man recht schnell erkennt, ist, daß *die Anzahl* der distribute/tape-merge-Operationen logarithmisch mit der Anzahl der Elemente auf dem Ausgangsband läuft. Jeder distribute geht offenbar linear mit der Anzahl der Elemente und das gleiche gilt für den tape-merge. Tatsächlich sind beides i.w. Kopieroperationen, die also jeweils aus zwei Bandzugriffen bestehen. D.h. für distribute sind es $2n$ Zugriffe und genauso für tape-merge. Somit gestaltet sich die gesamte Komplexitätsanalyse beachtlich einfach: Wir benötigen

$$\lceil \log_2(n/2) \rceil \cdot 2 \cdot \ 2 \cdot n = 4 \cdot \lceil \log_2(n/2) \rceil \cdot n$$

Lese- bzw. Schreiboperationen für das Geradeaus-Mischen, also in der Tat $\Theta(n \log n)$.

5.2 Verbesserungen

Eine unendliche Geschichte! Im Prinzip gibt es neben dem Geradeaus-Mischen eine ganze Reihe von Mischverfahren, die auf einer ähnlichen Idee basieren und gewisse praktische Verbesserungen bringen.

Eine ganz einfache Beobachtung an unserem Entwurf ist die, daß wir die Forderung nach einem Speicherbedarf von $O(1)$ zu wörtlich genommen haben: Wir haben so wenig Speicher wie möglich verwendet. Wenn man *soviel* Speicher wie möglich verwenden will, kann man z.B. interne Sortierverfahren anstelle der ersten paar Runden verwenden und so eine konstante Anzahl von distribute/tape-merge-Paaren einsparen. Das hat den Vorteil, daß man für kürzere Bänder unmittelbar intern sortiert.

Am schlausten ist es dabei, nach dem ersten `distribute` keinen normalen `tape-merge` zu veranstalten, sondern eine Version, in der die Werte von `t1` und `t2` in eine *priority queue* (s. 4.2.3) gepackt werden, die dann sukzessive — durch andauerndes Entnehmen des kleinsten Elements — nach `t` zurückgeschrieben wird. So ist die Codierung am einfachsten.

Aufgabe: *Geben Sie Pseudo-Code für eine Prozedur mit dem Namen* `tape-merge-pq` *an, die zwei Bänder* `t1` *und* `t2` *durch Einschalten einer priority queue der Größe* 2^k *auf das Band* `t` *mischt.*

Wenn man dann diese priority queue durch einen Heap realisiert, hat man implizit **heapsort** (s. 4.2) für die Sortierung benutzt und kann damit eine Reihe von Durchläufen (u.U. alle) durch die eigentliche Schleife einsparen.

Ein anderes Problem ensteht dadurch, daß das Geradeaus-Mischen keinerlei Rücksicht auf eventuell bereits vorhandene sortierte Teilsequenzen nimmt. Insbesondere zeigt es ein abenteuerliches Verhalten auf vorsortierten Dateien. Das sogenannte **natürliche Mischen** (engl. natural merge) geht dementsprechend anders vor als das Geradeaus-Mischen, weil es keine Vorannahmen über die Länge der sortierten Folgen auf den Bändern macht. Straight merge tut das ja, weil diese Länge immer als 2^k beim k-ten Durchlauf angenommen wird.

Stattdessen muß man beim natürlichen Mischen in der Prozedur `tape-merge` *dynamisch* feststellen, ob man sich auf einem der Bänder noch innerhalb einer sortierten Teilfolge befindet und solange weitermischen, bis beide Teilfolgen optimal abgearbeitet wurden. Das erreicht man dadurch, daß man jeweils ein Element weiter nach vorn guckt als man müßte (auf `eof` aufpassen!). Außerdem sollte man sich beim `distribute` einiges einfallen lassen, denn jeder Verteilungslauf sollte der tatsächlichen Sortierung Rechnung tragen.

Ohne in alle Einzelheiten zu gehen, kommt hier noch ein wenig Pseudo-Code für das natürliche Mischen (nach [Wirth, 1979]). Die Bezeichnung für eine maximale sortierte Teilfolge auf einem Band ist dabei *Lauf* (engl. run). Die erste Prozedur `copyrun` kopiert einen Lauf vom Band `x` auf das Band `y` des triple-tape `t` :

```
procedure copyrun(t,x,y)
  repeat
     copyrec(t,x,y)
  until {Ende des Laufs auf Band x}
```

Zwei Läufe mischt man von `aux1` und `aux2` nach `main` recht einfach:

```
procedure mergerun(t)
  repeat
     if buff(t,aux1) <= buff(t,aux2) then
        writerec(t,aux1,main)
        getrec(t,aux1)
        if {Lauf auf aux1 beendet} then
           copyrun(t,aux2,main)
     else
        writerec(t,aux2,main)
        getrec(t,aux2)
        if {Lauf auf aux2 beendet} then
           copyrun(t,aux1,main)
  until {Lauf auf aux1 beendet} or
        {Lauf auf aux2 beendet}
```


Nun kann man sich die gesuchte Prozedur `distribute` schreiben, die die Sortierung innerhalb der Läufe erhält:

```
procedure distribute(t)
  toread(t,main)
  towrite(t,aux1)
  towrite(t,aux2)
  repeat
     copyrun(t,main,aux1)
     if not endfile(t,main) then
        copyrun(t,main,aux2)
  until endfile(t,main)
```

So weit, so gut! Man muß sich nun allerdings vor einem kleinen Denkfehler hüten. Aus der Inspektion des Codes von `distribute` kann man auf die (falsche!) Idee kommen, daß die Anzahl der Läufe auf den Bändern `aux1` und `aux2` entweder gleich ist oder daß auf `aux1` genau *ein* Lauf mehr steht als auf `aux2`. Daß das nicht so ist, sieht man an einem kleinen Gegenbeispiel. Die einzelnen Läufe sind in dieser Darstellung eingeklammert:

```
main:  [8] [2 5 9] [6 13 19] [17 23 31] [14 37 43] [41 47 59] [37 61 71] [67]
aux1:  [8] [6 13 19] [14 37 43] [37 61 71]
aux2:  [2 5 9] [17 23 31] [41 47 59] [67]
```

So sind die Läufe auf die beiden Bänder verteilt worden. Das Problem dabei ist, daß sich auf `aux2` für die nächste Runde *nur noch ein einziger* Lauf präsentiert. Denn die Elemente auf diesem Band sind bereits sortiert! So sieht es nämlich „in Wirklichkeit" aus:

```
aux1:  [8] [6 13 19] [14 37 43] [37 61 71]
aux2:  [2 5 9 17 23 31 41 47 59 67]
```

Wir haben also auf `aux1` vier Läufe und auf `aux2` gerade mal einen!

Aufgabe: *Stellen Sie den Inhalt von* `aux1` *und* `aux2` *nach der Verteilung folgender Eingabe auf dem* `main`*-Band auf:*

[3] [2 5 11] [7 13 19] [17 23 31] [29 37 43] [41 47 59] [57 61 71] [67]

Überlegen Sie sich, was beim laufweisen Mischen im nächsten Schritt passiert.

Wenn man nun überlegt, wie die eigentliche Mischphase abläuft, muß man sich vor Augen halten, daß nach dem Mischen der beiden ersten Läufe auf `aux1` und `aux2` auf beiden noch

eine unbekannte Anzahl weiterer Läufe stehen kann. Das heißt, daß man nach dem Aufruf von `mergerun` noch beide Bänder solange abklappern muß, bis *physikalisch* mit Sicherheit nichts mehr drauf ist. Zählen nützt also nichts! Zusätzlich zählen wir die Anzahl der Läufe auf dem `main`-Band mit, weil wir später noch ein gescheites Abbruchkriterium brauchen (Anzahl der Läufe auf `main` ist 1).

```
procedure tape-merge(t; var number-of-runs)
  toread(t,aux1)
  toread(t,aux2)
  towrite(t,main)
  number-of-runs := 0

  repeat
     mergerun(t)
     number-of-runs := number-of-runs + 1
  until endfile(t,aux1) or endfile(t,aux2)

  {Beide Hilfsbaender abklappern}

  while not endfile(t,aux1) do
     copyrun(t,aux1,main)
     number-of-runs := number-of-runs + 1;

  while not endfile(t,aux2) do
     copyrun(t,aux2,main)
     number-of-runs := number-of-runs + 1
```

Das war's dann auch schon! Das natürliche Mischen sieht dann noch viel einfacher aus als das Geradeaus-Mischen:

Algorithmus 5.5 *Externes Sortieren durch natürliches Mischen.*

Precondition: ---
Postcondition: Inhalt des main-Bands ist sortiert

```
procedure natural-merge(t)
  repeat
     distribute(t)
     tape-merge(t,number-of-runs)
  until number-of-runs = 1
```

Kapitel 6

Hashing

Hashing (s. z.B. [Knuth, 1973]) ist eine sehr verbreitete Technik, deren Vorteil darin besteht, daß eine potentiell sehr große Menge von Schlüsseln ziemlich effizient auf einen vergleichsweise kleinen aktuellen Speicher abgebildet werden. Typisch für Anwendungen von Hashing sind alle Arten von Zeichenketten-Speicherung, wie man sie z.B. im Compilerbau bei der Bezeichnerverwaltung benötigt.

Obwohl meistens sehr einfach zu implementieren, sind Hashing-Verfahren theoretisch nicht ganz ohne Tücken! Man hat in der Auswahl des speziellen Verfahrens einige Freiheiten, deren Auswirkungen man kennen sollte, um keine leichtfertigen Entscheidungen zu treffen. Eine falsch gewählte Konstante z.B. kann das gesamte Speicherungssystem in seiner Effizienz gewaltig verschlechtern, also sind Vorsicht, Nachdenken und Rechnen gefordert. Aus diesem Grund enthält dieser Abschnitt einiges an Mathematik.

6.1 Einführung

Hashing ist in gewisser Weise eine Verallgemeinerung des von Arrays bekannten *Zugriffs über einen Index*. Natürlich ist es am angenehmsten, wenn man die gegebenen Schlüssel unmittelbar als Array-Indizes verwenden kann: Es reicht dann einfach aus, den direkten Zugriff auf den Speicher mit dem Schlüssel vorzunehmen. Leider weigert sich z.B. Pascal, so etwas zu akzeptieren:

```
TYPE Speicher = ARRAY [STRING] OF ...
```

Das liegt natürlich daran, daß die potentielle Schlüsselmenge viel zu groß ist, um für jeden Schlüssel einen Platz reservieren zu können. Andererseits ist es natürlich auch so, daß zu jedem Moment nur ein verschwindend kleiner Teil aller möglichen Schlüsselwerte benötigt wird, so daß dieser hypothetische Riesenspeicher auch so ziemlich leer stehen dürfte.

Beide Überlegungen führen dazu, daß man versucht, einen Kompromiß so zu finden, daß man

- den angebotenen *Adreßbereich* so klein macht, daß er bequem in einem Programm gehandhabt werden kann, aber dafür
- darauf verzichtet, den *Schlüsselbereich* bijektiv auf den Adreßbereich abzubilden.

6.2 Das Szenario

Stellen wir zunächst mal zusammen, welche Akteure bei diesem Unternehmen eine Rolle spielen:

Der Adreßbereich ist eine endliche Menge von Werten $\mathcal{A} = \{0, 1, \ldots, m-1\}$, die z.B. als Indizes in einem Array dienen können.

Die Hash-Tabelle ist der eigentliche Speicher, der durch die Adressen aus $\mathcal{A}$ adressiert wird. Sie wird in einer höheren Programmiersprache typischerweise als Array realisiert.

Der Schlüsselbereich ist eine endliche Menge von Werten $\mathcal{S} = \{s_0, s_1, \ldots, s_{n-1}\}$, die auf den Adreßbereich abgebildet werden soll.

Die Abbildungsfunktion h bildet also $\mathcal{S}$ in $\mathcal{A}$ ab.

Von h sprechen wir ab sofort als *Hash-Funktion*! Der Fall $m \geq n$ ist völlig harmlos, denn es gibt dann ausreichend Platz für alle Schlüssel im Adreßbereich. Interessant ist also der Fall $m < n$, d.h. es stehen weniger Adressen als Schlüssel zur Verfügung. Für h bedeutet dies zweierlei

1. h ist nicht mehr injektiv,
2. h ist nicht mehr umkehrbar.

Die Tatsache, daß h nicht mehr injektiv ist, bedeutet also, daß es nun zwei Schlüssel s_1 und s_2 gibt, für die $h(s_1) = h(s_2)$ gilt. Sollten zwei solche Schlüssel tatsächlich einmal zusammen auftreten, sprechen wir von einer *Kollision*. Die Behandlung von Kollisionen ist eins der Hauptprobleme dieses Verfahrens (s. 6.3).

Was kann man in der Praxis von h verlangen? Bedenkt man, daß zu jedem Zugriff über einen solchen Schlüssel s der Wert $h(s)$ bestimmt werden muß, kommt man auf die erste praktische Anforderung

- h sollte *schnell zu berechnen* sein.

Eine triviale, schnell berechenbare Hash-Funktion wäre nun etwa

$$\begin{aligned} h : \mathcal{S} &\rightarrow \mathcal{A} \\ h(s) &= 0, \end{aligned}$$

die in der Tat alle einsichtigen Anforderungen erfüllt. Dennoch scheint sie ungeeignet zu sein, denn sie läßt zunächst fast alle möglichen Adressen außer acht, verschwendet also Adreßraum. Ferner führt sie dazu, daß bereits bei zwei Schlüsseln eine Kollision auftritt, was immer Probleme mit sich bringt. Wir fordern also

- h sollte *surjektiv* sein, d.h. jeder Wert des Adreßbereichs sollte als Bildwert $h(s)$ für einen Schlüssel s auftreten können.

6.2.1 Beispiele für Hash-Funktionen

Unabhängig von einer bestimmten Anwendung betrachten wir einmal kurz Vor- und Nachteile bestimmter Hash-Funktionen, wobei wir immer als Schlüsselmenge $\mathcal{S}$ die Menge von

Strings beliebiger Länge annehmen wollen.

Wir demonstrieren hier die beiden wichtigsten Möglichkeiten, Hash-Funktionen zu konstruieren.

Die Divisions-Methode

Recht einfach — abgesehen von eventuellen Bereichsüberläufen — ist die Methode, die Zahlenwerte der einzelnen Zeichen des Strings (in `Pascal` mit der `ORD`-Funktion, in `C` einfach so) zu einem Wert v aufzuaddieren und als h-Wert v modulo m zu nehmen:

```
v := 0
for i := 1 to length(S) do
   v := v + ORD(S[i])
return (v mod m)
```

Wann treten Kollisionen auf? Klar: Bei Anagrammen, also Strings, die dieselben Zeichen an verschiedenen Stellen enthalten, aber auch bei anderen.

Rechnet man mit dieser Hash-Funktion und einem Wert von $m = 307$, so kommen z.B. die Kollisionen zustande, daß die Strings „`DISJUNKTE`", „`FARBKORREKTUR`", „`RINGLISTE`" und „`ZEICHENVORRAT`" alle auf die Adresse 75 abgebildet werden.

Die Multiplikations-Methode

Etwas anders kann man an die Geschichte herangehen, wenn man die resultierende Zahl v im obigen Beispiel nicht einfach modulo m nimmt, sondern mit einer Konstanten $0 < c < 1$ multipliziert, davon die Nachkommastellen nimmt, diese Zahl, die nun < 1 ist, mit m multipliziert und vom Ergebnis die Vorkommastellen als Funktionswert verwendet:

```
return trunc(m*frac(v*c))
```

Hier hängt seltsamerweise viel von der Wahl der Konstaten c ab, wie man sich leicht überlegt. Nimmt man z.B. den extremen Fall $c = 0.5$, so entsteht durch den Teilausdruck `frac(v*c)` entweder eine 0 (wenn v gerade ist) oder eine 0.5 (wenn `v` ungerade ist).

Das nun mit m multipliziert und abgeschnitten ergibt immer entweder 0 oder $\lfloor \frac{m}{2} \rfloor$, was dazu führt, daß alle unsere Strings auf *zwei* Adressen abgebildet werden und das ist nun überhaupt nicht das, was wir eigentlich wollen!

Allgemein betrachten wir mal eine Wahl von $c = n_1/n_2$, wobei n_1 und n_2 teilerfremd sein sollen. Für einen gegebenenen Wert v_1 rechnen wir zunächst $v_1 \cdot c = (n_1 \cdot v_1)/n_2$ aus und vergleichen das mal mit der Rechnung für $v_2 = v_1 + k \cdot n_2$, $k = 1, \ldots$, also für einen Wert, der sich additiv um ein Vielfaches des Nenners von v_1 unterscheidet. Es ergibt sich

$$v_2 \cdot c = \frac{n_1 \cdot v_2}{n_2} = \frac{n_1 \cdot (v_1 + k \cdot n_2)}{n_2} = \frac{n_1 \cdot v_1}{n_2} + \frac{k \cdot n_1 \cdot n_2}{n_2} = v_1 \cdot c + k \cdot n_1.$$

Nimmt man also nun per `frac` die Nachkommastellen dieser Ausdrücke, so ergibt sich für v_1 und v_2 gar kein Unterschied mehr! Sie landen letztlich auf derselben Adresse.

Es muß also bei der Multiplikationsmethode auf alle Fälle beachtet werden, daß bei einer rationalen Konstanten $c = n_1/n_2$ der Nenner $n_2 > m$ ist[1].

6.3 Kollisionsbehandlung

Wir wissen nun einiges über den Aufbau von Hash-Funktionen mit dem Ziel, Kollisionen weitgehend zu minimieren. Dennoch ist es in der Praxis natürlich kaum möglich, ohne eine vernünftige Behandlung von Kollisionen auszukommen.

Es gibt i.w. zwei Möglichkeiten, mit einer Kollision fertig zu werden:

1. Man organisiert die Hash-Tabelle so, daß die mehrfachen Einträge unter einer Adresse als verkettete Liste realisiert werden (Chaining).
2. Man errechnet aus dem Original-Funktionswert von h einen neuen und versucht, den Schlüssel dort einzutragen (Re-Hashing, Probing oder Sondieren).

6.3.1 Verkettung

Dies ist das theoretisch einfachste Verfahren. Man speichert in der Hash-Tabelle nicht die Schlüssel, sondern Köpfe einer verketteten Liste, in der Schlüssel enthalten sind (s. Abb. 6.1).

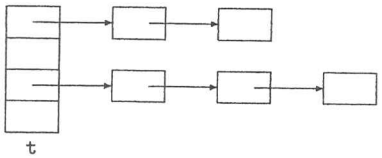


Abbildung 6.1: Kollisionsbehandlung durch Verkettung

Beim Einfügen und Suchen eines Schlüssels handelt es sich also um elementare Operationen auf einer verketteten Liste und man muß sich lediglich darüber klar werden, ob man eine eventuelle *Ordnung* auf den Schlüsseln ins Kalkül ziehen will.

Hier als einziges Beispiel das Einfügen:

Algorithmus 6.1 *Einfügen des Schlüssels* `key` *in die Hash-Tabelle* `htab` *unter Verwendung der Hash-Funktion* `h`. *Kollisionsbehandlung durch Verkettung. Zwischen den Schlüsseln ist eine Ordnung „<“ definiert. Vergl. auch Algorithmus 3.2.*

Precondition: `key` $\notin$ $\texttt{htab}_{pre}$
Postcondition: $\texttt{htab}_{post}$ = $\texttt{htab}_{pre}$ + `key`

[1] Das ist insofern wichtig, weil in der Computerarithmetik alle Zahlen rational sind!


```
procedure chained-insert(key)

  new(x)                    {Knoten fuer "key" bauen}
  key(x)   := key
  next(x)  := nil

  index    := h(key)        {Hash-Funktion auf "key" anwenden}

  p        := htab[index]   {Kettenkopf aus der Tabelle holen}

  {Klassisches Einfuegen in die Kette p}
  if p = nil then           {Kette ist leer}
     htab[index] := x
  else
     if key(p) >= key       {Vorne einfuegen}
        next(x)     := head
        htab[index] := key
     else                   {Dahinter einfuegen}
        while (p <> nil) and (key(p) < key) do
           pv := p
           p  := next(p)

        next(x)  := p       {Einklinken}
        next(pv) := x
```

Aufgabe: *Geben Sie Pseudocode für die Suchoperation* `chained-lookup` *und die Löschoperation* `chained-delete` *an.*

Wie müssen die Operationen aussehen, wenn auf der Schlüsselmenge keine Ordnung definiert ist?

Komplexität

Bevor wir uns im Detail um die Komplexität dieser Verfahren kümmern, müssen wir festlegen, wovon die Laufzeit denn abhängen soll. Dazu brauchen wir natürlich den *aktuellen* Inhalt der Tabelle, also die Menge S von Schlüsseln, die im Moment angespeichert sind. Setzen wir $N = |S|$, so bildet N die Grundlage der Analyse.

Die Komplexität der Berechnung von h spielt offenbar jedesmal eine Rolle. Wir nehmen hier grundsätzlich an, daß es sich dabei um eine Operation mit der Zeitkomplexität $O(1)$ handelt.

Beim Einfügen ohne Beachtung der Ordnung kommen wir natürlich auch mit $O(1)$ aus, wenn wir den neuen Schlüssel immer an den Anfang der Kette setzen.

Bei der Komplexität der Suchfunktion bleibt also das Durchlaufen durch die Kette, was natürlich von ihrer Länge abhängt. Im schlechtesten Fall haben wir natürlich $\Theta(N)$ als

Komplexität, wenn nämlich (fast) alle Schlüssel in der entsprechenden Kette hängen. In diesem Fall liegt das Problem wohl eher in der Funktion h!

Andererseits zeigt diese Überlegung recht deutlich, daß manchmal die Analyse des schlechtesten Falls doch keinen Aufschluß über die Qualität eines Verfahrens liefert. In der Tat kann man von einer vernünftigen Hash-Funktion in der Praxis viel bessere Werte erwarten.

Dazu müssen wir uns doch ein wenig mit der „Analyse des mittleren Falls" befassen.

Definition 6.1 *Gegeben eine Hash-Tabelle mit m Adressen und N gespeicherten Schlüsseln. Der Wert $\alpha = N/m$ heißt* **Auslastung** *(engl. load factor) der Tabelle.*

Dieses α beschreibt etwas mehr als die einfache Auslastung der Tabelle. Nehmen wir mal an: $\alpha = 3$. Wir haben also dreimal soviele Elemente abgespeichert wie es Adressen gibt. Wir können dann den Mittelwert der Kettenlängen angucken und finden natürlich auch diesen Wert 3. α gibt also auch noch den Mittelwert der Kettenlängen an!

Natürlich wissen wir nicht, ob sich im allgemeinen Fall Kettenlängen ergeben, die alle nahe bei α liegen, das hängt von der Hash-Funktion und den aktuell verwendeten Schlüsseln ab. Wir können allerdings die *Wahrscheinlichkeit* angucken, mit der so etwas passiert. Um so eine Situation annehmen zu können, sollte h die möglichen Schlüssel möglichst gleichförmig über den Adreßbereich $\mathcal{A}$ verteilen:

Definition 6.2 *Eine Hash-Funktion h heißt* **uniform**, *wenn h die Elemente von $\mathcal{S}$ gleichmäßig über $\mathcal{A}$ verteilt, d.h.*

$$\forall i, j \in \mathcal{A} \text{ ist } |h^{-1}(i)| = |h^{-1}(j)|$$

Ein Schlüsselbereich $\mathcal{S}$ heißt **uniform**, *wenn alle Elemente als aktuelle Schlüssel gleichwahrscheinlich sind, d.h. die Wahrscheinlichkeit für das Auftreten eines Schlüssels $s^* \in \mathcal{S}$ ist die Wahrscheinlichkeit $p(s = s^*) = 1/|\mathcal{S}| = 1/n$.*

Es ist meistens ausgesprochen schwierig, für einen gegebenen Schlüsselbereich und eine gegebene Hash-Funktion h zu beweisen, daß h uniform ist; meistens wird man Uniformität der Hash-Funktion als „wünschenswert" ansehen und informal nachweisen. Die Uniformität eines Schlüsselbreichs ist zwar ganz hübsch, für praktische Anwendungen aber ziemlich irrelevant. Die Uniformitätsannahme ist massiv abhängig von der Anwendung, die zugrundeliegt. Will man z.B. deutsche Wörter hashen, so sind beileibe nicht alle Strings gleichwahrscheinlich[2]; will man so etwas als *Bezeichnerverwaltung* in einem Compiler einsetzen, gelten ganz andere Regeln[3].

Nun aber weiter in der Rechnerei! Wir nehmen für einen Moment einen uniformen Schlüsselbereich $\mathcal{S}$ und eine dafür uniforme Hash-Funktion h an. Man sieht mit einigem Nachdenken:

[2] *Ich* kenne kein deutsches Wort, in dem der Teilstring „**xyzzy**" vorkommt.
[3] Eine Variable „**xyzzy**" hab' ich schon gesehen!

Aussage 6.1 *Für einen uniformen Schlüsselbereich $\mathcal{S}$, einen Adreßbereich und eine für $\mathcal{S}$ uniforme Hash-Funktion h sind die Funktionswerte von h* gleichverteilt, *d.h.*

$$p(h(s_k) = i) = 1/|\mathcal{A}| = 1/n \quad \text{für alle } s_k \in \mathcal{S}, i \in \mathcal{A}.$$

Untersuchen wir nun schnell den Aufwand für das *erfolglose* Suchen eines Schlüssel unter der Auslastung α. Wir nehmen dabei der Einfachheit halber an, daß wir keine Ordnung auf dem Schlüsselbereich haben, sonst wird die Geschichte mathematisch noch etwas haariger. Wir brauchen $O(1)$ für die Auswertung von h und müssen dann alle Elemente der entsprechenden Kette durcharbeiten[4]. Die Länge dieser Kette ist im Mittel ja gerade gleich α, also

Aussage 6.2 *Erfolgloses Suchen eines Schlüssels bei Verkettung ist von der Laufzeitkomplexität $\Theta(\alpha + 1)$.*

Komplizierter zu analysieren ist das Verhalten bei *erfolgreicher* Suche! Das hängt natürlich davon ab, wieviele Schlüssel man in der Kette am Eintrag `t[h(s)]` abklappern muß, bevor man s findet. Skizze! Schnell!

$$\boxed{\texttt{t}[h(s)]} \longrightarrow \underbrace{k_1 \longrightarrow \boxed{s} \longrightarrow k_2}_{L = k_1 + k_2 + 1 \leq \lceil \alpha \rceil}$$

Also: Eine Menge von k_1 Schlüsseln hängt *vor* s und eine Menge von k_2 Schlüsseln *dahinter*. Die Kette habe eine Länge von L, was im Mittel zwischen $\lfloor \alpha \rfloor$ und $\lceil \alpha \rceil$ liegt. In dieser Situation müssen wir offenbar $k_1 + 1$ Schlüssel angucken bis wir s gefunden haben. Unter der Gleichverteilungsannahme kann dieses k_1 die Werte von 0 bis $L - 1$ mit derselben Wahrscheinlichkeit $1/L$ annehmen. Der Erwartungswert für k_1 ergibt sich dann durch die Formel

$$\frac{1}{L}\sum_{i=1}^{L} i = \frac{1}{L} \cdot \frac{L(L+1)}{2} = \frac{L+1}{2} \leq \frac{\lceil \alpha \rceil + 1}{2} \leq \frac{\alpha + 2}{2} = \frac{\alpha}{2} + 1$$

Das sagt auch nicht sehr viel mehr, als daß wir erwarten können nach spätestens $\alpha/2 + 1$ Schritten unser s gefunden zu haben, was i.w. auch einsichtig ist. Zusammen ergibt sich

Aussage 6.3 *Erfolgreiches Suchen eines Schlüssels bei Verkettung ist von der Laufzeitkomplexität $\Theta(\alpha + 1)$.*

Hier einige experimentell bestimmte Daten (Tabelle 6.1). Es wurden 32-bit Ganzzahlen in eine Tabelle gehasht, wobei drei verschiedene Hash-Funktionen verwendet wurden:

1. D arbeitet mit der einfachen Divisionsmethode.
2. M_1 arbeitet mit der Multiplikationsmethode, aber einem sehr schlechten $c = 4/m$.
3. M_2 arbeitet mit der Multiplikationsmethode, aber einem besseren c.

α	$\alpha+1$	D	M_1	M_2
0.1	1.1	1.16	2.20	1.11
0.2	1.2	1.27	3.54	1.20
0.5	1.5	1.82	6.90	1.51
0.8	1.8	2.23	10.11	1.81
1.0	2.0	2.64	12.76	2.07
1.5	2.5	3.16	19.45	2.61
2.0	3.0	3.89	25.24	3.08
3.0	4.0	5.45	35.27	4.22
4.0	5.0	6.95	48.44	5.16
5.0	6.0	8.52	59.75	6.18
8.0	9.0	13.04	95.48	9.33
10.0	11.0	14.73	120.75	11.61
13.0	14.0	20.60	162.22	14.65

Tabelle 6.1: Suchaufwand bei Kollisionsbehandlung durch Verkettung

Das ist zwar recht ermutigend, aber nur dann aussagekräftig, wenn wir näheres über α wissen. Natürlich kann es sein, daß z.B. $N = \Theta(2^m)$ ist, wir also exponentiell viele Einträge in der Tabelle haben. Dann ist $\alpha = \Theta(2^m/m) = O(2^m) = \Theta(N)$. Unsere Suchoperation wird also langsam linear, also asymptotisch nicht besser als das einfache Suchen in einer verketteten Liste.

Passiert ja nicht! Jeder vernünftige Mensch wird — wenn er die Möglichkeit dazu hat — die Größe m der Hash-Tabelle so wählen, daß $N = \Theta(m)$ ist, also die zu erwartende Anzahl der Schlüssel vorher ganz grob abschätzen. Dann aber hat das Suchen und haben auch die übrigen Operationen eine Laufzeitkomplexität, die man für praktische Zwecke als *konstant* annehmen kann.

6.3.2 Sondieren

Die Idee des „Sondieren" geht dem Kollisionsproblem anders zuleibe. Haben wir beim Verketten alle kollidierenden Schlüssel in eine Kette gepackt, versucht man beim Sondieren, an anderer Stelle der Tabelle einen freien Platz zu finden.

Der Vorteil ist offensichtlich der, daß kein Aufwand für das Durchsuchen von Ketten getrieben werden muß; der Nachteil besteht darin, daß die Anzahl der speicherbaren Schlüssel durch die Anzahl der verfügbaren Adressen m beschränkt ist. Insbesondere kann die Auslastung α nicht über 1 steigen.

Sondieren wird also bei einer Kollision auf dem Wert $h(s)$ verschiedene weitere Adressen ausprobieren[5], um einen Platz zu finden, an dem s untergebracht werden kann. Prinzipiell wird das so aussehen, daß zuerst an der Stelle $h(s)$ probiert und bei Kollision an weiteren Adressen geguckt wird, die um irgendwelche Werte δ_i (natürlich modulo m gerechnet) verschoben sind (s. Abb. 6.2).

[4]Genau das müssen wir bei einer Ordnung auf $\mathcal{S}$ ja *nicht* tun!
[5]Daher der Name „Probing". Im Ernst!

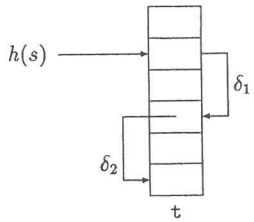


Abbildung 6.2: Kollisionsbehandlung durch Sondieren

Prinzipiell verallgemeinert sich bei diesem Verfahren die verwendete Hash-Funktion zu einer, die neben dem Schlüssel s noch eine laufende Nummer i als Argument nimmt, wobei i die Nummer des Versuchs — beginnend bei 0 — angibt:

$$h_p : \mathcal{S} \times \mathbb{N}_0 \rightarrow \mathcal{A},$$

wobei h_p die eigentliche Hash-Funktion h verwendet.

Hat man diese Idee, so kann man sich eine recht abstrakte Operation für das Einfügen mit Sondieren überlegen. Zunächst überlegt man sich, daß unter jeder Wahl von h_p spätestens dann Schluß mit der Sucherei ist, wenn m Proben genommen worden sind. Damit haben wir auf jeden Fall ein Abbruchkriterium für die Sondierungsschleife.

Das nächste Problem besteht darin, von einer Adresse $h_p(s,i)$ auf die nächste $h_p(s,i+1)$ zu kommen — am besten, ohne allzuviel zu rechnen. Solange wir noch nichts genaues über die Verarbeitung des Versuchszählers wissen, deuten wir die entsprechende Operation im Pseudocode nur durch einen (*) an.

Algorithmus 6.2 *Einfügen des Schlüssels* `key` *in die Hash-Tabelle* `htab` *unter Verwendung der Hash-Funktion* `h`. *Kollisionsbehandlung durch Sondieren.*

Precondition: `key` $\notin$ $\texttt{htab}_{pre}$
Postcondition: $\texttt{htab}_{post}$ = $\texttt{htab}_{pre}$ + `key`

```
procedure probe-insert(key)
  {Erster Versuch!}
  num  := 0
  pos  := h(key,num)

  {Sondierungsschleife}
  while (num < m) and "htab[pos] besetzt" do
     num := num + 1
     pos := pos + "offset"                    (*)

  if "htab[pos] besetzt" then
     {Tabelle ist voll}
  else
     htab[pos] := key
```

Aufgabe: *Entwickeln Sie einen Algorithmus* `probe-lookup`, *der nach dem gegebenen Schnittmuster von Algorithmus 6.2 nach einem Schlüssel sucht.*

Bevor wir uns an verschiedene Strategien des Sondierens heranmachen, muß ein gewichtiger Nachteil dieses Ansatzes explizit erwähnt werden: *Das Löschen aus einer Hash-Tabelle, bei der die Kollisionsbehandlung durch Sondieren erfolgt, ist ausgesprochen umständlich zu realisieren.*

Der Grund liegt natürlich darin, daß man einen Schlüssel s nicht einfach aus der Tabelle herausnehmen kann. Abb. 6.3 zeigt das Problem genauer.

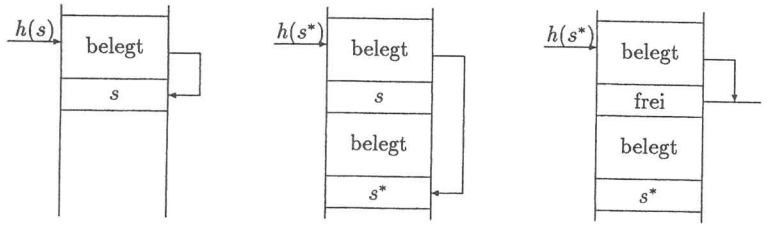


Abbildung 6.3: Das Löschproblem beim Sondieren

Hier wurde der Schlüssel s eingefügt, wobei er durch den belegten Bereich weiter nach unten gefallen ist (linker Teil). Später wurde ein Schlüssel s^* ebenfalls in den belegten Bereich vor s gehasht, fiel aber *an s vorbei* noch weiter nach unten (mittlerer Teil). Wird nun der Eintrag von s einfach entfernt, so entsteht eine Lücke, an der der Suchprozeß nach s^* einfach steckenbleibt (rechter Teil). Die Hash-Tabelle glaubt, daß s^* nicht da ist!

Aufgabe: *Überlegen Sie sich Mittel und Wege, diesem Problem beizukommen. Eine Möglichkeit besteht darin, bei Einfügen und Suchen verschiedene Markierungen für „frei" zu verwenden; eine andere läßt die Lücke dadurch verschwinden, daß weiter unten liegende Schlüssel nach oben geschoben werden (Kompaktifizierung nennt man das).*

Nun aber zu den verschiedenen Ideen, wie man Sondierung betreiben kann.

Lineares Sondieren

Die einfachste Idee besteht natürlich darin, zunächst $h(s)$ auszurechnen und bei einer Kollision einfach in der Tabelle „nach unten" den nächsten freien Platz zu belegen. Das entspricht der Hash-Funktion $h_p(s,i) = (h(s) + i) \bmod m$. Allgemeiner kann man das dadurch verfeinern, daß man eine andere Schrittweite $w \in \mathbb{N}$ als 1 wählt, also zu einer Hash-Funktion $h_p(s,i) = (h(s) + w \cdot i) \bmod m$ übergeht. In Algorithmus 6.2 ergibt sich dann die einfache Ergänzung

```
pos := (pos + w) mod m
```


Aufgabe: *Kann man* jede *Schrittweite w nehmen?*

Der Defekt ist zu sehen! Betrachten wir der Einfachheit halber mal die Schrittweite 1 und eine freie Adresse k. Vor[6] k wird eine gewisse Anzahl d von Plätzen belegt sein, davor wieder ein freier Platz, s. Abb. 6.4.

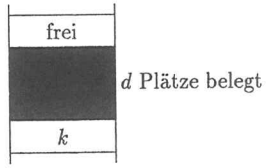


Abbildung 6.4: Zur Clusterbildung beim linearen Sondieren

Nun sieht man die Tücke. Die Wahrscheinlichkeit, daß als nächstes der Platz k belegt wird, hängt nun auch noch von d ab, denn jeder Schlüssel, der in den belegten Bereich gehasht wird, landet sicher auf Platz k. Insgesamt ergibt sich für die Belegung von k eine Wahrscheinlichkeit von $(d+1)/m$. Mit dieser Wahrscheinlichkeit wird sich also der belegte Bereich um mindestens einen vergrößern; noch heftiger wird es, wenn dadurch zwei belegte Bereiche verbunden werden. Der nächste freie Platz hinter k wird dann mit einer noch größeren Wahrscheinlichkeit das nächste Opfer, usw.

In der Tat hat diese Form des Sondierens die unangenehme Eigenschaft, sogenannte „Cluster" (Haufen oder Ballungen) zu bilden, die sich selbst immer schneller vergrößern und natürlich zu immer längeren Probenfolgen führen. Daß das auch bei anderen Schrittweiten passiert, sieht man durch eine ähnliche Überlegung, die Cluster hängen dann allerdings nicht direkt zusammen, sondern verteilen sich mit w.

Quadratisches Sondieren

Hier wird versucht, dem Cluster-Problem dadurch aus dem Weg zu gehen, daß man nicht linear sondern in quadratisch wachsender Schrittweite durch die Tabelle läuft. Die Hash-Funktion hat also die Form $h_p(s,i) = (h(s) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$, wobei $c_1, c_2 \neq 0$. In Algorithmus 6.2 schreiben wir also

```
pos := (pos + c1 + num*num*c2) mod m
```

Das vermeidet zwar die beim linearen Sondieren auftretenden Cluster, führt aber zu nicht geringen Problemen, was die Wahl von c_1 und c_2 angeht. Es muß nämlich sichergestellt werden, daß jede Adresse als Wert $h_p(s,i)$ auftreten kann[7], so daß c_1 und c_2 nicht unabhängig voneinander und von der Tabellengröße m gewählt werden können.

Aufgabe: *Denken Sie sich für $m = 307$ zwei Konstanten c_1 und c_2 aus, mit denen nicht alle m Adressen erreicht werden.*

[6] „Vor" heißt natürlich unter Berücksichtigung des „mod", also nötigenfalls auch danach..
[7] Formal: h_p muß ja (wie bekannt) surjektiv sein.

Dieser Umstand macht quadratisches Sondieren in der Praxis recht unhandlich und es zeigt auch dasselbe prinzipielle Problem wie das lineare Sondieren: *Gilt für zwei verschiedene Schlüssel s_1 und s_2 $h(s_1) = h(s_2)$, so erzeugen sie identische Probenfolgen: $h_p(s_1, i) = h_p(s_2, i)$, für alle i.*

Doppeltes Sondieren

Dieses Problem umgeht das doppelte Sondieren, indem es *zwei* einfache Hash-Funktionen h_1 und h_2 zu h_p kombiniert: $h_p(s,i) = (h_1(s) + i \cdot h_2(s)) \bmod m$. Die entsprechende Modifikation von Algorithmus 6.2:

```
pos := (pos + h2(key)) mod m
```

Natürlich rechnet man `h2(key)` *nur einmal* aus! Da machen wir nicht mal eine Übungsaufgabe draus.

Auch hier ergeben sich einige Probleme, die primär mit h_2 zu tun haben. Schnell stellt man fest, daß h_2 keinen Schlüssel auf 0 abbilden darf. Gemein wird die Geschichte, wenn für einen Schlüssel s der Wert $h_2(s)$ und die Tabellengröße m einen gemeinsamen Teiler $d > 1$ besitzen. Dann ist also $m = n_1/d$ und $h_2(s) = n_2/d$. Im ersten Sondierungsschritt bekommen wir als Adresse $h_p(s,0) = h_1(s) \bmod m$, das ist klar. Unschön ist dann der Wert $h_p(s, n_1)$:

$$\begin{aligned} h_p(s,n_1) &= (h_1(s) + n_1 \cdot h_2(s)) \bmod m \\ &= (h_1(s) + (n_1 \cdot n_2)/d) \bmod m \\ &= (h_1(s) + n_2 \cdot m) \bmod m \\ &= h_1(s) \bmod m \\ &= h_p(s,0) \end{aligned}$$

Also: Im Fall, daß für einen Schlüssel s die Werte $h_2(s)$ und m nicht teilerfremd sind, wird *nicht* die gesamte Tabelle durchsucht! Schlimmstenfalls führt das natürlich dazu, daß ein solcher Schlüssel gar nicht eingefügt werden kann, obwohl noch genug Platz in der Tabelle ist.

Am einfachsten ist es, z.B. für m eine Primzahl zu wählen und in h_2 den Wert 0 „manuell" verhindern. Eine andere Möglichkeit wäre es, m als Zweierpotenz zu wählen und h_2 immer ungerade sein zu lassen.

Die Qualität des doppelten Sondierens besteht natürlich darin, daß zwei Schlüssel, die mit h_1 auf denselben Platz gehasht werden, dann durch h_2 wieder getrennt werden. Der Nachteil ist natürlich darin zu sehen, daß man zwei Hash-Funktionen ausrechnen muß, um sondieren zu können.

Komplexität von Sondierung

Will man sich dem Problem der Komplexitätsanalyse von Hashing mit Sondierung widmen, steht vor ganz ordentlichen Problemen, die nicht nur die Uniformität von Schlüsselbereich und Hash-Funktion(en), sondern auch noch Eigenschaften des speziellen Sondierungsverfahrens einbeziehen muß. Hier also nur die wichtigsten Überlegungen und Ergebnisse im Telegrammstil.

Zunächst gehen wir also mal davon aus, daß das Sondierungsverfahren keine der angesprochenen groben Fehler enthält, d.h. eine Probenfolge $\langle h_p(s,0), h_p(s,1), \ldots, h_p(s,m-1)\rangle$ ist nichts weiter als eine Permutation der Adressfolge $\langle 0, 1, \ldots, m-1\rangle$. Entsprechend den bekannten Definitionen nehmen wir an, daß alle Probenfolgen gleichwahrscheinlich ($p = 1/m$) sind. Noch einigermaßen einfach zu sehen ist

Aussage 6.4 *Unter den angegebenen Annahmen über h_p und bei gegebener Tabellenauslastung $\alpha < 1$ müssen für das Einfügen wie auch das erfolglose Suchen im Mittel $E_M = \frac{1}{1-\alpha}$ Proben genommen werden.*

Beweisidee (Genaue Beweise finden sich in [Cormen *et al.*, 1990] und in [Mehlhorn, 1988]): Eine Probe auf $h_p(s,0)$ müssen wir auf jeden Fall nehmen. Mit der Wahrscheinlichkeit α ist dieser Platz belegt, mit der Wahrscheinlichkeit α^2 auch der nächste, mit der Wahrscheinlichkeit α^3 auch der übernächste usw. Bildet man nun den Erwartungswert, so benötigt man an Proben

$$1 + \alpha + \alpha^2 + \cdots + \alpha^m < \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

Aufgabe: *Warum benötigt das Einfügen genausoviele Proben wie das erfolglose Suchen?*

Schwieriger zu zeigen ist (s. [Cormen *et al.*, 1990], [Mehlhorn, 1988]):

Aussage 6.5 *Unter denselben Annahmen wie in 6.4 benötigt man für eine erfolgreiche Suche im Mittel $S_M = \frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}$ Proben.*

In Tabelle 6.2 sind wieder einige experimentelle Daten zusammengestellt. Es bedeuten

- E_{LS} Einfügen mit linearer Sondierung.
- E_{QS} Einfügen mit quadratischer Sondierung.
- E_{DS} Einfügen mit doppelter Sondierung.
- S_{LS} Erfolgreiches Suchen mit linearer Sondierung.
- S_{QS} Erfolgreiches Suchen mit quadratischer Sondierung.
- S_{DS} Erfolgreiches Suchen mit doppelter Sondierung.

α	E_{LS}	E_{QS}	E_{DS}	E_M	S_{LS}	S_{QS}	S_{DS}	S_M
0.1	4.75	5.06	1.090	1.111	3.00	3.31	1.13	1.054
0.2	8.90	9.74	1.210	1.250	4.38	4.76	1.10	1.116
0.3	12.56	15.24	1.580	1.429	7.38	7.93	1.20	1.189
0.4	16.53	19.31	1.580	1.667	6.92	10.37	1.16	1.277
0.5	20.07	27.55	2.030	2.000	11.13	12.58	1.25	1.386
0.6	25.03	33.35	2.550	2.500	12.34	16.26	1.44	1.527
0.7	30.85	42.54	3.300	3.333	14.68	19.71	1.82	1.720
0.8	43.56	59.13	4.500	5.001	17.27	21.97	2.37	2.012
0.9	86.83	85.01	11.800	10.006	22.95	25.70	2.81	2.559
0.95	312.39	119.89	19.300	20.011	32.96	24.49	2.71	3.154
0.99	3252.25	278.32	91.350	100.382	55.08	35.83	5.69	4.655
0.999	4581.94	1390.79	1040.650	1028.917	23.50	36.99	4.90	6.943

Tabelle 6.2: Aufwand bei Kollisionsbehandlung durch Sondieren

6.4 Schlußbemerkungen

Wir haben gesehen, daß Hashing ein zwar mächtiges aber nicht ganz triviales Mittel ist, große Schlüsselmengen auf kleine Speicher abzubilden. Zwar ist jede der beiden Hauptmethoden (Verkettung und Sondierung) an sich sehr leicht zu implementieren, aber im Detail steckt doch so manche Teufelei. In praktischer Hinsicht ist Verkettung aus zwei Gründen — meiner Meinung nach — dem Sondieren vorzuziehen:

1. Es erlaubt Auslastungen $\alpha > 1$, was beim Sondieren nicht möglich ist.
2. Es unterstützt das Löschen, was beim Sondieren recht kompliziert ist.
3. Es wird einen gegebenen Schlüssel *in jedem Fall* abspeichern, was beim Sondieren bei grob falscher Wahl der Sondierungsstrategie nicht sichergestellt ist.

Sondieren hat Vorteile in Fällen, in denen die Tabelle ausreichend groß ist, um die anfallenden Schlüssel zu halten, und wenn Löschen nicht erforderlich ist:

1. Die Zugriffsoperationen sind wesentlich effizienter als bei der Verkettung.
2. Bei gegebener Sondierungsstrategie sind die Algorithmen einfacher zu implementieren. Es muß nicht mit Zeigern hantiert werden, was immer ein gewisses Risiko beinhaltet. Insbesondere ist leichter festzustellen, wann die Speichermöglichkeiten erschöpft sind ($N = m$) als beim Verketten (kein dynamischer Speicher mehr frei).

Kapitel 7

Suchen in Baumstrukturen

Grundlegende Begriffe im Bereich von Bäumen — insbesondere von Binärbäumen — sind in 3.2.2 bereits dargestellt worden. Bäume eignen sich gut zur Speicherung von Daten, die durch einen Schlüssel identifiziert werden und damit als Implementation des ADT „Dictionary“ (s. 1.4.6). Den ersten Ansatz dafür bieten die *sortierten Binärbäume*, eine Spezialvariante geordneter Binärbäume,

7.1 Sortierte Binärbäume

Wir definieren uns zunächst einmal einen Binärbaum als ADT. Im Überblick sieht er so aus:

Binärbaum	

Schnittstelle:	`left`
	`right`
	`parent`
	`key`
	`bintree-insert`
	`bintree-delete`

Invariante:	left(x) $\neq$ right(x) left(x) $\neq$ parent(x) right(x) $\neq$ parent(x)

Abbildung 7.1: Der ADT Binärbaum

Schnittstelle:

1. `left(x)`
   Precondition: `x <> nil`
   Postcondition: `---`
   Liefert den linken Nachfolger des Knotens `x`. Falls der nicht existiert, so ist der Wert von `left` die Konstante `nil`.

2. `right(x)`
   Precondition: `x <> nil`
   Postcondition: `---`
   Liefert den rechten Nachfolger des Knotens `x`. Falls der nicht existiert, so ist der Wert von `left` die Konstante `nil`.

3. `parent(x)`
   Precondition: `x <> nil`
   Postcondition: `---`
   Liefert den unmittelbaren Vorgänger von `x` bzw. `nil`, falls `x` die Wurzel des Baums ist.

4. `key(x)`
   Precondition: `x <> nil`
   Postcondition: `---`
   Liefert den Schlüssel des Knotens `x`.

5. `bintree-insert(T,k)`
   Precondition: `---`
   Postcondition: `Baum enthaelt Knoten k`
   Der Knoten `k` wird in den Baum `T` eingefügt.

6. `bintree-delete(T,k)`
   Precondition: `k <> nil`
   Postcondition: `Baum enthaelt Knoten k nicht mehr`
   Entfernt den Knoten `k` aus dem Baum `T`.

Zur weiteren Abkürzung nehmen wir sogar an, daß diese Ausdrücke auch *auf der linken Seite* einer Zuweisung stehen dürfen. So soll eine Zuweisung

```
left(x) := y
```

bedeuten, daß der linke Nachfolger des Kontens `x` mit dem (Verweis auf den) Knoten `y` besetzt wird.

Für *sortierte Binärbäume* nehmen wir an, daß auf den Schlüsseln eine *vollständige Ordnung* definiert ist, d.h. zwei Schlüssel sind bezüglich der Eigenschaften „<“ und „=“ grundsätzlich vergleichbar. Außerdem legen wir ein für allemal fest, daß wir nur sortierte Binärbäume angucken wollen, in denen *alle Schlüssel verschieden sind*. Ein sortierter Binärbaum ist dann folgendermaßen definiert:

Definition 7.1 *Ein geordneter Binärbaum heißt* **sortierter Binärbaum** *(binärer Suchbaum), wenn für jeden Knoten* `k` *die folgenden beiden Bedingungen gelten:*

1. *Im* linken *Unterbaum von* `k` *sind alle Schlüssel* kleiner *als der Schlüssel von* `k`.
2. *Im* rechten *Unterbaum von* `k` *sind alle Schlüssel* größer *als der Schlüssel von* `k`.

Da sich diese Eigenschaft in der Schnittstelle des ADT nicht bemerkbar macht, ist lediglich die Invariante betroffen:

Sortierter Binärbaum	

Schnittstelle:	`left`
	`right`
	`parent`
	`key`
	`bintree-insert`
	`bintree-delete`

Invariante:	left(x) $\neq$ right(x) left(x) $\neq$ parent(x) right(x) $\neq$ parent(x) Def. 7.1

Abbildung 7.2: Der ADT Sortierter Binärbaum

7.1.1 Elementare Operationen auf sortierten Binärbäumen

Damit kommen wir sehr schnell zu einfachen Verfahren zum Suchen eines Elements und zum Einfügen:

Algorithmus 7.1 *Suchen eines Knotens mit dem Schlüssel* `k` *in einem sortierten Binärbaum unterhalb und einschließlich des Knotens* `n`. *Ergebnis ist (Zeiger auf den) Knoten mit dem Schlüssel* `k` *oder* `nil`, *falls kein solcher Knoten existiert.*

Precondition: `---`
Postcondition: `---`

```
function search-bin-tree(n,k)
  if n = nil then
     return nil
  else
     if key(n) = k then
        return n
     else
        if key(n) > k then
           return search-bin-tree(left(n),k)
        else
           return search-bin-tree(right(n),k)
```

Man überzeugt sich, daß die Anzahl der besuchten Knoten durch die Höhe h des Baums nach oben begrenzt wird. Das ist gerade der Sinn dieser ganzen Konstruktion. Formal heißt das, daß Suchen in einem sortierten Binärbaum von der Kompexität $O(h)$ ist.

Ganz schnell gehen einige triviale Operationen auf solchen sortierten Binärbäumen:

Algorithmus 7.2 *Suchen des Knotens mit dem kleinsten bzw. größten Schlüssel in einem sortierten Binärbaum unterhalb und einschließlich des Knotens* `n`.

Precondition: `n <> nil`
Postcondition: `---`

```
function tree-min(n)  {Minimum eines sortierten Binaerbaums}
  while left(n) <> nil do
     n := left(n)
  return n

function tree-max(n)  {Maximum eines sortierten Binaerbaums}
  while right(n) <> nil do
     n := right(n)
  return n
```

Auch für diese beiden Verfahren finden wir die Komplexität $O(h)$.

Ein etwas anderes Problem stellt das Einfügen eines Knotens in einen sortierten Binärbaum dar. Die Sortierungseigenschaft ist ja eine *Invariante* des ADT, d.h. wir können davon ausgehen, daß der Baum *vor* dem Einfügen sortiert ist und müssen nur sicherstellen, daß er *nach* dieser Operation immer noch (oder schon wieder) sortiert ist.

Algorithmus 7.3 *Einfügen eines Knotens* `k` *in einen sortierten Binärbaum unterhalb und einschließlich des Knotens* `n` *mit Verkettung auf den Vorgänger* `p`.

Precondition: `---`
Postcondition: $\texttt{Baum}_{post} = \texttt{Baum}_{pre} + \texttt{k}$

```
procedure bintree-insert(var n; k,p)
  if n = nil then        {Leerer Baum}
     n          := k
     left(k)    := nil
     right(k)   := nil
     parent(k)  := p
  else
     if key(k) < key(n) then
        bintree-insert(left(n),k,n)
     else
        bintree-insert(right(n),k,n)
```

Man muß diese Prozedur mit dem p-Argument auf `nil` aufrufen, es dient nur dazu, beim rekursiven herunterklettern den Vorgängerknoten mitzuschleppen. Auch hier sieht man an der rekursiven Formulierung, daß die Anzahl der besuchten Knoten durch h nach oben begrenzt wird, also haben wir auch hier die Komplexität $O(h)$.

Die eigentlich ziemlich unschönen Eigenschaften solcher sortierten Binärbäume werden durch Abb. 7.3 (links) illustriert. Mit Algorithmus 7.3 wurden der Reihe nach folgende neun Werte eingetragen:

$$12, 1, 3, 5, 2, 4, 7, 6, 0.$$

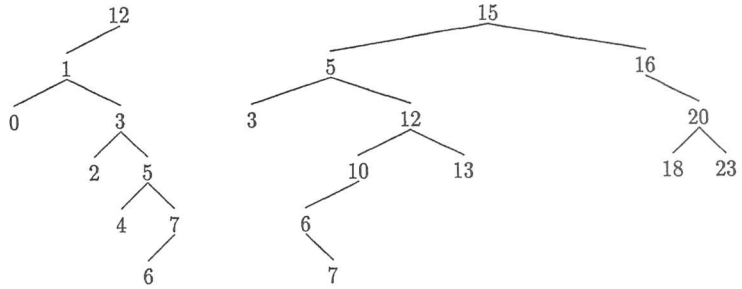


Abbildung 7.3: Ein schlecht und ein besser balancierter Binärbaum

Das Ergebnis deutet bereits an, wo später das Problem beim Suchen liegen wird: Bestimmte Werte, wie etwa die 6, sind nur mit vergleichsweise hohem Aufwand zu finden. Der unschöne Effekt besteht offenbar darin, daß für diesen Baum die Höhe h unverantwortlich groß ist. Sehr ungünstig wird die gesamte Angelegenheit, wenn man z.B. auf die Idee verfallen sollte, *auf- oder absteigend sortierte* Werte in einen solchen Binärbaum zu füttern. Man erkennt dann nämlich, daß h nicht logarithmisch mit der Anzahl der Knoten läuft, sondern linear.

Aufgabe: *Zeichnen Sie den sortierten Binärbaum auf, der nach dem Einfügen der fünf Werte* $1, 2, 3, 4$ *und* 5 *entsteht.*

7.1.2 Der Ordnungs-Nachfolger

Nächstes Problem! Betrachten wir einmal den etwas besser balancierten Baum aus Abb. 7.3, rechts.

Wir wollen uns an diesem Beispiel über einige ganz interessante Eigenschaften von sortierten Binärbäumen klar werden. In erster Linie geht es uns dabei um die Frage, wo man im Baum den *unmittelbaren Nachfolger* eines gegebenen Knotens *in der Reihenfolge der Elementsortierung* finden kann. Wir nennen diesen Knoten kurz den *Ordnungs-Nachfolger* (**ON**), um ihn von den Nachfolgern im Baum zu unterscheiden.

Definition 7.2 *Der* **Ordnungs-Nachfolger** *(ON) eines Knotens* k *ist derjenige Knoten mit der Eigenschaft, daß er den kleinsten Schlüssel im Baum enthält, der größer als der Schlüssel von* k *ist.*

Das Problem, den ON zu finden, hört sich zunächst ganz einfach an, ist algorithmisch aber nicht so ganz ohne. Zunächst gucken wir uns mal den Knoten mit der Nummer 5 an. Der unmittelbare Nachfolger trägt die Nummer 6. Kann man einen Algorithmus angeben, der zu einem gegebenen Knoten diesen Nachfolger findet? Man kann, aber ganz trivial ist es nicht. Das Problem ist nämlich, daß der ON nicht unbedingt *unterhalb* des angegebenen Knotens liegen muß, wie man erkennt, wenn man mal den ON der 13 sucht. Das ist nämlich die 15, also der Wurzelknoten.

Aufgabe: *Suchen Sie die ON der Knoten 10 und 18 und legen Sie ein Kriterium fest,* wann *man den ON eines Knotens* weiter oben *im Baum findet. Lesen Sie bitte erst weiter, wenn Ihnen eine brauchbare Idee gekommen ist.*

Richtig! Es hängt damit zusammen, ob der entsprechende Knoten einen *rechten* Unterbaum besitzt. Wenn er einen hat, findet sich der ON dort und zwar als kleinstes Element (dafür haben wir schon einen Algorithmus; `tree-min` heißt er). Und wenn der Knoten keinen rechten Unterbaum hat?

Betrachten wir dazu einmal aufwärtsgerichtete Knotenfolgen mit der Eigenschaft, daß man immer nur *von links unten* in den Vorgänger eintritt, z.B. 6 — 10 — 12 oder 3 — 5 — 15. Schon wieder richtig: Die Schlüssel wachsen streng monoton an (weil wir ja keine doppelten Schlüssel im Baum haben).

Bei Knotenfolgen, bei denen wir immer *von rechts unten* kommen (wie z.B. 13 — 12 — 5), funktioniert das Prinzip genau anders herum. Die Schlüssel fallen streng monoton. Daraus können wir nun den Weg bestimmen, den wir nach oben nehmen müssen, um den ON zu finden.

Nehmen wir also an, wir suchten den ON des Knotens p_0. Wir beginnen damit, daß wir uns dem Vorgänger von p_0 zuwenden. Besitzt der einen kleineren Schlüssel als p_0, so laufen wir einfach weiter nach oben. Nehmen wir mal an, wir würden so über die Knoten $p_1, p_2, \ldots, p_{k-1}$ hinweglaufen. Wenn wir auf diesem Weg nun zum ersten Mal einen Knoten o_k mit *größerem* Schlüssel finden, gelten nun folgende beiden Tatsachen:

1. Der gesuchte ON kann *nicht* im rechten Unterbaum von p_k liegen, denn dort sind die Schlüssel ja noch größer.
2. Die Schlüssel in den Knoten $p_1, p_2, \ldots, p_{k-1}$ sind alle kleiner als der von p_0 und damit natürlich auch die in deren linken Unterbäumen, die wir uns gar nicht angeguckt haben. Da kann der gesuchte ON also nicht gelegen haben.

Damit bleibt nur ein Schluß übrig: p_k ist der Knoten, der den gesuchten ON enthält. Damit haben wir allerdings so ziemlich alles zusammen, was wir für den entsprechenden Algorithmus brauchen.

Algorithmus 7.4 *Aufsuchen des Ordnungs-Nachfolgers zu einem Knoten* n.

Precondition: `n <> nil`
Postcondition: `---`

```
function tree-succ(n)
  if right(n) <> nil then
     return tree-min(right(n))
  else
     pred := parent(n)                {Start des Hochkletterns}
     while pred <> nil and key(pred) <= key(n) do
        pred := parent(pred)

     return pred
```


Dieser Algorithmus arbeitet in zwei Alternativen. Der einfachste Fall tritt auf, wenn der Knoten n einen rechten Unterbaum hat. Dann suchen wir uns einfach den Knoten mit dem kleinsten Schlüssel aus diesem Teilbaum raus; das muß dann der unmittelbare Nachfolger sein. Ansonsten beginnen wir, den Baum nach oben zu klettern, bis wir entweder oben über die Wurzel herausgeschossen sind oder wir einen Knoten mit größerem Schlüssel gefunden haben.

Aufgabe: *Offenbar kann die Funktion* `tree-succ` *als Ergebnis auch* `nil` *liefern, wie man erkennt, wenn man die Nachbedingung der Schleife betrachtet. Was bedeutet dies für den Eingangsknoten* n*?*

Die Komplexitätsanalyse von Algorithmus 7.4 ist auch einfach. Im ersten Fall wird lediglich `tree-min` aufgerufen, das bekannterweise die Komplexität $O(h)$ hat. Das Heraufklettern im anderen Teil arbeitet sich auf einem Ast nach oben und geht niemals wieder nach unten. Auch in diesem Fall stellt h eine obere Schranke für die Anzahl der besuchten Knoten dar, also ist die gesamte Komplexität $O(h)$.

Eine interessante Eigenschaft des ON brauchen wir noch weiter unten.

Aussage 7.1 *Wenn in einem sortierten Binärbaum ein Knoten n einen rechten Unterbaum besitzt, so besitzt sein ON s höchstens einen rechten Unterbaum — unter Umständen ist s ein Blatt.*

Beweis: Da n einen rechten Unterbaum besitzt, muß sich s dort befinden. Hätte s nun einen linken Sohn l, so wäre dessen Schlüssel kleiner als der von s (wegen der Sortierungseigenschaft) aber größer als der von n (weil s und damit auch l im rechten Unterbaum von n liegen). Damit hätte man aber

$$\text{key}(n) < \text{key}(l) < \text{key}(s)$$

und das kann nicht sein, weil s der ON von n sein soll.

Nun gucken wir uns ein grundlegendes Problem an, das bei der dynamischen Verwaltung von Datenstrukturen andauernd auftritt: *Das Löschen eines Knotens.*

7.1.3 Löschen aus einem sortierten Binärbaum

Das Löschen ist deshalb nicht so einfach, weil man

1. einen Knoten nicht so einfach aus dem Baum herausnehmen kann, wenn man sich nicht darum kümmert, was aus seinen eventuellen Nachfolgern wird und
2. die Sortierungseigenschaft (Entwurfsinvariante!) beibehalten muß.

In der Tat gilt es, *drei* mögliche Fälle zu beachten:

1. Der zu entfernende Knoten n hat keinen Nachfolger (er ist also ein Blatt). Das wird durch das bereits angedeutete Löschen des Knotens ohne weitere Operationen erledigt.

2. Der zu entfernende Knoten n hat lediglich einen Nachfolger (links oder rechts). Der Knoten kann dann einfach gelöscht werden und man trägt seinen einzigen Nachfolger in den Vaterknoten von n an die Stelle ein, an der n vorher stand. Man überzeugt sich durch langes Nachdenken davon, daß diese Operation die Sortierungseigenschaft erhält.
3. Der zu entfernende Knoten n hat zwei Nachfolger. Dieser Fall bedarf einiger Analyse.

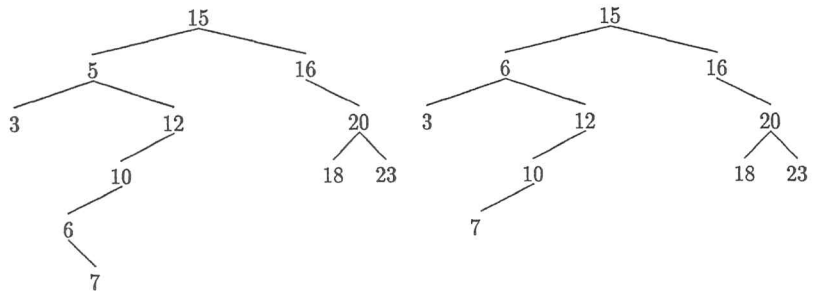


Abbildung 7.4: Löschen aus einem sortierten Binärbaum

Die ersten beiden Fälle sind so einfach, daß wir sie nicht lange betrachten wollen. Versuchen wir aber einmal, den Knoten mit dem Schlüssel 5 aus Abb. 7.4 (links) zu entfernen. Wenn wir uns daran erinnern, daß für jeden Knoten mit einem rechten Unterbaum und somit natürlich auch für jeden Knoten mit *zwei* Unterbäumen der ON höchstens einen (rechten) Unterbaum hat, kommen wir ziemlich schnell auf die richtige Idee.

Würden wir anstelle der 5 im entsprechenden Knoten den Wert des ON, also 6 eintragen, wäre die Sortiereigenschaft nicht verletzt. Den linken Teilbaum *kann* es nicht tangieren, den dort sind sowieso alle Knoten kleiner als 5 und erst recht kleiner als 6. Dem rechten Teilbaum ist das auch egal, denn die 6 ist nach Konstruktion ja das minimale Element davon, d.h. bis auf die 6 selbst sind alle Schlüssel dort größer als 6.

Das entscheidende an dieser Idee ist nun, daß der ON (in diesem Fall also der Knoten mit dem Schlüssel 6) höchstens einen Sohn besitzt, den wir also mit einem der beiden einfachen Verfahren löschen können.

Damit ist das Verfahren aber klar! Man sucht sich mittels `tree-succ` den Nachfolger des zu löschenden Knotens, kopiert sich dessen Schlüssel hinein und löscht dann den ON (u.U. rekursiv), wobei man sich darauf verlassen darf, daß der einfach zu löschen ist. Das Ergebnis sieht man in Abb. 7.4 (rechts).

Es ist bei der endgültigen Codierung des Algorithmus noch auf den Fall zu achten, daß es sich beim zu löschenden Knoten um die Wurzel des Baums handelt. In diesem Fall muß diese Änderung nach außen bekannt gemacht werden. Das müssen wir bei der Definition der Prozedur beachten (die Wurzel muß als `var`-Parameter übergeben werden). Fangen wir also langsam an.

Algorithmus 7.5 *Löschen des Knotens* `k` *aus einem sortierten Binärbaum mit der Wurzel* `root`.

Precondition: `k <> nil`
Postcondition: Baum_{post} = Baum_{pre} - k

```
procedure bintree-delete(var root; k)

  {Zuerst den wirklich zu loeschenden Knoten bestimmen}

  if left(k) = nil or right(k) = nil then   {Nur ein Sohn}
     todelete := k
  else                                       {Zwei Soehne...}
     todelete := tree-succ(k)                {... also der ON}

  {Sohn des Opfers bestimmen, der nach oben gekettet wird}
  if left(todelete) <> nil then
     son := left(todelete)
  else
     son := right(todelete)

  parent := parent(todelete)

  {Es kann son = nil sein, dann ist todelete ein Blatt}
  if son <> nil then        {Wenn nicht, nach oben ketten}
     parent(son) := parent

  {parent = nil, dann ist (war) todelete die Wurzel}

  if parent = nil then
     root := son
  else                          {Eintragen in den neuen Vater}
     if todelete = left(parent) then
        left(parent) := son     {Links eintragen}
     else
        right(parent) := son    {Rechts eintragen}

  {todelete ist nun ausgekettet. Falls es nicht der wirklich
   zu loeschende Knoten war, muss sein Inhalt kopiert werden}

  if todelete <> k then         {Es war Fall 3!}
     key(k) := key(todelete)
     {U.U. noch andere Felder des Knotens kopieren!}
```

Hier kommt zu einer recht bemerkenswerten Komplexitätsanalyse. Man erkennt, daß die eigentliche Arbeit, also das Umhängen von Verweisen, keinerlei Traversierungsaufwand erfordert, also von der Komplexität $O(1)$ ist. Leider kommt im schlechtesten Fall (k hat zwei Nachfolger) der Aufwand für die Suche nach dem ON von k dazu und das ist $O(h)$. Damit haben wir für das Löschen eine obere Schranke, die durch die obere Schranke von `tree-succ` gegeben ist, insgesamt (leider) die Komplexität $O(h)$.

Insgesamt stellen wir also fest, daß alle Operationen auf sortierten Binärbäumen von der Komplexität $O(h)$ sind, wobei h die Höhe des gegebenen Baums ist. Nun wollen wir dies gar nicht unbedingt wissen. Interessiert sind wir an der Komplexität der Algorithmen in Abhängigkeit von der *Knotenanzahl* N. Gut wäre eine Höhe von $\Theta(\log N)$, aber das ist ja nicht garantiert, wie wir wissen.

Das erreicht man zweckmäßigerweise dadurch, daß man *höhen-balancierte* Bäume betrachtet, bei denen also die Höhe logarithmisch mit der Anzahl der Knoten wächst und nicht linear (vergl. 3.2.2). Was wir suchen, ist also das beste aller möglichen Welten: Sortierte höhen-balancierte Binärbäume. Was man erwarten kann, ist dabei, daß der algorithmische Aufwand für das Einfügen und Löschen von Knoten noch ein wenig höher wird. Solche Bäume werden wir nun betrachten.

7.2 Rot/Schwarz Bäume

Zu den bisher bekannten sortierten Binärbäumen bauen wir uns nun eine Erweiterung auf, die im Hinblick auf ihre Balance wesentlich angenehmere Eigenschaften hat als die Originalversionen. Diese Konstruktionen heißen im englischen Sprachraum anschaulich *Red/Black trees*, also Rot/Schwarz Bäume [Bayer, 1972]. Wie der Name schon andeutet, fangen wir damit an, die Knoten des Binärbaums farbig zu markieren. Einige Knoten werden nun schwarz angemalt, andere eben rot.

7.2.1 Definition und elementare Eigenschaften

Die neue Idee bei Rot/Schwarz Bäumen die die eines *externen* Knotens. Ein externer Knoten steht für einen `nil`-Zeiger innerhalb des Baums, d.h. in der Praxis, daß die Söhne von Blättern externe Knoten sind und ebenso fehlende Söhne von Knoten. Wir nennen solche hinzugedichteten Knoten *externe Blätter* im Gegensatz zu den (normalen) *internen Blättern*.

Definition 7.3 *Ein* **Rot/Schwarz-Baum** *(RS-Baum, engl. Red/Black tree, RB tree) ist ein sortierter Binärbaum mit folgenden Eigenschaften:*

1. *Jeder Knoten ist entweder* rot *oder* schwarz.
2. *Jedes externe Blatt ist* schwarz.
3. *Wenn ein Knoten* rot *ist, so sind seine beiden Söhne* schwarz.
4. *Alle Pfade von einem Knoten zu den darunterliegenden Blättern enthalten dieselbe Anzahl* schwarzer *Knoten.*

Der ADT sieht fast so aus, wie der eines sortierten Binärbaums, abgesehen von einigen Umbenennungen, der neuen Invariante und einer neuen Schnittstellenfunktion und ist in Abb. 7.5 dargestellt.

Die neue Schnittstellenfunktion:

Rot/Schwarzbaum	

Schnittstelle:	`left`
	`right`
	`parent`
	`key`
	`color`
	`rb-insert`
	`rb-delete`

Invariante:	left(x) $\neq$ right(x) left(x) $\neq$ parent(x) right(x) $\neq$ parent(x) Def. 7.3 color(root) = black

Abbildung 7.5: Der ADT Rot/Schwarz-Baum

1. `color(x)`
   Precondition: `x <> nil`
   Postcondition: ---
   Liefert den „Farbe" des `x`, also `red` oder `black`.

Zunächst einmal ein Beispiel für einen solchen Baum, s. Abb. 7.6. Die Qualität dieser Datenstruktur erkennt man daran, daß er durch die gleiche Folge von Einfügeoperationen entstanden ist, wie der unbalancierte Binärbaum aus Abb. 7.3, links. Man muß nur wissen, daß ein Knoten mit einem durchgezogenen Kästchen drum herum als *schwarz* gilt, die mit den punktierten Kästchen sind *rot*. Die internen Blätter 0, 2, 4, 6 und 12 tragen die bereits angesprochenen *externen* schwarzen Blätter, die hier nur als leere Kästchen dargestellt sind. Wir machen das einfach zu einer graphischen Konvention.

Eine wichtigste Eigenschaft von RS-Bäumen besteht darin, daß sie immer noch sortiert sind und dementsprechend die Operationen `bin-tree-search`, `tree-min`, `tree-max` und `tree-succ` identisch sind. Die aus der vorigen Abbildung bereits ahnbare Eigenschaft von RS-Bäumen läßt sich nun präzise formulieren (s. [Cormen *et al.*, 1990]):

Aussage 7.2 *Die* Balanceeigenschaft von RS-Bäumen: *Ein RS-Baum mit n internen Knoten hat eine Höhe von höchstens $2 \cdot \log_2(n+1)$. Damit sind die oben angegebenen Operationen alle von der Komplexität $O(\log_2 n)$.*

Man kann natürlich erwarten, daß die Operationen für das Einfügen und das Löschen von Knoten in solchen Bäumen ein wenig aufwendiger sind als es das für normale sortierte Binärbäume der Fall war. Blindes Einfügen, wie oben geschildert, kann dazu führen, daß die Rot-Schwarz Eigenschaft verletzt wird. Man muß sich also nach etwas umsehen, das diese Eigenschaft invariant läßt. Das Stichwort lautet *Rotation*.

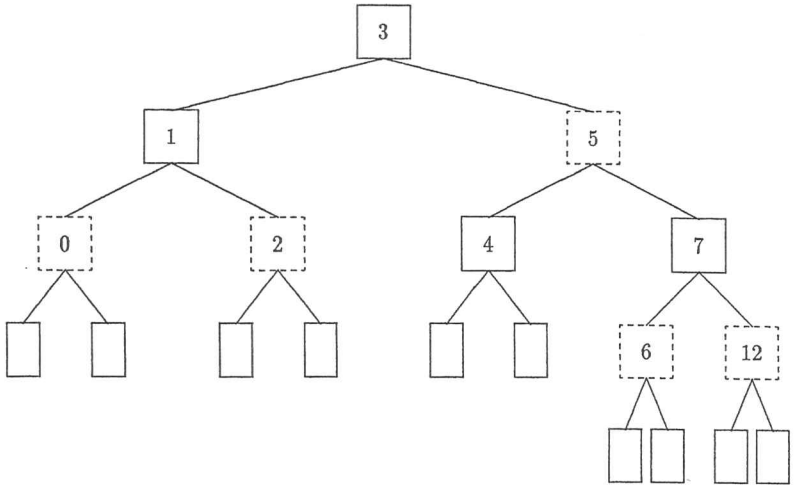


Abbildung 7.6: Ein Rot/Schwarz-Baum

7.2.2 Rotation

Rotation ist eine *lokale* Operation, die Knoten unter Invarianz der Sortierungseigenschaft innerhalb des Baums vertauscht. Wir kennen zwei Arten von Rotation: **Links-** und **Rechtsrotation** (s. Abb. 7.7).

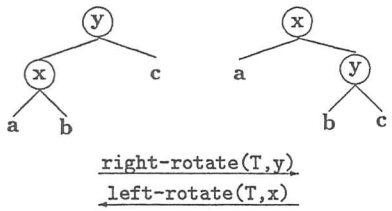


Abbildung 7.7: Die beiden Rotations-Varianten

Man muß sich nun nur noch davon überzeugen, daß beide Arten der Rotation die Eigenschaft der Sortierung des Baums erhalten.

Wie wir wissen, kann man aus einem sortierten Binärbaum die Sortierfolge durch *Inordnungs-Traversierung* ablesen. Wenn wir mal annehmen, daß die durch a, b und c angedeuteten Unterbäume lediglich durch diesen Wurzelknoten dargestellt werden, ergibt sich im linken Fall bei *Inordnungs-Traversierung* (s. 3.2.2):

a x b y c.

Man überzeugt sich schnell, daß das im rechten Fall genauso ist.

Zunächst werden wir den Pseudocode für die Linksrotation konstruieren. Man überlegt sich allerdings schon vorher, daß es sich hierbei um ein einfaches Umhängen einiger Zeiger handeln wird, *dessen Komplexität nicht von der Anzahl der Knoten im Baum abhängt*, es ist also eine lokale Operation mit der Komplexität $O(1)$.

Nehmen wir zunächst die Linksrotation. Halten wir fest, was sowohl *vorher* als auch *nachher* gilt und was geändert werden muß. Wir gehen allerdings dabei davon aus, daß x in der Tat einen rechten Sohn hat, sonst ist die Rotation nicht definiert.

VORHER	NACHHER	ÄNDERN
left(x) = a	left(x) = a	*nein*
right(x) = y	right(x) = b	*ja*
left(y) = b	left(y) = x	*ja*
right(y) = c	right(y) = c	*nein*
parent(y) = x	parent(x) = y	*ja*

Tabelle 7.1: Notwendige Änderungen für eine Linksrotation

Algorithmus 7.6 *Linksrotation des Knotens* x.

Precondition: x <> nil and right(x) <> nil
Postcondition: Tabelle 7.1

```
procedure left-rotate(var root; x)
  {Der Uebersichtlichkeit halber die Akteure besetzen}
  y := right(x)
  a := left(x)
  b := left(y)
  c := right(y)

  {Schritt 1: right(x) = b herstellen}
  right(x) := b
  if b <> nil then
     parent(b) := x

  {Schritt 2: y zum Vater von x machen}
  parent := parent(x)
  parent(y) := parent

  {Gefahr! x war u.U. die Wurzel. Dann ist es nun y!}
  if parent = nil then
     root := y
  else
     {y an den richtigen Anschluss von x' Vater einhaengen}
     if x = left(parent) then
        left(parent) := y
```

```
    else
       right(parent) := y

  {Schritt 3: left(y) = x herstellen}
  left(y)   := x
  parent(x) := y     {Nicht vergessen!}
```

Aufgabe: *Sie haben es sich sicher schon gedacht: Geben Sie Pseudocode für die Operation* `right-rotate(root,y)` *an.*

7.2.3 Einfügen in einen RS-Baum

Nun gehen wir mal ans Einfügen eines Knotens `x`. Wir halten zunächst einige Eigenschaften des bekannten `bintree-insert` (Algorithmus 7.3) fest.

Zunächst bekommt der Knoten `x` die Position eines inneren Blatts, darf also mit *rot* eingefärbt werden (denn seine beiden Nachfolger sind `nil` und damit *schwarz*). Damit sind die Eigenschaften 1 und 2 eines RS-Baums immer noch erfüllt. Interessanterweise gilt das auch für Eigenschaft 4. Denn durch das Einfügen als inneres Blatt ist im entsprechenden Pfad ein schwarzes `nil` weggefallen und dafür `x` als roter Knoten mit jeweils einem schwarzen `nil` dran hinzugekommen, d.h. die Anzahl der schwarzen Knoten auf dem Pfad ist gleichgeblieben.

Aber Eigenschaft 3 ist u.U. verlorengegangen! Es könnte ja sein, daß der Vater von `x` auch rot ist. Das muß durch den Algorithmus repariert werden. Wir geben nun zunächst einmal eine Grobversion des Algorithmus `rb-insert` an, um nicht den Überblick zu verlieren.

```
procedure rb-insert(var root; x)
  {assert: color(root) = black}

  bin-tree-insert(root,x,nil)  {Normal einfuegen}
  color(x) := red

  if color(parent(x)) = red then
     {Roter Knoten mit einem roten Sohn}
     {Repariere die Farbkonfiguration des Baums}
```

Was muß aber nun geschehen, wenn sowohl `x` als auch sein Vater rot sind? Es nützt *nichts*, den Vater einfach schwarz zu färben, wie man im ersten Moment vermuten würde. Das bringt nämlich wiederum unsere Eigenschaft 4 durcheinander, weil auf einmal ein neuer schwarzer Knoten auftaucht. Das kann man aber dadurch kompensieren, daß man auch den *Onkel* von `x` schwarz einfärbt. Der Onkel ist natürlich der andere Sohn des *Großvaters* von `x`. Also sind nun zwei Fälle zu unterscheiden:

Fall 1: Der Onkel von `x` ist rot. Dann kann man ihn und den Vater von `x` schwarz einfärben.

Fall 2: Der Onkel von `x` ist schwarz bzw. `nil` (`nil`s sind ja auch schwarz!). Dann muß man zu härteren Maßnahmen greifen und rotieren!

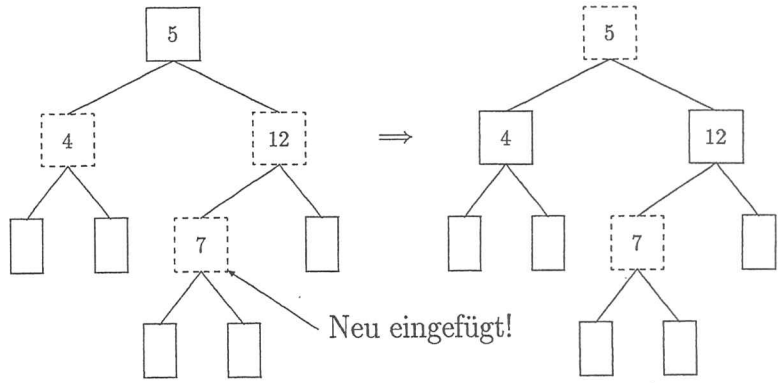


Abbildung 7.8: Einfügen in einen RS-Baum (Fall 1)

Schauen wir uns zunächst wieder unseren roten Onkel an (Abb. 7.8). Hier nehmen wir einmal an, der Knoten 7 sei eben eingefügt worden und stellen entsetzt fest, daß sein Onkel mit der Nummer 4 rot (angelaufen) ist (Abb. 7.8, links).

Außerdem gucken wir uns zunächst nur Fälle an, bei denen der Vater des eingefügten Knotens *linker* Sohn seines Vaters ist. Die andere Möglichkeit wird symmetrisch abgehandelt.

Wir färben den Vater 12 und den Onkel 4 schwarz ein. Nun haben wir leider auf jedem Pfad, der vom Großvater 5 ausgeht, einen schwarzen Knoten mehr bekommen und somit färben wir den Großvater rot ein (Abb. 7.8, rechts).

Das führt dazu, daß wir dasselbe Theater mit dem Großvater bekommen, das wir schon vorher mit `x` hatten — nur zwei Etagen höher. Also arbeiten wir uns in dieser Variante immer in Doppelschritten nach oben und reparieren die Effekte der Umfärbung des Großvaters entsprechend.

Nun der Fall mit dem schwarzen Onkel. Hier gibt's wieder zwei Fälle zu unterscheiden:

Fall 2a: `x` ist linker Sohn seines Vaters. Das ist der einfachere Fall.

Fall 2b: `x` ist rechter Sohn seines Vaters. Das ist etwas komplizierter, wird aber in den Fall **2a** umgewandelt.

Zunächst einmal **Fall 2a:** In Abb. 7.9 sehen wir die Situation, in der wir nach dem Einfügen des Knotens 1 einen roten Vater 3 und einen schwarzen Onkel `nil` vor uns haben.

Um nun zur Lösung rechts zu gelangen, sind drei Schritte vonnöten. Zunächst färben wir den Vater 3 schwarz, womit der Konflikt wegfällt.

Wenn wir nun den Großvater 12 rot einfärben, stimmt wieder die Anzahl der schwarzen Knoten drunter nicht mehr, denn rechts unterhalb der 12 ist dann einer zu wenig. Aber

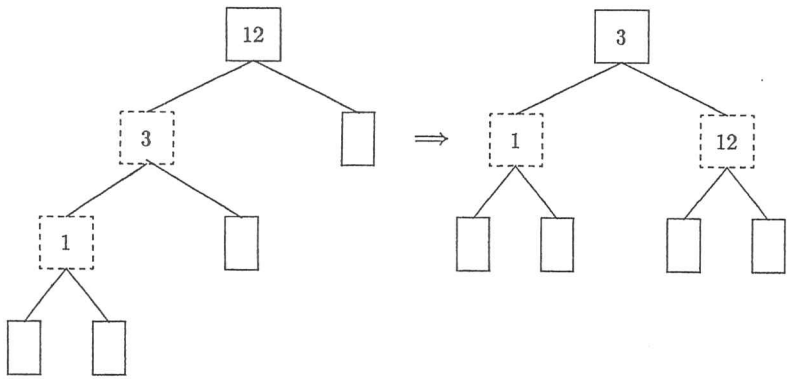


Abbildung 7.9: Einfügen in einen RS-Baum (Fall **2a**)

durch eine *Rechtsrotation* um die 12 biegen wir die gesamte Sache so hin, daß am Ende alles wieder stimmt. Nun geht's noch um den Fall **2b**, d.h. **x** ist rechter Sohn seines Vaters.

Ausgehend von der Situation links oben in Abb. 7.10 erledigen wir das Problem in zwei Schritten:

1. Wir rotieren den Baum *links* herum um den Vater des neu eingefügten Knotens 4, also um die 3. Damit kommen wir zum zweiten Bild, das offenbar unter Fall **2a** fällt.
2. Nun agieren wir genauso wie im Fall **2a**, wobei sich **x** und sein Vater durch die Rotation vertauscht haben und bekommen den gewünschten Endzustand.

Faßt man alles zusammen, wird aus der Bedingung in der Grobversion

```
color(parent(x)) = red
```

eine Schleife, mit der wir uns im Fall **1** nach oben hangeln. So kommen wir zur nächsten Verfeinerung:

```
procedure rb-insert(var root;x)

  bintree-insert(root,x,nil)
  color(x) := red

  while x <> root and color(parent(x)) = red do
     p := parent(x)          {p ist Vater von x}
     g := parent(p)          {g ist Grossvater von x}

     if p = left(g) then
        u := right(g)        {u ist Onkel von x}

        if {Fall 1} then     {Fall 1: Roter Onkel}
```

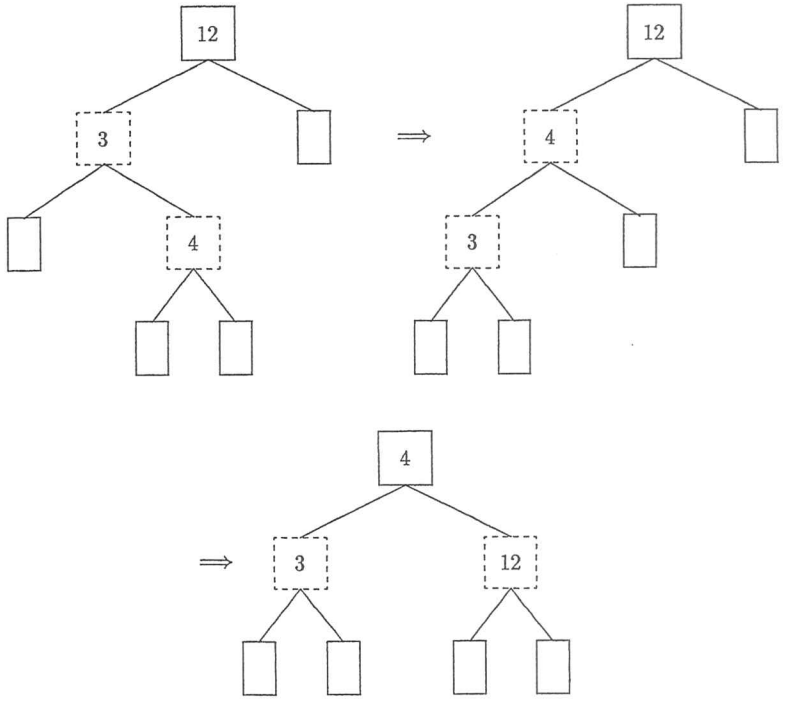



Abbildung 7.10: Einfügen in einen RS-Baum (Fall **2b**)

```
           {Behandle Fall 1 wie beschrieben}
           {x weitersetzen auf Grossvater}
        else                    {Fall 2: Onkel schwarz}
           if {Fall 2b} then
              {Forme 2b nach 2a um}
           {assert: Fall 2a}
           {Behandle Fall 2a wie beschrieben}
           {Aufpassen, dass die Schleife stoppt}
     else
        {Vater ist rechter Sohn des Gro"svaters}
        {also: Das ganze nochmal unter Vertauschung von}
        {left und right, auch bei den Rotationen!}

  color(root) := black      {Wurzel ist immer schwarz}
```

Nun suchen wir uns aus den Beschreibungen der Fälle die entsprechenden Verfeinerungen zusammen. Vergewissern Sie sich bitte anhand der verbalen Beschreibungen, daß die angegebenen Verfeinerungen tatsächlich das gewünschte leisten.

```
{Behandle Fall 1 wie beschrieben}
{x weitersetzen auf Grossvater}
```

$\Longrightarrow$

```
color(p)  := black
color(u)  := black
color(g)  := red
x         := g
```

```
{Forme 2b nach 2a um}
```

$\Longrightarrow$

```
x := p
left-rotate(root,p)
p := parent(x)
g := parent(p)
```

```
{Behandle Fall2a wie beschrieben}
{Aufpassen, dass die Schleife stoppt}
```

$\Longrightarrow$

```
color(p) := black
color(g) := red
right-rotate(root,g)
```

Nun aber können wir den gesamten Algorithmus formulieren:

Algorithmus 7.7 *Einfügen eines Knotens* `x` *in einen RS-Baum mit der Wurzel* `root`.

Precondition: `x <> nil`
Postcondition: Baum_{post} = Baum_{pre} + x

```
procedure rb-insert(var root;x)

  bintree-insert(root,x,nil)
  color(x) := red

  while x <> root and color(parent(x)) = red do
     p := parent(x)                  {p ist Vater von x}
     g := parent(p)                  {g ist Grossvater von x}

     if p = left(g) then
        u := right(g)                {u ist Onkel von x}

        if color(u) = red then       {Fall 1: Roter Onkel}
           color(p) := black         {Vater umfaerben}
           color(u) := black         {Onkel umfaerben}
           color(g) := red           {Grosvater umfaerben}
           x        := g             {2 Stufen hoch weiter}

        else                         {Fall 2: Onkel schwarz}
          if x = right(p) then       {Fall 2b nach Fall 2a}
             x := p
             left-rotate(root,p)
             p := parent(x)
             g := parent(p)
                                     {Fall 2a folgt nun}
          color(p) := black          {Vater umfaerben}
          color(g) := red            {Grossvater dito}
```



```
             right-rotate(root,g)
             {Iteration bricht nun ab, weil}
             {color(parent(x)) = black}
        else
           {Das ganze nochmal unter Vertauschung von}
           {left und right, auch bei den Rotationen}

  color(root) := black
```

Aufgabe: *Formulieren Sie die fortgelassene Variante mit den Vertauschungen von* `left` *und* `right`.

7.2.4 Löschen aus einem RS-Baum

Noch etwas haariger ist das Löschen eines Knotens aus einem solchen Baum. Man sollte sich dabei das Leben so einfach wie nur irgend möglich machen, um einen allzu unübersichtlichen Code zu vermeiden.

Dazu nehmen wir eine technisch einfache Modifikation in der Realisierung von RS-Bäumen vor, die sich mit dem `nil`-Zeiger beschäftigt. Bis jetzt sah es ja so aus, daß ein solches `nil` einen fehlenden Nachfolger eines Knotens markiert hat. Nun modifizieren wir alle `nil`-Einträge derart, daß sie alle auf ein einziges „`nil`" zeigen, das wir als `tree-nil` bezeichnen wollen. Dieses `tree-nil` ist ein Knoten wie jeder andere auch, er hat alle Felder (`left`, `right`, `parent` und `key`), er kann als Vorgänger und als Nachfolger eines Knotens auftreten usw. Der Effekt ist, daß man bei einem ausgeketten Knoten noch einen `parent`-Zeiger definieren kann.

Damit funktioniert das Löschen in einem RS-Baum fast analog zum Löschen in einem normalen sortierten Binärbaum. Das einzige Problem ist dabei der Fall, daß der effektiv gelöschte Knoten (`todelete` hieß er in der Prozedur `bintree-delete`, Algorithmus 7.5) die Farbe *schwarz* hat! Damit würde in der Tat die Eigenschaft 4 verletzt, denn auf dem Pfad, auf dem er lag, fehlt nun ein schwarzer Knoten. Das muß dann explizit durch eine Prozedur `rb-fixcolors` wettgemacht werden.

Zunächst eine kleine Bitte: Schauen Sie sich die Idee hinter dem Löschen eines Knotens in einem normalen sortierten Binärbaum (Algorithmus 7.5) noch einmal an, insbesondere die Idee, einen ganz anderen Knoten zu löschen als den, den wir eigentlich loswerden wollten. Mit einigen kleinen Änderungen ergibt sich der entsprechende Algorithmus für RS-Bäume:

Algorithmus 7.8 *Löschen des Knotens* `k` *aus einem RS-Baum mit der Wurzel* `root`. *Anschließend u.U. Korrektur der Farbverteilung.*

Precondition: `k <> tree-nil`
Postcondition: Baum_{post} = Baum_{pre} - k

```
procedure rb-delete(var root; k)
  if left(k) = tree-nil or right(k) = tree-nil then
     todelete := k
  else
     todelete := tree-succ(k)

  if left(todelete) <> tree-nil then
     son := left(todelete)
  else
     son := right(todelete)

  parent := parent(todelete)

  parent(son) := parent   {ACHTUNG: Aenderung!!}

  if parent = tree-nil then
     root := son
  else
     if todelete = left(parent) then
        left(parent) := son
     else
        right(parent) := son

  if todelete <> k then
     key(k) := key(todelete)
     {U.U. noch andere Felder des Knotens kopieren!}

  if color(todelete) = black then     {Noch eine Aenderung!}
     rb-fixcolors(root,son)           {Farben hindrehen!}
```

Verschaffen wir uns zunächst noch einmal ein Überblick über die Rollen, die die Bezeichner in dieser Prozedur spielen:

Bezeichner	Bedeutung
`todelete`	Effektiv zu löschender Knoten
`son`	*Einziger* Nachfolger von `todelete` bzw. `tree-nil`, falls `todelete` ein Blatt ist
`parent`	Vorgänger von `todelete`

Die Bedeutung von `todelete` ist wohl klar, ebenso die von `parent`. Machen wir uns also klar, was hinter `son` steckt, mit dem wir nun in die Reparaturprozedur klettern. Offenbar handelt es sich um einen Nachfolger des Knotens `todelete`, also desjenigen Knotens, den wir effektiv ausgekettet haben. Andererseits wissen wir ja, daß per Konstruktion dieser Knoten *höchstens einen* Nachfolger hatte, weil wir den ja vergleichsweise einfach löschen konnten. Also: `son` ist der einzige Sohn von `todelete` (falls der einen hatte) oder `tree-nil`, falls `todelete` ein Blatt ist/war.

Die erste Änderung sorgt nun dafür, daß der Vorgänger von `son` nun gleich dem von `todelete` ist, unabhängig davon, ob `son` ein „normaler" Knoten ist oder `tree-nil`. Das ist der Hauptzweck dieser Operation mit dem `tree-nil`.

Nochmal: Wir haben also einen schwarzen Knoten `todelete` gelöscht und auf seinem (ehemaligen) Pfad fehlt nun ein schwarzer Knoten, d.h. Eigenschaft 4 ist verletzt. Die „Idee" ist nun, daß wir einen Moment annehmen, daß sein Nachfolger `son` „ein Schwarz zuviel" hat, also sozusagen „tiefschwarz" ist. Wir werden nun versuchen, dieses tiefschwarz langsam nach oben aus dem Baum rauszutreiben, bis wir entweder einen roten Knoten treffen, den wir schwarz einfärben oder dieses extra schwarz oben aus der Wurzel rausquillt. Wir verabreden, daß der tiefschwarze Knoten immer „`x`" heißen soll.

Die ganze Geschichte wird dann auch wieder mal symmetrisch, je nach dem, ob `x` linker oder rechter Nachfolger seines Vorgängers ist. Genau wie beim Einfügen geht's in diesen beiden Fällen spiegelbildlich zu und wir gucken uns (willkürlich) den Fall an, in dem `x` *linker* Nachfolger eines Vaters ist.

Um die Farben nun richtig hinzudrehen, müssen wir zunächst *zwei* Fälle unterscheiden, die sich alle mit dem *Bruder* von `x` beschäftigen.

1. Dieser Bruder (D) ist *rot*. Wir nehmen eine kleine Transformation vor, die dafür sorgt, daß er *schwarz* wird und gehen damit in den nächsten Fall über.
2. Der Bruder ist *schwarz*. Dann erst fangen wir an, zu überlegen.

Erster Anlauf:

```
procedure rb-fixcolors(root, x)
  brother := {Bruder von x, noetigenfalls tree-nil!}
  if color(brother) = red then
     {Transformiere die Konstruktion so,}
     {dass brother schwarz wird}

  {assert: color(brother) = black}
  {Farben drehen bei schwarzem Bruder}
```

Diese erste Transformation nennen wir mal *Fall 1*. Fall 1 überführt also eine Situation mit einem roten Bruder von `x` in eine mit einem schwarzen Bruder von `x`. Wie das passiert, gucken wir uns gleich an und nehmen nun an, der Bruder sei schwarz! Nun unterscheiden wir *drei* Fälle anhand der Nachfolger dieses Bruders (Neffen bzw. Nichten).

Also die neuen Fälle (unter der Bedingung: `color(brother) = black`). Hier geht es darum, die Fälle zu unterscheiden, in denen `brother` einen roten Nachfolger hat oder nicht. Wenn er keinen hat (Fall 2) verfahren wir wie gleich beschrieben, das ist recht simpel. Wenn er einen hat, sorgen wir dafür, daß dieser rote Nachfolger *rechts* an `brother` hängt und überlegen weiter. Also die Fälle...

Fall 2: Beide Nachfolger von `brother` (C und E) sind *schwarz*. Wir entfernen je ein schwarz von `x` und `brother`. `x` wird „normal schwarz" und `brother` wird rot. Dieses schwarz schieben wir hoch zum Vater `daddy` von `x` und versuchen es weiter, indem wir `x` nach oben schieben! Das führt dazu, daß das ganze in einer Schleife abläuft.

Die Fälle **1** und **2** und ihre Lösung zeigt Abb. 7.11.

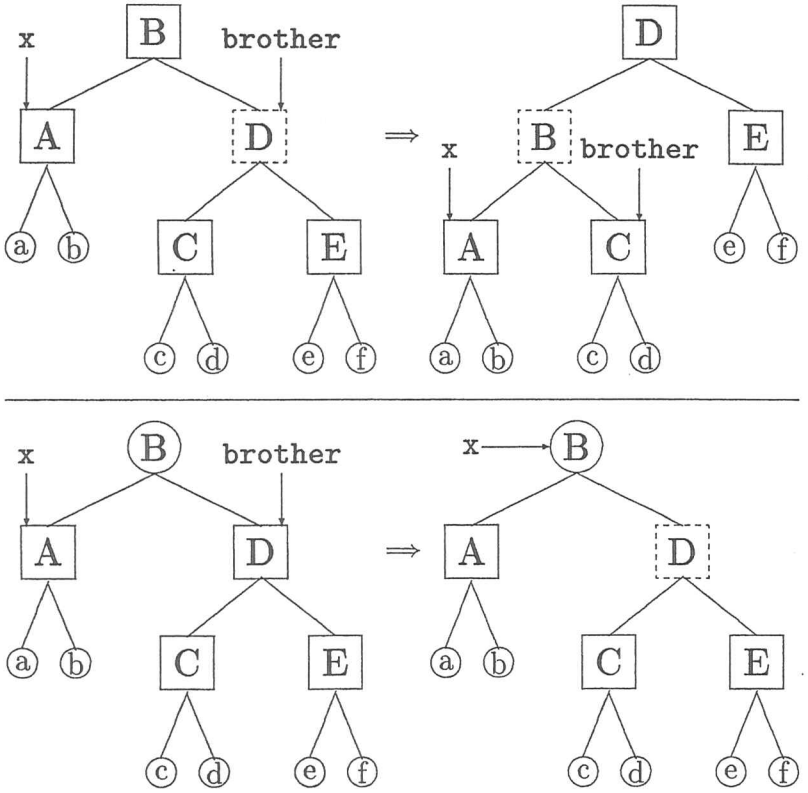


Abbildung 7.11: Löschen aus einem RS-Baum Fall 1 (oben) und Fall 2 (unten)

Fall 3: Der *linke* Nachfolger (C) von `brother` ist rot und der rechte ist schwarz. Das ist falsch und wir rotieren die ganze Geschichte so, daß `brother` einen *roten* rechten Nachfolger hat und vertagen die Geschichte damit bis zum Fall 4. Aus dem „rechten Neffen" E wird dann der rechte rote Neffe D.

Fall 4: Hier ist es nun! Der rechte Nachfolger von `brother` (E) ist rot. Guckt man sich diese Situation im Bild an, so zählt man im linken Ast von B zwei schwarz (beide in `x`) und rechts eins. Eigenschaft 4 ist ja nicht verletzt (s. oben) also stimmt die rot/schwarz Bilanz in diesem Bild. Wir müssen nun dafür sorgen, daß die Knoten in dieser Szene so umgeschaufelt werden, daß im linken Ast *zwei* schwarze Knoten und im rechten Ast immer noch *ein* schwarzer Knoten hängt. In diesem Fall können wir das „extra schwarz" aus `x` rausnehmen und einfach aufhören.

Man sollte sich noch durch Betrachten der Abb. 7.11 und 7.12 davon überzeugen, daß alle Operationen die Sortierungseigenschaft des Baums invariant lassen. Begründung: Man guckt sich die Inordnungs-Traversierungen an, wobei a, b, c, d, e und f irgend-

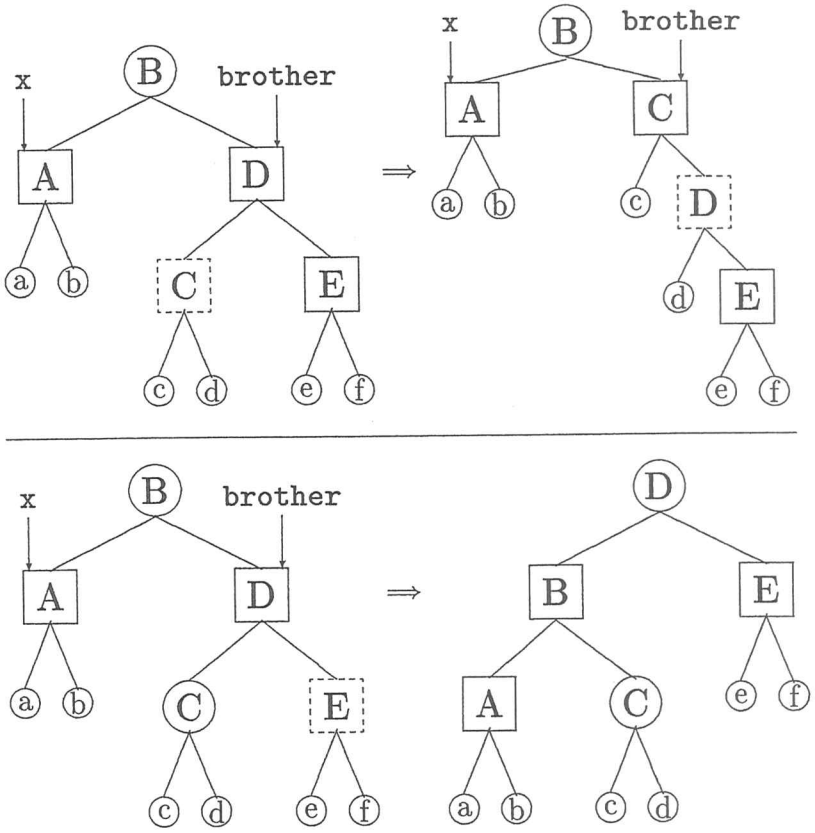


Abbildung 7.12: Löschen aus einem RS-Baum Fall 3 (oben) und Fall 4 (unten)

welche Teilbäume sind, die wir ja nicht mehr anfassen. In allen Fällen ergibt sich dieselbe Inordnungs-Traversierung:

a A b B c C d D e E f.

Also nächster Ansatz:

```
procedure rb-fixcolors(root; x)
  while {noch nicht fertig} do {x ist der tiefschwarze Knoten}
     daddy := parent(x)

     if x = left(daddy) then
        brother := right(daddy)
```

```
if color(brother) = red then
   {Fall 1: Transformiere die Konstruktion so,}
   {        dass brother schwarz wird}

{assert: color(brother) = black}
{Farben drehen bei schwarzem Bruder}

if color(left(brother)) = black and
   color(right(brother)) = black then
   {Fall 2: x nach oben rausschieben}
else
   if color(right(brother)) = black then
      {Fall 3: Nach Fall 4 ueberfuehren}

   {assert: color(right(brother)) = red}
   {Also Fall 4: Umschaufeln, so dass im linken Ast
                 zwei schwarze und im rechten Ast
                 ein schwarzer Knoten haengt}
   {Sorge fuer Abbruch}
```

Nun müssen wir noch die letzten Kleinigkeiten ausfüllen. Zunächst Fall 1, also die Passage, mit der man den Bruder schwarz bekommt:

```
color(brother) := black                {D wird schwarz}
color(daddy)   := red                  {B wird rot}
left-rotate(root,daddy)                {D nach oben, B runter}
daddy          := parent(x)            {Umbenennen nach Rotation}
brother        := right(daddy)         {Nochmal umbenennen}
```

Sauber! Nun der Fall 2, also brother mit zwei schwarzen Nachfolgern (C und E):

```
color(brother) := red                  {D wird rot}
x              := daddy                {x hochschieben}
```

Der nächste Fall (Fall 3, also ein schwarzer rechter Neffe E) ist auch recht einfach:

```
color(left(brother)) := black          {C wird schwarz}
color(brother)       := red            {D wird rot}
right-rotate(root,brother)             {Rumwirbeln!}
brother              := right(parent)  {Das sollte nun C sein}
```

Die allerletzte Verfeinerung behandelt nun den Fall 4. Hier sorgen wir auch noch für den Abbruch, indem wir uns das auf einer booleschen Variable done merken, die wir natürlich am Anfang geeignet besetzen:


```
color(brother)        := color(daddy)  {D bekommt die Farbe von C}
color(daddy)          := black         {B wird schwarz}
color(right(brother)) := black         {E auch}
left-rotate(root,daddy)                {Rumdrehen}
done                  := true          {Abbruch!}
```

So! Nun nur noch die Frage, wann das alles abbricht. Es bricht ab, wenn x die Wurzel ist (dann läuft das extra Schwarz oben raus), wenn x rot ist (dann färben wir es schwarz) oder wenn wir in Fall 4 fertig geworden sind.

Algorithmus 7.9 *Wiederherstellen der RS-Eigenschaft bei einem tiefschwarzen Knoten* x *in einem RS-Baum mit Wurzel* root.

Precondition: ---
Postcondition: Def. 7.3

```
procedure rb-fixcolors(root; x)

  done := false

  {Tiefschwarz nach oben treiben}
  while x <> root and color(x) = black and not done do
     daddy := parent(x)
        if x = left(daddy) then
           brother := right(daddy)

           if color(brother) = red then
              color(brother) := black
              color(daddy)   := red
              left-rotate(root,daddy)
              daddy := parent(x)
              brother := right(daddy)

           if color(left(brother)) = black and
              color(right(brother)) = black then
              color(brother) := red
              x := daddy
           else
              if color(right(brother)) = black then
                 color(left(brother)) := black
                 color(brother)       := red
                 right-rotate(root,brother)
                 brother := right(daddy)

              color(brother)        := color(daddy)
              color(daddy)          := black
              color(right(brother)) := black
              left-rotate(root,daddy)
```

```
            done := true
    else
       {x ist rechter Nachfolger:}
       {alles spiegelbildlich nochmal}

  {Knoten x ist nun einfach schwarz}
  color(x) := black
```

Komplexität müssen wir noch angucken. Offenbar sind alle Zeigeroperationen und die entsprechenden Rotationen alle $O(1)$ und es ist ja lediglich der Fall 2, der uns erneut in die Schleife zwingt. In diesem Fall aber ist sichergestellt, daß wir x aber nach oben im Baum schieben und wieder mal einen echten „Ast-Algorithmus" der Komplexität $O(\log_2 n)$ vor uns haben. Da aber das „normale" Löschen auch $O(\log_2 n)$ kostet, bleibt für das gesamte Löschen in RS-Bäumen dasselbe übrig.

Verlassen wir nun die binären Bäume! Generalisiert man das, was wir bisher gesehen haben, so kommt man schnell auf die Idee, die Anzahl der Nachfolger eines Knotens nicht mehr auf zwei zu beschränken, sondern mehr zuzulassen. Das erfordert einen gewissen Aufwand für die bekannten Operationen, was sich allerdings auszahlt. Die Idee ist einfach: Will man eine gewisse Anzahl n von Knoten in einem binären Baum abspeichern, so bekommt man eine Höhe von $O(\log_2 n)$. Läßt man $m > 2$ Nachfolger eines Knotens zu, so kann man erwarten, daß die Höhe des resultierenden Baums $O(log_m n)$ beträgt. Der Baum wird also flacher, wenn auch nur um einen konstanten Faktor.

7.3 B-Bäume

B-Bäume stellen eine Verallgemeinerung balancierter sortierter Binärbäume dar, bei denen nun noch die Eigenschaft fehlt, *binär* zu sein. Ganz allgemein fallen sie damit unter die Kategorie der *Mehrwegbäume*. Mehrwegbäume sind Bäume vom Grad $m > 2$ ($m = 2$: binäre Bäume). Entsprechend den binären Suchbäumen gibt es somit auch allgemeine *m-Weg Suchbäume*:

Definition 7.4 *Ein* **m-Weg Suchbaum** *für eine gegebene Zahl $m \geq 2$ ist ein Baum mit folgenden Eigenschaften*

1. *Jeder Knoten enthält maximal $m - 1$ Daten.*
2. *Die Daten in einem Knoten sind* aufsteigend *sortiert.*
3. *Ein Knoten hat maximal m Nachfolger.*
4. *Die Daten der linken (rechten) Teilbäume sind kleiner (größer) als das Datum der Wurzel dieses Teilbaums.*
5. *Alle Teilbäume sind wiederum m-Weg-Suchbäume.*

Als Beispiel zunächst ein 4-Weg Baum, dargestellt in Abb. 7.13.

Das wesentliche bei dieser Struktur besteht darin, daß man sich die einzelnen Schlüssel (etwa in der Wurzel) als Wurzeln eines eigenen sortierten Baums vorstellen muß. Der

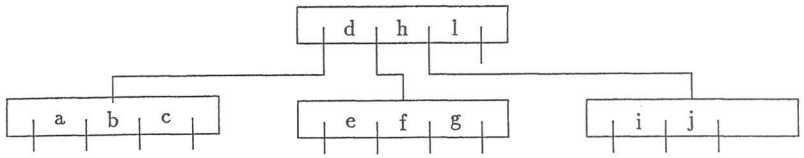


Abbildung 7.13: Ein 4-Weg Baum

Schlüssel „d" im Wurzelknoten besitzt den linken Nachfolger mit den Schlüsseln „a", „b" und „c"; sein rechter Nachfolger enthält die Schlüssel „e", „f" und „g". Die Nachfolgeknoten hängen also „zwischen" zwei Schlüsseln.

Nach allem, was wir über *Balance* von Bäumen gehört haben, ist kaum anzunehmen, daß wir uns sehr lange mit allgemeinen m-Weg Bäumen aufhalten werden. Es geht also darum, einen bestimmten Typ von m-Weg Bäumen einzuführen, der „gute" Balanceeigenschaften aufweist. Entartung ist hier ja genauso möglich wie beim binären Suchbaum.

Das Entarten eines m-Weg Baums verhindert man durch eine Erweiterung der Definition [Bayer und McCreight, 1972]. So kommt man dann zu den B-Bäumen.

7.3.1 Benutzung von B-Bäumen

Bevor wir uns eingehend mit diesem Typ von Bäumen beschäftigen, zunächst einige Bemerkungen darüber, wo sie so häufig angewendet werden. Ihre Hauptrolle spielen sie in der *Datenverwaltung auf Massenspeichern*, also vornehmlich Platten.

Bekannt sind ja sequentielle Dateien. In vielen Anwendungen, etwa im Bereich der Datenbanken, möchte man aber alles mögliche mit Dateien tun, aber *nicht* sequentiell lesen. Einer der Gründe ist, daß man — ähnlich wie beim Array — möglichst schnell über einen Index (oder Schlüssel) auf einen Satz in einer Datei zugreifen möchte. Wir wir alle wissen, geht das bei Arrays in $O(1)$ Zeit. Ganz so schnell bekommen wir es mit Dateien nicht hin, aber besser als in $O(n)$ Schritten werden wir es durch die B-Bäume schon schaffen.

Das abstrakte Modell einer solchen *indiziert zugreifbaren* Datei stellen wir uns so vor, daß wir uns eine Magnetplatte als in *Seiten* (pages) eingeteilt vorstellen. So eine Seite kann z.B. einen Platten-*Sektor* umfassen, an den wir schnell herankommen können, wenn wir seine Nummer wissen. Die *mittlere Zugriffszeit* zu einer Seite (einem Sektor) können wir uns im Bereich von 5 bis 20 msec vorstellen. Sie ist ein Leistungsmerkmal des vorhandenen Laufwerks. Wenn der Lese-/Schreibarm einmal die physikalische Position erreicht hat, auf der die Seite zu erwarten ist, und wenn dann schließlich die entsprechende Information auf ihrer Rundfahrt vorbeigekommen ist, ist der Transport der Daten in den/aus dem Hauptspeicher eher von geringem Aufwand.

Gemein ist also lediglich, daß die Zeit zum Beschaffen einer Seite groß gegen den Aufwand ist, der im Hauptspeicher mit dem Inhalt dieser Seite getrieben wird (darf man wenigstens annehmen). Insbesondere wird man also darauf achten, daß man bei der Komplexitätsbetrachtung die *Anzahl der Plattenzugriffe* und die *interne Rechenzeit* getrennt zu beachten hat.

Man hat im allgemeinen Fall bei solchen Anwendungen das Problem zu lösen, aus einem gegebenen Schlüssel (Name, Personalnummer, Kontonummer, Abteilung,...) eine schnelle Umrechnung in einen Platten-*Index* zu bekommen, mit dem man dann in den angegebenen 20 msec an die entsprechende Seite herankommt.

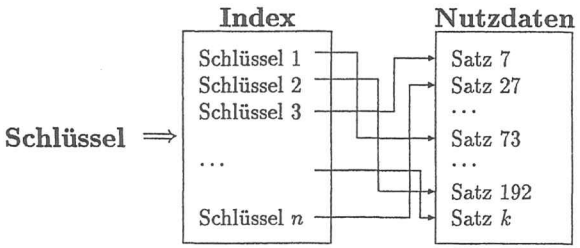


Abbildung 7.14: Dateizugriff mit Schlüsseln über einen Index

Generell sieht unser Zugriffsproblem also so aus wie schematisch in Abb. 7.14 dargestellt: Wir haben irgendeinen Schlüssel, mit dem wir auf unsere „Nutzdaten" zugreifen wollen. Die gesamten Schlüssel sind in einem separaten *Index* organisiert, aus dem wir für jeden Schlüssel, sofern er vorhanden ist, die Adresse des zugehörigen Nutzdatums finden können. Eine Möglichkeit, dies zu tun, besteht einerseits in den bereits bekannten *Hash*-Verfahren (vergl. Kap. 6), die ja genau dies tun. Ein anderes benutzt halt B-Bäume, d.h. man wird den Index zwecks schnellem Zugriff als balancierten m-Weg Baum organisieren.

Ein praktischer Aspekt von B-Bäumen ist die Tatsache, daß sie ein Sonderfall von m-Weg Bäumen darstellen; ein Knoten in einem B-Baum enthält also eine zwar feste, aber beliebig große Maximalzahl von Schlüsseln. In der Praxis werden das *sehr viele* Schlüssel sein. Der Effekt, der durch das m (die Anzahl der Nachfolger eines Knotens) ins Spiel kommt, ist der, daß die Tiefe des Baums vergleichsweise klein ist: Rechnen Sie doch zur Illustration einmal aus, wieviele Schlüssel Sie in einem solchen Baum abspeichern können, wenn in jedem Knoten mindestens 100 gespeichert sind, jeder Knoten, der kein Blatt ist, also wenigstens 100 Nachfolger hat und der Baum nur drei Ebenen tief ist.

Schematisch ergibt sich also folgende Aufgabe für einen solchen Baum: Man wirft einen Schlüssel hinein und als Antwort bekommt man irgendein assoziiertes Datum (z.B. die Adresse des zugehörigen Nutzdatums) wieder heraus (Abb. 7.15).

Zwei Probleme entstehen dann natürlich durch die vergleichsweise großen Knoten, die ja eine ganze Menge Schlüssel gespeichert haben: Zunächst wird der Suchaufwand natürlich eine gewisse Verschlechterung dadurch erfahren, daß wir auch *innerhalb* eines Knotens suchen müssen, weil da ja mehr als ein Schlüssel drin ist. Weiterhin nehmen die Knoten relativ viel Platz weg, so daß wir gar nicht auf die Idee kommen sollten, den gesamten B-Baum im Hauptspeicher zu halten. Wenn er irgendwas in der Kehre von 10^6 Schlüsseln enthält, ist das offenbar völlig hoffnungslos! Also kommt man auf die schlaue Idee, auch den Index in einer Datei, der *Indexdatei*, abzuspeichern.

So kommt dann eine zweistufige Dateiorganisation zusammen: Der Index (realisiert über einen B-Baum) wird knotenweise in einer Indexdatei abgelegt, lediglich der Wurzelknoten

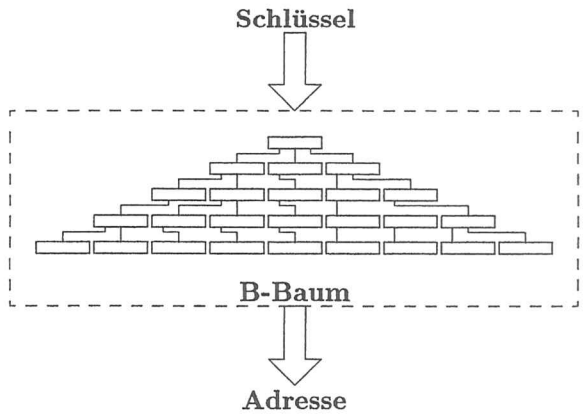


Abbildung 7.15: Indexorganisation mit einem B-Baum

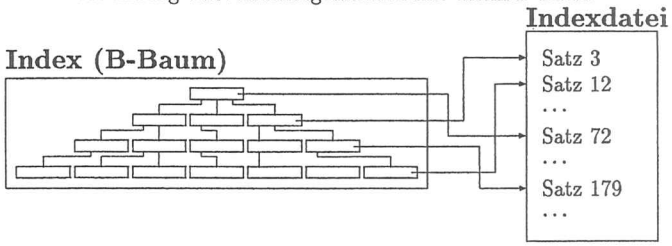


Abbildung 7.16: Speicherung eines B-Baums in einer Indexdatei

des B-Baums liegt resident im Hauptspeicher vor. Das führt dazu, daß man bei jedem Zugriff auf einen anderen Knoten einen Zugriff auf die Indexdatei vornehmen muß und bei jeder Änderung den geänderten Knoten (auch die Wurzel) zurückschreiben muß. Aus diesem Index kommen wir dann mit dem Schlüssel an die eigentlichen Nutzdaten heran.

Nun aber endlich zur zentralen Frage: Was ist denn nun ein B-Baum?

7.3.2 Struktur von B-Bäumen

Hier muß man sich die Anforderungen an die Struktur ganz genau klarmachen, sonst kommt man später bei den dazu gehörenden Algorithmen in Schwierigkeiten. Also zuerst einmal eine

Definition 7.5 *Es sei $t > 0$. Ein* **B-Baum** *der Ordnung t ist entweder leer oder er hat folgende Eigenschaften:*

1. *Jeder Pfad von der Wurzel bis zu einem Blatt hat die gleiche Länge.Anders ausgedrückt: Alle Blätter eines B-Baums liegen auf derselben Ebene.*
2. *Jeder Knoten x hat $n(x)$* in aufsteigender Reihenfolge sortierte *Schlüssel, $k_1(x), \ldots\ldots, k_{n(x)}(x)$, wobei $t-1 \leq n(x) \leq 2t-1$ gilt. Ausnahme: Für die Wur-*

zel r gilt $1 \leq n(r) \leq 2t - 1$. *Lediglich für einen leeren B-Baum enthält die Wurzel keinen einzigen Schlüssel. Ein Knoten heißt* **voll**, *wenn er genau* $2t - 1$ *Schlüssel enthält.*

3. *Jeder Knoten* x *ist ein Blatt oder hat genau* $n(x) + 1$ *Nachfolger. Das Prädikat* `leaf(x)` *ist wahr genau dann, wenn* x *ein Blatt ist. Die Nachfolger sind realisiert als Zeiger auf (Indizes von) die entsprechenden Seiten,* $p_1(x), \ldots, p_{n(x)+1}(x)$. *Die entsprechenden Felder in einem Blatt sind undefiniert und dürfen nicht angefaßt werden.*

4. *Die Schlüssel* $k_i(x)$ *teilen die Bereiche der Schlüssel in den Unterbäumen auf wie in Abb. 7.17 dargestellt (wir lassen das* x *fort, wenn klar ist, um welchen Knoten es sich handelt; insbesondere steht nun* n *für* $n(x)$*)*

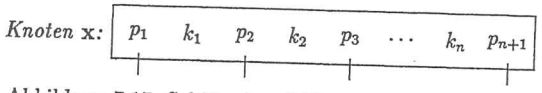


Abbildung 7.17: Schlüssel und Verweise in einem B-Baum-Knoten

Für die Schlüssel s *im Unterbaum, der über den Verweis* p_i *erreichbar ist, gilt:* $k_{i-1} \leq s \leq k_i, i = 2, \ldots, n$. *Die Schlüssel im Baum* p_1 *sind alle* $\leq k_1$; *die im Baum* p_{n+1} *sind alle* $\geq k_n$.

Natürlich folgt für einen inneren Knoten (also für einen Knoten, der kein Blatt ist) die Eigenschaft 3 aus der Eigenschaft 2. Denn die n Schlüssel lassen sich von $n+1$ Unterbäumen „einrahmen".

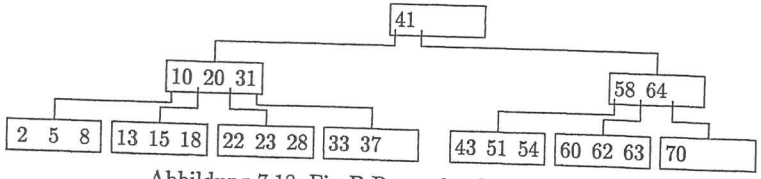


Abbildung 7.18: Ein B-Baum der Ordnung $t = 2$

Als Beispiel ein B-Baum zweiter Ordnung (Abb. 7.18). Das bedeutet, daß alle Knoten außer der Wurzel zwischen einem $(t - 1)$ und drei $(2t - 1)$ Schlüssel enthalten und entsprechend einen Verweis auf einen Unterbaum mehr.

Eine kleine Bemerkung noch, bevor wir die entsprechenden Algorithmen entwerfen. Die hier gegebene Definition von B-Bäumen ist *nicht* die einzig mögliche. Der entscheidende Unterschied zwischen den einzelnen Varianten besteht lediglich darin, daß einige (wie bei uns) eine *ungerade* Anzahl von Schlüsseln enthalten, während es natürlich auch die Möglichkeit gibt, eine *gerade* Anzahl von Schlüsseln in einem Knoten zu fordern. Das macht keinen großen Unterschied, wie man sich leicht vorstellen kann, es ist lediglich dann einfacher, mit einer ungeraden Anzahl zu arbeiten, wenn wir uns irgendwann einmal um so etwas wie den *mittleren* Schlüssel in einem Knoten Sorgen machen müssen.

7.3.3 Suchen in einem B-Baum

Zunächst einmal das Problem der Suche eines Elements in einem B-Baum. Wir schauen uns das ganze Problem in zwei Schritten an:

1. Suchen *innerhalb* eines Knotens.
2. Suchen *unterhalb* eines Knotens.

Dazu macht man sich zunächst einmal klar, wie es innerhalb eines solchen Knotens denn genau ausschaut: Wir finden dort prinzipiell eine Folge von Werten, die abwechselnd aus Zeigern (auf Nachfolger) $p_i, i = 1, \ldots, n + 1$ und Schlüsseln $k_j, j = 1, \ldots, n$ besteht, wobei am Anfang und am Ende immer ein Zeiger steht. Vergleichen Sie bitte noch einmal Abb. 7.17.

Die Ausgangssituation ist dadurch gegeben, daß wir in einem gewissen Knoten x nach einem Schlüssel k suchen. Man wird also zunächst einmal im Knoten die k_i angucken, etwa durch eine geeignete lineare Suche. Entweder finden wir k unter den Schlüsseln oder wir finden einen Verweis auf einen Unterknoten, in dem dieser Schlüssel stehen könnte.

Algorithmus 7.10 *Suchen des Schlüssels* `k` *im Knoten* `x` *eines B-Baums.*

Precondition: `x <> nil`
Postcondition: `---`

```
function b-tree-search(x,k)
  i := 1
  while i <= x.n and k > k[i] do      {Lineare Suche}
     i := i + 1

  if i <= x.n and k = k[i] then       {Gefunden?}
    return x,i                        {Resultat: Knoten/Index}

  if {x ist ein Blatt} then           {Unterhalb eines Blatts}
     return nil                       {ist Schluss!}
  else
     disk-read(p[i])                  {Knoten beschaffen}
     return b-tree-search(p[i],k)     {rekursiv runter!}
```

Aufmerksame Leser haben schon gemerkt, was man hier noch besser machen kann: Man kann sich darauf verlassen, daß die Schlüssel innerhalb eines Knotens ja sortiert vorliegen! Und mit dieser Überlegung gelangt man dann schnell zur Lösung folgender

Aufgabe: *Wie kann die Suche innerhalb eines Knotens asymptotisch verbessert werden? Modifizieren Sie* `b-tree-search` *entsprechend.*

Um einen B-Baum zu initialisieren, müssen wir nicht viel tun: Wir müssen uns einen Knoten beschaffen, ihn als Blatt markieren, die Anzahl der Schlüssel auf 0 setzen, ihn auf die Platte schreiben und als Wurzelknoten des Baums vermerken.

Algorithmus 7.11 *Aufbau eines leeren B-Baums* T*:*

Precondition: ---
Postcondition: T besteht aus einer leeren Wurzel

```
procedure b-tree-create(T)
  x       := new-node
  leaf(x) := true
  x.n     := 0
  disk-write(x)
  root(T) := x
```

7.3.4 Einfügen in einen B-Baum

Das Einfügen eines Werts in einen B-Baum ist nicht ganz trivial, aber mit etwas Nachdenken einigermaßen nachzuvollziehen. Bei „normalen" Binärbäumen oder auch Rot/Schwarz Bäumen ging es noch einigermaßen glatt ab, aber bei den B-Bäumen haben wir das Problem, daß wir die perfekte Balance-Eigenschaft beachten müssen. Aus diesem Grund muß bei der u.U. unvermeidbaren Vertiefung des Baums mit einer gewissen Vorsicht vorgegangen werden. Die Hauptoperation, die wir dazu brauchen ist das *Spalten* eines Knotens, was im B-Baum Jargon als *Split* bezeichnet wird.

Spalten eines Knotens

Ein Knoten muß aufgespalten werden, wenn er voll ist, wenn also die maximale Anzahl von Schlüsseln $(2t-1)$ in diesem Knoten enthalten ist. Zusätzlich wollen wir annehmen, daß der zu spaltende Knoten einen Vorgänger hat, der nicht voll ist. In Abb. 7.19 (oben) sieht man die Situation für einen Knoten aus einem B-Baum der Ordnung 4.

Was man tun muß, ist ganz einfach: Man nimmt sich aus dem Knoten y den *Median-Schlüssel* (den schon angesprochenen mittleren Schlüssel, hier die 11) mit dem Index t (der Ordnung des Baums). Diesen Schlüssel fügt man im Vorgänger x an die richtige Stelle ein (also hier zwischen 7 und 15) und teilt y dann in zwei Knoten mit jeweils $t-1$ Schlüsseln auf, die man richtig an den Vorgänger ankettet. Das Ergebnis des Splits von y sieht dann aus, wie in Abb. 7.19, unten.

Bei der Programmierung dieses Verfahrens gibt es einige notationelle Probleme. Wir werden wechselweise Schlüssel und Verweise in zwei Knoten ansprechen, weil wir diese Einträge ja herumkopieren müssen. Wir verabreden, daß wir den Schlüssel mit dem Index j im Knoten x kurz als `x.k[j]` bezeichnen. Mit den Verweisen und anderen Knoten geht es natürlich genau so.

Algorithmus 7.12 *Spalten eines vollen Knotens* y*. Der Vorgänger* x *ist dabei nicht voll.* y *hängt am Verweis* x.p[i]*). Zu den Bezeichungen vergleiche Abb. 7.19.*

Precondition: y ist voll, x ist nicht voll
Postcondition: y ist nicht voll

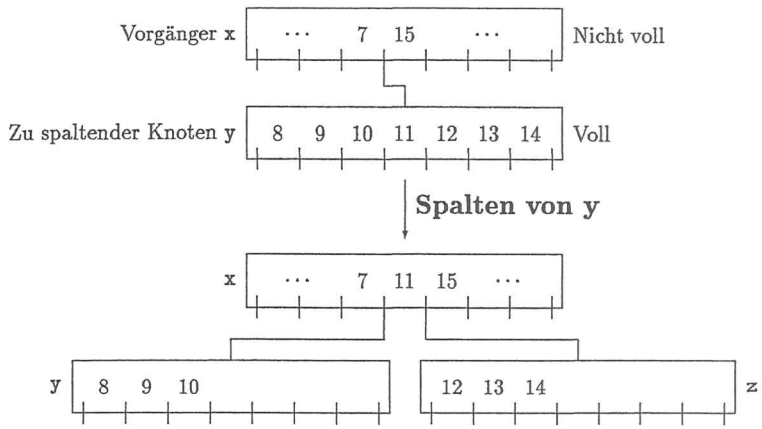


Abbildung 7.19: Spalten eines Knotens in einem B-Baum

```
procedure b-tree-split-child(var x; i; var y)

  {Zuerst einen zusaetzlichen Knoten beschaffen}
  {z wird dann die hoeheren Schluessel enthalten}
  z   := new-node
  z.n := t-1          {Anzahl Schluessel in z}

  {z ist Blatt, wenn y ein Blatt ist}
  leaf(z) := leaf(y)

  {Hohe Schluessel von y nach z schieben}
  for j := 1 to t-1 do
     z.k[j] := y.k[j+t]

  {Falls y Nachfolger hat, die Verweise auch nach z kopieren}
  if not leaf(y) then
     for j := 1 to t do
       z.p[j] := y.p[j+t]

  {Den "Rest" in y abhacken, indem man einfach}
  {die Anzahl der Elemente runtersetzt!}
  y.n := t-1

  {Split beendet. Nun den Median in x einfuegen}
  {Erster Teil: Platz fuer den Verweis auf z schaffen}
  {             und z dort eintragen}
  for j := x.n+1 downto i+1 do
     x.p[j+1] := x.p[j]
```

```
x.p[i+1] := z

{Zweiter Teil: Schluessel in x verschieben, um Platz fuer}
{              den Median zu schaffen, Median eintragen}
for j := x.n downto i do
   x.k[j+1] := x.k[j]
x.k[i] := y.k[t]       {Achtung hier!!}

x.n := x.n + 1         {x hat nun einen Schluessel mehr}

disk-write(x)          {Alle geaenderten Knoten zurueckschreiben}
disk-write(y)
disk-write(z)
```

Diese Spalterei hat lediglich den Sinn, das *Einfügen* eines Schlüssels in einen B-Baum vorzubereiten. Dabei kann es in der Tat vorkommen, daß man sich in einem vollen Knoten wiederfindet, den man dann entsprechend aufspalten muß. Zunächst aber einige Überlegungen, wie man einen Schlüssel k in einen Knoten x einfügen muß, der *nicht* voll ist.

Einfügen in einen nicht-vollen Knoten

Zunächst stellen wir fest, daß Schlüssel *nur in Blätter* eingefügt werden! Wie dies? Wenn man einen Schlüssel in einen inneren Knoten einfügen wollte, müßte man sich ja um die entsprechenden „flankierenden" Unterbäume kümmern, die dann ja hinzukämen.

Nehmen wir also an, wir hätten einen nicht vollen Knoten x, in den wir den Schlüssel k einfügen wollten. Es sind dann zwei Fälle zu unterscheiden:

1. x ist ein Blatt. Nach Voraussetzung ist x nicht voll und man kann k durch einfaches Einfügen eintragen.
2. x ist ein innerer Knoten. Die Hauptaufgabe besteht darin, den Verweis auf den Nachfolgeknoten zu finden, in dem man weiterklettern muß, um das richtige Blatt zu finden.

Man bekommt als erste Grobversion:

```
procedure b-tree-insert-nonfull(x,k)
   if leaf(x) then
      {Suche den Platz fuer k und trage es dort ein}
   else
      {Suche den Nachfolgeknoten und fuege k dort (rekursiv) ein.}
      {Es ist darauf zu achten, dass dieser Knoten nicht voll ist}
```

Die Verfeinerung von

```
{Suche den Platz fuer k und trage es dort ein}
```


lautet dann...

```
i := x.n      {Anzahl der Schluessel im Knoten x}

{Platz fuer k suchen, dort eintragen}
{Verfahren: Einfaches Einfuegen durch Hochschieben}
while i >= 1 and k < x.k[i] do
   x.k[i+1] := x.k[i]
   i        := i - 1

{assert: i < 1 or k >= x.k[i]}
x.k[i+1] := k       {Also: k muss auf Indexposition i+1}

x.n := x.n + 1      {Ein neuer Schluessel}
disk-write(x)
```

Etwas aufwendiger ist das, was man tun muß, wenn es sich bei x um einen inneren Knoten handelt:

```
{Suche den Nachfolgeknoten und fuege k dort (rekursiv) ein.}
{Es ist darauf zu achten, dass dieser Knoten nicht voll ist}
```

Verfeinern...

```
{Suche Index i des Nachfolgeknotens}
{Sorge dafuer, dass dieser Knoten nicht voll ist}
b-tree-insert-nonfull(x.p[i],k) {rekursiv!}
```

Nun ist der Rest ganz einfach:

```
{Suche Index i des Nachfolgeknotens}
```

wird zu...

```
i := x.n      {Anzahl der Schluessel im Knoten x}
while i >= 1 and k < x.k[i] do
   i := i - 1

{assert: i < 1 or k >= x.k[i]}
{Also: Nachfolger auf Position i+1}
i := i + 1    {Nachfolger-Index}
y := x.p[i]   {Nachfolger-Knoten}

disk-read(y)  {Seite y angucken}
```

Nun sorgen wir dafür, daß Knoten y nicht voll ist. Wenn nicht, spalten wir ihn in Richtung x. Dann müssen wir uns nochmal den (neuen durch den Split enstandenen Schlüssel) angucken, um endgültig den richtigen Index i zu bestimmen:

```
if y.n = 2t - 1 then      {y ist voll ===> Spalten}
   b-tree-split-child(x,i,y)

{In welchen Nachfolger muss k denn nun?}
if k > x.k[i] then
   i := i + 1
```

Algorithmus 7.13 *Einfügen eines Schlüssels* k *in den nicht-vollen Knoten* x.

Precondition: x ist nicht voll
Postcondition: x enthält k

```
procedure b-tree-insert-nonfull(x,k)
  i := x.n
  if leaf(x) then
     while i >= 1 and k < x.k[i] do
        x.k[i+1] := x.k[i]
        i        := i + 1

     x.k[i+1] := k
     x.n := x.n + 1
     disk-write(x)
  else
     while i >= 1 and k < x.k[i] do
        i := i - 1

     i := i + 1              {Nachfolger-Index}
     y := x.p[i]             {Nachfolger-Knoten}
     disk-read(y)

     if y.n = 2t - 1 then    {y ist voll ===> Spalten}
        b-tree-split-child(x,i,y)

        if k > x.k[i] then
           i := i + 1

        b-tree-insert-nonfull(x.p[i],k)
```

Wenn wir in einen „nicht-vollen" Knoten einfügen können, ist das generelle Einfügen nun kein großes Problem mehr. Der ganze Einfall besteht lediglich darin, daß wir B-Bäume nicht mehr nach unten sondern *nach oben* wachsen lassen.

Algorithmus 7.14 *Einfügen des Schlüssels* k *in den B-Baum* T.

Precondition: ---
Postcondition: $T_{post} = T_{pre} + k$


```
procedure b-tree-insert(T,k)
  rootnode := root(T)                {Wurzel des Baums}

  if rootnode.n = 2t - 1 then        {Ist sie voll?}
     {Ja! Wurzel spalten: Neuen Knoten s drueber setzen}
     s       := new-node
     leaf(s) := false
     s.n     := 0
     s.p[1]  := rootnode

     {s wird dann die neue Wurzel}
     root(T) := s

     {Nun den einzigen Nachfolger von s
      - die ehemalige Wurzel - spalten}
     b-tree-split-child(s,1,rootnode)

     {s ist nun nicht voll: Einfach einfuegen}
     b-tree-insert-nonfull(s,k)
  else
     {Wurzel nicht voll}
     b-tree-insert-nonfull(rootnode,k)
```

Man beachte bitte, daß es unnötig ist, hier irgendwelche Knoten zurückzuschreiben, weil das in den entsprechenden Prozeduren zum Spalten und Einfügen schon gemacht wird.

Ein Beispiel

Analysieren wir dieses Verfahren an einem Beispiel. Wir nehmen an $t = 3$, also kann jeder Knoten maximal fünf Schlüssel enthalten.

Wir starten also damit, daß wir etwa in Abb. 7.20 (oben) den Schlüssel 24 einfügen wollen. Zunächst stoßen wir auf das Problem, daß der Wurzelknoten voll ist und spalten die Wurzel in der Mitte (52) nach oben. Der Effekt ist, daß man nun mit der bekannten Einfügungsprozedur arbeiten kann (Abb. 7.20, unten).

Offenbar geht es beim Einfügen der 24 nun so weiter, daß wir im (neuen) ganz linken Unterbaum der (neuen) Wurzel weiterarbeiten. Schauen wir uns nun den Fall an, daß wir von dort in einen vollen Knoten geraten, der ja „zwischen" den Schlüsseln 20 und 43 liegen muß.

Nun sagt uns `btree-insert-nonfull`, daß wir diesen Knoten nach oben spalten müssen: Also die 31 hochziehen und zwei Knoten konstruieren.

Man beachte bitte, daß durch diese Operation der Baum nicht tiefer wird! Nun soll der Knoten mit den Schlüsseln 22 und 25 endlich ein Blatt sein (Abb. 7.21, oben), so daß wir unsere 24 dort nun einfach einfügen können (Abb. 7.21, unten).

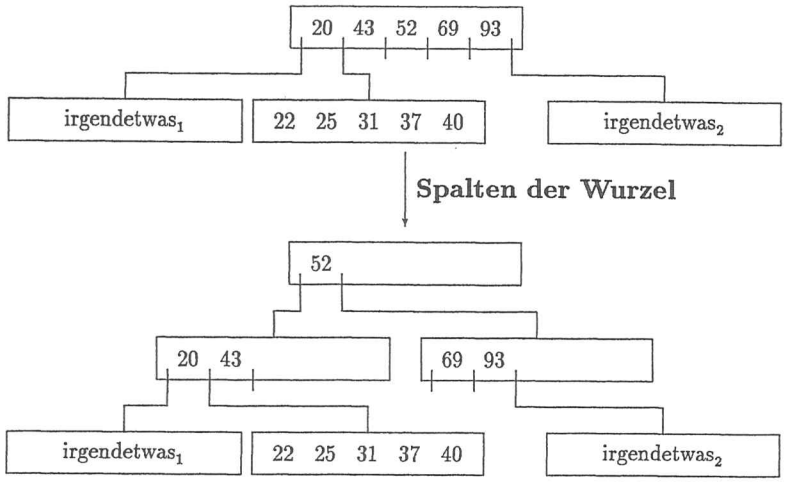


Abbildung 7.20: Einfügen in einen B-Baum, erste Phase

7.3.5 Löschen aus einem B-Baum

Nächster Punkt: Löschen eines Schlüssels k in einem B-Baum! Wir werden das ausführlich analysieren, weil da einige kleine Probleme auftreten. Zwei Fälle sind zu unterscheiden:

Fall 1: Der Knoten x, in dem k liegt, ist ein Blatt.
Fall 2: Der Knoten x, in dem k liegt, ist ein innerer Knoten.

Das „einzige" Problem im ersten Fall: Man kann natürlich k einfach rausnehmen, aber es könnte sein, daß dann in diesem Knoten x zuwenig Schlüssel übrigbleiben! Im obigen Baum (nach dem Einfügen der 24, Abb. 7.21, unten) wäre es einfach, die 24 wieder zu löschen, es blieben ja noch $t-1=2$ Schlüssel im Knoten. Was aber geschieht, wenn man etwa die 37 löschen will?

Einfach rausnehmen geht nicht, es bleibt nur die 40 übrig und das ist zuwenig. Es entsteht ein Knoten, der weniger als $t-1$ Schlüssel enthält; so etwas nennen wir einen *untervollen* Knoten (s. Abb. 7.22, links oben).

Das gesamte Problem beim Löschen besteht offenbar darin, diese untervollen Knoten möglichst mit einer lokalen Operation loszuwerden. In diesem Fall ist es ganz einfach. Die Idee: Man „borgt" sich einen Schlüssel aus dem Nachbarknoten z, schaufelt ihn in den Vaterknoten y als neuen Schlüssel und holt sich den dort vorher vorhandenen Schlüssel in unseren „untervollen" Knoten x ans linke Ende. Abb. 7.22 (links unten) zeigt das Ergebnis dieses „Borgens".

Man überzeugt sich schnell, daß es zwei Möglichkeiten zu borgen gibt: einmal vom linken und einmal vom rechten Nachbarknoten.

Aufgabe: *Basteln Sie sich ein Beispiel dafür zusammen, wie man vom rechten Nachbarknoten dessen linkes Element borgt.*

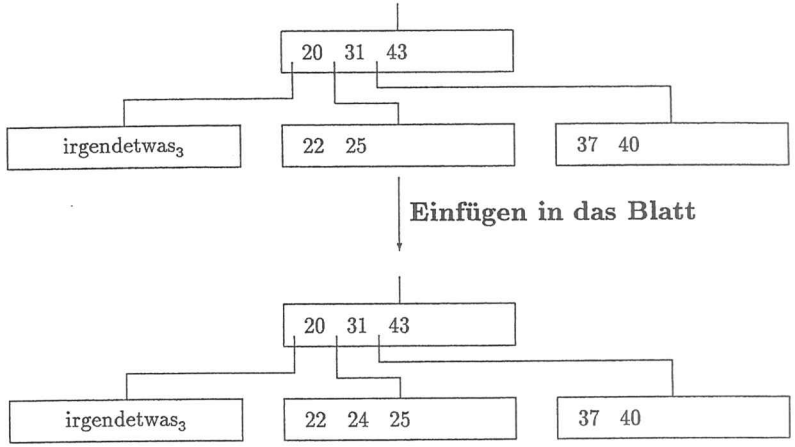


Abbildung 7.21: Einfügen in einen B-Baum, zweite Phase

Löschen wir nun aber die 40, so stehen wir vor dem nächsten Problem: Im Nachbarknoten z ist kein Schlüssel zum „Borgen" mehr da, dann würde der zu klein. Aber nun sind in zwei Knoten ausreichend wenig Schlüssel vorhanden, um sie zu *verschmelzen*. In Abb. 7.22 (rechts unten) ist zu sehen, was nach dem Verschmelzen übrigbleibt. Auf jeden Fall *nicht* der Knoten x, der ist nun verschwunden!

Auch hier erkennt man schnell, daß es sich um eine lokale Operation handelt, deren Aufwand allerdings auch noch von t abhängt, denn es müssen ja die $t-2$ Schlüssel und $t-1$ Verweise von x nach z kopiert werden. Das gesamte Verfahren zur Beseitigung eines untervollen Knotens besteht also darin, daß man entweder borgt oder verschmilzt. Allgemein bezeichnen wir diese Operation als *Balancieren* des Knotens x.

Balancieren eines Knotens

Hier und nur hier ist das Problem beim Löschen, das beim Codieren etwas Überlegung erfordert. Natürlich kann man „Borgen" und „Verschmelzen" als einfache Unterprogramme aufbauen und entsprechend aufrufen. Nur beim Verschmelzen kann es passieren, daß auch weiter oben der Knoten y balanciert werden muß, wenn der untervoll wird.

Nun kann man sich die Sache einfach machen und genau in diesem Fall hoch nach y klettern (zum Balancieren von x brauchen wir y ja sowieso), dort feststellen, an welchem Knoten und an welchem Verweis dort y hängt und entsprechend balancieren. Praktisch ist das aber wegen des vergleichsweise hohen Suchaufwands nicht. Legen wir also das Szenario zum Löschen eines Schlüssels k genauer fest.

Wir werden also mit k in den Wurzelknoten des B-Baums einsteigen, dort den Knoten suchen, der k enthält (sofern es so etwas gibt) und dann mit dem eigentlichen Löschen ansetzen. Dann zeigt es sich aber, daß man alle eventuell zu balancierenden Knoten auf

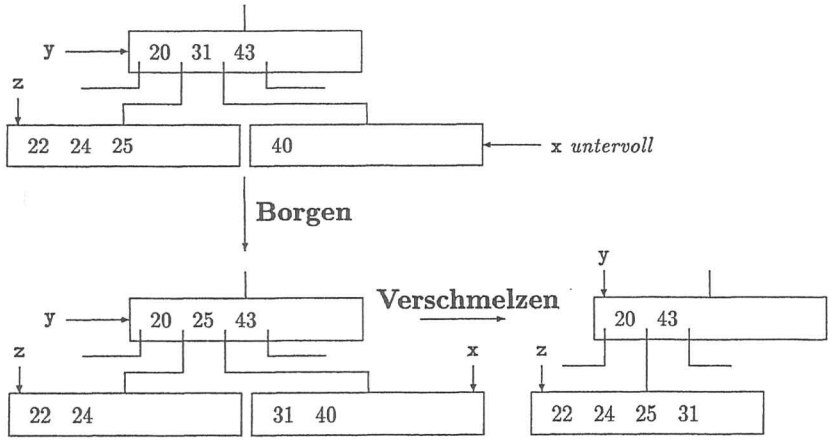


Abbildung 7.22: Balancierungsoperationen in einem B-Baum

dem Suchpfad bereits einmal in der Hand gehabt hat. Dann ist es einfacher, beim Rückzug aus dem Baum die notwendigen Balancierungsarbeiten vorzunehmen.

Jedes Unterprogramm, das sich mit dem Löschen beschäftigt, wird also ein Signal an das von oben rufende Unterprogramm zurückgeben, das dem anzeigt, daß eine eine Balancierung notwendig ist. Schematisch schauen unsere Prozeduren also wie folgt aus:

```
procedure irgendwas-mit-dem-Blatt(x; var must_balance)

  succ := {irgendein Nachfolgeknoten von x}
  irgendwas-mit-dem-Blatt(succ,must_balance)

  {Falls must_balance wahr ist, so ist succ untervoll}
  {Also: Balanciere succ unterhalb von x}

  must_balance := {x ist nun untervoll}
```

Eine weitere Konsequenz besteht darin, daß man sowohl beim Borgen als auch beim Verschmelzen auf die Position eventueller Nachfolgeknoten Rücksicht nehmen muß. Es ist ja durchaus möglich, daß nicht nur Blätter sondern auch innere Knoten belanciert werden müssen.

Zunächst das Verfahren, mit dem man vom linken Nachbarn z des Knotens x ein Element borgt. Als einzige Vorbedingung nehmen wir lediglich an, daß der „Spender" z genug Schlüssel enthält, um beim Ausleihen des rechten nicht untervoll zu werden. Daß x zu wenig Schlüssel enthält, ist ebenfalls anzunehmen, sonst würden wir das gar nicht versuchen. Aus diesem Grund kann danach keiner der beteiligten Knoten untervoll werden, so daß wir hier keine Meldung nach oben reichen müssen. Das wird erst beim Verschmelzen der Fall sein.

Das Problem entsteht lediglich in der Analyse des neuen Platzes für den rechten Unterbaum des geborgten Schlüssels. Man erkennt aber an Abb. 7.23, daß er links in den freiwerdenden Schlitz im Knoten x gehört.

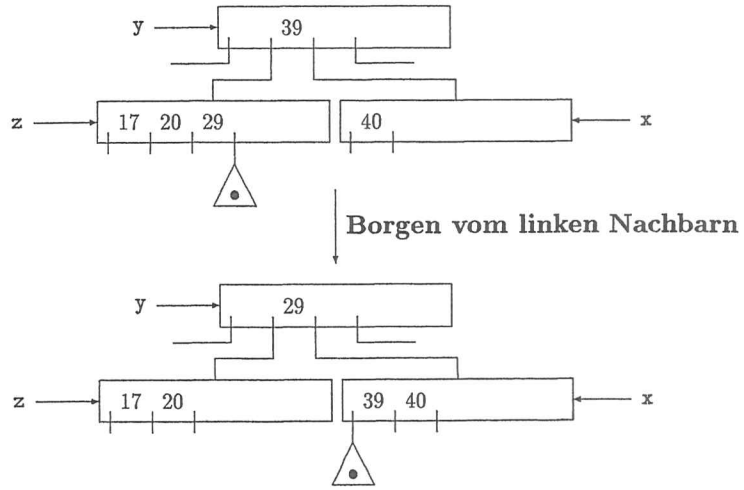


Abbildung 7.23: Borgen eines Schlüssels

Was wir also brauchen, sind die Knoten x, y und z, sowie den Index i, der uns sagt, wo x in y angeklemmt ist. Nachher müssen wir lediglich dafür sorgen, daß wir das beim Aufruf wissen.

Algorithmus 7.15 *Borgen eines Elements vom* linken *Nachbarn* z *des Knotens* x, *das am Zeiger* y.p[i+1] *hängt:*

Precondition: z enthält genug Schlüssel
Postcondition: x nicht untervoll

```
procedure b-tree-borrow-from-left(y,i,z,x)
  nz := z.n        {Groesse des Knotens z}

  {Schritt 1: Die Elemente von x um eine Stelle nach rechts}
  for j := x.n downto 1 do
     x.k[j+1] := x.k[j]
     x.p[j+1] := x.p[j]

  {Nicht vergessen: Nun ist es einer mehr!}
  x.n := x.n + 1

  {Schritt 2: Trennenden Schluessel nach x auf Position 1 ziehen}
  x.k[1] := y.k[i+1]
```

```
{Schritt 3: Ganz rechten Unterbaum von z ganz links an x}
x.p[1] := z.p[nz+1]

{Schritt 4: Rechten Schluessel von z hochziehen}
y.k[i+1] := z.k[nz]
z.n      := nz - 1

disk-write(x)
disk-write(y)
disk-write(z)
```

Aufgabe: *Codieren Sie* `b-tree-borrow-from-right`, *das das entsprechende Borgen vornimmt, wenn* `z` *rechter Nachbar von* `x` *ist.*

Hier nun der Fall, in dem zwei Knoten x und z verschmolzen werden. Im Prinzip handelt es sich um eine Art Umkehrung des Aufspaltens eines Knotens. Wir klären zunächst die Frage, wie sich die entsprechenden Teilbäume verhalten müssen. Dazu ein Beispiel, in dem wir die entsprechenden Unterbäume nur symbolisch angedeutet haben (Abb. 7.24). Wie man sieht, gehen wir dabei davon aus, daß z *linker* Nachbar von x ist.

Man sieht schnell, daß sich in der Reihenfolge der Schlüssel und der eventuellen Unterbäume nichts ändert, wenn man sie von links nach rechts liest:

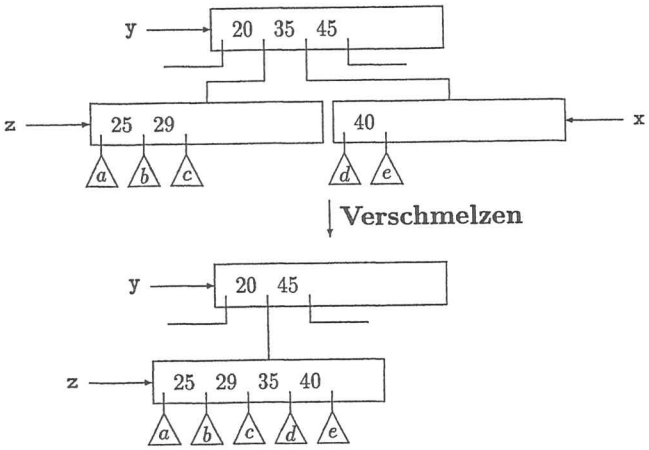


Abbildung 7.24: Verschmelzen von Knoten

Hier ist nun auch der Fall, daß man unter Umständen nach oben signalisieren muß, daß y zu balancieren ist, falls der nämlich untervoll wird. Das geschieht durch den Parameter `must_balance`.

Hier also der Algorithmus zum Verschmelzen von zwei Knoten eines B-Baums:

Algorithmus 7.16 *Verschmelzen der Knoten* `x` *und* `z` *unterhalb* `y`. `x` *hängt am Verweis* `y.p[i+1]`. `z` *ist* linker *Nachbar von* x. `must_balance` *wird* true, *wenn* `y` *untervoll wird und balanciert werden muß.*

Precondition: ---
Postcondition: `x` nicht untervoll

```
procedure b-tree-join(y,i,x,z; var must_balance)
  nx := x.n       {Anzahl Schluessel in x}
  nz := z.n       {Anzahl Schluessel in z}

  {1. Trennschluessel aus y runterziehen}
  z.k[nz+1] := y.k[i]

  {2. Alle Schluessel aus x kopieren}
  for j := 1 to nx do
     z.k[nz+j+1] := x.k[j]

  {3. Die Unterbaeume auch umkopieren}
  for j := 1 to nx+1 do
     z.p[nz+j+2] := x.p[j]

  {4. Schluessel und Verweise in y komprimieren}
  for j := i to y.n do
     y.k[j] := y.k[j+1]
     y.p[j] := y.p[j+1]

  {5. Anzahlen aendern}
  y.n          := y.n - 1
  must_balance := (n(y) < t-1)    {y untervoll ==> balancieren!}
  z.n          := nz + nx + 1     {Die 1 kommt aus y!}

  dispose(x)
  disk-write(y)
  disk-write(z)
```

Nun haben wir so ziemlich alles zusammen, um einen untervollen Knoten x auszubalancieren. Leider stellt sich in der Praxis das Problem, sich zwischen einem eventuellen linken bzw. rechten Nachbarn z zu entscheiden, mit dem der Ausgleich vorgenommen werden soll. Hier kann man einige Ideen einbringen, die z.B. dafür sorgen, daß wenn möglich geborgt werden soll, um eventuelle große Umorganisationen des Baums zu vermeiden, die beim Verschmelzen von Knoten ja auftreten können. Wir geben hier also nur eine Skizze an.

Algorithmus 7.17 *Balancieren der Seite* `x` *am Verweis* `y.p(i+1)` *der Seite* `y`. `must_balance` *wird* true, *wenn* `y` *balanciert werden muß.*

Precondition: `x ist untervoll`
Postcondition: `x ist nicht untervoll`

```
procedure b-tree-balance(x,y,i; var must_balance)
  z := {Linker oder rechter Nachbar von x}

  if z.n >=  t then          {Genug Schluessel in z: Borgen}
     must_balance := false  {Keine Balance noetig!}
     if {z ist linker Nachbar von x} then
        b-tree-borrow-from-left(y,i,x,z)
     else
        b-tree-borrow-from-right(y,i,x,z)
  else
     {Beim Verschmelzen an die Reihenfolge von x und z denken!}
     if {z ist linker Nachbar von x} then
        b-tree-join(y,i,z,x,must_balance)
     else
        b-tree-join(y,i,x,z,must_balance)
```

Löschen in einem inneren Knoten

Das nächste Problem stellt sich, wenn wir den Schlüssel k in einem inneren Knoten finden. Das lösen wir dadurch, daß wir uns derselben Idee bedienen, den wir beim Löschen in sortierten Binärbäumen (s. 7.1.3). schon kennengelernt haben: Wir löschen einfach einen anderen Schlüssel, der in einem Blatt liegt. Das entspricht i.w. der Suche nach dem ON eines Knotens in einem solchen Baum und läuft nach derselben Idee ab: Man geht in den rechten Nachfolgebaum des zu löschenden Schlüssels und läuft dann immer den ganz links stehenden Verweis runter bis man an ein Blatt kommt.

Offenbar ist dort der ganz linke (kleinste) Schlüssel der unmittelbare Nachfolger des zu löschenden Schlüssels. Den löschen wir aus dem Blatt, gleichen u.U. aus und ersetzen den eigentlich zu löschenden Schlüssel durch den eben gelöschten. Abb. 7.25, links: Der Schlüssel 35 im Knoten x ist zu löschen.

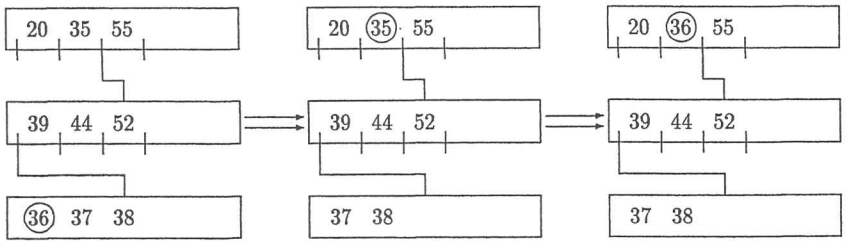


Abbildung 7.25: Löschen aus einem inneren Knotens eines B-Baums

Man geht im rechten Nachfolgebaum der 35 runter in den Knoten 39, 44, 52 und immer weiter links entlang, bis man zum Blatt 36, 37, 38 kommt. Dort wirft man die 36 raus (Abb. 7.25, Mitte) und trägt sie oben anstelle der 35 ein (Abb. 7.25, rechts).

Man beachte dabei, daß es Geschmackssache ist, ob man im rechten Nachfolgebaum den

ganz links stehenden Schlüssel löscht oder im linken Nachfolgebaum den ganz rechts stehenden. [Wirth, 1979] macht es genau anders herum als hier.[1]

Zunächst also bearbeiten wir den Fall, diesen Schlüssel zu finden und zu löschen. Das erledigen wir in einem Gang mit dem Problem, ihn ganz nach oben zurückzugeben, weil wir ihn dort ja an der Stelle des eigentlich zu löschenden Knotens einsetzen wollen.

Nun die Frage nach der Balancierung. Die Konvention ist ja die, daß der Parameter `must_balance` immer dann `true` ist, wenn das Blatt, das wir gerade behandelt haben, untervoll geworden ist. Insbesondere wird also der Fall, daß nach dem tatsächlichen Löschen des linken Schlüssels im am weitesten links stehenden Blatts dieses Blatt untervoll ist, gar nicht sofort abgehandelt, sondern nach oben signalisiert. Die notwendige Balance wird dann vom rufenden Programm abgehandelt.

Algorithmus 7.18 *Löschen und Zurückliefern des Schlüssels* `k`*, der in oder unterhalb des Knotens* `x` *im am weitesten links stehenden Blatt ganz links steht.* `must_balance` *ist* true*, wenn* `x` *balanciert werden muß.*

Precondition: `x <> nil`
Postcondition: $\texttt{Baum}_{post} = \texttt{Baum}_{pre} - \texttt{k}$

```
procedure b-tree-delete-leftmost(x; var k; var must_balance)
  if leaf(x) then
     k := x.k[1]              {Das ist er! Merken}

     {Loesche den ersten Schluessel im Blatt x}
     {durch Herunterschieben der anderen}
     for i := 1 to x.n - 1 do
        x.k[i] := x.k[i+1]

     {Anzahl der Schluessel in x heruntersetzen}
     {Feststellen, ob x untervoll geworden ist}
     x.n          := x.n - 1
     must_balance := (x.n < t-1)
     disk-write(x)
  else
     succ := x.p[1]           {Nachfolger anfassen}
     disk-read(succ)
     b-tree-delete-leftmost(succ,k,must_balance)
     if must_balance then    {succ ist untervoll!}
        b-tree-balance(succ,x,0,must_balance)

     {Achtung: must_balance bezieht sich nun auf x}
```

Damit ist das Löschen eines Schlüssels aus einem inneren Knoten weitgehend einfach. Eine Tücke wieder mal: Wenn man nach dem Aufruf von `b-tree-delete-leftmost` gemeldet bekommt, daß die rechte Nachfolgeseite `x.p[i+1]` untervoll geworden ist, so darf

[1] Ein Student hat mich darauf aufmerksam gemacht, daß es geschickter ist, so wie Wirth vorzugehen. Finden Sie eine Begründung dafür?

man *unter keinen Umständen* die notwendige Balancierung vornehmen, bevor man den Schlüssel k eingetragen hat. Die Gefahr, daß das Original beim Balancieren verloren geht und in irgendeine Nachfolgeseite abdriftet, ist nicht zu vernachlässigen!

Algorithmus 7.19 *Löschen des Schlüssels* `x.k[i]` *des inneren Knotens* `x`.

Precondition: `x <> nil and x.p[i+1] <> nil`
Postcondition: `x.k[i] gelöscht, x nicht untervoll`

```
procedure b-tree-delete-inner(x,i)

  {Rechter Nachfolgeknoten hat den Verweis x.p[i+1]}
  right := x.p[i+1]
  disk-read(right)

  {Reingehen und dort den am weitesten links in einem Blatt}
  {stehenden Schluessel k loeschen}
  b-tree-delete-leftmost(right,k,must_balance)

  {must_balance ist true, wenn dieser rechte Folgeknoten}
  {untervoll ist. Noch nicht balancieren! Erst eintragen!}
  x.k[i] := k

  {Jetzt erst balancieren, wenn noetig}
  if must_balance then
     b-tree-balance(right,x,i-1,must_balance)  {Warum 1 abziehen?}
  else
     disk-write(x)
```

Nun können wir den Code für das Löschen eines Schlüssels k in bzw. unterhalb eines Knotens x angeben. Wir verfahren dazu in zwei Schritten:

1. Wir suchen in x nach k bzw. nach demjenigen Nachfolger von x, in dem k stehen könnte.
2. Hier gibt's zwei Fälle zu unterscheiden:
   (a) Wenn wir k in x gefunden haben, löschen wir ihn raus.
   (b) Wenn nicht, gucken wir in der Nachfolgeseite rekursiv weiter.

Gleich den Code dafür, bevor wir endgültig die Lust verlieren. Den Suchteil lassen wir weg, er sieht i.w. genauso aus wie in Algorithmus 7.10.

Algorithmus 7.20 *Löschen des Schlüssels* `k` *in oder unter dem Knoten* `x`. `must_balance` *ist wahr, genau dann, wenn* `x` *untervoll ist.*

Precondition: `x <> nil`
Postcondition: `k in bzw. unter x nicht mehr vorhanden`


```
procedure b-tree-delete-from-node(x,k; var must_balance)

  disk-read(x)

  {Suche Schluessel k im Knoten x:
        found = Schluessel in x gefunden
        ind   = Index von k, falls found
              = Index der Folgeseite, falls not found}

  if found then            {Schluessel k ist im Knoten x}

     if leaf(x) then       {Gefunden in einem Blatt}
        x.n := x.n - 1     {k rausnehmen}
        for j := ind to x.n do
           x.k[j] := x.k[j+1]
        {must_balance = true, falls x untervoll ist}
        must_balance := (x.n < t-1)
        disk-write(x)

     else                  {Gefunden in einem inneren Knoten}

        b-tree-delete-inner(x,ind,must_balance)
        {must_balance = true, falls x untervoll ist}

  else                     {Nicht gefunden! Weiter runter}
     if leaf(x) then       {Endstation! Fehler!}
        {Fehlerbehandlung}
        must_balance := false
     else
        succ := x.p[ind]
        b-tree-delete-from-node(succ,k,must_balance)

        if must_balance then    {succ ist untervoll}
           b-tree-balance(succ,x,ind-1,must_balance)
           {must_balance bezieht sich nun auf x!}
```

Nun ist bis auf ein letztes Problem auch das generelle Löschen eines Schlüssels k in einem B-Baum mit der Wurzel `root` klar:

```
b-tree-delete-from-node(root,k,must_balance)
```

Was bedeutet aber in diesem Fall `must_balance = true`? Zunächst nicht viel, es heißt lediglich `root.n` $< t-1$ und das ist nicht so schlimm. Schlimm wäre allerdings der Fall `root.n` $= 0$. Dann wäre der Wurzelknoten schlüsselfrei und das muß man sich genauer angucken! Ein Beispiel für diesen Fall zeigt Abb. 7.26.

Rechts oben ist überall der minimale Füllungsgrad der Knoten erreicht; jedes Löschen führt zu einer drastischen Reorganisation. Beispielsweise löschen wir mal die 20 und erreichen die Situation rechts oben.

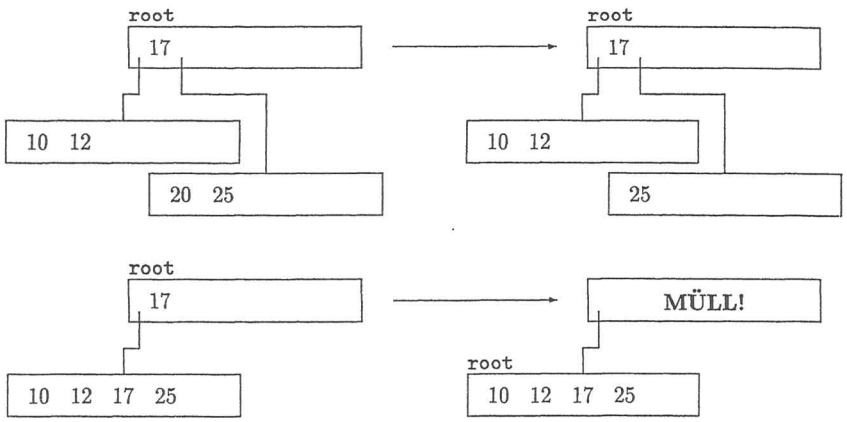


Abbildung 7.26: Verschwinden der Wurzel eines B-Baums

Der Knoten, in dem die 20 stand, ist nun untervoll und wird balanciert, indem er mit seinem linken Nachbarn verschmolzen wird (links unten)

Und schon ist es passiert: Die Wurzel ist schlüsselfrei; lediglich ein einziger Zeiger hängt noch dran. Und nun ist auch klar, was man tun muß: Die Wurzel geht in den Müll und der einzige Nachfolger wird die neue Wurzel (rechts unten). Dies ist auch der einzige Fall, in dem ein B-Baum flacher wird:

Nun also der einfache Algorithmus zum generellen Löschen eines Schlüssels aus einem B-Baum:

Algorithmus 7.21 *Löschen des Schlüssels* k *in einem B-Baum* T.

Precondition: `root(T) vorhanden`
Postcondition: T_{post} = T_{pre} - k

```
procedure b-tree-delete(T,k)

  rootnode := root(T)

  b-tree-delete-from-node(rootnode,k,must_balance)

  if rootnode.n = 0 then        {Wurzel ist leer}
     succ := rootnode.p[1]      {Nachfolger...}
     delete(rootnode)
     root(T) := succ            {... wird neue Wurzel}
```

7.3.6 Komplexität

Soll man über die Komplexität der einzelnen Operationen noch etwas sagen? Natürlich, denn es spielt ja bei vielen nicht nur die Anzahl der Knoten eine Rolle, sondern auch der

Grad des B-Baums. Asymptotisch sind wir natürlich an der Abhängigkeit von t gar nicht so sehr interessiert, weil wir das per Definition des Baums in der Hand haben, wie groß dieses t denn sein soll.

Die Bedeutung des Grads t läßt sich in zwei Regeln fassen:

1. Je größer t ist, desto flacher wird der Baum.
2. Je kleiner t ist, desto geringer wird der Aufwand zum Suchen innerhalb eines Knotens.

Stellen wir zunächst das Verhältnis zwischen der Höhe h, dem Grad t und der Anzahl der Schlüssel n eines B-Baums fest:

Aussage 7.3 *Falls* $n \geq 1$ *und* $t \geq 2$, *dann gilt* $h \leq \log_t \frac{n+1}{2}$.

Einen Beweis findet man in [Cormen *et al.*, 1990]. Das hat ziemlich gewichtige Konsequenzen, wenn man mit dem t ein bißchen herumspielt und das n konstant läßt. Anders ausgedrückt: Man speichert dieselbe Anzahl von Schlüsseln in B-Bäumen unterschiedlichen Grads. Nehmen wir mal an wir hätten grob gerechnet $h_t = \log_t n$ und verdoppeln unser t. Dann bekämen wir die neue Höhe $h_{2t} = \log_{2t} n$. Nun rechnen wir mal das Verhältnis zwischen h_t und h_{2t} aus und bekommen $h_t = h_{2t} \cdot (1 + \log_t 2)$, oder allgemein für $c > 1$ $h_t = h_{ct} \cdot (1 + \log_t c)$.

Natürlich bringt ein großes t bei großem n eine ganze Menge, aber wie wir aus der Mathematik wissen, ist der Effekt linear, fällt asymptotisch also gar nicht ins Gewicht. Algorithmen, die sich entlang von Ästen bewegen, sind damit von der Komplexität $O(\log_t n)$, wenn man den Aufwand, der innerhalb eines Knotens getrieben wird, ignoriert. Im Beispiel des Suchens eines Schlüssels mit linearer Suche bekommen wir ja i.w. $O(t) \cdot O(\log_t n) = O(\log_t n)$, was wir mit Binärsuche zu $O(\log_2 t) \cdot O(\log_t n) = O(\log_t n)$ verbessern können.

Ein wenig komplizierter ist die Geschichte mit den Plattenzugriffen, die ja die Gesamtperformanz stark beeinflussen können. Betrachtet man das recht aufwendige Verfahren zum Löschen eines Schlüssels, so stellt man fest, daß im Prinzip ein kompletter Ast eingelesen *und resident gehalten* wird.

Wieder betrachten wir überschlägig, was eine Änderung von t für Konsequenzen hat. Wir setzen mal ganz grob den Speicherbedarf für einen Ast in einem B-Baum der Ordnung t an mit $s_t = h_t \cdot (2t - 1) \approx h_t \cdot 2t$, also Länge des Asts ($h_t$) multipliziert mit der Größe der Knoten $(2t - 1)$. Nun vergrößern wir mal das t um den bekannten Faktor $c > 1$ und bekommen den neuen Speicherbedarf $s_{ct} = h_{ct} \cdot (2ct - 1) \approx h_{ct} \cdot 2ct$. Von oben wissen wir grob, daß $h_{ct} = h_t/(1+\log_t c)$ und somit $s_{ct} \approx h_t/(1+\log_t c) \cdot 2ct = h_t \cdot 2t \cdot [c/(1+\log_t c)] = s_t \cdot [c/(1 + \log_t c)]$.

Dieser Faktor $\delta_{ct} = c/(1 + \log_t c)$ gibt also an, wie sehr der Speicheraufwand für das Festhalten eines kompletten Asts ungefähr steigt, wenn man das t um den Faktor c erhöht.

Hier sind ein paar Zahlenbeispiele ($c = 2$: Verdoppeln, $c = 10$: Verzehnfachen von t):

t	c	δ_{ct}
10	2	1.537
10	10	5.000
100	2	1.738
100	10	6.667
1000	2	1.818
1000	10	7.500

Und das ist ausgesprochen unangenehm! Der Wechsel von einem B-Baum der Ordnung $t = 100$ zu einem der Ordnung $t = 200$ erhöht den Speicheraufwand beim Löschen (in unserem Ansatz) um einen Faktor, der deutlich über 1 liegt (mehr sollte man nicht sagen, wo wir so viel vereinfachende Annahmen in der Herleitung hatten).

Das führt natürlich dazu, daß wir uns bei der Konstruktion eines solchen Baums vorher über den notwendigen Speicheraufwand klarwerden müssen. Wenn das zu groß wird (in unserer Codierung liegen die Knoten des Asts ja alle auf dem Laufzeitkeller!), muß anders gearbeitet werden, indem man z.B. Knoten vor dem Abstieg in die Rekursion wegwirft und bei Bedarf nachliest, was dann wieder auf die Laufzeit geht. Unser Code liest und schreibt einen Knoten des Asts höchstens einmal, was ganz günstig ist. Im schlimmsten Fall muß man einen Knoten zweimal lesen: Einmal beim Abstieg und einmal beim Balancieren.

Kapitel 8

Suchen in Graphen

Das Suchen in einem allgemeinen Graphen stellt eine Generalisierung des bekannten Suchens in einem Baum dar. Zusätzlich stellt sich hier noch das Problem der *Traversierung*. Unter einer *Traversierung* eines beliebigen Graphen verstehen wir einen Algorithmus, der garantiert jede Ecke dieses Graphen einmal besucht. Für Bäume ist dieses Problem schon behandelt worden: Suchen in Bäumen haben wir meist als *rekursive Tiefensuche* realisiert, vergl. 3.2.2 und 7.1.1. Bei allgemeinen Graphen treten Komplikationen lediglich dadurch auf, daß wir Ecken ziemlich oft besuchen oder gar in Kreise hineinlaufen können. Bleiben wir zunächst mal bei der Tiefensuche (depth-first search, DFS).

Das angenehme dabei ist, daß wir zunächst keinen Unterschied zwischen gerichteten und ungerichteten Graphen machen müssen. Der eigentliche Unterschied liegt ja lediglich im Begriff der „benachbarten" Ecke, was wir aber schon in der Adjazenzliste (s. 3.2.3) realisiert haben.

8.1 Tiefensuche

Die Idee bei der Tiefensuche, die dabei ins Spiel kommt, ist die, so zu tun, als wäre der Graph ein Baum. Die Tatsache, daß er keiner ist, wird dadurch berücksichtigt, daß wir Ecken unter Umständen als *bereits besucht* markieren wollen. Das führt dazu, daß wir zu jeder Ecke noch einen booleschen Wert mitführen, der uns anzeigt, ob wir im Verlauf der Traversierung dort schon mal gelandet sind oder nicht.

Am Anfang wird man also dafür sorgen, daß alle Ecken als „nicht besucht" markiert werden. Allein dieser Vorgang erfordert natürlich schon $|V|$ Schritte. Normalerweise kann man das auch nicht verhindern und braucht also einen linearen Durchgang durch die Eckenliste. Wir gehen also am besten von der Darstellung in der Form einer Adjazenzliste (s. 3.2.3) aus.

Zunächst macht man sich klar, daß der Graph nicht unbedingt zusammenhängend sein muß. Er kann ja aus mehreren *Komponenten* bestehen. Es reicht also nicht aus, an irgendeiner Ecke v zu starten und zu hoffen, so alle Ecken besuchen zu können. Was man bestenfalls schafft, ist durch Verfolgen der Kanten alle Ecken in der Komponente zu besuchen, in der v liegt. Also wird man das Verfahren zweistufig entwerfen:

1. Ein rekursives Verfahren, das ausgehend von einer unmarkierten Ecke v, alle unmarkierten Ecken in der Komponente von v besucht.
2. Ein Verfahren, das sich damit befaßt, das erste auf alle Komponenten anzuwenden.

Das erste ist trivial: Man startet an einer unmarkierten Ecke und besucht rekursiv alle unmarkierten Nachbarecken.

Algorithmus 8.1 *Tiefentraversierung der Komponente eines Graphen, in der die unmarkierte Ecke v liegt.*

Precondition: ---
Postcondition: `Alle Ecken, die von v erreichbar sind, sind markiert`

```
procedure dfs-vertex(v)
  marked(v) := true            {Markiere v als besucht}
  v_next := adj[v]             {Erste benachbarte Ecke}

  while v_next <> nil do
     if not marked(v_next) then
        dfs-vertex(v_next)

     v_next := succ(v_next)    {Naechste benachbarte Ecke}
```

Nun kann man sich schnell überlegen, wie man alle Komponenten des Graphen systematisch besuchen kann:

Algorithmus 8.2 *Tiefentraversierung eines Graphen G.*

Precondition: ---
Postcondition: `Alle Ecken in G markiert`

```
procedure dfs(G)

  for v in V(G) do             {Alle Marken entfernen}
     marked(v) := false

  for v in V(G) do             {Alle unmarkierten Ecken besuchen}
     if not marked(v) then
        dfs-vertex(v)
```

Man überzeugt sich vergleichsweise schnell von der Reihenfolge, in der dieser Algorithmus die Ecken unseres gerichteten Beispielgraphens aus Abb. 3.8, Seite 44, (rechts) besucht:

$$1, 2, 4, 5, 3, 6.$$

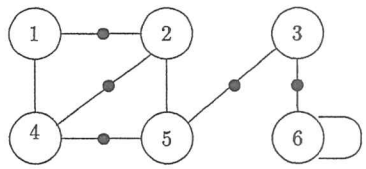


Abbildung 8.1: Bei Tiefentraversierung überquerte Kanten

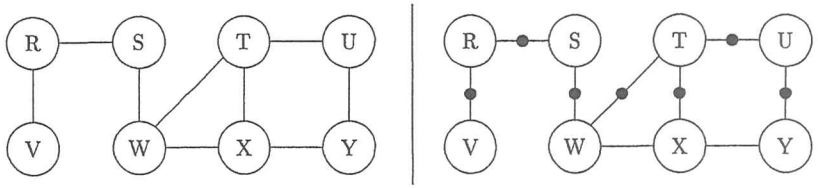


Abbildung 8.2: Eine weitere Tiefentraversierung

Wenn man sich in Abbildung 8.1 noch anguckt, welche Kanten dabei überquert wurden (die mit dem Punkt), stellt man fest, daß er sie an eine ununterbrochene Folge von Kanten aufgefädelt hat, was aber Zufall ist. Was dieser Algorithmus auf einem zusammenhängenden Graphen — der also nur eine Komponente hat — bewirkt, sehen wir an der nächsten Abbildung 8.2.

Hier sind wir mit unserer Tiefentraversierung auf der Ecke S gestartet und haben uns dann entlang der in der rechten Abbildung punktierten Kanten durch den Graphen gearbeitet. Natürlich hängt das Ergebnis im Einzelfall davon ab, wie die Nachbarecken in der Adjazenzliste stehen. Eine Möglichkeit der Tiefentraversierung ist im rechten Teil dargestellt.

Dort sind lediglich die beiden unmarkierten Kanten nicht betreten worden. Das Ergebnis dieser Geschichte ist ein Baum mit der Wurzel S. In der Tat findet Tiefensuche, so wie wir es hier gemacht haben, ein Gerüst eines zusammenhängenden Graphens, wenn auch nicht unbedingt ein minimales. Wenn der Graph *nicht* zusammenhängend ist, bekommen wir einen Wald von Gerüsten: Eins für jede Komponente.

Wie teuer ist `dfs`? Offenbar wird `dfs-vertex` nur für eine unmarkierte Ecke betreten, die dann sofort markiert wird. Das sind dann also genau $|V|$ Aufrufe. In `dfs-vertex` gucken wir uns die benachbarten Ecken *alle* an, obwohl wir einige unter Umständen wegwerfen. Bei einem ungerichteten Graphen fassen wir also jede Kante *zweimal* an (einmal von jeder Seite), bei einem gerichteten tun wir das nur *einmal* (die Adjazenzliste ist ja nicht so umfangreich).

Insgesamt haben wir also einen Aufwand für `dfs`, der proportional zu $\max(|V|, |E|)$ ist, also $O(\max(|V|, |E|))$ und das ist schon ganz erfreulich.

Aufgabe: *Stellen Sie den entsprechenden Algorithmus für eine Darstellung mittels einer Adjazenzmatrix auf und untersuchen Sie seine Komplexität.*

Wenn man `dfs` auf einen ungerichteten Graphen anwendet, kann man damit sogar die Komponenten des Graphen lokalisieren.

Aufgabe: *Nehmen wir an, wir könnten zu jeder Ecke eines ungerichteten Graphen eine beliebige Zahl abspeichern. Modifizieren Sie* `dfs` *und* `dfs-vertex` *derart, daß alle Ecken innerhalb derselben Komponente mit derselben Zahl markiert werden und Ecken in verschiedenen Komponenten verschiedene Zahlen aufweisen.*

Man kann also offenbar während der Tiefentraversierung eines Graphens erheblichen Unsinn mit den Ecken anstellen. Hier kommt noch einer.

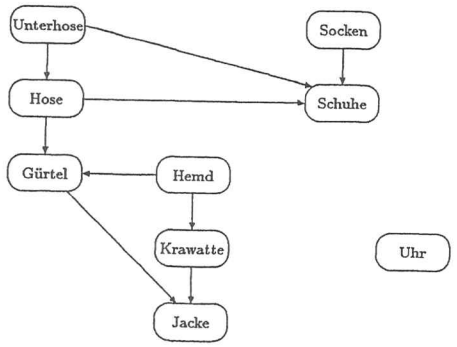


Abbildung 8.3: Ein gerichteter Graph

In Abbildung 8.3 sehen wir einen gerichteten Graphen, der Professor Hastings beim morgendlichen Ankleiden helfen soll, weil das immer soviel Probleme macht, wenn man andauernd über wichtige Dinge nachdenken muß. Es sind in diesem Graphen die wichtigsten Verbindungen zwischen den einzelnen Tätigkeiten des Ankleidens durch gerichtete Kanten dargestellt. Die Bedeutung ist, daß die Tätigkeit der Anfangsecke der Kante *unbedingt vor* der Tätigkeit der Endecke der Kante auszuführen ist: Also ist es zum Beispiel notwendig, die Socken anzuziehen bevor man in die Schuhe steigt, usw.

Wir stellen zunächst fest, daß es sich hierbei um einen gerichteten Graphen handelt. Zeitliche, kausale oder auch manche räumliche Anordnungen von Handlungen, Ereignissen, Gegendtänden, Orten oder was auch immer resultieren grundsätzlich in einer gerichteten Struktur.

Bestimmte solche Strukturen müssen – gemäß der Anschauung – die Eigenschaft haben, *kreisfrei* zu sein. Das heißt, daß man von keiner Ecke mit einer Kantenfolge wieder in diese Ecke zurückgelangen kann. Das ist typisch für zeitliche und kausale Beziehungen. Hier handelt es sich also um einen DAG (s. 3.2.1).

In einem DAG gibt es eine bestimmte Menge von Ecken, die in der Praxis ganz erhebliche Bedeutung haben: Das sind die Ecken, die in der Struktur *keinen Vorgänger* haben. Man kann sich überlegen, daß es in einem *nicht-leeren* DAG immer wenigstens eine solche Ecke

gibt. Wir nennen eine solche Ecke anschaulich einen *Einstieg* in den DAG. Im obigen Bild gibt es also die Einstiege *Unterhose, Socken, Hemd* und *Uhr*.

Wir gehen davon aus, daß wir zu einer Ecke e durch das Prädikat `entry` erkennen, ob sie ein solcher Einstieg ist.

Aufgabe: *Gegeben ein DAG als* Adjazenzmatrix. *Woran kann man einen Einstieg erkennen? Wie schaut es bei einer* Adjazenzliste *aus?*

Zurück zu Professor Hastings! Die graphische Darstellung der Beziehungen allein nützt ihm nicht so viel; wir wollen ihm eine Liste aufbauen, *in welcher Reihenfolge* er denn nun die verschiedenen Kleidungsstücke anzulegen hat. Wir nehmen zunächst einmal eine Art Protokollierung innerhalb der Prozedur `dfs-vertex` vor, die uns anzeigt, wann eine besuchte Ecke denn verlassen wird. Wir sehen gleich, wie uns das weiterhilft. Also:

```
procedure dfs-vertex(v)
  marked(v) := true            {Markiere v als besucht}
  v_next    := adj[v]          {Erste benachbarte Ecke}

  while v_next <> nil do
     if not marked(v_next) then
        dfs-vertex(v_next)
     v_next := succ(v_next)    {Naechste benachbarte Ecke}

  write(v)                     {Das ist neu!}
```

Nun können wir allerdings bei der Tiefentraversierung nicht blind in den DAG hineintapsen, sondern wir werden uns auf Einstiege (s.o.) beschränken. Wichtig dabei ist nur, daß wir dabei alle Ecken des Graphen tatsächlich erreichen, aber das ist klar. Weiterhin erkennt man, daß unter dieser Bedingung ein Einstieg niemals markiert werden kann, es sei denn, man würde direkt über ihn in den DAG einsteigen. Da ein Einstieg ja keinen Vorgänger hat, kann er durch den Kern des Tiefensuchverfahrens niemals erreicht werden. Diese Verantwortlichkeit stecken wir in die übergeordnete Prozedur `dfs` :

```
procedure dfs(G)

  for v in V(G) do
     marked(v) := false

  for v in V(G) do
     if entry(v) then        {Ist v ein Einstieg?}
        dfs-vertex(v)
```

Wir nehmen nun einmal den „Trace“ einer systematischen Tiefentraversierung vor, die – sagen wir mal – mit der Ecke „Hemd“ beginnt. Rechts rausgesetzt protokollieren wir, wann was ausgegeben wird.

```
dfs-vertex(Hemd)
    dfs-vertex(Krawatte)
        dfs-vertex(Jacke)
                                    Jacke
                                    Krawatte
    dfs-vertex(Gürtel)
                                    Gürtel
                                    Hemd
```

Der Teil wäre erledigt. Nun versuchen wir z.B. mal die Ecke „Uhr":

```
dfs-vertex(Uhr)
                                    Uhr
```

Nächster Einstieg: Sagen wir mal „Unterhose":

```
dfs-vertex(Unterhose)
    dfs-vertex(Hose)
        dfs-vertex(Schuhe)
                                    Schuhe
                                    Hose
                                    Unterhose
```

Bleibt noch einer übrig: „Socken":

```
dfs-vertex (Socken)
                                    Socken
```

Gucken wir uns nun die Reihenfolge an, in der die Kleidungsstüusgegeben worden sind, stellen wir fest: *Genau falsch herum!* Anders ausgedrückt: So zieht man sich **aus** und nicht **an**! Wie man das beheben kann, ist offensichtlich.

Dieses Verfahren ist ein ganz wichtiges *Sortierverfahren* und gehörte eigentlich in das Kapitel 4, wenn es sich in der Aufgabenstellung nicht deutlich von den klassischen Verfahren unterschiede: Bei den bekannte Sortierverfahren waren zwei Elemente grundsätzlich miteinander vergleichbar; für zwei Elemente a und b war also immer $a < b$, $a = b$ oder $a > b$. Beim Sortieren von DAGs war das durchaus nicht immer der Fall, z.B. sind die Ecken *Socken* und *Uhr* keineswegs miteinander vergleichbar. Ihre Reihenfolge ist unbestimmt bzw. beliebig. Das hat zur Konsequenz, daß es mehrere mögliche sortierte Reihenfolgen der Ecken eines DAGs geben kann.

Diese Technik nennt man **topologisches Sortieren** (topological sorting). Warum das so heißt, ist schwer zu erklären und ich bin auch nicht ganz sicher. *Topologie* beschäftigt sich als mathematische Disziplin mit den Eigenschaften von Objekten, die nur aus ihrer Lage *zueinander* resultieren, nicht aus ihrer absoluten Lagen in irgendeinem Koordinatensystem. Das wäre die Aufgabe der *Geometrie*.

In unserem Beispiel ist das auch der Fall: Wir wissen nur, welche Aktivitäten vor anderen ausgeführt werden müssen. Wir wissen *nicht*, *wann* diese Aktivitäten denn tatsächlich ausgeführt werden müssen. Das kann man u.U. aus einer topologische Sortierung nachträglich ablesen. Also hier die endgültige Codierung des topologischen Sortierens:

Algorithmus 8.3 *Topologisches Sortieren eines DAGs* `G` *mit einem globalen Keller* `s`.

Precondition: ---
Postcondition: Jede Ecke von G vor allen ihren Nachfolgern ausgegeben

```
procedure toposort(G)

  {S ist ein globaler Keller}
  s := empty-stack

  {Eigentliche Tiefentraversierung}

  for v in V(G) do
     marked(v) := false

  for v in V(G) do
     if entry(v) then      {Ist v ein Einstieg?}
        dfs-vertex(v)

  {Ausgabe vom Keller}
  while not empty(s) do
     write(top(s))
     pop(s)

{-----------------------------------------------------------}

procedure dfs-vertex(v)
  marked(v) := true            {Markiere v als besucht}
  v_next := adj[v]             {Erste benachbarte Ecke}

  while v_next <> nil do
     if not marked(v_next) then
        dfs-vertex(v_next)
     v_next := succ(v_next)    {Naechste benachbarte Ecke}

  push(s,v)                    {Nicht schreiben! Auf den Keller tun!}
```

Es sollte nun klar sein, daß der Keller genau die Funktion hat, die eigentlich falsche Reihenfolge der Ecken umzukehren.

Natürlich kann man Tiefentraversierung auch nicht-rekursiv machen, wenn wir wir uns schon eines expliziten Kellers bedienen. Studieren wir dafür den folgenden Pseudocode als „Ersatz" für `dfs-vertex`:

Algorithmus 8.4 *Tiefentraversierung einer Ecke v mit einem Keller.*

Precondition: ---
Postcondition: Alle Ecken, die von v erreichbar sind, sind markiert

```
procedure dfs_vertex'(v)

   t := empty-stack   {t ist ein lokaler Keller}

   marked(v) := true
   push(t,v)

   while not empty(t) do
      w      := top(t)
      w_next := adj[w]

      while w_next <> nil and marked(w_next) do
         w_next := succ(w_next)

      if w_next <> nil then
         marked(w_next) := true
         push(t,w_next)
      else
         pop(t)
```

Hier wurde der Laufzeitkeller der rekursiven Version durch einen expliziten Keller `t` ersetzt. Traversieren wir in unserem Ankleidebeispiel mal die Ecke *Hemd*, also `dfs-vertex'(Hemd)`.

Zunächst wird **Hemd** markiert und auf den Keller `t` gepusht, bevor wir die äußerste `while`-Schleife betreten. Wir geben den Kellerinhalt so an, daß das oberste Kellerelement immer ganz links steht:

t:	Hemd
markiert:	Hemd

Es wird also in der äußeren `while`-Schleife `w` mit **Hemd** besetzt; nehmen wir an, es stünde **Gürtel** als erstes in der Adjazenzliste von **Hemd** : Es wird markiert und wandert auf den Keller:

t:	Gürtel Hemd
markiert:	Hemd, Gürtel

Nun wird `w` (= `top(t)`) also mit **Gürtel** besetzt. Dort finden wir als einizge (noch unmarkierte) Nachbarecke **Jacke**, die markiert wird und auf den Keller wandert:

t:	Jacke Gürtel Hemd
markiert:	Hemd, Gürtel, Jacke

`w` wird nun also zur **Jacke**, deren Adjazenzliste aber leer ist. Also: `w_next = nil` und die Jacke verschwindet vom Keller:

t:	Gürtel Hemd
markiert:	Hemd, Gürtel, Jacke

In der nächsten Runde kommt also der Gürtel wieder zum Vorschein, dessen einzige Nachbarecke (Jacke) bereits markiert ist, so daß auch der Gürtel vom Keller verschwindet:

t:	Hemd
markiert:	Hemd, Gürtel, Jacke

Also `w` = **Hemd**. Da gibt's einen Nachbarn, der noch nicht markiert ist: **Krawatte**. Wir markieren ihn und tun ihn auf den Keller:

t:	Krawatte Hemd
markiert:	Hemd, Gürtel, Jacke, Krawatte

Die Krawatte liegt oben auf dem Keller (`w` = Krawatte), aber ihr einziger Nachbar Jacke ist schon markiert, so daß sie vom Keller verschwindet:

t:	Hemd
markiert:	Hemd, Gürtel, Jacke, Krawatte

Nun gibt's aber keinen einzigen Nachbarn von **Hemd**, der noch nicht markiert ist, so daß wir es vom Keller nehmen, der damit leer wird. Die Ecken wurden also in folgender Reihenfolge betreten:

Hemd, Gürtel, Jacke, Krawatte

Aufgabe: *Machen Sie sich die verbleibenden Schritte der Tiefentraversierung mit den übriggebliebenen Einstiegen klar. Wann muß in dieser Version für die topologische Sortierung eine Ecke auf den globalen Keller getan werden?*

Nun gibt's aber eine zum Keller „spiegelbildliche" Datenstruktur. Der Keller gehorcht ja der Regel „last in, first out" (LIFO). Im anderen Fall heißt es dann „first in, first out" (FIFO) und das Ding hört auf den Namen *Schlange* (engl. queue), wie wir wissen (s. 1.4.6 und 3.1.3). Es kommt eine ganz andere Reihenfolge heraus, wenn man den Keller durch eine Schlange ersetzt: Die *Breitentraversierung* oder auch *Breitensuche*.

8.2 Breitensuche

Formal entsteht die Breitentraversierung (breadth-first search) aus der Tiefentraversierung dadurch, daß man alle Operationen auf dem Keller `t` durch die entsprechenden Operationen auf einer Schlange ersetzt.

Der Effekt dabei ist, daß das Traversierungsverfahren nicht mehr direkt nach unten im Graphen läuft, sondern – wie wir gleich an einem Beispiel sehen werden – zunächst *alle* Nachbarn der aktuellen Ecke (die also eine Kante entfernt sind) abarbeitet. In der nächsten „Runde" sind dann alle deren Nachbarn dran (zwei Kanten von der Ausgangsecke entfernt) usw.

Bedeutung	Keller-Operation	Schlangen-Operation
Initialisierung	empty-stack	empty-queue
Element einfügen	push	enqueue
Element entnehmen	pop	dequeue
Element anschauen	top	head
Test auf leer	empty	empty

Tabelle 8.1: Unterschiede zwischen Keller und Schlange

Gucken wir uns zunächst mal das an, was oben als einander „entsprechende" Operationen bezeichnet wurde, Tabelle 8.1 zeigt sie.

Die Breitentraversierung sieht dann also genau entsprechend zur Tiefentraversierung aus:

Algorithmus 8.5 *Breitentraversierung einer Ecke* v *mit einer Schlange.*

Precondition: ---
Postcondition: Alle Ecken, die von v erreichbar sind, sind markiert

```
procedure bfs_vertex'(v)

  t := empty-queue   {t ist eine lokale Schlange}

  marked(v) := true
  enqueue(t,v)

  while not empty(t) do
     w      := head(t)
     w_next := adj[w]

     while w_next <> nil and marked(w_next) do
        w_next := succ(w_next)

     if w_next <> nil then
        marked(w_next) := true
        enqueue(t,w_next)
     else
        dequeue(t)
```

Was geschieht, wenn wir mal wieder auf der Ecke **Hemd** einsteigen? Kurzdurchgang:

t: Hemd
markiert: Hemd

Nun kommt's! Der **Gürtel** kommt *nach* dem Hemd in der Schlange zu stehen und nicht *davor*, wie beim Keller!

t: Hemd Gürtel
markiert: Hemd, Gürtel

Die restlichen Schritte bedürfen nun wohl keiner Erläuterung mehr:

t: Hemd Gürtel Krawatte
markiert: Hemd, Gürtel, Krawatte

t: Gürtel Krawatte
markiert: Hemd, Gürtel, Krawatte

t: Gürtel Krawatte Jacke
markiert: Hemd, Gürtel, Krawatte, Jacke

t: Krawatte Jacke
markiert: Hemd, Gürtel, Krawatte, Jacke

t: Krawatte Jacke
markiert: Hemd, Gürtel, Krawatte, Jacke

Hier gibt es also eine andere Reihenfolge:

Hemd, Gürtel, Krawatte, Jacke

Breitentraversierung ist also eine Technik, die sich immer gleich weit von der Ausgangsecke entfernt. Diese Entfernung wird lediglich dann vergrößert, wenn nichts gefunden wurde. Im Gegensatz dazu versucht Tiefentraversierung, sich so schnell und so weit wie möglich von dieser Ecke zu entfernen.

Tiefentraversierung ist gewöhnlich einfach (nämlich rekursiv) zu implementieren, während Breitentraversierung da etwas komplizierter ist. Aus diesem Grund benutzt man auch meistens Tiefentraversierung, wenn nicht ganz gewichtige Gründe dafür sprechen, in der Breite zu suchen.

Einen solchen Grund kann man skizzieren:

> *Man sucht die Verbindung zwischen zwei Ecken im Graphen. Tiefensuche liefert eine entsprechende Kantenfolge, wenn es eine gibt. Breitensuche liefert garantiert die kürzeste!*

8.3 Kürzeste Wege I: Breitensuche

Wir können also Breitensuche einsetzen, um das einfachste Problem der kürzesten Wege in einem Graphen zu lösen. Allgemein formuliert besteht dieses Problem darin, zu zwei Ecken eines Graphen einen Weg (eine Kantenfolge) zu finden, der die kürzeste Verbindung zwischen diesen Ecken darstellt. Ganz allgemein wird man dieses Problem auf gerichteten und gewichteten Graphen zu lösen versuchen, bei denen die Kantengewichte den „Aufwand" des Übergangs zwischen den beiden Ecken darstellen, also z.B. die Entfernung zwischen ihnen.

Der angesprochene einfachste Fall behandelt zunächst ungewichtete Graphen, die allerdings als gewichtete Graphen mit Kantengewichten von 1 aufgefaßt werden können. Lösungen für gewichtete Graphen werden wir in den Kapiteln über Optimierung behandeln, 9.5 und 10.1.

Genauer formuliert: Gesucht ist ein Breitensuchverfahren, das einen Weg von der Ecke s (source) zur Ecke d (drain) findet, der über *die geringste Anzahl von Kanten* verläuft und gleichzeitig diesen Weg so ablegt, daß man ihn hinterher wiederfinden kann.

Man braucht eine kleine Erweiterung des obigen Algorithmus. Die Idee besteht darin, zu jeder Ecke einen Verweis (pred) auf den aktuellen Vorgänger im Pfad speichern. Sobald eine Ecke in die queue wandert, wird diejenige Ecke, die dafür verantwortlich ist, als entsprechender Vorgänger abgespeichert. Am Ende bekommt man dann von der Ecke d ausgehend durch Abspulen des pred-Zeigers den entsprechenden Weg (allerdings genau auf dem Kopf!) wieder heraus.

Algorithmus 8.6 *Breitensuche der Ecke* d *ausgehend von der Ecke* s *mit einer Schlange. Der Weg von* s *zu* d *wird rückwärts über den* pred*-Eintrag rekonstruiert.*

Precondition: s <> nil, d <> nil
Postcondition: Alle Ecken, die von v erreichbar sind, sind markiert

```
procedure bf-search(s,d)
  t          := empty-queue    {t ist eine lokale Schlange}
  marked(s)  := true
  pred(s)    := nil            {s hat keinen Vorgaenger}

  enqueue(t,s)

  while not empty(t) and head(t) <> d do
     w      := head(t)
     w_next := adj[w]

     while w_next <> nil and marked(w_next) do
        w_next := succ(w_next)

     if w_next <> nil then
        marked(w_next) := true
        pred(w_next)   := w        {Vorgaenger merken}
        enqueue(t,w_next)
     else
        dequeue(t)

  if empty(t) then
     {Kein Pfad von s nach d}
```

Kapitel 9

Greedy-Optimierung

Stellen wir uns zur Illustration der Prinzipien von Optimierung ein Problem: Wir brechen nachts in eine Villa ein und stehlen wertvolle Gegenstände. Als informatisch geschulte Einbrecher möchten wir dabei

- soviel Wert mitnehmen, wie wir tragen können und
- so schnell wie möglich wieder raus.

Wir stehen dabei unter folgenden bekannten Randbedingungen:

1. Die Bewohner der Villa sind verreist und wir haben viel Zeit, uns gemütlich umzusehen.
2. Von allen Wertgegenständen kennen wir ihren Wert[1].
3. Wir können allerdings nicht *alles* mitschleppen, sondern nur ein bestimmtes Gewicht. Ansonsten wäre „Nimm alles mit“ ein optimales Verfahren.
4. Die Gewichte kennen wir auch (Handwaage ist immer dabei!).

Dies ist (in krimineller Verkleidung) eins der berühmtesten Optimierungsproblem der Informatik: Das *Rucksack-Problem* (knapsack problem). Allgemein formuliert stellt es sich wie folgt dar:

1. Es ist gegeben eine Menge M von n Gegenstände mit den *Werten* $v_i, i = 1, \ldots, n$ und den *Gewichten* $w_i, i = 1, \ldots, n$. Gesucht ist eine Teilmenge $M' \subseteq M$ mit der Eigenschaft

$$\sum_{i \in M'} w_i \leq w_{max}$$

   d.h. die Summe der Gewichte in M' liegt unter einem gegebenen Gewicht w_{max}.
2. Ferner soll der Wert der Gegenstände in M' so groß wie möglich sein:

$$\sum_{i \in M'} v_i = \text{Max!}$$

[1] Seltsame Annahme! Vielleicht haben die Leute die Preisschilder an den Sachen drangelassen, um ihren Freunden zu imponieren?

Wie gehen wir am besten vor? Die einfachste Idee zuerst:

1. Wir sortieren die Gegenstände in M absteigend nach dem Verhältnis von Wert zu Gewicht, d.h. der Gegenstand i mit dem höchsten Quotienten v_i/w_i bekommt den Index 1.
2. Dann nehmen wir solange „von oben" die Gegenstände an uns, wir wir sie noch tragen können.

Man nennt diese Strategie *gierig*. Im englischen heißt so etwas dementsprechend ein *greedy* Algorithmus. Klappt das immer?

Nein! Hier ist das Gegenbeispiel: Wir haben drei Gegenstände

1. Ein Lehrbuch über Algorithmen. Gewicht $w_1 = 10\,\text{kg}$, Wert $v_1 = 60\,\text{DM}$. Verhältnis 6 DM/kg.
2. Eine Taschenrechner mit eingebautem B-Baum (7.3). Gewicht $w_2 = 20\,\text{kg}$, Wert $v_2 = 100\,\text{DM}$. Verhältnis 5 DM/kg.
3. Eine allegorische Statue, die die ewige Schönheit von Heapsort (4.2) symbolisiert. Gewicht $w_3 = 30\,\text{kg}$, Wert $v_3 = 120\,\text{DM}$. Verhältnis 4 DM/kg.

Tragen können wir nun maximal 50 kg. Sortiert nach relativem Wert haben wir die Sachen nun auch schon. Also greifen wir uns zunächst das Lehrbuch und dann den Taschenrechner. Die Statue lassen wir liegen, weil wir sie nicht mehr tragen können. Der Wert unserer Beute beträgt nun 160 DM, obwohl wir durch den Taschenrechner und die Statue auf 220 DM kämen und beide Sachen auch tragen könnten.

Klappt *Greedy* denn überhaupt einmal? Dazu modifizieren wir unser Problem ein wenig und stellen uns folgende potentielle Beute vor:

1. Ein Haufen Platinstaub. Gewicht 10 kg, Wert 60 DM. Verhältnis 6 DM/kg.
2. Ein Haufen Goldstaub. Gewicht 20 kg, Wert 100 DM. Verhältnis 5 DM/kg.
3. Ein Haufen Silberstaub. Gewicht 30 kg, Wert 120 DM. Verhältnis 4 DM/kg.

Das entscheidende dabei ist nun, daß wir überhaupt nicht darauf angewiesen sind, ganze Haufen einzustecken. Wir können uns anteilig bedienen! Und auf einmal klappt es *greedy*! Wir nehmen uns alles Platin, alles Gold und noch 20 kg vom Silber mit einem Gesamtwert von 240 DM. Man überzeugt sich davon, daß es nicht besser geht!

Eine kleine Modifikation des Problems erlaubt es uns anscheinend, mit einer einfachen Strategie ein optimales Ergebnis zu erzielen. Beide Varianten des Rucksack-Problems haben dementsprechend verschiedene Namen:

1. Das erste Problem heißt **0-1 Rucksack** (engl. 0-1 knapsack), weil für jeden Gegenstand eine Entscheidung getroffen werden muß, ob man ihn liegenläßt (0) oder einpackt (1).
2. Das zweite Problem heißt **gebrochener Rucksack** (engl. fractional knapsack), weil man auch einen Bruchteil eines „Gegenstandes" einpacken kann.

Salopp gesprochen: Der gebrochene Rucksack läßt sich *greedy* optimieren, der 0-1 Rucksack nicht. Da wird man sich nach einem anderen Verfahren umschauen müssen.

9.1 Allgemeiner Aufbau von Greedy-Algorithmen

Zunächst betrachten wir einmal die Arbeitsweise ein Greedy-Algorithmus recht abstrakt. Normalerweise haben wir folgendes zur Verfügung:

1. Eine Menge von *Kandidaten* c, aus denen wir die Lösung konstruieren wollen.
2. Eine Teilmenge $s \subseteq c$, die wir bereits ausgewählt haben.
3. Eine boolesche Funktion *solution*, die uns sagt, ob eine Menge von Kandidaten eine *legale* Lösung des Problems darstellt, unabhängig davon, ob diese Lösung optimal ist.
4. Eine Testfunktion *feasible*, die uns sagt, ob eine gewisse Teillösung u.U. zu einer kompletten legalen Lösung erweitert werden kann.
5. Eine *Auswahlfunktion select*, die uns denjenigen noch unbenutzten Kandidaten liefert, der im Sinne der Greedy-Strategie der erfolgsversprechendste ist.
6. Eine *Zielfunktion val*, die uns den Wert einer gewissen Lösung angibt.

Die Greedy-Strategie startet mit einer leeren Lösungsmenge und erweitert sie schrittweise um das höchstwertige passende Element der Kandidatenmenge:

```
function greedy(c) : set
  {c ist die gesamte Menge der Kandidaten}
  {Gesucht ist eine "optimale" Teilmenge von c}
  {Die wird in die Loesungsmenge s hineinkonstruiert}

  s := emptyset          {Loesungsmenge ist am Anfang leer}

  while not solution(s) and c <> emptyset do
     x := {dasjenige Element von c, das select maximiert}

     c := c - {x}       {x wird nur einmal angefasst und}
                        {aus c unwiderruflich entfernt}

     if feasible(s + {x}) then      {Geht es?}
        s := s + {x}                {Geht! Es bleibt}
                                    {auf ewig in s drin!}

  if solution(s) then
     return s
  else
     {Es gibt keine Loesung!}
```

Am Ende liefert `val(s)` den Wert der gefundenen Lösung. Daß diese Lösung den Wert von `val` tatsächlich optimiert, ist die wesentliche Eigenschaft greedy lösbarer Probleme.

Einige Optimierungsprobleme auf Graphen lassen sich greedy lösen, am bekanntesten sind die Algorithmen von Kruskal und Dijkstra, s. [Maas, 1993]. Wir werden uns beide hier anschauen.

9.2 Minimalgerüste

Das Problem, zu einem gegebenen zusammenhängenden gewichteten Graphen G ein Minimalgerüst zu finden, also einen Baum,

1. dessen Kanten alles Kanten von G sind,
2. der alle Ecken von G miteinander verbindet und
3. bei dem die Summe der Kantengewichte minimal ist,

kann mit dem Verfahren von Kruskal greedy gelöst werden.

Das Verfahren beruht auf der Idee, die Kantenmenge von G einmal aufsteigend zu sortieren und dann jede Kante dann in den anfangs leeren Baum einzusetzen, wenn sich dadurch kein Kreis ergibt. Es entsteht dann das Bild, daß sich in der Eckenmenge von G kleine Bäume bilden, die sich durch eine solche Kante zu einem größeren verbinden, bis schließlich nur ein einziger Baum — eben das Minimalgerüst — vorhanden ist. Zur Theorie konsultiere man [Maas, 1993].

Das einzig interessante bei der Implementation dieses Verfahrens besteht offenbar darin, eine Organisation der einzelnen Bäume zu finden, die wir schließlich zu einem einzigen zusammenwachsen lassen wollen. Diese Bäume stellen wir am einfachsten dadurch dar, daß wir die Ecken, die zu ihnen gehören, in ein System von *disjunkten Mengen* speichern, das wir lediglich in geeigneter Weise verwalten müssen.

9.2.1 Datenstrukturen für disjunkte Mengen

Zunächst sollten wir festlegen, was unter einem System disjunkter Mengen zu verstehen ist und welche Schnittstelle der entsprechende ADT aufweisen soll.

Unter einem System disjunkter Mengen verstehen wir eine Sammlung von Mengen

$$\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$$

derart, daß $S_i \cap S_j = \emptyset$, falls $i \neq j$.

Diese Mengen stellen wir uns dynamisch vor, so daß wir sie identifizieren und verändern können.

Die Identifikation einer solchen Menge S_x geschieht durch einen *Repräsentanten* $x \in S_x$. Das ist eindeutig, weil ein Element ja nur in einer dieser Mengen sein kann. Gerechnet wird also immer nur auf einem solchen Repräsentanten.

Man steht nun vor dem Problem, daß man irgendein Element einer Menge gegeben hat und daraus diese Menge bestimmen möchte. Das heißt natürlich, daß man den Repräsentanten dieser Menge haben möchte.

Insgesamt ergibt sich folgende Schnittstelle des ADT:

`make-set(x)` erzeugt eine neue Menge, deren einziges Element x ist. x ist damit auch automatisch der Repräsentant dieser Menge. Es muß — als Vorbedingung — sichergestellt sein, daß x nicht in einer anderen Menge enthalten ist (sonst wären die Mengen ja nicht disjunkt!)

`union(x,y)` bildet *destruktiv* die Vereinigung der Mengen, die durch x und y repräsentiert werden. Der Repräsentant der entstehenden Vereinigung ist wieder ein Element der entstehenden Vereinigungsmenge, aber es muß nicht x oder y sein. Die alten Mengen verschwinden aus der Struktur und werden durch die Vereinigung ersetzt.

`find-set(x)` liefert (einen Zeiger auf) den Repräsentanten der Menge, in der x liegt.

Natürlich kann man einen solchen ADT mit etwas Sorgfalt z.B. durch verkettete Listen realisieren. Eine sehr interessante Alternative der Implementation besteht aber darin, daß man die Elemente einer solchen Menge in einen Baum zu packen, der nach oben verkettet ist. Also: Jedes Element zeigt auf seinen Vorgänger im Baum; die Wurzel dient als Repräsentant.

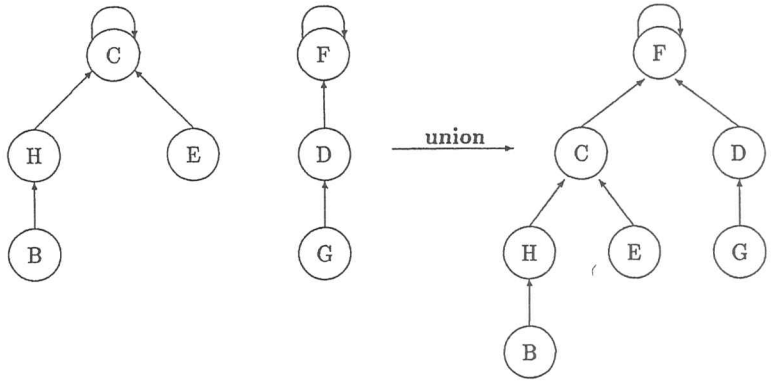


Abbildung 9.1: Darstellung disjunkter Mengen durch Bäume

Man beachte, daß es sich hierbei nicht um einen Binärbaum handeln muß, obwohl die Abbildung 9.1 das suggeriert.

Die Vereinigung zweier Mengen bekommt man dadurch, daß man die Wurzel des einen Baums einfach auf die Wurzel des anderen zeigen läßt, wie in Abbildung 9.1 dargestellt. Diese Konstruktion nennen wir einen *Wald von disjunkten Mengen* (DM-Wald, engl. disjoint set forest).

Das Auffinden des Repräsentanten ist dann offenbar ein $O(h)$ Problem, wobei h die Höhe des Baums ist. Wir klettern ja von einem Knoten immer nach oben. Man kann allerdings mit einem Kunstgriff den Baum immer sehr flach halten.

Die Idee ist einfach: Wann immer man mittels `find-set` den Repräsentanten eines Elements der Menge bestimmt, tut man das ja automatisch auch für alle anderen Elemente des entsprechenden Asts. Das kann man dadurch festhalten, daß man alle diese Elemente sofort auf diesen Repräsentanten zeigen läßt. Dies ist natürlich kein Verfahren, das uns eine gewisse maximale Höhe *garantiert*, aber es führt in der Praxis zu einer drastischen Reduktion der Komplexität. So eine Verbesserung nennen wir eine *Heuristik*, weil ihr Effekt nur schwer abzuschätzen ist. Diese hier heißt *Pfadkompressions-Heuristik* (engl. path compression heuristic). Man überzeugt sich, daß es einfach zu programmieren ist.

Algorithmus 9.1 *Finden des Repräsentanten für das Element* x *in einem DM-Wald. Der entsprechende Pfad wird parallel komprimiert.*

Precondition: ---
Postcondition: ---

```
function find-set(x)
  if x <> parent(x) then
     parent(x) := find-set(parent(x))
     return parent(x)
  else
     return x
```

Eine weitere Heuristik für DM-Wälder besteht darin, bei der Vereinigung zweier Mengen die Höhe der beiden Bäume in Betracht zu ziehen. Das ist nun ein wenig komplizierter, so daß wir hier darauf verzichten wollen.

9.2.2 Der Kruskal-Algorithmus

Also: Wir brauchen

1. Eine Liste von *Kanten*, aus der wir *gierig* immer die mit dem kleinsten Gewicht herausgreifen.
2. Ein System *disjunkter Mengen* von *Ecken*, die wir sukzessive miteinander vereinigen.

Hier also ganz schnell der Pseudocode für den Kruskal-Algorithmus, wobei wir eine kleine Modifikation gleich mit vornehmen: Wenn unser DM-System nur noch aus einer einzigen Menge besteht, brauchen wir offenbar nicht mehr weiterzumachen, der MST ist fertig. Das kann bei einem sehr *dichten* Graphen eine ganze Menge ausmachen:

Algorithmus 9.2 *MST-Algorithmus nach Kruskal.* tree *ist eine Menge von Kanten.*

Precondition: ---
Postcondition: tree enthält die Kanten eines MST

```
procedure kruskal(G; var tree)
  tree := emptyset          {Kanten des MST}

  for v in V(G) do
     make-set(v)

  nsets := |V(G)|           {Anzahl disjunkter Mengen}
  edges := E(G)

  sort(E(G))                {aufsteigend nach Gewicht}

  {Greedy loop}
  while edges <> emptyset and nsets > 1 do
```



```
     (u,v) := {Naechste Kante aus edges}

     {Die Kante (u,v) wurde gierig ausgewaehlt}
     {Wenn sie zwei unterschiedliche Baeume verbindet:}
     {1. Als neue Kante in den MST packen}
     {2. Die beiden Baeume zu einem zusammenfassen}

     if find-set(u) <> find-set(v) then
        {Verschiedene Baeume!}
        tree := tree + (u,v)
        union(u,v)
```

Beispiel her! Man betrachte den Graphen in Abbildung 9.2.

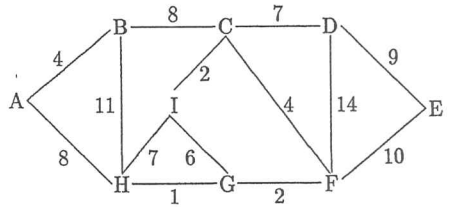


Abbildung 9.2: Beispielgraph für den Kruskal-Algorithmus

Hier im Kurzdurchgang, was so alles passiert: Nach der Initialisierung des Systems der disjunkten Mengen, haben wir also *neun* disjunkte Mengen. Wir schreiben das System etwas kompakter, indem wir die einzelnen disjunkten Mengen durch ein Komma trennen und die einzelnen Elemente einfach hintereinanderschreiben. Also:

DM-Wald: A,B,C,D,E,F,G,H,I
MST: $\emptyset$ (Länge 0)

Die Kantenliste haben wir auch schnell sortiert:

(G,H)	(F,G)	(C,I)	(A,B)	(C,F)	(G,I)	(H,I)
(C,D)	(A,H)	(B,C)	(D,E)	(E,F)	(B,H)	(D,F)

Der Anfang der Geschichte besteht nun darin, daß wir die erste Kante (G,H) nehmen, feststellen, daß G und H in verschiedenen Mengen liegen, die Kante in den Baum einfügen und die Mengen, in denen G und H liegen, vereinigen:

DM-Wald: A,B,C,D,E,F,GH,I
MST: (G,H) (Länge 1)

Die nächsten Kanten (F,G), (C,I), (A,B) und (C,F) gehen genauso in den MST hinein, wie man sich durch eine ausführliche Handsimulation überzeugt. Danach sieht die ganze Situation wie folgt aus:

DM-Wald: AB,D,E,CFGHI
MST: (G,H) (F,G) (C,I) (A,B) (C,F) (Länge 13)

Nun sind's gerade mal noch vier disjunkte Mengen. Der Effekt ist, daß nun die ersten Fehlversuche auftreten. Die Kanten (G,I) und (H,I) werden zurückgewiesen, weil sie schon

ein einem gemeinsamen Baum liegen. Lediglich (C,D) wird in den MST eingefügt und führt zur Vereinigung der entsprechenden Bäume:

DM-Wald: AB,CDFGHI,E
MST: (G,H) (F,G) (C,I) (A,B) (C,F) (C,D) (Länge 20)

Nächster Schritt! (A,H) geht als nächste Kante rein:

DM-Wald: ABCDFGHI,E
MST: (G,H) (F,G) (C,I) (A,B) (C,F) (C,D) (A,H) (Länge 28)

Die letzte Kante, die zum MST gehört, ist (D,E) und damit ist das Verfahren beendet:

DM-Wald: ABCDFGHIE
MST: (G,H) (F,G) (C,I) (A,B) (C,F) (C,D) (A,H) (D,E) (Länge 37)

Die Komplexität des Kruskal-Verfahrens ist in dieser Implementation sehr schwer genau zu bestimmen. Der Grund liegt darin, daß wir keine gescheite Abschätzung für die Kosten eines `find-set` haben, weil da die Pfadkompressions-Heuristik mitspielt, die uns bei jedem Zugriff den entsprechenden Baum optimiert. Was wir wissen ist immerhin folgendes:

1. Die Konstruktion der $|V|$ disjunkten Teilmengen zur Initialisierung der einzelnen Bäume wird in $O(|V|)$ Schritte erledigt.
2. Wir sortieren die Kanten in $O(|E| \log |E|)$ Schritten.
3. Die äußerste Schleife — der *greedy* loop — wird $O(|E|)$ mal durchlaufen. Die Verbesserung durch das Mitzählen der Bäume (= disjunkten Mengen) ist rein praktischer Natur und geht in die Analyse nicht ein!
4. Wir machen also $O(|E|)$ mal das fragliche `find-set` und auch — im schlechtesten Fall — $O(|E|)$ mal `union`.

`union` ist eine $O(1)$ Operation, wie man sich überzeugt, wenn man am Anfang des Kapitels nochmal nachguckt. Was ist aber mit `find-set`? Man kann sich überlegen, daß seine Komplexität irgendwo zwischen $O(1)$ und $O(n)$ für eine Menge mit n Elementen liegt. Wir machen in der Praxis keinen großen Fehler, wenn wir diese Operation als $O(1)$ (!) annehmen. Eine genauere Analyse der Komplexität von `find-set` findet sich in [Cormen *et al.*, 1990].

Im *greedy*-loop haben wir also einen Zeitbedarf von $O(|E|)$, was gegen den Sortieraufwand verschwindet. Bleiben wir also bei der Summe

$$O(|V| + |E| \log |E|).$$

9.3 Scheduling-Probleme

Eine praktisch hochinteressante Anwendung von Greedy-Algorithmen besteht in der Lösung eines speziellen Falls des sogenannten *Scheduling-Problems* (Planungsproblem, aber dieser Ausdruck ist nicht sehr genau). Man kann sich dieses Scheduling-Problem an einem Beispiel klarmachen:

In einer Fabrik steht *eine einzige* wertvolle *Maschine* (server). Auf dieser Maschine sollen *Aktivitäten* ablaufen, von denen wir annehmen, daß sie alle gleichlange brauchen, um fertig zu werden. Für uns heißt das, daß wir vereinfachend annehmen, daß diese Zeit immer gleich 1 ist. Die Gemeinheit ist nur, daß jede dieser Aktivitäten A_i zwei Eigenschaften hat:

1. Wenn A_i bearbeitet worden ist, kann das entstandene Produkt für einen Betrag w_i verkauft werden.
2. Aber *nur*, wenn A_i spätestens zu einem Zeitpunkt t_i fertig wird. Ansonsten bekommen wir gar nichts dafür.

Die zweite Eigenschaft nennt man einen *Endtermin* (engl. *deadline)*, das hier betrachtete Problem ist somit ein *Scheduling-Problem mit konstanter Bearbeitungszeit, einem Server und deadlines*. Unsere Aufgabe besteht nun darin, die Aktivitäten so auf unsere Maschine zu verteilen, daß der erzielte Erlös maximal wird.

Hier ist ein kleines Beispiel:

Aktivität	1	2	3	4
w_i	50	10	15	30
t_i	2	1	2	1

Betrachten wir erstmal alle legalen Arbeitsfolgen an und bestimmen den optimalen Plan manuell:

Arbeitsfolge	**Erlös**
1	50
2	10
3	15
4	30
1,3	65
2,1	60
2,3	25
3,1	65
4,1	80 ← *Optimum!*
4,3	45

Aufgabe: *Machen Sie sich klar, warum die Folge 3,2 hier gar nicht auftritt. Sie ist illegal. Warum?*

Wir wollen also eine Menge von Aktivitäten konstruieren, mit denen wir unseren Erlös maximieren können. Das Problem ist lediglich, daß wir etwas mehr wollen. Wir wollen nicht nur wissen, *welche* Aktivitäten auszuführen sind, sondern auch *in welcher Reihenfolge*. Hier kann man das aus der Menge $\{1, 4\}$ nötigenfalls mit der Hand die Reihenfolge bestimmen, wenn die Menge allerdings mehr als 100 Elemente enthält, ist das auch nicht mehr ganz trivial.

Eine Menge s von Aktivitäten ist in Ordnung im Sinn von *feasible* (s. 9.1), wenn es eine Reihenfolge der Aktivitäten in s gibt, die es erlaubt, alle Aktivitäten vor ihrem Endtermin fertigzustellen.

Die Auswahlfunktion *select* besteht lediglich darin, die Aktivität *mit dem höchsten Erlös* aus der Kandidatenmenge zu entnehmen.

In diesem Beispiel würde man also zunächst die Aktivität 1 auswählen. Danach schauen wir uns gleich die Nummer 4 an, weil w_4 den größten Wert im Rest hat. Schnell (?) stellen wir fest, daß es eine Folge gibt, mit der die Menge $\{1,4\}$ innerhalb der Zeit abgearbeitet werden kann. Die nächste Menge $\{1,3,4\}$ (w_3 ist der größte Wert im Rest) geht nicht mehr: Aktivität 3 wird zurückgewiesen. Der Aktivität 2 geht es auch nicht besser, so daß wir mit der Menge $\{1,4\}$ übrigbleiben.

Nun denkt man sich, daß die Prüfung einer Menge von Aktivitäten M mit k Elementen auf Legalität darauf hinausläuft, daß man alle $k!$ Permutationen prüfen muß, ob sie eine korrekte Abarbeitungsfolge ergeben. Zum Glück reicht es aber aus, die Aktivitäten nach ihrem Endtermin zu sortieren.

Aussage 9.1 *Angenommen eine Menge M von k Aktivitäten. Sie ist legal* genau dann, wenn *die nach Endterminen sortierte Folge $\sigma = \langle s_1, s_2, \ldots, s_k \rangle$ legal ist.*

Diese Folge σ hat also die Eigenschaft $t_1 \leq t_2 \leq \ldots \leq t_k$.

Beweisidee: Eine Richtung ist klar: Wenn die sortierte Folge legal ist, ist die gesamte Menge sicherlich legal; denn es gibt ja wenigstens eine solche Folge. Die andere Richtung erfordert etwas Überlegung.

Nehmen wir also an, es gäbe eine Sortierung der Aktivitäten ρ in M, die legal ist. Es ist zu zeigen, daß dann auch die oben definierte Folge σ legal ist. Das machen wir, indem wir ρ in mehreren Schritten in σ umformen und dabei immer legal bleiben. Also gucken wir uns diese legale Folge

$$\rho = \langle r_1, r_2, \ldots, r_k \rangle$$

einmal an. Sie soll nicht gleich σ sein, sonst wären wir schon fertig. Also gibt es einen kleinsten Index a, in dem sich σ und ρ unterscheiden, also $s_a \neq r_a$ und $s_i = r_i, \forall i < a$. Nun suchen wir uns in ρ die Aktivität r_b raus, die s_a in σ entspricht, also $r_b = s_a$. Offenbar ist $b > a$. Nun kann der Endtermin von r_a nicht vor dem Endtermin von s_a liegen, denn dann wäre σ nicht sortiert und r_a müßte an der Stelle a in σ liegen. Also gilt $t_{r_a} \geq t_{s_a} = t_{r_b}$. Skizze:

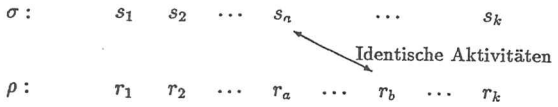


Abbildung 9.3: Zwei legale Aktivitätsfolgen

Da nun ρ eine legale Folge sein soll, kann man r_a mit r_b vertauschen; denn r_a muß nicht vor r_b fertig werden und r_b darf man sicherlich auch früher als in ρ geplant ausführen. Es ergibt sich durch diese Vertauschung die modifizierte Folge

$$\rho' = \langle r_1, r_2, \ldots, r_b, \ldots, r_a, \ldots, r_k \rangle,$$

die nun in den Positionen 1 bis a übereinstimmen. Nun wiederholt man das ganze Verfahren mit σ und ρ' und der erste Index mit verschiedenen Aktivitäten wird nun hinter a liegen. Das machen wir solange, bis $\rho^{(n)} = \sigma$ ist. Damit ist aber alles bewiesen, denn alle Vertauschungen haben die Legalität der entstehenden Folge beibehalten.

Nun die Frage, wie man zeigt, daß die von der Greedy-Strategie zusammengestellte Menge (nicht Folge!) von Aktivitäten in der Tat optimal ist. Nehmen wir also zwei Dinge an:

1. Der Greedy-Algorithmus hat die Menge $\mathcal{I}$ zusammengestellt.
2. Es gibt eine Menge $\mathcal{J} \neq \mathcal{I}$, die legal und optimal ist.

Als erstes nehmen wir uns je eine legale *Folge* aus $\mathcal{I}$ und $\mathcal{J}$, genannt $F_\mathcal{I}$ und $F_\mathcal{J}$, die muß es ja in beiden geben. Dann formen wir diese Folgen so um, daß gemeinsame Aktivitäten zum gleichen Zeitpunkt ausgeführt werden. Das kann bedeuten, daß unter Umständen Lücken in den Folgen auftreten können. Skizze:

$$\begin{array}{lcccccccc} F_\mathcal{I}: & \mathrm{p} & \mathrm{y} & \mathrm{q} & \mathrm{x} & \mathrm{r} & & & \\ F_\mathcal{J}: & \mathrm{r} & \mathrm{s} & \mathrm{t} & \mathrm{p} & \mathrm{u} & \mathrm{v} & \mathrm{q} & \mathrm{w} \end{array}$$

Nach dieser Umformung entstehen also zwei neue legale Folgen $F'_\mathcal{I}$ und $F'_\mathcal{J}$, wobei $\square$ eine Lücke darstellt:

$$\begin{array}{lcccccccc} F'_\mathcal{I}: & \mathrm{x} & \mathrm{y} & \square & \mathrm{p} & \mathrm{r} & \square & \mathrm{q} & \square \\ F'_\mathcal{J}: & \mathrm{u} & \mathrm{s} & \mathrm{t} & \mathrm{p} & \mathrm{r} & \mathrm{v} & \mathrm{q} & \mathrm{w} \end{array}$$

Zunächst stellt man fest, daß es nicht möglich ist, daß in $F'_\mathcal{J}$ eine Lücke auftritt, wenn an dieser Stelle in $F'_\mathcal{I}$ eine Aktivität A geplant ist. Diese Aktivität A kann in $F'_\mathcal{J}$ nicht vorkommen, sonst stünde sie ja an dieser Stelle. Also wäre es eine neue Aktivität, die man getrost anstelle der Lücke einschieben könnte. Das würde ja den Erlös erhöhen. Insgesamt wäre also $\mathcal{J}$ gar nicht optimal!

Umgekehrt geht's allerdings auch nicht! Gäbe es eine Aktivität B in $F'_\mathcal{J}$ und an dieser Stelle stünde in $F'_\mathcal{I}$ eine Lücke, so wäre die Menge $\mathcal{I} \cup B$ legal und der Greedy-Algorithmus hätte B auch auswählen müssen.

> **Erste Zusammenfassung:** Entweder stehen in den beiden Folgen zwei Lücken oder zwei Aktivitäten übereinander. Lücke und Aktivität geht nicht! *Die oben dargestellte Situation kann es also schon gar nicht geben!*

Also nehmen wir an, es gäbe zwei Aktivitäten A aus $\mathcal{I}$ und B aus $\mathcal{J}$, die in unserer Anordnung untereinander stehen und $A \neq B$. Es sind drei Fälle zu unterscheiden:

1. $w_A > w_B$: Dann könnte man A statt B in $\mathcal{J}$ einsetzen und damit die Erlöse in $\mathcal{J}$ erhöhen. $\mathcal{J}$ wäre also gar nicht optimal.
2. $w_A < w_B$: Dann hätte der Greedy-Algorithmus anstelle von A die Aktivität B gewählt, denn

- B wäre durch die Auswahlfunktion *vor* A betrachtet worden, denn sie ist wertvoller.
- B wäre auch gewählt worden, denn die entstandene Menge $(\mathcal{I} \setminus A) \cup B$ ist legal.

3. Es bleibt lediglich die Möglichkeit $w_A = w_B$ übrig.

Zweite Zusammenfassung: In beiden Folgen stehen an einer Stelle entweder
- zwei Lücken,
- zwei identische Aktivitäten, oder
- zwei Aktivitäten mit gleichem Wert.

Damit ist aber alles klar; denn $\mathcal{I}$ und $\mathcal{J}$ haben den gleichen Gesamtwert und da $\mathcal{J}$ optimal ist, ist es $\mathcal{I}$ auch.

Nun können wir uns an die Codierung des Verfahrens machen. Natürlich nutzen wir die Eigenschaft aus, die wir weiter oben festgestellt haben und halten die Lösung gleich sortiert nach Endterminen zusammen. Der Effekt ist, daß wir damit auch gleich eine legale Ausführungsreihenfolge bekommen.

Was wir nur brauchen, ist eine Liste der Endtermine, die nach absteigenden Werten der entsprechenden Aktivitäten sortiert ist. Diese Liste nennen wir `t` und realisieren es als Array. Wir nehmen ferner an, daß wir n Aktivitäten zu verplanen haben. Die Anzahl der effektiv verplanten Aktivitäten sei dann k und die Aktivitäten — genauer: die Nummern der Aktivitäten — liegen dann in einem Array namens `s`. Grobversion:

```
procedure greedy_scheduler(t,n; var s; var k)
  k    := 1          {Erste Aktivitaet ist immer in der Loesung!}
  s[1] := 1

  for i := 2 to n do
     if {Aktivitaet i kann noch genommen werden} then
        {Schaffe den Platz fuer i in der sortierten}
        {Folge der Loesungsaktivitaeten s}
        {i in s eintragen}
        k := k + 1
```

Daß die erste Aktivität immer in der Lösung enthalten sein muß, kann man sich systematisch klar machen.

Das einzig schwierige ist nun, wie man feststellt, ob die Aktivität i noch in die Lösung übernommen werden kann. Wir gehen als *Invariante* immer davon aus, daß `s` nach Endterminen sortiert ist. Ein Beispiel ($k = 5$):

j:	1	2	3	4	5
t[s[j]]:	1	3	5	6	7

Das ist offenbar legal, denn alle Aktivitäten werden nicht später als ihr Endtermin fertig. Formal: `t[s[j]]` $\geq$ `j`.

Nun finden wir beispielsweise als nächsten Kandidaten i eine Aktivität mit dem Endtermin `t[i]` $= 4$.

Offenbar muß diese Aktivität an eine der Positionen 1 bis 4 gespeichert werden, damit sie rechtzeitig fertig wird. Gleichzeitig muß sichergestellt werden, daß die Aktivitäten in `s`, die dann hinter i zu stehen kommen, auch tatsächlich um eine Stelle nach rechts dürfen. Sonst kämen die ja zu spät.

Also läuft man von hinten durch `s` hindurch und sucht einen Index, ab dem man alles um eine Stelle nach rechts verschieben dürfte:

```
index := k
while t[s[index]] > index do
   index := index - 1

{assert: t[s[index]] <= index}
```

Die Aktivitäten ab Position `index` $+ 1$ dürfen also um eine Einheit nach rechts geschoben werden. Das terminiert natürlich fehlerfrei nur in dem Fall, in dem wenigstens eine Aktivität in `s` ist, die nicht mehr verschoben werden darf. Das muß aber gar nicht sein, u.U. kann man bereits die erste Aktivität noch um eine Zeiteinheit auf später verschieben! Also führt man ein *Wächter-Element* (engl. sentinel) ein, der das sicherstellt: `s[0]` mit dem Wert 0.

Man prüft im Beispiel, daß hier als `index` der Wert 1 rauskäme. Man könnte alle Aktivitäten ab Position 2 um eine Position nach rechts schieben:

j:	1	2	3	4	5	6
t[s[j]]:	1	$\triangle$	3	5	6	7

Prinzipiell dürfte i an die durch $\triangle$ gekennzeichnete Stelle rein, aber dann wäre `s` nicht mehr richtig sortiert, weil der einzufügende Wert 4 wäre. Um den Punkt zu finden, an den i unter Beibehaltung der Sortierung eingetragen werden dürfte, nehmen wir eine ähnliche Schleife:

```
index := k
while t[s[index]] > t[i] do
   index := index - 1

{assert: t[s[index]] <= t[i]}
```

Im Beispiel würde diese Schleife mit `index` $= 2$ terminieren und i könnte wieder an die Position `index + 1` $= 3$, also sähe es folgendermaßen aus:

j:	1	2	3	4	5	6
t[s[j]]:	1	3	$\triangle$	5	6	7

Nun wollen wir aber *beides* haben: Sortierung und Legalität! Also muß auch *beides* beim Abbruch der Schleife gelten:

```
index := k
while t[s[index]] > t[i] and t[s[index]] > index do
   index := index - 1
```

Nun müssen wir noch gucken, ob i tatsächlich an die Position `index + 1` gesetzt werden darf. Dazu müssen zwei Bedingungen erfüllt sein:

1. *Sortierung*: `t[s[index]] <= t[i]`
2. *Legalität*: `t[i] > index`

Damit ist nun aber alles klar:

Algorithmus 9.3 *Planung von n Aktivitäten gleicher Dauer mit dem Greedy-Verfahren.*

Precondition: `t[0:n]` ist absteigend sortiert
Postcondition: `s[0:n]` enthält die Lösung

```
procedure greedy_scheduler(t,n; var s; var k)
  t[0] := 0                {Zwei Waechter}
  s[0] := 0

  k    := 1                {Erste Aktivitaet ...}
  s[1] := 1                {... ist immer in der Loesung}

  for i := 2 to n do       {Restliche Kandidaten testen}

     {Waere die Erweiterung um i legal?}
     index := k
     while t[s[index]] > t[i] and t[s[index]] > index do
        index := index - 1

     {i sollte nun auf die Position index+1}
     {Ist das in Ordnung?}

     if t[s[index]] <= t[i] and t[i] > index then
        {Platz fuer i auf Position index+1 schaffen}
        {Elemente oberhalb davon hochschieben}

        for j := k downto index + 1 do
           s[j+1] := s[j]

        {i eintragen}
        s[index+1] := i
        k          := k + 1
```


9.4 Datenkompression

Datenkompression ist ein Thema für alle Leute, deren Festplatte zu klein ist[2] oder die Daten über eine Leitung versenden wollen. Es geht dabei darum, eine möglichst platzsparende Speicherung der Daten vorzunehmen, aus der der Originalzustand fehlerfrei rekonstruiert werden kann. Das bedeutet, daß ein brauchbarer Kompressor in zwei Phasen arbeiten muß:

Kompression: Ein Prozeß, der die Ausgangsdaten in eine andere Form überführt, die signifikant weniger Platz verbraucht.

Expansion: Ein Prozeß, der aus der komprimierten Form die Ausgangsdaten rekonstruiert.

Grundlagen

Am einfachsten werden die ersten Überlegungen, wenn man zwei extreme Fälle von Originaldaten betrachtet[3]:

1. Der Text besteht aus 256000 Zeichen, wobei jedes der 256 möglichen Zeichen genau 1000-mal auftritt.
2. Der Text besteht aus 256000-mal dem Zeichen „a".

Beide Originaldaten nehmen $256000 \times 8 = 2048000$ Bit Plattenplatz weg, also eine ganze Menge. Denkt man nun daran, daß ein Bit nichts weiter darstellt als eine Entscheidung zwischen zwei gleichwahrscheinlichen Ereignissen, so erkennt man schnell, daß

- im ersten Fall, in dem ja alle 256 „Ereignisse" — Auftreten eines bestimmten Zeichens — gleichwahrscheinlich sind, also die vollen 8 Bit wirklich benötigen und daß
- im zweiten Fall nur ein Ereignis auftreten kann, so daß man bei der Kompression im Idealfall damit auskäme, anzugeben *welches* Zeichen (das a) *wie oft* auftritt (256000-mal). Das kann man allerdings mit 30 Bit codieren, wie man ausprobiert.

In der Tat ist es so, daß man im ersten Fall keine signifikante Kompression erzielen kann, im zweiten Fall eine ziemlich große. Auf diesen Betrachtungen beruht das wesentlich Prinzip vieler Kompressionsverfahren: Man nutzt die Tatsache aus, daß einige Zeichen häufiger vorkommen als andere.

Codierungen

Um diesen Umstand nun zur Einsparung des Platzes verwenden zu können, wird man eine Codierung vornehmen, die dem gerecht wird. Die naheliegende Codierung bedient sich dabei (wie jede Codierung auf Computern) der *Binärdarstellung* von Daten als Folgen von 0 und 1. Der für Textdateien fast durchgängig verwendete ASCII-Code (American Standard Code for Information Interchange) codiert jedes Zeichen mit einer *festen* Länge von 8 Bit und das ist natürlich nicht so gut. Man wird eher versuchen, häufig auftretende

[2] ...also für alle, die eine Festplatte besitzen!

[3] Wir gehen dabei grundsätzlich davon aus, daß wir es mit Texten zu tun haben, wobei ein Zeichen mit einem Byte, also acht Bit, codiert ist.

Zeichen mit wenig Bits aber dafür selten auftretende Zeichen mit mehr Bits zu codieren. So eine Codierung nennt man *variabel*.

Eine variable Codierung soll also den Ausdruck

$$\sum_{i=1}^{n} l(c_i)f(c_i)$$

minimieren, wobei n die Anzahl der vorkommenden Zeichen, $l(c_i)$ die Länge der Codierung des Zeichens c_i und $f(c_i)$ die Häufigkeit des Auftretens von c_i im Text ist.

Dann steht man natürlich vor einem Problem: Woher weiß man beim Expandieren, *wie lang* den nun der Code des nächsten Zeichens ist? Das kann bei ungeschickter Wahl des Codes nicht mehr festzustellen sein. Beispiel: Ein Text mit 6 Zeichen nach [Cormen *et al.*, 1990]:

	a	b	c	d	e	f
Häufigkeit	45	13	12	16	9	5
ASCII-Code	01100001	01100010	01100011	01100100	01100101	01100110
Code 1	0	011	100	101	0011	1100
Code 2	0	101	100	111	1101	1100

Die beiden unteren Codierungen sind variabel und haben bereits die Eigenschaft, daß seltene Zeichen längere Codierungen als häufige Zeichen haben. Für die Expansion ist Code 1 allerdings ziemlich unbrauchbar, wie man z.B. an 001100 sieht.

Aufgabe: *Was ist an dieser Eingabe so unschön?*

Code 1 ist also nur dann zu expandieren, wenn man weiß, wie lang die nächste Folge von Binärziffern ist. Es hat natürlich keinen Sinn, die Längen mitzuspeichern. Daß das aber nicht so schlimm ist, zeigt der Code 2, bei dem das seltsamerweise *nicht* passieren kann. Die Eigenschaft, ohne explizite Längenangabe expandierbar zu sein, kann man der gewählten Codierung sofort ansehen:

Definition 9.1 *Eine Codierung heißt* **präfix-frei**, *wenn kein Code Anfangsstück (Präfix) eines anderen Codes ist.*

Eine solche präfix-freie Codierung entspricht einem binären Entscheidungsbaum, dessen Blätter genau die zu codierenden Zeichen sind. Für Code 2 ist dieser Baum in Abb. 9.4 zu sehen, wobei eine 0 immer für den linken und eine 1 für den rechten Nachfolger eines Knotens steht. Läuft man nun die Kanten von der Wurzel zu einem Blatt hinunter, so ergibt sich der Code dieses Blatts durch die Kantenfolge.

Damit ist die Aufgabe allerdings klar: Gegeben eine Datei mit zu komprimierenden Daten, man konstruiere einen binären Entscheidungsbaum, der zu einer optimalen präfix-freien Codierung des Dateiinhalts führt. Die bekannteste und am weitesten verbreitete Form dieser Codierung ist die *Huffman-Codierung*.

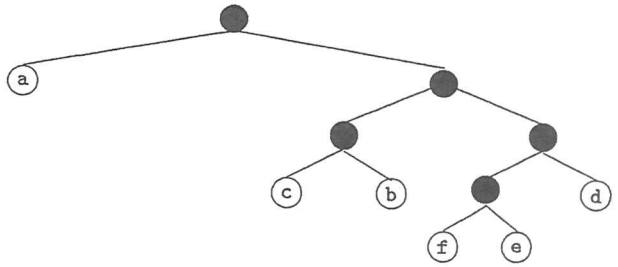


Abbildung 9.4: Präfix-freie Codierung durch einen binären Entscheidungsbaum

Huffman-Codierung

Die Idee der Huffman-Codierung besteht darin, diesen Baum auf der Basis der Zeichenhäufigkeit der gegebenen Datei aufzubauen und eine dementsprechende Codierung zu wählen, die für genau diese Datei optimal ist. Das einzige Problem besteht dabei darin, daß man in der komprimierten Datei die Information über diese Codierung mit ablegen muß, sonst klappt die Expansion nicht. Das führt dazu, daß man mit einem gewissen zusätzlichen Speicheraufwand rechnen muß, so daß diese Form der Codierung für größere Dateien effektiver ist als für kleinere. Bei großen Dateien kann man in der Praxis mit einer Platzersparnis von 20 – 70% rechnen, je nach dem.

Der grobe Ablauf der Huffman-Codierung besteht aus fünf Schritten:

1. Erster Durchlauf durch die zu komprimierende Datei, um für jedes mögliche Zeichen seine aktuelle Häufigkeit festzustellen.
2. Aufbau des Codebaums auf der Basis der eben festgestellten Häufigkeiten.
3. Festlegen der einzelnen Codes und der Längen aus dem Codebaum.
4. Abspeichern der Codeinformation für die Expansion.
5. Zweiter Durchlauf durch die zu komprimierende Datei, um die einzelnen Zeichen zu codieren.

Über den ersten Schritt — das Zeichenzählen — muß man wohl nicht viel sagen. Wir gehen ab sofort davon aus, daß nach dem Abschluß dieser Phase auf dem Array `freq` für jedes Zeichen seine Häufigkeit gespeichert ist. Zeichen, die in der Datei nicht vorkommen, haben also einen Wert von 0.

9.4.1 Aufbau des Codebaums

In der Tat haben wir es hier mit einem Optimierungsproblem zu tun, denn der zu konstruierende Baum soll eine optimale Codierung ergeben. Außerdem ist Huffman-Codierung ein gieriges Verfahren, weil die Häufigkeit eines Zeichens nur ein einziges Mal angeguckt wird.

Nehmen wir mal einen einfachen Text:

STUDENTEN PENNEN OFT

Die erste Phase liefert uns folgende Häufigkeiten (_ ist das Leerzeichen):

c	_	D	E	F	N	O	P	S	T	U
freq[c]	2	1	4	1	5	1	1	1	3	1

Um nun an den Baum zu gelangen, denken wir uns zunächst man „von oben" in ihn hinein, obwohl wir ihn später „von unten" konstruieren werden. Man möchte zuerst die vorkommenden Zeichen — die anderen interessieren uns überhaupt nicht — in zwei Gruppen so zerlegen, daß die Häufigkeiten möglichst gleich groß sind.

Insgesamt haben wir 20 Zeichen im Text und eine mögliche Zerlegung wäre etwa

	Zeichen	Gesamte Häufigkeit
Gruppe 1	E D F O P	8
Gruppe 2	N T _ S U	12

Dieser erste Schritt ist in Abb. 9.5 dargestellt.

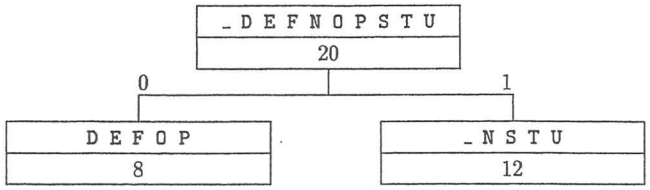


Abbildung 9.5: Idee hinter der Huffman-Codierung (Erster Schritt)

Diese Gruppen werden dann links bzw. rechts vom Wurzelknoten angeordnet und dann geht es in diesen Gruppen genauso weiter. Führt man dieses Verfahren beispielsweise mit der ersten Gruppe weiter, so passiert etwas, was für die Huffman-Codierung entscheidend: Es entsteht ein Nachfolger, der nur noch ein einziges Zeichen enthält, das E. In Abb. 9.6 ist das zu sehen.

Damit aber liegt bereits die Codierung des ersten Zeichens fest: Das E bekommt den Code „00". Würde man das Verfahren nun fortsetzen, so bekäme einen Codebaum heraus, wie er in Abb. 9.7 dargestellt ist.

Man erkennt schnell, daß es ziemlich willkürlich ist, welchen Knoten man nach links und welchen nach rechts tut, das betrifft lediglich die Reihenfolge der Binärziffern in einem Code, nicht die Länge und schon gar nicht die Eigenschaft des Codes, präfix-frei zu sein.

Aus dem Baum in Abb. 9.7 kann nun schon die Codierung abgeleitet werden:

c	_	D	E	F	N	O	P	S	T	U
freq[c]	2	1	4	1	5	1	1	1	3	1
Code	1110	0100	00	0101	10	0110	0111	11110	110	11111
Länge	4	4	2	4	2	4	4	5	3	5

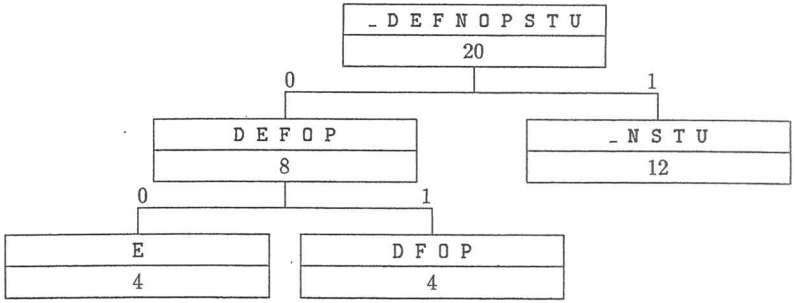


Abbildung 9.6: Idee hinter der Huffman-Codierung (Zweiter Schritt)

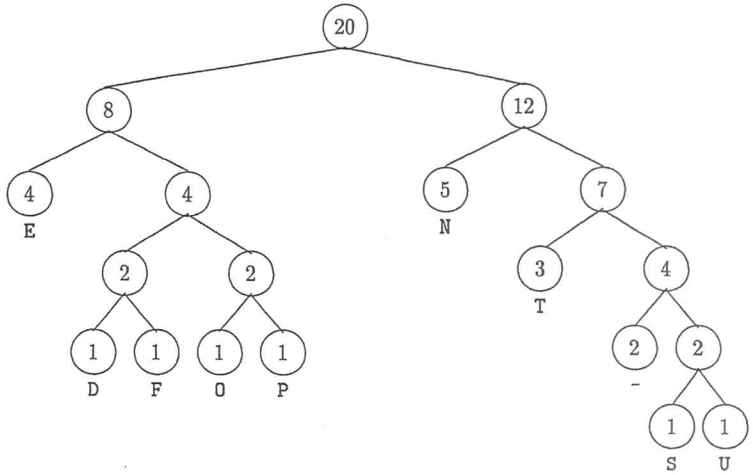


Abbildung 9.7: Ein fertiger Codebaum

Unser Text STUDENTEN PENNEN OFT sieht in dieser Codierung so aus:

111101101111101000010110001011100111001010001011100110010111 0

Aufgabe: *Machen Sie sich klar, daß diese Bitfolge bei gegebener Codierungstabelle eindeutig zu expandieren ist.*

Soviel zur Idee, programmieren kann man das so nur mit Schwierigkeiten. Wie schon angedeutet, werden wir den Baum von den Blättern her codieren, denn die Blätter kennen wir ja schon aus der Häufigkeitsbestimmung der ersten Phase. Zu zwei solchen Blättern werden wir dann zunächst ihren Vorgänger im Codebaum konstruieren, dessen Häufigkeitswert der Summe der beiden Häufigkeiten in den Blättern entspricht.

Betrachtet man Abb. 9.7, so sieht man, daß einige Blätter, wie z.B. das E, zu Anfang keinen Nachbarn haben, der muß erst noch konstruiert werden. Die Blätter, deren Vorgänger sofort zu konstruieren ist, haben alle eine geringere Häufigkeit aufzuweisen. Und damit wird die Struktur des Verfahrens klar:

1. Man startet mit den Blättern und ihrer Häufigkeit.
2. Man suche die beiden Blätter mit den kleinsten Häufigkeiten, etwa S und U.
3. Man konstruiere einen Knoten, dessen linker Nachfolger das S und dessen rechter Nachfolger das U ist. Sein Häufigkeitswert (2) ergibt sich als Summe der Häufigkeiten von S und U.

Diesen Knoten fügen wir zu unserer Sammlung hinzu und entfernen S und U. Im nächsten Schritt konstruieren wir einen Vorgänger für O und P, einen für D und F und haben nun nur noch Knoten mit einer Häufigkeit ≥ 2 vorrätig. Nun kann man z.B. den anfangs konstruierten Vorgänger von S und U mit dem Blatt _ (Häufigkeit 2) kombinieren und so arbeitet man sich langsam den Baum hinauf.

Bevor wir an den Pseudocode herangehen, sollten wir noch festlegen, wie eine solche Sammlung von Knoten denn aussehen soll. Offenbar brauchen wir eine *Prioritätswarteschlange* (priority queue), die wir mit `insert` und `remove-smallest` angehen, vergl. 1.4.6 und 4.2.3. Die Elemente der PQ besteht aus komplexen Knoten, die folgende Komponenten enthalten sollen:

- `leaf`: Ist wahr, wenn es sich um ein Blatt des Codebaums handelt, also ein Zeichen darstellt.
- `char`: Das dargestellte Zeichen, falls der Knoten ein Blatt ist.
- `freq`: Gesamte Häufigkeit der durch einen Knoten dargestellten Menge von Zeichen. Das ist gleichzeitig die Priorität des Knotens.
- `left`: Verweis auf den linken Nachfolger.
- `right`: Verweis auf den rechten Nachfolger.

Also...

Algorithmus 9.4 *Aufbau des Codebaums für den Huffman-Code mit einer Prioritätswarteschlange* PQ.

Precondition: ---
Postcondition: PQ enthält lediglich den Wurzelknoten des Codebaums

```
procedure huffman-tree(var PQ)
  {Berechne Haeufigkeiten im Text auf freq}

  init(PQ)   {Nicht vergessen!}

  {Eintragen der Blaetter in die PQ}
  for {moegliche Zeichen c} do
     if freq[c] <> 0 then
        n        := new-node
        leaf(n)  := true
        left(n)  := nil
        right(n) := nil
        char(n)  := c
        freq(n)  := freq[c]
        insert(PQ,n)

  {Aufbau des Codebaums}
  while {PQ enthaelt wenigstens zwei Elemente} do
     x        := remove-smallest(PQ)
     y        := remove-smallest(PQ)
     n        := new-node
     leaf(n)  := false
     left(n)  := x
     right(n) := y
     freq(n)  := freq(x) + freq(y)
     insert(PQ,n)
```


In Abb. 9.8 sind die ersten Schritte dargestellt.

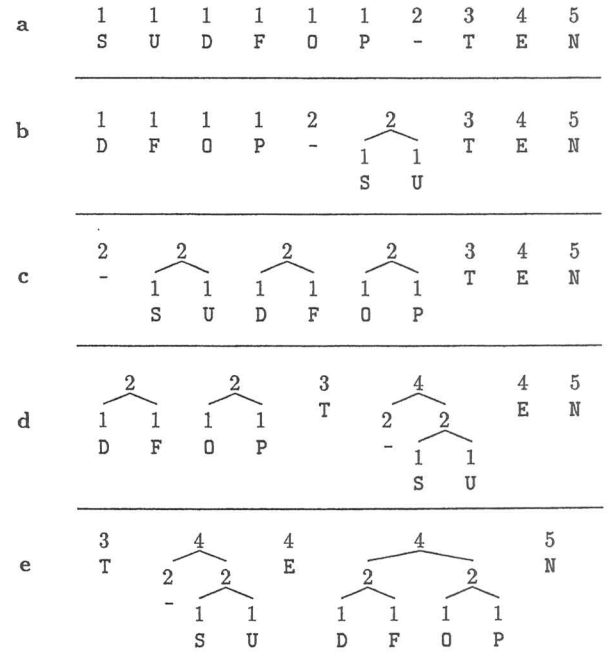


Abbildung 9.8: Aufbau des Codebaums 1

Die PQ ist hier in der Reihenfolge angeben, in der Elemente entnommen werden. Das Einfügen der Knoten mit der Häufigkeit der einzelnen Zeichen ergibt einen Zustand, der

unter **a** angegeben ist. Die beiden kleinsten Knoten `S` und `U` werden entnommen, durch einen Vorgänger ersetzt, der die Häufigkeit 2 aufweist und an geeigneter Stelle wieder in PQ eingefügt wird (**b**). Dasselbe geschieht mit den vier anderen Knoten der Häufigkeit 1 (**c**). Nun stehen zwei 2er Knoten vorn, die zu einem 4er verschmolzen werden, der noch etwas warten muß, bis die beiden anderen 2er und der 3er behandelt worden sind (**d**). Die beiden 2er verschmelzen zu einem 4er und damit rutscht das `T` mit der Häufigkeit 3 nun nach vorne (**e**). In Abb. 9.9 folgen dann die nächsten Schritte.

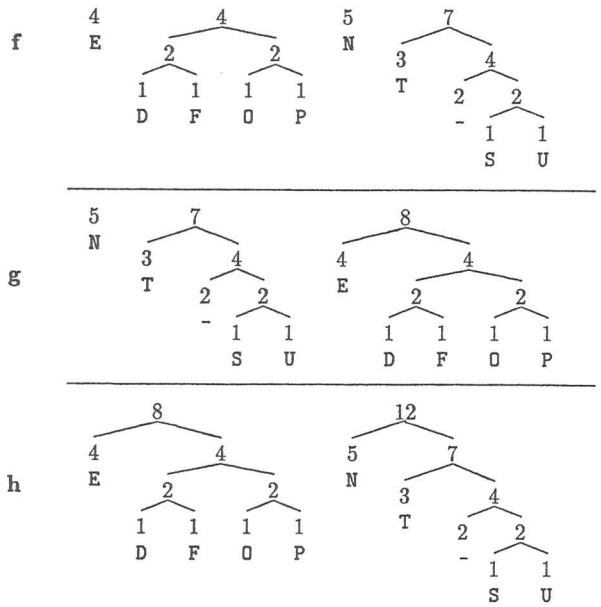


Abbildung 9.9: Aufbau des Codebaums 2

Das `T` verschmelzen wir mit dem vorhin konstruierten 4er zu einem Knoten mit der Häufigkeit 7, der dann ganz ans Ende rutscht (**f**). Aus den beiden 4er-Knoten wird ein 8er gemacht, der auch ganz nach hinten läuft (**g**) und die dann vorne stehenden Knoten mit den Häufigkeiten 5 und 7 ergeben dann den mit der Häufigkeit 12 (**h**). Setzt man nun noch diese beiden Bäume zusammen, kommt man zu der in Abb. 9.7 dargestellten Situation. Am Ende dieses Teils enthält unsere PQ nur noch einen einzigen Knoten: die Wurzel des Codebaums.

Man muß noch einmal darauf hinweisen, daß aus diesem Text viele verschiedene Codebäume erzeugt werden können. Das hängt natürlich davon ab, was links und was rechts angebracht wird und wie die PQ Elemente mit gleicher Priorität behandelt.

9.4.2 Komplexität

Die Komplexität des Aufbaus des Codebaums ist eigentlich ganz einfach zu bestimmen, wenn man die Kosten für `insert` und `remove-smallest` kennt. Natürlich implementiert man die PQ mit einem *min-Heap*, s. 4.2.2. Da dann bei gegebener Anzahl n von möglichen Zeichen beide Operationen von der Komplexität $O(\log n)$ sind, benötigt das Einfügen am Anfang also $O(n \log n)$ Schritte. Da nun bei jedem Schritt zum Aufbau des Codebaums zwei Elemente entnommen aber nur eins zurückgeschrieben wird, wird die entsprechende Schleife $n-1$ mal durchlaufen, also hat auch die Aufbauphase eine Komplexität von $O(n \log n)$. Was *in der Praxis* sicher dominiert, ist somit wohl der erste Teil, das Feststellen der Häufigkeiten.

Aufbau der Codetabelle

Der nun notwendige Aufbau der Codetabelle ist nun wieder ganz einfach: Man traversiert den Baum in Tiefentraversierung, wie in 3.2.2 beschrieben, wobei man die Binärzahl und ihre Länge mitführt. Wir konstruieren so folgende Arrays:

- `code`, das zu jedem Zeichen seinen Code als Zahl enthält und
- `len`, das zu jedem Zeichen die Länge des Codes enthält.

Die Implementation der Bit-Schieberei wird hier durch Multiplikation mit 2 und eventueller Addition von 1 vorgenommen. Meistens sehen Programmierumgebungen auch unmittelbar nutzbare Operationen für so etwas vor. In `C` ist es sowieso eingebaut.

Algorithmus 9.5 *Konstruktion der Arrays* `code` *und* `len` *aus dem Codebaum. Aufruf* `construct-code(root,0,0)`.

Precondition: `node <> nil`
Postcondition: `code und len enthalten die Codierungen und Längen der im Text vorkommenden Zeichen`

```
procedure construct-code(node, code, len)
  if leaf(node) then
     code[char(n)] := code
     len[char(n)]  := len
  else
     construct-code(left(node),code*2,len+1)
     construct-code(right(node),code*2+1,len+1)
```

9.4.3 Komprimierung

Expandieren kann man den Bitstring natürlich nur, wenn die `code`- und `len`-Informationen bekannt sind. Bei der Kompression müssen diese beiden also zuerst gespeichert werden, sie werden bei jedem Transport mitgeschickt. Natürlich reicht es aus, nur für die tatsächlich vorkommenden Zeichen seinen `code` und seine `len` anzugeben. Die Expansionsphase ist damit völlig ausreichend bedient.

Nun geht es „nur noch" darum, die zu komprimierende Datei noch einmal durchzulaufen und jedes vorkommende Zeichen in seiner Binärcodierung in Worte zu 16 Bit gepackt abzuspeichern.

9.4.4 Expansion

Der Ablauf der Expansion ist i.w. spiegelbildlich zur Kompression, der Codebaum wird nun aber nicht aus dem Text, sondern aus den Angaben aus dem Kompressionslauf erzeugt. Man konstruiert zunächst einen Wurzelknoten `root`, dessen `left`- und `right`-Zeiger mit `nil` initialisiert werden, also einen leeren Codebaum.

Im nächsten Schritt wird jedes Zeichen `char` mit seinem Code `code` und dessen Länge `len` in der einfachsten denkbaren Weise in den Baum eingefügt, wobei man im Code — durch das entsprechende Bit — immer die Entscheidung ablesen kann, ob man links (0) oder rechts (1) weiterlaufen muß.

Algorithmus 9.6 *Erweitern des Codebaums für das Zeichen* `char`, *seinen Code* `code` *und dessen Länge* `len`.

Precondition: `node <> nil`
Postcondition: `char steht an derselben Stelle im Baum wie bei der Kompression`

```
procedure build-node(var node; char,code,len)
  if len = 0 then
     leaf(node) := true
     char(node) := char
  else
     leaf(node) := false

     if {Bit in code an position len gesetzt} then
        {nach rechts}
        if right(node) = nil then
           new-node(right(node))
           left(right(node))  := nil
           right(right(node)) := nil
        build-node(right(node),char,code,len-1)
    else
       {nach links}
       if left(node) = nil then
          new-node(left(node))
          left(left(node))  := nil
          right(left(node)) := nil
       build-node(left(node),char,code,len-1)
```

Die eigentlich Expansionsphase besteht nun darin, die einzelnen Binärziffern aus dem komprimierten Text zu holen, sich bei einer 0 nach links und bei einer 1 nach rechts zu orientieren, bis man an ein Blatt gelangt. Das Zeichen in diesem Blatt ist es. Danach natürlich nicht vergessen, wieder auf die Wurzel des Codebaums zurückzugehen!

9.5 Kürzeste Wege II: Single-Source Shortest Path

Man kann einen Greedy-Algorithmus einsetzen, um das schon in 8.3 beschriebene Problem der kürzesten Wege in einem (nun gewichteten) Graphen zu lösen. Genauer stellt sich hier zunächst das Problem, *von einer festen Ecke s aus die kürzesten Entfernungen zu allen anderen Ecken des Graphen zu bestimmen.* Im Jargon handelt es sich also um einen *single-source shortest path* (SSSP) Algorithmus.

Eine andere Klasse von solchen Wegproblemen wird durch die sogenannten *all-pair shortest path* (APSP) Algorithmen gelöst, die das nicht für eine Ausgangsecke, sondern *für alle Paare von Ecken* im Graphen machen. Ein typisches Beispiel dafür sind die Entfernungstabellen, die im Anhang einer Straßenkarte manchmal enthalten sind.

Nach allem, was wir uns so denken können, ist SSSP „einfacher" als APSP. Tatsächlich geht SSSP *greedy* — Dijkstras Algorithmus ist ein Greedy-Algorithmus. Einen APSP-Algorithmus lernen wir in 10.1 kennen.

Die Wirkungsweise des „klassischen" Dijkstra-Algorithmus wird z.B. in [Maas, 1993] ausführlich dargestellt. Hier also nur kurz die Idee:

Gegeben eine Startecke s, von der aus für alle anderen Ecken des Graphen der kürzeste Weg dorthin bestimmt werden soll. Jeder Ecke v wird ein vorläufiger Schätzwert $d(v)$ zugeordnet, der den Abstand zu s angibt. Dieser Wert wird im Verlauf des Verfahrens immer weiter verbessert, bis er am Ende *genau* den Abstand angibt.

Der Dijkstra-Algorithmus betrachtet nun jeweils die Ecke v mit dem aktuell kleinsten d-Wert und alle von dort wegführenden Kanten (v, v'). Die Nachbarecke v' könnte nun über v von s aus erreicht werden, so daß sich dann als Wert $d(v)$ die Summe $d(v) + w(v, v')$ ergäbe. Ist dieser Wert kleiner als der bisherige Wert $d(v')$, so wird er durch die neue Summe ersetzt. Eine einmal betrachtete Ecke wird *nie wieder* angefaßt.

Das Verfahren endet, wenn alle Ecken überprüft worden sind.

9.5.1 Implementation des Dijkstra-Algorithmus

Hier also einiges zur konkreten Implementation: Die in einer Ecke abgespeicherte Schätzung wird also im Verlauf des Verfahrens immer weiter verbessert, ein Vorgang, der als *Relaxation* der Ecke bezeichnet wird (der Begriff stammt aus citecormen:leiserson:rivest:algorithms).

Wir schauen uns zunächst an, was man also innerhalb einer Ecke im Graphen abzuspeichern hat. Das hängt natürlich stark davon ab, was wir letztlich haben wollen. Haben wollen wir wenigstens *zwei* Informationen zur Ecke `t` :

1. Die kleinste Entfernung von der Startecke `s` zu `t`.
2. Die Kantenfolge, die zu dieser Entfernung geführt hat.

Natürlich reicht es bei SSSP aus, zu jeder Ecke `t` den unmittelbaren Vorgänger in dieser Kantenfolge festzuhalten. Damit kann man sich ja nötigenfalls rückwärts wieder zu `s` hinhangeln.

Eine Ecke `v` hat also zwei zusätzliche Komponenten:

1. d[v] ist der Schätzwert der Entfernung von s.
2. p[v] ist der (ein Zeiger auf den) Vorgänger im entsprechenden kürzesten Weg.

Dann ist aber schon klar, wie vor Beginn des Verfahrens die Eckenmenge des Graphen zu initialisieren ist. Gegeben ist also ein Graph G und eine Ecke s von G, die als Ausgangsecke des Verfahrens dient:

```
procedure init-sssp(G,s)

  for v in V(G) do
     d[v] := infinity
     p[v] := nil

  d[s] := 0
```

Also: Jede Ecke bekommt als Schätzwert ∞ (schlechter geht's bestimmt nicht!) und hat noch keinen Vorgänger. d und p werden im Verlauf des Verfahrens aktualisiert.

Der eigentliche Relaxationsschritt über eine Kante (v_1, v_2) ist ganz einfach. Wir gehen davon aus, daß wir von v_1 kommend v_2 betreten und gucken uns den Wert d[v_1] + w(v_1, v_2) an. Der entspricht ja der Annahme, der kürzeste Weg von s nach v_2 liefe über v_1. Wenn dieser Wert kleiner als der aktuelle Wert d[v_2] ist, so ersetzen wir ihn dadurch und merken uns, daß der Vorgänger von v_2 auf diesem Weg halt v_1 ist:

```
procedure relax(v1,v2)
  newd := d[v1] + w(v1,v2)
  if newd < d[v2] then
     d[v2] := newd
     p[v2] := v1
```

Nun ist der Dijkstra-Algorithmus auch noch ganz kurz. Wir müssen lediglich noch die Frage klären, wie wir die Kandidatenmenge, also die Eckenmenge des Graphen, am besten verwalten.

Was wissen wir über den Algorithmus? Wir greifen uns ja immer diejenige Ecke aus der Kandidatenmenge heraus, die den kleinsten aktuellen d-Wert besitzt. Ausgehend von dieser Ecke werden dann alle Kanten durch relax gejagt. Ganz abstrakt ergibt sich dadurch etwa folgende Grobversion des Dijkstra-Algorithmus:

```
procedure dijkstra(G,s)

  init-sssp(G,s)
  c := V(G)       {Eckenmenge des Graphen}

  while c <> emptyset do
     {Gieriger Auswahlschritt}
     v1 := {diejenige Ecke mit dem kleinsten aktuellen d-Wert aus c}
```



```
     c := c - {v1}

     {Alle Nachbarn von v 1 angucken und deren}
     {d-Werte verbessern, falls moeglich}

     v2 := adj[v1]      {Erste benachbarte Ecke}
     while v2 <> nil do
        relax(v1,v2)
        v2 := succ(v2) {Naechste benachbarte Ecke}
```

Die wirklich einzige nicht-triviale Fragestellung besteht in der Realisierung der Kandidatenmenge! Man möchte ja die Ecke mit dem kleinsten aktuellen d-Wert möglichst schnell bekommen. Als ADT kennen wir so etwas: wieder mal die *Priority Queue*, s. 1.4.6.

Wir nehmen also an, die Eckenmenge ist derart als PQ realisiert, daß wir mittels remove-smallest diejenige verbleibende Ecke bekommen, die als Schlüssel den kleinsten d-Wert besitzt. Gleichzeitig soll sie ja auch aus der PQ entfernt werden.

Nun gibt's ja viele Möglichkeiten, eine solche PQ zu implementieren. Wir wählen hier natürlich wieder die Implementation mit einem *min-Heap* (vergl. 4.2.2).

Mit dem, was wir nun schon alles wissen, geht der Initialisierungsteil des Verfahrens schnell. Nach der Initialisierung der d- und p-Werte in init-sssp müssen wir den min-Heap aufbauen. Dazu kann man natürlich das Einfügen mittels insert benutzen, aber aus Komplexiätsgründen machen wir es uns etwas einfacher. Wir wissen ja, daß es lediglich die Ecke s ist, die einen d-Wert besitzt, der nicht unendlich ist. Also speichern wir s als erste in den Heap (sie kommt auch als erste wieder raus!) und speichern dann die anderen Ecken in beliebiger Reihenfolge dahinter:

```
c[1]     := s
heapsize := 1

for v in V(G) do
   if v <> s then
      heapsize    := heapsize + 1
      c[heapsize] := v
```

Der gierige Auswahlschritt heißt dann

```
remove-smallest(c,heapsize,v1)
```

Nun kann sich natürlich in relax unter Umständen der d-Wert einer Ecke ändern: Er wird kleiner. Was wir brauchen, ist also eine Möglichkeit, den Heap wieder zu reorganisieren, wenn sich der Schlüsselwert eines Knotens ändert. Das muß natürlich in der PQ realisiert werden.

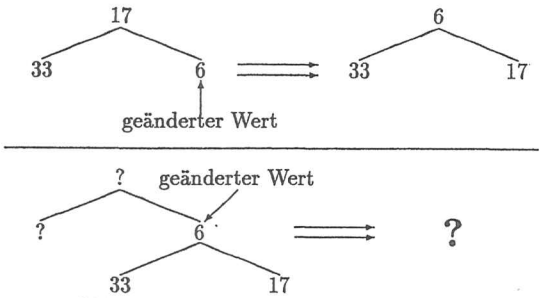


Abbildung 9.10: Reorganisation eines min-heap

In Abbildung 9.10 kann man sehen, in welcher Form diese Änderung vorzunehmen ist: Irgendwie ist die Situation entstanden, daß diese falsche 6 im Heap erscheint (vorher war der Wert vielleicht 42). Die eine min-Heap Eigenschaft ist also verletzt.

Das kann man leicht reparieren, wie die Abbildung schon zeigt. Zuerst stellt man fest, daß sich durch einen Relaxationsschritt der d-Wert des Heap-Knotens *erniedrigt* hat. Es kann also passieren, daß der Knoten *nach oben* verschoben werden muß.

Man vergleicht also den neuen (niedrigeren) d-Wert also mit dem seines Vorgängers im Heap. Es gibt dann zwei Möglichkeiten:

1. Der d-Wert des Vorgängers ist größer oder gleich dem des verbesserten Knotens. Dann ist die Heap-Eigenschaft nicht gestört und wir können aufhören.
2. Der d-Wert des Vorgängers ist echt kleiner als der des verbesserten Knotens. Dann vertauschen wir die beiden. Es kann dann sein, daß dieselbe Situtation eine Etage weiter oben auftritt. Also gehen wir in eine entsprechende Schleife.

Bei der Codierung gehen wir davon aus, daß die Knoten des Heap i.w. mit den Ecken des Graphen identisch sind. Also greifen wir auf den d-Wert als Komponente des Knotens zu:

Algorithmus 9.7 *Wiederherstellen der möglicherweise im Index* `i` *zerstörten min-Heap Eigenschaft von* `h`. `heapsize` *ist letzter belegter Index des Heap* `h`.

Precondition: `Lediglich h verletzt Def. 4.2`
Postcondition: `Def. 4.2`

```
procedure restore-heap(var h; heapsize,i)
  pred := i div 2
  indx := i
  temp := h[indx]
  comp := temp.d

  while pred > 0 and h[pred].d > comp do
     h[indx] := h[pred]
     indx    := pred;
     pred    := indx div 2

  h[indx] := temp
```


Diese Änderung muß man natürlich bei jeder erfolgreichen Relaxation einer Ecke ausgeführt werden, so daß `relax` der Punkt ist, an dem das zweckmäßigerweise zu implementieren ist. Man weiß dort allerdings nicht mehr, an welchem Index v_2 im Heap steht! Wir gehen also davon aus, daß es einen Eintrag `heap-index` in jeder Ecke gibt, der uns den aktuellen Index der Ecke im Heap angibt.

Algorithmus 9.8 *Relaxation der Ecke* `v2` *durch die Ecke* `v1` *mit folgender Korrektur im Heap* `c`.

Precondition: `---`
Postcondition: `d[v2] und p[v2] enthalten verbesserte Werte`

```
procedure relax(v1,v2)
  newd := d[v1] + w(v1,v2)
  if newd < d[v2] then
     d[v2] := newd
     p[v2] := v1

  restore-heap(c,heapsize,v2.heap-index)
```

Nun ist unser Dijkstra aber schon fertig!

Algorithmus 9.9 *SSSP nach Dijkstra. Die Kandidatenmenge wird in einem min-Heap* `c` *verwaltet.*

Precondition: `---`
Postcondition: `d und p enthalten für jede Ecke die kürzeste Entfernung zu s bzw. den Zeiger auf den Vorgänger im kürzesten Weg`

```
procedure dijkstra(G,s)

  init-sssp(G,s)

  {PQ initialisieren: In den min-Heap fuellen}
  c[1]     := s
  heapsize := 1

  for v in V(G) do
     if v <> s then
        heapsize    := heapsize + 1
        c[heapsize] := v

  while heapsize > 0 do      {Greedy loop!}
     {Gieriger Auswahlschritt!}
     remove-smallest(c,heapsize,v1)

     {Alle Nachbarn von v1 angucken und deren}
```

```
{d-Werte verbessern, falls moeglich}
v2 := adj[v1]              {Erste benachbarte Ecke}

while v 2 <> nil do
   relax(v1,v2)
   v2 := succ(v2)          {Naechste benachbarte Ecke}
```

9.5.2 Beispiel

In Abbildung 9.11 ist ein Beispiel dargestellt.

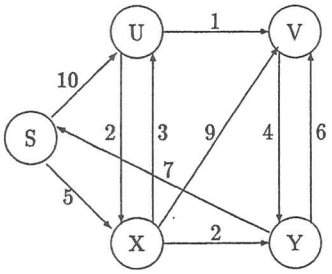


Abbildung 9.11: Beispielgraph für SSSP

Die Ausgangsecke s ist mit S markiert. Das nun ist das, was passiert:

1. `remove-smallest:` Ecke S mit d-Wert 0
   `relax(S,X)` d[X] $\infty \Rightarrow 5$
   p[X] = S
   `relax(S,U)` d[U] $\infty \Rightarrow 10$
   p[U] = S
2. `remove-smallest:` Ecke X mit d-Wert 5
   `relax(X,U)` d[U] $10 \Rightarrow$ d[X] $+ 3 = 5 + 3 = 8$
   p[U] = X
   `relax(X,Y)` d[Y] $\infty \Rightarrow$ d[X] $+ 2 = 7$
   p[Y] = X
   `relax(X,V)` d[V] $\infty \Rightarrow$ d[X] $+ 9 = 14$
   p[V] = X
3. `remove-smallest:` Ecke Y mit d-Wert 7
   `relax(Y,S)` Keine Verbesserung von d[S] $= 0$
   `relax(Y,V)` d[V] $14 \Rightarrow$ d[Y] $+ 6 = 13$
   p[V] = Y
4. `remove-smallest:` Ecke U mit d-Wert 8
   `relax(U,X)` Keine Verbesserung von d[X] $= 5$
   `relax(U,V)` d[V] $13 \Rightarrow$ d[U] $+ 1 = 9$
   p[V] = U
5. `remove-smallest:` Ecke V mit d-Wert 9
   `relax(V,Y)` Keine Verbesserung von d[V] $= 9$

Nach Ablauf des Dijkstra-Algorithmus sind also folgende Werte bestimmt worden:

```
d[S] = 0  p[S] = nil
d[U] = 8  p[U] = X
d[V] = 9  p[V] = U
d[X] = 5  p[X] = S
d[Y] = 7  p[Y] = X
```

9.5.3 Komplexität

Um im folgenden nicht immer $V(G)$ oder $E(G)$ schreiben zu müssen, halten wir den Graphen G fest und schreiben kurz V bzw. E, wenn wir die Eckenmenge bzw. Kantenmenge meinen.

Die hauptsächliche Abschätzung des Verfahrens ist bekannt: Durch den *Greedy loop* gehen wir $|V|$ mal und machen dort jeweils ein `remove-smallest`. Jedes solche `remove-smallest` kostet $O(\log |V|)$, was auch für `restore-heap` gilt (nachrechnen!). Rechnen wir zusammen:

1. Aufbau des min-Heap: $O(|V|)$
2. $|V|$–mal `remove-smallest`: $O(|V| \log |V|)$
3. `relax` inkl. `restore-heap`: $O(\log |V|)$
4. dasselbe $|E|$-mal: $O(|E| \log |V|)$

2 und 4 zusammengezählt (1 geht unter!): $O((|V| + |E|) \log |V|)$.

9.6 Weitere greedy lösbare Problem

Hier noch einige Probleme, die greedy lösbar sind:

Minimierung der Aufenthaltszeit in einem System mit einer Maschine: Eine Maschine soll n Aufträge abwickeln, die eine Bearbeitungszeit $t_i, 1 \leq i \leq n$ haben. Die Summe der Verweilzeiten aller Aufträge, also Wartezeit + Bearbeitungszeit, ist zu minimieren. Auswahlfunktion: Derjenige verbleibende Auftrag mit der kürzesten Bearbeitungszeit. Dies nennt man „Shortest Job Next“ Strategie (SJN-Strategie).

Minimierung von Erholungspausen: Eine Maschine kann eine gewisse Maximalzahl von Zeiteinheiten n ununterbrochen arbeiten, spätestens dann muß sie eine Pause einlegen, von der wir annehmen, daß sie konstante Länge hat. Danach kann sie dann wieder n Einheiten lang arbeiten. Diese Pausen können allerdings — aus Gründen des Arbeitsablaufs — nur zu bestimmten Zeitpunkten t_i genommen werden, die mit der Maschine selbst nichts zu tun haben. Es ist die Anzahl der Pausen zu minimieren. Auswahlfunktion: Man nehme denjenigen Zeitpunkt für die nächste Pause, der soweit wie möglich in der Zukunft liegt, aber von der letzten höchstens n Einheiten entfernt ist.

Maximierung nicht-überlappender Aktivitäten auf einer Maschine: Wieder einmal haben wir eine einzige Maschine, auf der eine gewisse Anzahl n von Aktivitäten ablaufen sollen. Jede Aktivität i hat einen verbindlichen *Anfangstermin* a_i und einen verbindlichen *Endtermin* e_i, in dem sie diese Maschine belegt. Zwei Aktivitäten sind verträglich, wenn sich die resultierende Belegungen nicht überlappen. Die Anzahl der verträglichen Aktivitäten ist zu maximieren. Auswahlfunktion: Man wähle diejenige Aktivität i mit dem nächstmöglichen *End*termin e_i, die mit der letzten geplanten Aktivität verträglich ist.

Das Greedy-Verfahren ist also vergleichsweise einfach, aber nur begrenzt anwendbar. Wenn man nun einen ausreichend friedlichen Kunden hat, kann es sein, daß der mit einer guten, aber nicht optimalen Lösung schon zufrieden ist.

In diesem Fall kann man natürlich ein an sich nicht greedy optimierbares Problem trotzdem greedy lösen, auch wenn sich im allgemeinen Fall keine optimale Lösung ergibt. Man nennt den Einsatz eines Greedy-Algorithmus in einem solchen eigentlich nicht-optimalen Fall eine *Greedy-Heuristik*, weil die Korrektheit fraglich ist. Greedy-Heuristiken klammern wir hier aber zunächst aus und tasten uns an ein mächtigeres aber auch aufwendigeres Optimierungsverfahren heran: Die *dynamische Programmierung*.

Kapitel 10

Dynamische Programmierung

Dieser Ausdruck bezeichnet keinen besonders kraftvollen Stil, Programme zu schreiben! Die Herkunft des Begriffs „Programmierung" in diesem Zusammenhang stammt wohl aus einer Kultur, die unter Programmierung alles verstand, was irgendwie mit Tabellen und ähnlichem zu tun hat. In diesem Zusammenhang steht auch der seltsame Begriff der „linearen Programmierung", die man heutzutage besser als „lineare Optimierung" bezeichnet.

Die Intuition hinter der dynamischen Programmierung (DP) ist nicht so ganz einfach zu vermitteln. Im Prinzip geht es darum, ein Optimierungsproblem nicht „von oben" zu lösen, wie das beim Greedy-Verfahren der Fall ist, sondern eher „von unten", indem man sich die resultierenden Restprobleme anguckt, bevor man das eigentliche Problem angeht.

Am besten starten wir gleich mit einem richtigen Optimierungsproblem! Thema noch einmal: *Kürzeste Wege*.

10.1 Kürzeste Wege III: All-Pair Shortest Path

Hier geht's nun um das schon angesprochene Problem, *für alle Paare von Ecken* die Längen der kürzesten Wege und u.U. auch noch diese Pfade festzustellen. Hier wollen wir uns das allereinfachste Verfahren angucken: Den Algorithmus von *Floyd und Warshall*.

10.1.1 Erste Überlegungen

Studieren wir die Situation bei der Konstruktion kürzester Wege einen Moment, so fällt auf, daß es eine wichtige Eigenschaft hat, die wir sehr vorteilhaft ausnutzen können. Betrachten wir dazu zwei Ecken v_i und v_j. Es gibt zwei Möglichkeiten für den kürzesten Weg zwischen ihnen:

1. er verläuft über die Kante zwischen ihnen oder
2. er verläuft über eine dritte Ecke v_k.

Der zweite Fall soll symbolisch durch $v_i \rightsquigarrow v_k \rightsquigarrow v_j$ notiert werden, wobei natürlich noch weitere Ecken an diesem Weg beteiligt sein können. Das soll uns im Moment aber noch nicht so sehr interessieren.

Was wissen wir nun über die Wege $v_i \rightsquigarrow v_k$ und $v_k \rightsquigarrow v_j$? Richtig: Auch sie sind die kürzesten Wege!

Aufgabe: *Warum?*

Dies ist eine charakteristische Eigenschaft von Problemen, die mit DP gelöst werden können. Deshalb formulieren wir sie noch einmal explizit:

> *Charakteristische Eigenschaft 1 für DP: Die Optimale Teilstruktur-Eigenschaft* (engl. optimal substructure property, OTE, OSP). Ein Problem hat die OTE, wenn eine optimale Lösung des Problems optimale Lösungen aller Teilprobleme enthält. Unter dieser Bedingung kann man also sich zunächst damit befassen, die kleineren Teilprobleme zunächst zu lösen und dann daraus die Lösung des nächst-größeren Problems zusammenzusetzen.

Wenn man erkennt, daß ein Problem die OTE besitzt, kann man die Anwendung von DP ernsthaft in Betracht ziehen. Das APSP-Problem hat also die OTE und ist damit ein Kandidat für dieses Verfahren.

Das allein ist noch ziemlich banal, weil man sich bereits auf der Basis dieser Eigenschaft einen rekursiven Algorithmus überlegen kann, der APSP löst. Man benötigt eine Darstellung des Graphen in Form seiner *Adjazenzmatrix* a, bei der vereinbarungsgemäß gilt (vergl. 3.2.3):

- a[i,j] = 0, falls i = j.
- a[i,j] = w(v_i,v_j), also das Gewicht der Kante von v_i nach v_j, falls es so eine Kante gibt.
- a[i,j] = ∞, falls $i \neq j$ und keine Kante zwischen v_i und v_j existiert.

Ohne alles auszuprogrammieren, hier kurz die rekursive Idee: Um die Länge des kürzesten Wegs zwischen v_i und v_j zu bestimmen, beginnt man mit a[i,j], der direkten Entfernung zwischen ihnen. Dann guckt man sich alle *anderen* Eckenn v_k an und bestimme rekursiv die Längen der kürzesten Wege $v_i \rightsquigarrow v_k$ und $v_k \rightsquigarrow v_j$. Die Summe dieser beiden Wege ist ein neuer Kandidat für den kürzesten Weg zwischen v_i und v_j. Man muß lediglich darauf achten, daß die Wege so konstruiert werden, daß beim Zusammensetzen keine Ecke mehr als einmal durchlaufen wird.

Aufgabe: *Programmieren Sie diese Skizze aus. Lassen Sie das Programm für Eckenzahlen zwischen 5 und 8 laufen. Wie sieht das Laufzeitverhalten experimentell aus?*

Der Effekt ist schon bei einer Eckenzahl von 4 zu beobachten, man betrachte Abb. 10.1, in der die Aufrufe zur Berechnung des kürzesten Wegs von v_1 nach v_2 dargestellt werden.

Man erkennt, daß einige Wege doppelt berechnet wurden und dieser Effekt verschlimmert sich noch bei größeren Eckenzahlen. Man kann zeigen, daß bei n Ecken jeder Weg $\Omega(2^{n-2}(n-2)!)$ oft berechnet wird und das führt dazu, daß dieses Verfahren für praktische Zwecke absolut unbrauchbar ist.

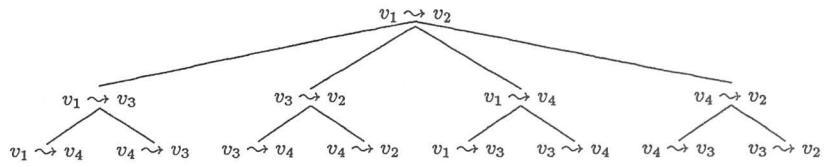


Abbildung 10.1: Rekursive Berechnung kürzester Wege

Die kennzeichnende Eigenschaft, die hier so störend auffällt, ist dann auch die zweite charakteristische Eigenschaft für DP-geeignete Probleme:

> *Charakteristische Eigenschaft 2 für DP: Auftreten überlappender Teilprobleme.* Wenn die rekursive Version des Optimierungsalgorithmus identische Probleme sehr häufig lösen muß, spricht man davon, daß das Problem überlappende Teilprobleme besitzt. Das Auftreten *relativ weniger* überlappender Teilprobleme ist ebenfalls charakteristisch für DP-geeignete Probleme.

Man erkennt, daß die OTE allein noch keinen Einfluß darauf hat, daß das Problem mittels DP gelöst werden sollte. Lediglich wenn die Eigenschaft der überlappenden Teilprobleme hinzukommt, steckt man im geeigneten Szenario. Die Idee ist nun, diese Teilprobleme einmal zu lösen und das Ergebnis in einer Tabelle oder Matrix festzuhalten.

Dazu ist es natürlich wichtig, daß es nur wenige solche Teilprobleme gibt und die eigentlich Komplikation des rekursiven Algorithmus durch eine sehr häufige Berechnung ein und desselben Teilproblems entsteht.

10.1.2 Der Floyd-Warshall Algorithmus

Dieser Algorithmus funktioniert in $\Theta(|V|^3)$ Schritten und ist in seiner Codierung sehr einfach — *viel* einfacher als Dijkstra!

Zur Frage, wie wir den Algorithmus ansetzen. Wir denken uns die Ecken des Graphen durchnummeriert von 1 bis $n = |V(G)|$, der Anzahl der Ecken des Graphen. Wir betrachten nun im k-ten Iterationsschritt die möglichen optimalen Trennungen aller Wege $v_i \rightsquigarrow v_t \rightsquigarrow v_j$, mit $1 \leq t \leq k$, $t \neq i$ und $t \neq j$. Wenn wir das n-mal gemacht haben, haben wir in der Tat *alle* möglichen Trennungen abgeklappert und sind fertig. Das Ergebnis tun wir dann in eine Matrix d, die im k-ten Iterationsschritt folgende Eigenschaften hat:

- d[i,j] = 0, falls i = j.
- d[i,j] = Länge des kürzesten Wegs zwischen v_i und v_j, der nur durch die von v_i und v_j verschiedenen Ecken $\{v_1, \ldots, v_k\}$ verläuft, falls so etwas existiert.
- d[i,j] = ∞, falls $i \neq j$ und kein solcher Weg zwischen v_i und v_j existiert.

Damit bekommen wir unsere erste Grobversion:

```
procedure floyd-warshall(G)
  for k := 1 to n do
     for i := 1 to n do
        for j := 1 to n do
           d[i,j] :=  {Laenge des kuerzesten Wegs}
                      {zwischen v_i und v_j, der}
                      {nur durch Ecken mit einem}
                      {Index <= k verlaeuft}
```

Am Ende haben wir nach Definition in `d[i,j]` die Länge des entsprechenden kürzesten Wegs stehen, weil wir k bis n haben laufen lassen.

Wie muß `d` initialisiert werden? Wir setzen einfach in der Definition `k` $= 0$: Also die Länge der kürzesten Wege, die durch *keine* andere Ecke als v_i bzw. v_j verläuft. Das ist gerade die Länge der Kanten (v_i, v_j), falls die existiert. Also: `d` wird mit der Adjazenzmatrix `a` initialisiert.

Wie bekommen wir nun im k-ten Schritt den neuen Wert von `d[i,j]`? Wir nehmen ja nun die Ecke v_k als mögliche Trennungsalternative hinzu, die anderen Ecken $\{v_1, \ldots, v_{k-1}\}$ haben wir ja bereits abgehandelt.

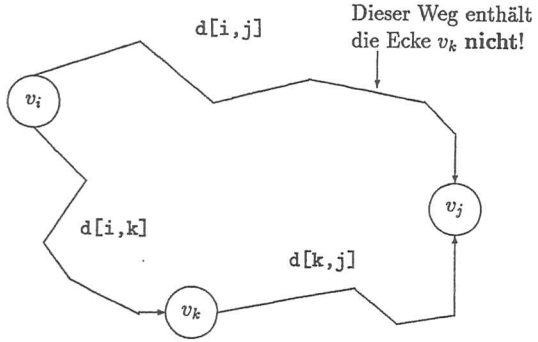


Abbildung 10.2: Zur Illustration des k-ten Iterationsschritts

Es ist also lediglich festzustellen, ob es einen besseren Weg zwischen v_i und v_j gibt, als den bisher besten — ohne v_k.

Das kann man aber prüfen: Der Weg $v_i \rightsquigarrow v_k \rightsquigarrow v_j$ ist besser als der bisher beste $v_i \rightsquigarrow v_j$ aus der $k-1$-ten Iteration, wenn die Summe `d[i,k] + d[k,j]` kleiner als `d[i,j]` ist.

Hier kommt dann auch schon der Kern des Floyd-Warshall-Algorithmus.

Algorithmus 10.1 *APSP-Algorithmus nach Floyd und Warshall.*

Precondition: `a[1:n,1:n] ist Adjazenzmatrix des Graphen`
Postcondition: `d[1:n,1:n] enthält die Längen der kürzesten Wege bzw. unendlich, wenn kein Weg existiert`


```
procedure floyd-warshall(a)
  d := a
  for k := 1 to n do
     for i := 1 to n do
        for j := 1 to n do
           if d[i,k] + d[k,j] < d[i,j] then
              d[i,j] := d[i,k] + d[k,j]
```

Nehmen wir den Graphen des Dijkstra-Verfahrens als Beispiel (Abbildung 9.11). Seine Adjazenzmatrix ist einfach aufzuschreiben, wenn wir uns die Ecken in der Reihenfolge S, X, U, Y, V nummeriert denken.

$$a = \begin{pmatrix} 0 & 5 & 10 & \infty & \infty \\ \infty & 0 & 3 & 2 & 9 \\ \infty & 2 & 0 & \infty & 1 \\ 7 & \infty & \infty & 0 & 6 \\ \infty & \infty & \infty & 4 & 0 \end{pmatrix}$$

So sieht `d` am Anfang also auch aus! Nun arbeiten wir die $n = |V|$ Schritte über k durch und bekommen nacheinander...

$$\begin{pmatrix} 0 & 5 & 10 & \infty & \infty \\ \infty & 0 & 3 & 2 & 9 \\ \infty & 2 & 0 & \infty & 1 \\ 7 & \infty & \infty & 0 & 6 \\ \infty & \infty & \infty & 4 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 5 & 10 & \infty & \infty \\ \infty & 0 & 3 & 2 & 9 \\ \infty & 2 & 0 & \infty & 1 \\ 7 & 12 & 17 & 0 & 6 \\ \infty & \infty & \infty & 4 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 5 & 8 & 7 & 14 \\ \infty & 0 & 3 & 2 & 9 \\ \infty & 2 & 0 & 4 & 1 \\ 7 & 12 & 15 & 0 & 6 \\ \infty & \infty & \infty & 4 & 0 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} 0 & 5 & 8 & 7 & 9 \\ \infty & 0 & 3 & 2 & 9 \\ \infty & 2 & 0 & 4 & 1 \\ 7 & 12 & 15 & 0 & 6 \\ \infty & \infty & \infty & 4 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 5 & 8 & 7 & 9 \\ 9 & 0 & 3 & 2 & 4 \\ 11 & 2 & 0 & 4 & 1 \\ 7 & 12 & 15 & 0 & 6 \\ 11 & 16 & 19 & 4 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 5 & 8 & 7 & 9 \\ 9 & 0 & 3 & 2 & 4 \\ 11 & 2 & 0 & 4 & 1 \\ 7 & 12 & 15 & 0 & 6 \\ 11 & 16 & 19 & 4 & 0 \end{pmatrix}$$

Am Ende haben wir also eine Matrix mit lauter Zahlen rausbekommen, die die Längen der kürzesten Wege enthält. Experimentell kann man das gerade mal an der ersten Zeile überprüfen. Dies sollten gerade die Längen der kürzesten Wege ausgehend von der Ecke S sein, also gerade das, was wir vor einiger Zeit mittels Dijkstra ausgerechnet haben.

Zeitkomplexität: $\Theta(|V|^3)$! Klarer geht's wohl kaum! Drei ineinandergeschachtelte Schleifen! Speicherkomplexität $\Theta(|V|^2)$, wegen der beiden Arrays `a` und `d`.

Nun nur noch ein Problem! Wir wollen nicht nur die Längen der kürzesten Wege, sondern vielleicht auch noch diese Wege selbst bekommen.

Die einzige Information, die wir im Verlauf des Algorithmus gewonnen haben, besteht ja in der Bestimmung des Index k der trennenden Ecke. In der Tat reicht diese Information völlig aus, um den kürzesten Weg rückwärts bestimmen zu können.

Wir definieren uns ein Array `t[1:n,1:n]`, das wir mit 0 vorbesetzen. Dazu noch eine kleine Änderung im Code:

```
if d[i,k] + d[k,j] < d[i,j] then
   d[i,j] := d[i,k] + [k,j]
   t[i,j] := k          {Index der trennenden Ecke festhalten}
```

Was wir als Ergebnis in der Matrix t bekommen, wenn unser Verfahren abgelaufen ist, sehen wir hier:

$$\mathtt{t} = \begin{pmatrix} 0 & 0 & 2 & 2 & 3 \\ 4 & 0 & 0 & 0 & 3 \\ 4 & 0 & 0 & 2 & 0 \\ 0 & 1 & 2 & 0 & 0 \\ 4 & 4 & 4 & 0 & 0 \end{pmatrix}$$

Wie bekommen wir denn daraus den kürzesten Weg von S nach V heraus? Dieser Weg muß über `t[1,5]` $= 3$ (das ist U) laufen. Der „linke" Weg $S \rightsquigarrow U$ läuft (wegen der OTE!) über `t[1,3]` $= 2$ (das ist X). Beim „rechten" Weg $U \rightsquigarrow V$ finden wir `t[3,5]` $= 0$! Was bedeutet dies? Anscheinend ist der Algorithmus nicht dazu gekommen, diesen Weg trennen zu müssen. Kein Wunder: Es gibt eine Kante dazwischen.

Wenn also in der t-Matrix eine 0 steht, gibt es offenbar eine Kante zwischen den beiden Ecken. Man wäre also fertig. Zurück zu $S \rightsquigarrow U$! Er läuft – wie wir wissen – über X und die entsprechenden Einträge `t[1,2]` und `t[2,3]` sind beide 0. Im Prinzip kann man sich das an einem Baum klarmachen:

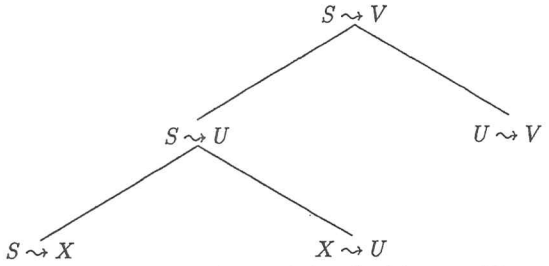


Abbildung 10.3: Rekonstruktion des kürzesten Wegs

Die Reihenfolge der Ecken liest man nun dadurch ab, daß man die Blätter von links nach rechts abarbeitet. Die müssen „wie Dominosteine" aneinanderpassen, sonst ist etwas falsch. Also schaut der kürzeste Weg so aus:

$$S \rightsquigarrow X \rightsquigarrow U \rightsquigarrow V.$$

Was nun kommt, ist klar:

Aufgabe: *Geben Sie einen rekursiven Algorithmus an, der aus der* t*-Matrix zu zwei Ecken, den kürzesten Pfad von einer zur anderen angibt!*

10.2 Der 0-1 Rucksack mit Dynamischer Programmierung

Nun aber wieder an's Rucksackpacken! Wir wollen zeigen, daß sich der *0-1 Rucksack* mit DP lösen läßt. Dazu muß man sich überlegen, daß der 0-1 Rucksack die OTE besitzt. Das kann man sich einfach klarmachen.

Nehmen wir an, wir hätten einen optimalen Rucksack R dessen Gesamtgewicht $w \leq w_{max}$ ist. Nehmen wir einmal einen beliebigen Gegenstand j heraus. Der übrigbleibende Rucksack R′ hat dann ein Gesamtgewicht von $w' = w - w_j$.

Er muß für dieses Gewicht w' optimal sein, wenn wir uns den Gegenstand j wegdenken. Das ist klar, denn gäbe es einen besseren Rucksack R″ mit diesem Gewicht, würden wir Gegenstand j in R″ einpacken und einen höheren Wert als mit R erzielen. Die Idee (und darauf muß man erst kommen!) liegt also darin, optimale Rucksäcke für alle Kapazitäten w mit $0 \leq w \leq w_{max}$ zu konstruieren.

Unsere erste DP-Tabelle werden wir also als Array `valtab[0:wmax]` definieren. Sie enthält auf dem Index i den größten Wert, den wir aus den gegebenen Gegenständen in den Rucksack packen können, ohne daß das Gesamtgewicht dieser Gegenstände größer als i wird. Offenbar gilt `valtab[0]` $= 0$.

Nun haben wir noch das Problem, daß wir für jede dieser Kapazitäten auch noch festhalten müssen, welche Gegenstände dazu beigetragen haben. Das kostet Platz! Am einfachsten wird das, wenn wir noch eine Matrix `taken[0:wmax,1:n]` definieren, die boolesche Werte enthält `taken[i,j]`, die anzeigen, ob für die optimale Füllung des i–ten Rucksacks der Gegenstand j genommen wurde. Damit bekommen wir allerdings einen Platzbedarf von $\Theta(w_{max} \cdot n)$.

Nun muß man nur überlegen, wie man den Rucksack $\mathtt{R}_k$ optimal füllt. Man geht lediglich davon aus, daß die optimalen Rucksäcke für die Kapazitäten $< k$ schon bekannt sind. Das ist aber genau das, was wir dann DP-mäßig sicherstellen werden.

Man betrachtet alle Gegenstände $i, i = 1, \ldots n$, daraufhin, ob sie überhaupt in $\mathtt{R}_k$ reinpassen *könnten*, ob sie also die Eigenschaft $w_i < k$ haben.

Wenn das der Fall ist, schauen wir im optimalen Rucksack $\mathtt{R}_{k-w_i}$ nach, der also das Gewicht $k - w_i$ hat, ob i schon drin ist. Wenn das der Fall ist, vergessen wir die Geschichte, denn wir können i ja nicht zweimal einpacken.

Sollte allerdings i nicht in diesem Rucksack sein, testen wir, ob sich der Wert von $\mathtt{R}_k$ durch Hinzunahme von i verbessern läßt. Erste Grobversion:

```
procedure knapsack-dp(w_max,n)

  {Lokale Daten: valtab[0:w_max]    ganzzahlig
                 taken[0:w_max,1:n] Wahrheitswerte}

  for i := 0 to w_max do      {Alle Rucksaecke}
     valtab[i] := 0           {initialisieren!}
```

```
      for j := 1 to n do
         taken[i,j] := false

   for k := 1 to w_max do      {Ueber alle Kapazitaeten}
      for i := 1 to n do       {Ueber alle Gegenstaende}
         if w[i] <= k then
            {Gegenstand i koennte passen}
            {Gucken im entsprechenden Rucksack}
            {Kann man durch i Rucksack k noch verbessern?}
```

Nun die Passage, in der wir im entsprechenden kleineren Rucksack nachgucken. Der hat offenbar den Index $k' = k - w_i$. Ob i drin ist, sieht man an `taken[k',i]`:

```
   k' := k - w[i];
   if not taken[k',i] then

      {Test, ob valtab[k] verbessert werden kann}
      {Wenn ja, Rucksack k' durch Hinzunehmen von i}
      {zum neuen besten Rucksack k machen}

      v_new := valtab[k'] + v[i]      {Hypothetischer Wert}

         if valtab[k] < v_new then    {Besser als der alte?}
            valtab[k] := v_new        {Ja! Neuen Wert merken}

            for j := 1 to n do        {Inhalt von k' nach k}
               taken[k,j] := taken[k',j]

            taken[k,i] := true        {zuzueglich i}
```

Algorithmus 10.2 *0-1 Rucksack mit begrenzter Anzahl von Gegenständen gelöst durch DP.*

Precondition: Gewichte `w[1:n]`, Werte `v[1:n]`, Maximalgewicht `w_max`

Postcondition: `valtab[0:w_max]` enthält Teilwerte, `taken[0:w_max,1:n]` Markierung mitzunehmender Gegenstände

```
   procedure knapsack-dp(w,v,w_max,n; var valtab,taken)

      for i := 0 to w_max do      {Alle Rucksaecke}
         valtab[i] := 0           {initialisieren!}
         for j := 1 to n do
            taken[i,j] := false

      for k := 1 to w_max do      {Ueber alle Kapazitaeten}
```



```
      for i := 1 to n do      {Ueber alle Gegenstaende}
         if w[i] <= k then
            k' := k - w[i];
            if not taken[k',i] then
               v_new := valtab[k'] + v[i]

               if valtab[k] < v_new then
                  valtab[k] := v_new

                  for j := 1 to n do
                     taken[k,j] := taken[k',j]

                  taken[k,i] := true
```

Ein Blick auf die Ergebnisse dieses Programms mit unserem Beispiel vom Anfang des Kapitels. Wir geben natürlich nicht *alle* Werte von `valtab` und `taken` an, sondern nur die interessanten. Die Spalte `w` gibt das aktuelle Gewicht des Rucksacks an:

k	valtab[k]	taken[k,1]	taken[k,2]	taken[k,3]	w
0	0	false	false	false	0
		...			
9	0	false	false	false	0
10	60	true	false	false	10
		...			
19	60	true	false	false	10
20	100	false	true	false	20
		...			
29	100	false	true	false	20
30	160	true	true	false	30
		...			
39	160	true	true	false	30
40	180	true	false	true	40
		...			
49	180	true	false	true	40
50	220	false	true	true	50

Natürlich ist die Laufzeit in diesem Fall sehr schlecht — man überzeugt sich, daß im schlechtesten Fall die Laufzeit $O(w_{max} \cdot n^2)$ ist — und auch der hohe Speicherbedarf ist unangenehm. Der Speicherbedarf war ja $\Theta(w_{max} \cdot n)$, was zu einem großen Teil darauf beruht, daß wir hier gleich eine ziemlich harte Rucksackvariante betrachtet haben: Jeder Gegenstand war nur einmal verfügbar und wir mußten uns für jeden Rucksack merken, ob er schon drin ist.

Läßt man diese Bedingung fort, so kommt man zu einer etwas einfacheren Variante des 0-1 Rucksacks, in der jeder der Gegenstände beliebig oft vorhanden ist. In unserem Beispiel hieße das, daß in der Villa eine ganze Kiste voller Algorithmenbücher, ein ganzes Regal mit Taschenrechnern und ein riesiger Satz von solchen Heapsort-Statuen herumsteht.

Aufgabe: *Überzeugen Sie sich davon, daß sie unter dieser Bedingung einfach 5 Algorithmenbücher einpacken sollten.*

Aufgabe: *Lösen Sie das entsprechende Problem mit folgenden Werten:*

w_{max} :	20				
n :	5				
w_i :	3	4	7	8	9
v_i :	4	5	10	11	13

Aufgabe: *Entwerfen Sie einen DP-Algorithmus, der dieses Problem allgemein löst. Sie brauchen das Array* `taken` *also nicht, weil sie jeden Gegenstand sooft einpacken können, wie sie wollen. Die Speicherkomplexität des Verfahrens wäre dann $\Theta(w_{max})$ und die Laufzeit $O(w_{max} \cdot n)$.*

Man muß dann allerdings aufpassen, daß man hinterher die Gegenstände erfährt, die zu der entsprechenden Füllung beigetragen haben! **Tip:** *Ähnlich wie beim APSP-Problem kann man sich zu jedem Rucksack das letzte Element i merken, das man hineingetan hat und daraus die optimale Füllung rückwärts ausrechnen.*

Aufgabe: *Wenn n „ausreichend klein" gegen w_{max} ist, kann es sich lohnen, nicht alle kleineren Rucksäcke anzugucken, sondern nur die, die sich tatsächlich aus diesen Gegenständen zusammensetzen lassen. Man erhält dann Varianten, die Speicherkomplexität von $\Theta(2^n \cdot n)$ anstelle von $\Theta(w_{max} \cdot n)$ (im Beispiel mit den einmaligen Gegenständen) bzw. $\Theta(2^n)$ anstelle von $\Theta(w_{max})$ (im Beispiel mit den beliebig oft vorhandenen Gegenständen) aufweisen. Entwerfen Sie eine Modifikation der beiden Algorithmen, die auf dieser Idee basieren.*

10.3 Parsing kontextfreier Sprachen

Der nächste DP-Algorithmus ist vielleicht einer der bekanntesten, die es so gibt. Er stellt ein Verfahren dar, mit dem jede kontextfreie Sprache (gegeben durch eine kontextfreie Grammatik) in kubischer Zeit geparst werden kann. Den theoretischen Hintergrund liefern [Aho *et al.*, 1988], [Hopcroft und Ullman, 1979] und [Stetter, 1988]. Aber (wie immer) gibt's zunächst etwas vorzubereiten. Stichwort: *Chomsky-Normalform.*

10.3.1 Die Chomsky-Normalform

Definition 10.1 *Eine kontextfreie Grammatik ist in* **Chomsky-Normalform** *(CNF), wenn jede Regel von der Form ist*

1. *$A \to BC$, mit nicht-terminalem B und C oder*
2. *$A \to a$, mit terminalem a.*

Es gilt

Aussage 10.1 *Zu jeder kontextfreien Sprache L ohne ϵ gibt es eine kontextfreie Grammatik G in CNF, so daß $L = L(G)$.*

Für unsere Fälle wichtiger ist

Aussage 10.2 *Für jede gegebene kontextfreie Grammatik G, die ϵ nicht erzeugt, kann eine äquivalente Grammatik G' in CNF effektiv konstruiert werden.*

Der entsprechende Algorithmus ist hier nicht von Interesse; man kann ihn etwa im „Drachenbuch" [Aho *et al.*, 1988] nachlesen. Warum auch die Komplexität dieses Algorithmus nicht von Interesse ist, sehen wir später. Der DP-Algorithmus geht nun davon aus, daß zuerst unsere Grammatik in CNF gebracht worden ist.

10.3.2 Der DP-Ansatz

Nehmen wir zunächst einmal an, wir hätten eine Grammatik mit dem nicht-terminalen Zeichenvorrat V_N, dem Startsymbol S, der Regelmenge P und ein gewisses Eingabewort $w = a_1 \ldots a_n \in V_T^*$ gegeben. Nun definieren wir uns eine Matrix `A[i,j]`, $\mathtt{i}, \mathtt{j} = 1, \ldots, n$ folgendermaßen:

$$\mathtt{A[i,j]} \stackrel{\text{def}}{=} \{S \in V_n \mid S \Rightarrow^* a_i \ldots a_j\} \text{ für } i \leq j.$$

Also: `A[i,j]` ist die Menge aller nicht-terminaler Symbole, aus denen man mit den Regeln der Grammatik das Teilwort $w = a_i \ldots a_j$ $(i \leq j)$ abgeleitet werden kann.

Erste Frage: Wie hilft uns A bei der Lösung des Parsingproblems? *Antwort:* Wir betrachten `A[1,n]` und prüfen, ob das Startsymbol S drin ist. Wenn ja, so ist das gesamte Eingabewort aus S ableitbar und gehört somit zur Sprache der Grammatik.

Offenbar kann man `A[i,j]` nicht mit einem Schlag berechnen. Wir brauchen also eine Möglichkeit, über eine gewisse Größe zu iterieren. Die wird durch die *Länge* `k` *des betrachteten Teilworts* geliefert.

Die Initialisierung beginnt naheliegenderweise mit `k` $= 1$, also mit den Teilworten, die nur ein Symbol enthalten. Also muß gelten $i = j$; wir starten also mit der Besetzung der Hauptdiagonalen `A[i,i]`.

Hier kommt zum ersten Mal die CNF ins Spiel: Man kann ja das Teilwort a_i nur über Regeln der Form $A \to a_i$ ableiten und das kann man durch einfaches Nachgucken in der Regelmenge erledigen.

Initialisierung:

```
for i := 1 to n do
   A[i,i] := {Alle nicht-terminalen Zeichen X}
             {mit einer Regel X ---> a_i}
```

Nun zum Iterationsschritt. Wir betrachten alle Teilworte der Länge l: Nehmen wir an, wir hätten ein solches Teilwort $a_i \dots a_j$ der Länge l. Offenbar gelten zunächst folgende Tatsachen:

1. $1 \leq i \leq n - l + 1$,
2. $j = i + l - 1$.

Nun greifen wir zur selben Methode wie beim Floyd-Warshall Algorithmus: Wir nehmen das Teilwort bei einem gewissen Index k auseinander, bekommen also folgende Situation

$$a_i \dots a_k \quad a_{k+1} \dots a_j, \quad i \leq k < j.$$

Betrachten wir die entsprechenden „Teilteilworte" $a_i \dots a_k$ und $a_{k+1} \dots a_j$, so stellen wir fest, daß ihre Länge echt kleiner als l ist. Die Werte A[i,k] und A[k+1,j] sind also bereits berechnet. Anders ausgedrückt: Wir kennen bereits diejenigen nicht-terminalen Symbole, aus denen sie abgeleitet werden können. Nehmen wir zwei nicht-terminale Symbole X und Y an, für die

1. $X \Rightarrow^* a_i \dots a_k$ und
2. $Y \Rightarrow^* a_{k+1} \dots a_j$

gilt. Dann läßt sich aus dem Wort XY unser Teilwort $a_i \dots a_j$ ableiten, was leider für den gesuchten Eintrag A[i,j] noch nicht ausreicht. Da wollen wir ja einzelne nicht-terminale Symbole haben. Die CNF erlaubt es uns aber, Regeln der Form $A \to XY$ zu finden, so daß wir in einem solchen Fall das Symbol A in A[i,j] aufnehmen können.

Also lassen wir l von 2 (1 hatten wir schon!) bis n laufen, betrachten alle Teilworte der Länge l und trennen sie an den geeigneten Stellen durch den Trennungsindex k. Dann alle Symbole X aus A[i,k] und Y aus A[k+1,j] angucken und die Symbole A, für die es eine Regel $A \to XY$ gibt, in A[i,j] aufnehmen. Das war's dann auch schon:

Algorithmus 10.3 *Parsen einer kontextfreien Grammatik in CNF durch DP.*

Precondition: ---
Postcondition: A[i,j] enthält die nicht-terminalen Symbole, die das entsprechende Teilwort erzeugen

```
procedure cf-parse
   {Initialisierung:}
   for i := 1 to n do
      A[i,i] := {Alle nicht-terminalen Zeichen X}
                {mit einer Regel X ---> a_i}

   {Iteration:}
   for l := 2 to n do                {Laenge des Teilworts}
      for i := 1 to n-l+1 do         {Start des Teilworts}
```



```
         j := i + l - 1              {Ende des Teilworts}
         A[i,j] := emptyset
         for k := 1 to j-1 do        {Trennungsindex}
            for X in A[i,k] do
               for Y in A[k+1,j] do
                  A[i,j] := A[i,j] union
                            {Alle nicht-terminalen Zeichen A}
                            {mit einer Regel A ---> XY}
```

10.3.3 Beispiel

Die „berühmteste" kontextfreie Sprache $a^n b^n, n \geq 1$, wird erzeugt durch die Grammatik

$$S \to aSb \mid ab$$

Umformung in CNF ergibt

$$\begin{aligned} S &\to AX \mid AB \\ X &\to SB \\ A &\to a \\ B &\to b \end{aligned}$$

Nehmen wir das Wort $w = aabb$ an, so ergibt sich $n = 4$. Die Initialisierung mit l = 0 besetzt die Hauptdiagonale von A wie in Matrix (1) zu sehen ist. Die Punkte in der Matrix bezeichnen Einträge, die noch nicht berechnet wurden:

$$(1) \begin{pmatrix} \{A\} & \cdots & \cdots & \cdots \\ \cdots & \{A\} & \cdots & \cdots \\ \cdots & \cdots & \{B\} & \cdots \\ \cdots & \cdots & \cdots & \{B\} \end{pmatrix} \quad (2) \begin{pmatrix} \{A\} & \emptyset & \cdots & \cdots \\ \cdots & \{A\} & \{S\} & \cdots \\ \cdots & \cdots & \{B\} & \emptyset \\ \cdots & \cdots & \cdots & \{B\} \end{pmatrix}$$

$$(3) \begin{pmatrix} \{A\} & \emptyset & \emptyset & \cdots \\ \cdots & \{A\} & \{S\} & \{X\} \\ \cdots & \cdots & \{B\} & \emptyset \\ \cdots & \cdots & \cdots & \{B\} \end{pmatrix} \quad (4) \begin{pmatrix} \{A\} & \emptyset & \emptyset & \{S\} \\ \cdots & \{A\} & \{S\} & \{X\} \\ \cdots & \cdots & \{B\} & \emptyset \\ \cdots & \cdots & \cdots & \{B\} \end{pmatrix}$$

Alle vier Einträge in der Hauptdiagonalen kommen also durch „$A \to a$" oder „$B \to b$" zustande.

Im nächsten Schritt werden die Teilworte der Länge l = 2 betrachtet. Für den Eintrag A[1,2] liegt also das Teilwort „aa" zugrunde. Der einzige Trennungsindex k ist 1, wir betrachten also die Elemente

- A[1,1] = $\{A\}$ und
- A[2,2] = $\{A\}$.

X „läuft“ über $\{A\}$ und Y „läuft“ (auch) über $\{A\}$, d.h. wir suchen eine Regel mit „AA“ auf der rechten Seite. Die gibt es nicht in der Grammatik und so kommt das $\emptyset$ zustande.

Anders ist das bei `A[2,3]`! Dort haben wir das Teilwort „ab“ am Wickel, das mit dem Trennungsindex `k` $= 2$ getrennt wird. Die zugehörigen Einträge sind

- `A[2,2]` $= \{A\}$ und
- `A[3,3]` $= \{B\}$.

Wir finden dann für $X = A$ und $Y = B$ die Regel $S \to AB$, die zum $\{S\}$ an der Stelle `A[2,3]` führt. Das $\{X\}$ an der Stelle `A[2,4]` kommt wie folgt zustande: Das Teilwort ist „abb”, was die beiden Trennungsindizes `k` $= 2$ und `k` $= 3$ zuläßt. Betrachten wir beide Fälle:

1. `k` $= 2$. Für `A[2,2]` finden wir $\{A\}$, aber `A[3,4]` ist $\emptyset$, d.h. das X finden wir zwar, aber es gibt kein geeignetes Y. Dieses `k` liefert also keinen Beitrag zu `A[2,4]`.
2. `k` $= 3$. Hier brauchen wir `A[2,3]` $= \{S\}$ und `A[4,4]` $= \{B\}$. Der einzige Fall, der behandelt wird ist also der mit $X = S$ und $Y = B$. In der Tat gibt es eine Regel mit „SB“ auf der rechten Seite: $X \to SB$ und damit ist alles klar.

Aufgabe: *Machen Sie sich analog klar, wie das $\{S\}$ an der Stelle* `A[1,4]` *nach dem letzten Iterationsschritt entsteht.*

Nun lesen wir in `A[1,4]` ab, daß S in dieser Menge enthalten ist und schließen daraus, daß $w = aabb$ zur Sprache unserer Grammatik gehört.

Hier noch ein Gegenbeispiel, um zu prüfen, daß der DP-Algorithmus auch negativ antworten kann. Wir setzen $w = aaabb$, was offenbar nicht zu unserer Sprache gehört. Es ergeben sich folgende Schritte:

$$(1)\quad \begin{pmatrix} \{A\} & \cdots & \cdots & \cdots & \cdots \\ \cdots & \{A\} & \cdots & \cdots & \cdots \\ \cdots & \cdots & \{A\} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \{B\} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \{B\} \end{pmatrix} \qquad (2)\quad \begin{pmatrix} \{A\} & \emptyset & \cdots & \cdots & \cdots \\ \cdots & \{A\} & \emptyset & \cdots & \cdots \\ \cdots & \cdots & \{A\} & \{S\} & \cdots \\ \cdots & \cdots & \cdots & \{B\} & \emptyset \\ \cdots & \cdots & \cdots & \cdots & \{B\} \end{pmatrix}$$

$$(3)\quad \begin{pmatrix} \{A\} & \emptyset & \emptyset & \cdots & \cdots \\ \cdots & \{A\} & \emptyset & \emptyset & \cdots \\ \cdots & \cdots & \{A\} & \{S\} & \{X\} \\ \cdots & \cdots & \cdots & \{B\} & \emptyset \\ \cdots & \cdots & \cdots & \cdots & \{B\} \end{pmatrix} \qquad (4)\quad \begin{pmatrix} \{A\} & \emptyset & \emptyset & \emptyset & \cdots \\ \cdots & \{A\} & \emptyset & \emptyset & \{S\} \\ \cdots & \cdots & \{A\} & \{S\} & \{X\} \\ \cdots & \cdots & \cdots & \{B\} & \emptyset \\ \cdots & \cdots & \cdots & \cdots & \{B\} \end{pmatrix}$$

$$(5)\quad \begin{pmatrix} \{A\} & \emptyset & \emptyset & \emptyset & \emptyset \\ \cdots & \{A\} & \emptyset & \emptyset & \{S\} \\ \cdots & \cdots & \{A\} & \{S\} & \{X\} \\ \cdots & \cdots & \cdots & \{B\} & \emptyset \\ \cdots & \cdots & \cdots & \cdots & \{B\} \end{pmatrix}$$

Aufgabe: *Machen Sie sich bitte die einzelnen Schritte klar, insbesondere vergewissern Sie sich bitte, daß der Eintrag* `A[2,5]` $= \{S\}$ *sinnvoll ist.*

10.3.4 Komplexität

Zum Schluß ein Wort zur Komplexität des Verfahrens. Der relevante Teil der Iteration sieht wie folgt aus:

```
for l := 2 to n do                                  (1)
   for i := 1 to n-l+1 do                           (2)
      ...
      for k := 1 to j-1 do                          (3)
         for X in A[i,k] do                         (4)
            for Y in A[k+1,j] do                    (5)
               {Regeln durchsuchen}                 (6)
```

Wir haben i.w. sechs (!) Schleifen — (1) bis (6) — zu analysieren. Entscheidend für die Komplexität sind dabei folgende Überlegungen:

- Schleife (1) wird $n - 1 = O(n)$-mal durchlaufen.
- Schleife (2) wird $O(n)$-mal durchlaufen.
- Schleife (3) wird $O(n)$-mal durchlaufen.
- Schleife (4) wird $O(|V_N|)$-mal durchlaufen, denn mehr nicht-terminale Symbole gibt es nicht.
- Dasselbe gilt für Schleife (5).
- Das Durchsuchen der Regelmenge (Schleife (6)) benötigt $O(|P|)$ Schritte.

Insbesondere hängt der Aufwand für die Schleifen (4) bis (6) *nicht* von der Länge des Eingabeworts ab! Damit aber ergibt sich für den Algorithmus eine Komplexität von $O(n^3)$, wenn wir als Maß diese Länge zugrundelegen. Aus demselben Grund setzen wir für die eventuell notwendige Umwandlung der Grammatik in CNF rigoros $O(1)$ (also *konstant*) als Komplexität an. Alles in allem haben wir es hier also mit einem $O(n^3)$ Algorithmus für das Parsingproblem kontextfreier Sprachen zu tun, obwohl hier (wie oft bei der Komplexitätsanalyse) ein leicht saurer Geschmack im Mund verbleibt: Der als konstant angenommene Anteil kann in der Praxis doch recht beträchtlich sein und für die interessanten Fälle den „eigentlichen“ Algorithmus dominieren. Ganz abgesehen davon, daß es viel mühsamer ist, eine beliebige kontextfreie Grammatik maschinell in CNF zu bringen als den eigentlichen DP-Algorithmus zu implementieren.

So theoretisch dieses Ergebnis auch aussehen mag, dieser Algorithmus hat auch eine praktische Daseinsberechtigung. Natürlich liefern uns Theorie und Praxis des Compilerbaus wesentlich effizientere Zerlegungsalgorithmen unter den Stichworte „LL-Analyse“ und „LR-Analyse“ [Aho *et al.*, 1988]. Andere Anwendungen der Theorie formaler Sprachen, wie sie etwa im Rahmen der „Künstlichen Intelligenz“ im Bereich „Erkennen und Verstehen natürlicher Sprache“ vorliegen, können sich nicht grundsätzlich auf *deterministische Sprachen* zurückziehen.

Parser für natürliche Sprache sind durchaus daran interessiert, auch wenn die gesamte Äußerung unsyntaktisch ist wenigstens Teile kategorisieren zu können. In der Praxis führt das dann dazu, daß für alle Teile der fraglichen Äußerung Ableitungen aufgestellt werden, wie es auch im zweiten Beispiel (der Abweisung) der Fall war. Die in der *Computerlinguistik*, also derjenigen Disziplin, die sich mit der Analyse natürlicher Sprache durch

effektive Algorithmen befaßt, sogenannten *Chart-Parser* stellen eine weitere Form solcher Zerlegungsalgorithmen dar.

Natürlich sind solche übergreifenden Algorithmen sehr aufwendig und nur in Bereichen anwendbar, in denen relativ „kurze" Eingaben zu behandeln sind. Das ist allerdings genau der Unterschied zwischen Compilern und Systemen zur Behandlung natürlicher Sprache: Ein Compiler muß Eingaben mit einigen 1000 Eingabesymbolen effizient behandeln können, wobei Äußerungen in natürlicher Sprache in der Größenordnung von nur wenigen Symbolen liegen.

10.4 Schlußbemerkungen

Die Dynamische Programmierung bietet ein sehr mächtiges Hilfsmittel zur Lösung von Optimierungsproblemen, die sich dem Greedy-Ansatz entziehen. Sie ist in den verschiedensten Aufgabenbereichen anwendbar, was z.B. durch den angegebenen Parsing-Algorithmus illustriert wird, der ja eigentlich gar kein Optimierungsproblem löst.

Eine wichtige Klasse von Problemen, die durch einen DP-Ansatz gelöst werden können, läßt sich durch das Stichwort „Konstruktion der transitiven Hülle einer zweistelligen Relation" charakterisieren. Dabei ist eine zweistellige Relation R über endlichen Mengen in einer Form gegeben, die nicht berücksichtigt, daß R transitiv ist. Das klassische Beispiel ist die Adjazenzmatrix eines Graphen (3.2.3), aus der abgeleitet werden soll, welche Ecken überhaupt miteinander verbunden sind. Dieses Problem kann mit einem DP-Algorithmus, der dem von Floyd und Warshall sehr ähnlich sieht, in kubischer Zeit gelöst werden.

Auch für das bekannte „Problem des Handlungsreisenden" (engl. Travelling Salesman Problem, TSP) (vergl. [Maas, 1993]) existiert ein — wenn auch recht aufwendiges — DP-Verfahren zu seiner Lösung.

Kapitel 11

Branch and Bound

Diese letzte Optimierungsmethode ist ein drittes charakteristisches Beispiel für den verzweifelten Versuch, mit der kombinatorischen Explosion mancher Optimierungsprobleme fertig zu werden. Wir werden das Verfahren exemplarisch wieder am Rucksackproblem verfolgen, wofür es sich sehr gut eignet. Dazu betrachten wir den 0-1 Rucksack als *binäres Entscheidungsproblem*.

11.1 Der 0-1 Rucksack

Im Prinzip kann man davon ausgehen, daß sich der 0-1 Rucksack durch einen binären Entscheidungsbaum veranschaulichen kann:

> Am Wurzelknoten ist noch keine einzige Entscheidung darüber gefallen, welcher Gegenstand einzupacken ist. Dieser Knoten hat zwei Nachfolger: der linke repräsentiert die Entscheidung, den ersten Gegenstand mitzunehmen, der rechte die Entscheidung, ihn liegenzulassen. Auf der nächsten Ebene teilt sich das Verfahren für Gegenstand zwei entsprechend auf usw.

Man erkennt schnell, daß es sich hier um nichts anderes handelt, als eine Darstellung, wie man systematisch die gesamte Potenzmenge der Gegenstände exploriert. Das klingt natürlich ziemlich ineffizient, aber man kann das Verfahren in der Praxis deutlich verbessern. Dazu folgende Überlegung:

> Man kann an der Wurzel — wenn also noch alle Gegenstände möglich sind — eine *optimistische* Schätzung darüber machen, wie hoch der maximale Wert wohl sein kann: Die Summe der Werte aller Gegenstände ist das beste, was erzielt werden kann. Wenn man nun vor einer Entscheidung steht, einen Gegenstand liegenzulassen oder mitzunehmen, kann die Schätzung entsprechend korrigiert werden:
>
> 1. Wenn man ihn mitnimmt, ändert sich die Schätzung bezüglich des zu erzielenden Gesamtwerts nicht.

2. Wenn man ihn liegenläßt, so kann sein Wert nicht mehr erzielt werden und man muß die Schätzung für die entsprechende Variante um diesen Wert heruntersetzen.

Was ist nun mit den Gewichten? An der Wurzel kann man noch das maximale Gewicht an Gegenständen w_{max} aufnehmen. Was geschieht bei einer Entscheidung?

1. Wenn man den Gegenstand mitnimmt, so verringert sich die verbleibende „Kapazität" um sein Gewicht.
2. Wenn man ihn liegenläßt, so ändert sich die „Kapazität" nicht.

Man sieht, wie beide Werte bei einer Entscheidung gegeneinanderlaufen:

1. Gegenstand mitnehmen: Schätzung des Werts unverändert
Kapazität nimmt ab

2. Gegenstand liegenlassen: Schätzung des Werts nimmt ab
Kapazität unverändert

Wir ordnen also jedem Knoten i des Baums *zwei* Werte zu:

1. b_i : Der maximale Wert, der unterhalb dieses Knotens im allerbesten Fall noch erzielt werden kann. Dieser Wert wird als *bound* (Grenze) des Knotens bezeichnet.
2. f_i : Die Kapazität, die in diesem Knoten noch zur Verfügung steht. Dieser Wert wird als *restriction* Schranke) des Knotens bezeichnet.

b_i ist also derjenige Wert, der dann zustandekäme, wenn man ab sofort nur noch Gegenstände mitnähme. Diese Schätzung ist völlig unabhängig davon, ob das tatsächlich geht und deshalb sehr optimistisch! f_i ist die Last, die man an diesem Knoten noch aufnehmen könnte.

Wir stellen jeden Knoten in der Form b_i/f_i dar. Wir nehmen uns mal zur Abwechslung einen anderen Rucksack her ($n = 5$):

i	1	2	3	4	5
w_i	17	16	21	8	12
v_i	11	14	16	8	7

w_{max} sei 37. Die Summe der Werte ist $11 + 14 + 16 + 8 + 7 = 56$. Der Wurzelknoten des Entscheidungsbaums heißt also 56/37.

Nun betrachten wir seine beiden Nachfolger. Wir treffen dabei die Vereinbarung, daß der linke Nachfolger immer heißt „Gegenstand mitnehmen" und der rechte immer „Gegenstand liegenlassen". Wenn wir nun Gegenstand 1 mitnehmen (linker Nachfolger), so bleibt der maximale Wert konstant 56, aber die Kapazität nimmt um sein Gewicht 17 ab. Der Knoten bekommt die Markierung 56/20.

Lassen wir Gegenstand 1 liegen (rechter Nachfolger), so können wir dann bestenfalls $56 - 11 = 45$ erzielen, aber wir können immer noch 37 Einheiten tragen. Dieser Knoten bekommt entsprechend die Markierung 45/37. Die ersten beiden Ebenen:

Die erste Regel dieses Verfahrens ist nun eine wesentliche Heuristik:

> *Verzweigungsregel:* Es wird immer in demjenigen Nachfolger weitergesucht, dessen bound am größten ist. Die anderen müssen noch warten.

Hiervon kommt dann auch letzlich der seltsame Name des Verfahrens: *Branch and Bound*, auf Deutsch „Verzweige und Begrenze". Dem Verzweigungsschritt (die Ermittlung der Nachfolgeknoten) folgt der Begrenzungsschritt, in dem der vielversprechendste Knoten (hier mit dem größeren bound) als nächster betreten wird.

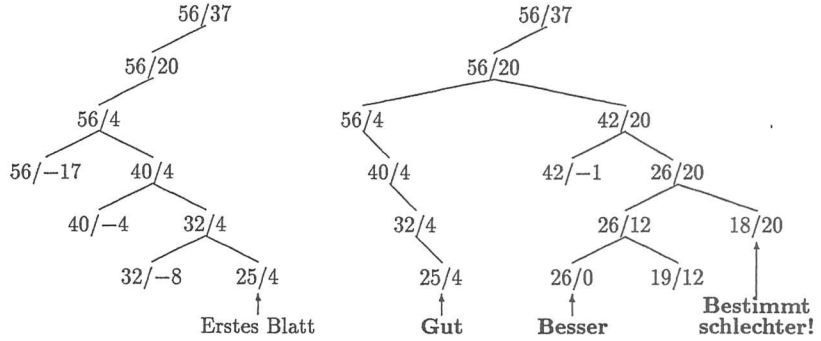


Abbildung 11.1: Anfang der Branch-and-Bound-Suche

In unserer Darstellung sieht das dann immer so aus, daß wir den linken Nachfolger zuerst weiterverfolgen. Suchen wir also unterhalb 56/20 weiter. Wir stehen nun vor der Entscheidung betreffend Gegenstand 2. Mitnehmen gibt den linken Nachfolger 56/4 und liegenlassen den rechten Nachfolger 42/20.

Wenn wir nun unterhalb von 56/4 nachgucken und Gegenstand 3 einpacken, bekommen wir den seltsamen Knoten 56/−17, also mit negativem f_i. Die Interpretation ist einfach: Wir wären überladen. Damit kommen wir zur zweiten Regel dieses Verfahrens:

> *Zulässigkeitsregel:* Wenn ein Knoten i eine negative restriction f_i hat, wird die Suche dort abgebrochen.

Wir steigen nun schnell zum ersten Blatt herab (s. Abb. 11.1, links). Einige unzulässige Knoten werden ignoriert und das Blatt mit der Markierung 25/4 wird erreicht.

Das ist nun die erste legale Lösung: Gegenstände 1 und 2 sind eingepackt, Wert ist 25 und wir könnten sogar noch 4 Einheiten mehr tragen, die aber nicht da sind.

Wir merken uns diesen garantiert erreichbaren Wert 25 (z.B. auf einer globalen Variablen). Wenn wir einmal so einen garantierten Wert haben, können wir eine ganz einfache Regel formulieren, die den dritten Teil dieses Verfahrens darstellt:

> *Schnittregel:* Wenn ein Wert V garantiert erreichbar ist, so brauchen Knoten mit einem bound $b \leq V$ *nicht weiter untersucht* zu werden.

Im rechten Teil von Abb. 11.1 sieht man die Schnittregel einmal in Aktion. Zunächst wurde also der Wert $V = 25$ im ersten zulässigen Blatt gefunden. Große Auswirkungen zeigt das allerdings nicht, denn beim Weiterverfolgen des Knotens 42/20, der ja nun anstünde, fällt lediglich 42/−1 raus, weil er nicht zulässig ist.

In diesem Stil arbeitet sich das Verfahren dann hinunter bis zur zweiten Lösung 26/0, die den globalen Wert V auf 26 verbessert. Diese Lösung ist also besser als die, die wir zuerst gefunden haben.

Der Knoten 18/20 (rechter Nachfolger von 26/20) ist bisher noch nicht untersucht worden. Das wird er auch nicht, denn das beste, was wir erzielen können, wären 18 und das wird von der Schnittregel verboten. Fazit: Der ganze Teilbaum mit der Wurzel 18/20 wird ignoriert! So ignorieren wir auch noch den Baum mit der Wurzel 19/12 (rechts unterhalb von 26/12), aber das bringt nicht mehr sehr viel, weil das sowieso ein Blatt ist! Einen Überblick über den gesamten Lösungsprozeß gibt Abb. 11.2.

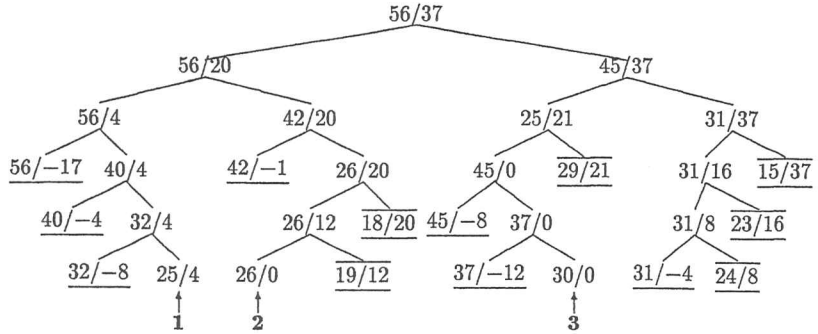


Abbildung 11.2: Kompletter Baum der Branch-and-Bound-Suche

Die unterstrichenen Knoten sind nicht zulässig und werden ignoriert. Der Knoten 30/0 wird als nächstes Blatt gefunden und verbessert das globale V auf den Wert 30. Das führt letzlich dazu, daß die unter- und überstrichenen Knoten 29/21, 24/8, 23/16 und erfreulicherweise wird auch der ziemlich weit oben angesiedelte 15/37 nicht mehr betreten.

Wir haben also mit dem Knoten 30/0 die optimale Lösung gefunden. Die Gegenstände, die wir dafür eingepackt haben, können wir offenbar an der Kantenfolge ablesen.

Der volle binäre Entscheidungsbaum für dieses Problem hätte $2^6 - 1 = 63$ Knoten gehabt, unser Verfahren hat gerade mal 31 davon betreten. Es gäbe $2^5 = 32$ Blätter, von denen nur 8 geprüft worden sind.

Nun aber an die Programmierung! Bevor wir uns aber an den Rucksack machen, einmal kurz sehr abstrakten Pseudocode für Branch and Bound. Wir nehmen der Einfachheit halber an, daß wir die Knoten des Baums ähnlich wie `Pascal RECORD`s anlegen. Die Idee ist bestechend, in der Praxis aber nicht ganz ungefährlich. Man sollte davon absehen, Knoten dynamisch zu kreieren, wenn man nicht ganz sicher ist, wann man sie wieder loswerden kann. Wie man das realisiert, sehen wir dann am Rucksackbeispiel.


```
procedure explore-if-legal(node)
  {Knoten node betreten, wenn der legal ist und}
  {wenn er das globale Optimum verbessern koennte}
  if node.f >= 0 and node.b > V then
     explore(node)


procedure explore(node)
  {Knoten node wurde soeben betreten}

  if {node ist ein Blatt} then
     if node.b > V then          {bound besser als globales Optimum}
        V := node.b              {Also verbessern!}

  else                           {Kein Blatt! Nachfolger berechnen}
     left  := {Linker Nachfolger}
     right := {Rechter Nachfolger}

     {Den mit dem besseren bound zuerst untersuchen}

     if left.b > right.b then
        explore-if-legal(left)    {left zuerst}
        explore-if-legal(right)   {right danach}
     else
        explore-if-legal(right)   {right zuerst}
        explore-if-legal(left)    {left danach}


procedure branch-and-bound
  {Initialisierung und Start von Branch and Bound}
  V     := -infinity            {Globales Optimum}
  start := {Wurzelknoten}       {Wurzelknoten aufbauen}
  explore(start)                {Ab in die Wurzel!!!}

  {V enthaelt nun den besten Wert!}
```

Es sind hier drei Unterprogramme angegeben:

1. `explore-if-legal` testet *vor* dem Eintreten in den Knoten `node`, ob der legal ist (`node.f >= 0`) und ob der bisher beste globale Wert `V` noch nicht unterschritten wurde.

2. `explore` ist das eigentliche Arbeitspferd des Verfahrens. Es prüft zunächst, ob es sich beim entsprechenden Knoten um ein Blatt handelt (legal ist es allemal). Wenn das der Fall ist, betrachtet es den bound des Blatts, der seinen Wert darstellt und verbessert das globale Ergebnis unter Umständen. Wenn es kein Blatt ist, werden die beiden Nachfolger `left` und `right` bestimmt und dann der mit dem höheren bound rekursiv zuerst betreten.

3. **branch-and-bound** tut nicht mehr sehr viel. Es initialisiert den besten globalen Wert **V** so, daß er auf dem ersten Blatt sofort modifiziert wird, baut den Startknoten auf und beginnt mit der Arbeit.

Nun der Rucksack: Wir deklarieren uns *zwei* Knoten, mit den Namen **bestnode** und **worknode**. **bestnode** wird die gesamte Information des bisher besten Blatts festhalten (also bound, restriction und was noch so alles gebraucht wird). **worknode** ist dann der Knoten, der die Information des aktuell explorierten Knotens enthält. Das einzige Kunststück ist dabei nur, **worknode** beim Verlassen der Rekursion so zu hinterlassen, wie man ihn vorgefunden hat. Wir nehmen also an:

- w_{max} ist das maximale Gewicht, das wir tragen können,
- n ist die Anzahl der Gegenstände, zwischen denen wir uns entscheiden können,
- b ist der *bound*-Eintrag eines Knotens,
- f ist der *restriction*-Eintrag eines Knotens und
- `taken[1:n]` ist ein boolesches Array, das für jeden Gegenstand angibt, ob er mitgenommen wurde (`true`) oder nicht (`false`).

Algorithmus 11.1 *Initialisierung der* Branch and Bound *Suche.*

Precondition: ---
Postcondition: `bestnode enthält optimale Lösung`

```
procedure knapsack-bb
  {bestnode initialisieren}
  bestnode.b := -infinity

  {worknode initialisieren}
  worknode.f := w_max                    {restriction}

  {bound ist Summe ueber alle v[i]}
  worknode.b := 0
  for i := 1 to n do
     worknode.b := worknode.b + v[i]

  for i := 1 to n do        {Kein Gegenstand eingepackt}
     worknode.taken[i] := false

  {worknode betreten; Gegenstand 1 angucken}
  explore(worknode,1)
```

Nun codieren wir unter denselben Bedingungen das Betreten eines Knotens:

Algorithmus 11.2 *Betreten eines Knotens bei* Branch and Bound *Suche.*

Precondition: `node.f >= 0`
Postcondition: ---


```
procedure explore(node,index)

  {Betreten von node}
  {assert: node.f >= 0}

  {1. Test, ob node ein Blatt ist}

  if index > n then                    {Dann ist node ein Blatt!}
     if node.b > bestnode.b then
        {node ist besser als bestnode}
        {Also bestnode verbessern}
        bestnode := node
  else
     {node ist kein Blatt: Nachfolger bestimmen}
     {Linker Nachfolger zuerst: Gegenstand index}
     {wird eingepackt, also b bleibt unveraendert,}
     {also linker Nachfolger wird zuerst untersucht}

     node.taken[index] := true                {Einpacken}
     node.f            := node.f - w[index]

     {Testen, ob legal und erforschungswuerdig}

     if node.f >= 0 and node.b > bestnode.b then
        explore(node,index+1)

     {Gegenstand index auspacken. f und b entsprechend}
     {geaendert ergibt dann den rechten Nachfolger}

     node.taken[index] := false
     node.f            := node.f + w[index]
     node.b            := node.b - v[index]

     {Testen, ob legal und erforschungswuerdig}
     if node.f >= 0 and node.b > bestnode.b then
        explore(node,index+1)

     {Letzte Gemeinheit! b wurde geaendert}
     {Also zuruecksetzen, bevor wir es nach oben zurueckgeben}

     node.b := node.b + v index
```

11.2 Schlußbemerkungen

Branch and Bound dient in erster Linie dazu, die Untersuchung eines sehr großen Problemraums dadurch zu ermöglichen, daß durch seine Regeln die Anzahl untersuchter Knoten re-

duziert wird. Insbesondere die Schnittregel ist dafür verantwortlich, daß unter Umständen große Teilbäume nicht untersucht werden müssen. Durch die Verzweigungsregel wird der Suchprozeß in eine Richtung gelenkt, in der wahrscheinlich (!) gute Lösungen zu finden sind, so daß dann schnell solche Schnitte auftreten. In dieser Hinsicht ist die Verzweigungsregel eine Heuristik: Sie kann den Aufwand für das Finden einer optimalen Lösung reduzieren, ihr genauer Effekt ist allerdings nur schwer zu quantifizieren.

Die Methode läßt sich mit etwas Phantasie für verschiedene Probleme einsetzen. So wird in [Maas, 1993] der Einsatz von Branch and Bound für ganzzahlige Lineare Optimierung dargestellt. Auch das bereits erwähnte TSP kann mit dieser Methode gelöst werden [Oberschelp und Wille, 1976].

Literaturverzeichnis

[Aho *et al.*, 1988] A. V. Aho, R. Sethi, und J. D. Ullman. *Compilers — Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1988. Auch in deutscher Sprache erhältlich.

[Bayer und McCreight, 1972] R. Bayer und E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[Bayer, 1972] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(3):290–306, 1972.

[Brassard und Bratley, 1988] G. Brassard und P. Bratley. *Algorithmics.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Cormen *et al.*, 1990] T. H. Cormen, C. E. Leiserson, und R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[Graham *et al.*, 1994] R. L. Graham, D. E. Knuth, und Patashnik O. *Concrete Mathematics — A Foundation for Computer Science.* Addison-Wesley, Reading, MA, 1994.

[Hoare, 1962] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[Hopcroft und Ullman, 1979] J. E. Hopcroft und J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979. Auch in deutscher Sprache erhältlich.

[Horowitz und Sahni, 1978] E. Horowitz und S Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, MD, 1978.

[Kemp, 1984] R. Kemp. *Fundamentals of the Average Case Analysis of Particular Algorithms.* Wiley–Teubner Series in Computer Science. John Wiley and Sons, Teubner Verlag, Stuttgart, 1984.

[Knuth, 1968] D. E. Knuth. *The Art of Computer Programming — Volume 1: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1968.

[Knuth, 1969] D. E. Knuth. *The Art of Computer Programming — Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, MA, 1969.

[Knuth, 1973] D. E. Knuth. *The Art of Computer Programming — Volume 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[Maas, 1993] C. Maas. *Graphentheorie und Operations Research für Studierende der Informatik.* Wißner Verlag, Augsburg, 1993.

[Mehlhorn, 1988] K. Mehlhorn. *Datenstrukturen und effiziente Algorithmen — Band 1: Sortieren und Suchen.* Leitfäden und Monographien der Informatik. Teubner Verlag, Stuttgart, 1988.

[Oberschelp und Wille, 1976] W. Oberschelp und D. Wille. *Mathematischer Einführungskurs für Informatiker.* Studienbücher Informatik. Teubner Verlag, Stuttgart, 1976.

[Reischuk, 1990] K. R. Reischuk. *Einführung in die Komplexitätstheorie.* Teubner Verlag, Stuttgart, 1990.

[Sedgewick, 1992] R. Sedgewick. *Algorithms in C++.* Addison-Wesley, Reading, MA, 1992. Auch in deutscher Sprache erhältlich.

[Stetter, 1988] F. Stetter. *Grundbegriffe der Theoretischen Informatik.* Springer Verlag, Berlin, 1988.

[Williams, 1964] J. W. J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, (7):347–348, 1964.

[Wirth, 1979] N. Wirth. *Algorithmen und Datenstrukturen.* Studienbücher Informatik. Teubner Verlag, Stuttgart, 1979.

Index

0-1 Rucksack, 164, 202
  mit Branch and Bound, 211
  mit DP, 201
3-Band-Mischen,
  s. Sortieren
3-Band-Sortieren,
  s. Sortieren

Abstrakter Datentyp,
  s. ADT
Adjazenzliste,
  s. Graph
Adjazenzmatrix,
  s. Graph
ADT, 12, 13
  Implementation, 16
  innerer Zustand, 15
  Invariante, 15, 104
  Notation, 16
  Schnittstelle, 14
  Spezifikation, 14
ADT tape, 76
ADT triple-tape, 79
Algorithmus, 11
  Effizienz, 21
  Komplexität, 21
    Ω-Notation, 25
    Θ-Notation, 26
    für rekursive A., 28
    O-Notation, 23
Allgemeine Baumeigenschaft, 53
APSP-Algorithmus, 187, 195,
  s. auch Floyd-Warshall-Algorithmus
  rekursiv, 196
Asymptotische Analyse, 22
Auswahlfunktion, 165, 172
Auswahlschritt, 172, 189

B-Baum, 126, 129
  Aufbau, 132
  Balancieren eines Knotens, 139, 143
  Borgen eines Schlüssels, 138, 140, 141
  Einfügen, 132, 134, 136
  Komplexität, 148
  Löschen, 138, 145, 146, 148
  Ordnung, 129
  Spalten eines Knotens, 132
  Suchen, 131
  Untervoller Knoten, 138
  Verschmelzen von Knoten, 139, 142, 143
  Voller Knoten, 130
Baum, 38
  balancierter, 127
  binärer, 41
  n-ärer, 40
Bester Fall, 21
Bezeichnertabelle, 92
Binädarstellung, 177
Binärbaum
  sortierter, 101
    Einfügen, 104
    Löschen, 108
    Suchen, 103
  voller, 41
  vollständiger, 41
    Realisierung durch Arrays, 42
Binärbaumeigenschaft, 53
Branch and Bound, 211, 213
  Begrenzungsschritt, 213
  Bound, 212
  Initialisierung, 216
  Restriction, 212
  Schnittregel, 214
  Verzweigungsregel, 213
  Verzweigungsschritt, 213
  Zulässigkeitsregel, 213
Breitensuche, 159, 160, 162

Codierung

feste, 177
präfix-freie, 178
variable, 178

DAG, 39, 45, 154
Einstieg, 155
Datenkompression, 177
Datenstruktur, 11
Datenstrukturen, 33
lineare und kreisförmige, 33
Datenverwaltung, 127
Depth-first search,
s. Tiefensuche
Deque, 37
Dictionary, 19
Dijkstra-Algorithmus,
s. auch Kürzeste Wege, 191
Relaxation einer Ecke, 187
Relaxationsschritt, 188, 190, 191
Directed acyclic graph,
s. DAG
Disjunkte Mengen, 166, 169
Pfadkompressions-Heuristik, 167
Repräsentant, 166
DP, 195
Parsing, 204
DP-Algorithmus,
s. DP
Dynamic Programming,
s. DP
Dynamische Programmierung,
s. DP

Externe Blätter, 110

Fehlerbehandlung, 13
FIFO-Liste,
s. Stack
Floyd-Warshall-Algorithmus, 195, 198
Konstruktion des kürzesten Wegs, 199

Gebrochener Rucksack, 164
Geradeaus-Mischen, 78,
s. Sortieren
Gerüst, 153
Wald, 153
Gierige Optimierung,
s. Greedy-Algorithmus
Graph, 38, 44
Adjazenzliste, 45, 151, 153, 155, 158
Adjazenzmatrix, 45, 46, 153, 155, 196, 199
Eckenmenge, 38, 45
gerichteter, 39, 45, 161
gewichteter, 45, 161
Kantenmenge, 45
Greedy-Heuristik, 194
Grenze,
s. Branch and Bound

Handhabbar, 27
Hashing, 87, 128
Adreßbereich, 88
Hash-Funktion, 88
Divisions-Methode, 89
Multiplikations-Methode, 89
uniforme, 92
Hash-Tabelle, 88
Hashtabelle
Auslastung, 92
Kollision, 88
Kollisionsbehandlung, 90
Doppeltes Sondieren, 98
durch Sondieren, 94
durch Verkettung, 90
Komplexität, 91
Lineares Sondieren, 96
Quadratisches Sondieren, 97
Komplexität, 99
Schlüsselbereich, 88
uniformer, 92
Heap, 54, 84, 185, 189, 190
als Array, 54
Einfügen, 55, 56
Entfernen des größten Elements, 56, 57
max-Heap, 54, 59
min-Heap, 54
Wiederherstellen, 60, 190
Heapsort,
s. Sortieren
Heuristik, 167
Huffman-Codierung, 177, 179
Aufbau der Codetabelle, 185

Aufbau des Codebaums, 182
Codebaum, 179
Expansion, 186
Komprimierung, 185

Index, 127, 128
Indexdatei, 128
Indiziert zugreifbare Datei, 127
Inordnungs-Traversierung, 43, 112, 122
Insertion Sort,
s. Sortieren
Interne Blätter, 110
Invariante, 174

Kandidaten, 165, 175
Kantengewicht, 46
Keller,
s. Stack
Komplexitätstheorie, 21
Komponente, 151, 153
Kontextfreie Grammatik,
s. Typ-2 Grammatik
Kreislisten, 36
Kürzeste Wege, 161, 187, 195

LIFO-Liste,
s. Queue
Lineare Optimierung, 218
Linksrotation, 112, 113
Lokale Teilordnungseigenschaft, 53, 55

m-Weg Suchbaum, 126
Markierung von Ecken, 151
Maschine, 171
max-Heap,
s. Heap
Maximierung nicht-überlappender Aktivitäten, 194
Media-Schlussel, 132
Mehrwegbäume, 126
Mergesort,
s. Sortieren
min-Heap,
s. Heap
Minimal Spanning Tree,
s. MST
Minimalgerüst,
s. MST
Minimierung der Aufenthaltszeit, 193
Minimierung von Erholungspausen, 193
Mischen, 75
Mittlerer Fall, 21
MST, 166
Kruskal-Algorithmus, 168

Nachbedingung, 15
Natural Merge,
s. Sortieren
Natural merge,
s. Natürliches Mischen
Natürliches Mischen,
s. Sortieren
Nicht handhabbar, 27

Optimal Substructure Property,
s. OTE
Optimale Teilstruktur-Eigenschaft,
s. OTE
Optimierung
Greedy-Algorithmus, 164
für Scheduling-Probleme, 170
Ordnung, 35, 90, 91, 102
Ordnungs-Nachfolger, 105, 106
OTE, 196

Parsing-Algorithmus, 204
Permutation, 172
Pfadkompressions-Heuristik, 170
Planung von Aktivitäten, 176
Plattenindex, 128
Plattensektor, 127
Postordnungs-Traversierung, 43
Priority Queue, 19, 38, 84, 182, 189
Implementation durch Heaps, 54
Prioritäts-Warteschlange,
s. Priority Queue
Präordnungs-Traversierung, 43
Pseudocode, 12

Queue, 18, 37, 54, 159
Quicksort, 75,
s. Sortieren

Rechenzeit, 21
Rechtsrotation, 112, 114, 116
Repräsentant, 166
Ressourcen, 22

Ressourcenfunktionen, 22
Ringliste, 45
Ringlisten, 36
Rot/Schwarz Baum,
s. RS-Baum
Rotation, 111
RS-Baum, 110
Balanceeigenschaft, 111
Einfügen, 114, 118
Farbkorrektur, 121, 125
Löschen, 119
Rucksack-Problem, 163

Scheduling-Problem, 170
Schlechtester Fall, 21
Schlüssel, 35, 88, 101, 127, 128
Schranke,
s. Branch and Bound
Selection Sort,
s. Sortieren
Sentinel,
s. Wächter-Element
Single-source shortest path Algorithmus,
s. SSSP-Algorithmus
Sortieren, 156
durch Mischen, 75
extern, 75
3-Band Sortieren, 76
Geradeaus-Mischen, 78, 83
Lauf, 84
Mischen, 76
Natürliches Mischen, 84, 86
Verteilen, 76, 81
Heapsort, 52, 62
Abbauphase, 62
Aufbauphase, 61
extern, 58
Insertion Sort, 49
Quicksort, 63, 67
auf kleinen Arrays, 72
Komplexität, 68
Median, 65, 70
Partitionierung, 64
Selection Sort, 48
topologisches, 156, 157
Sortierkriterium, 47
Sortierverfahren
elementare, 48
höhere, 48
Speicherplatzbedarf, 21
SSSP-Algorithmus, 187,
s. auch Disjkstra-Algorithmus
Stack, 17, 37, 157, 158, 220
Straight Merge,
s. Sortieren
Straight merge,
s. Geradeaus-Mischen
Syntaxanalyse
Chart-Parser, 210
LL-Analyse, 209
LR-Analyse, 209

tape-merge, 76
Teilweise sortierte Strukturen, 52
Testfunktion, 165
Tiefensuche, 43, 151–153, 155, 185
Tiefentraversierung,
s. Tiefensuche
nicht-rekursiv, 158
Topologisches Sortieren,
s. Sortieren
Transitive Hülle, 210
Traversierung
eines Baums, 43
eines Graphen, 151
TSP, 210, 218
Typ-2 Grammatik
Chomsky-Normalform, 204
Nicht-terminaler Zeichenvorrat, 205
Parsing, 206
Regelmenge, 205
Startsymbol, 205
Türme von Hanoi, 28, 29

Überlappende Teilprobleme, 197

Verkettete Liste, 33, 45
Vorbedingung, 14

Wald von disjunkten Mengen, 167
Warteschlange,
s. Queue
Wächter-Element, 175

Zielfunktion, 165