Datenstrukturen, Algorithmen und Programmierung II

Petra Mutzel
Markus Chimani
Carsten Gutwenger
Karsten Klein

Skript zur gleichnamigen Vorlesung von

Petra Mutzel

im Sommersemester 2009

VERSION 1.1
LEHRSTUHL 11 – ALGORITHM ENGINEERING
TECHNISCHE UNIVERSITÄT DORTMUND

Inhaltsverzeichnis

Vorwort

Algorithmen und Datenstrukturen sind ein wichtiges und zentrales Gebiet der Informatik. Was auch immer Sie später tun werden, ob während Ihres Studiums oder danach: die Wahrscheinlichkeit, dass Sie Basiswissen in Algorithmen und Datenstrukturen benötigen, ist sehr hoch.

Egal in welchem Gebiet Sie sich vertiefen werden, ob in Software-Entwicklung, Computergraphik, Datenbanken, Eingebettete Systeme, Kryptographie oder Künstlicher Intelligenz, Sie werden immer wieder auf die Grundbegriffe, die Sie in dieser Lehrveranstaltung lernen, stoßen.

Die Vorlesung **Datenstrukturen, Algorithmen und Programmierung 2** (DAP2) ist im Studienplan der Bachelorstudiengänge der Informatik angesetzt und behandelt den Entwurf und die Analyse von Algorithmen. Bisher haben Sie gelernt, wie man grundsätzlich programmiert. Sie sollten in der Lage sein, jede an Sie herangetragene Aufgabenstellung in ein lauffähiges Programm umzusetzen. Mit dem in DAP2 behandelten Basiswissen werden Sie auch in der Lage sein, **gut** und **effizient** zu programmieren, sowie die Qualität Ihrer Lösungen zu bewerten. Was dies genau bedeutet, werden Sie im Laufe der Vorlesung kennenlernen.

Hier nur ein kleines Beispiel, um Ihnen eine Vorstellung davon zu geben:

> Angenommen Sie haben nach Studienende eine Stellung als Projektleiter/in in der Software-Branche inne. Sie erhalten die Aufgabe, die Datenbank einer Firma neu aufzubauen, und erledigen das mit Ihrem Team. Nach Fertigstellung benötigt jede Abfrage (z.B. nach einer Kundennummer) ca. 2 Minuten.
>
> Ihr größter Konkurrent oder Ihre größte Konkurrentin, der/die den Stoff von DAP2 verstanden hat, schreibt hingegen ein Programm, das für jede Abfrage nur 1 Sekunde benötigt, also 120 Mal so schnell ist. Das ist offensichtlich schlecht für Sie und Ihre Zukunft in der Firma.
>
> Wenn Sie hingegen zu den Studierenden zählen, die sich in Algorithmen und Datenstrukturen auskennen, dann sind Sie denjenigen Mitarbeiter/inne/n aus der IT-Branche, die das nicht beherrschen, schon ein großes Stück voraus.

Wir, die Algorithmen und Datenstrukturen lehren, bemühen uns, dass es in Zukunft mehr Fachkräfte geben wird, die wissen, wie man gut und effizient programmiert. Wir hoffen, dass wir Sie dazu anregen können, dieses Wissen auch in die Praxis umzusetzen.

Wir hoffen Sie davon überzeugen zu können:

„ALGORITHMEN UND DATENSTRUKTUREN SIND SPANNEND“

und wünschen Ihnen *viel Erfolg*!

Petra Mutzel und das DAP2-Team

Organisatorisches

Das Modul *Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)* besteht aus den Lehrveranstaltungen:

- Vorlesung DAP2 (4 SWS, 6 Credits)
- Übungen zu DAP2 (2 SWS, 3 Credits)
- Praktikum zu DAP2 (2 SWS, 3 Credits)

Die Vorlesung DAP2 wird im Sommersemester 2009 von Prof. Dr. Petra Mutzel vom Lehrstuhl 11 (Algorithm Engineering) gehalten und findet dienstags von 12:15 bis 13:45 im *Auditorium Maximum* und donnerstags von 14:15 bis 15:45 Uhr im HG 2 / HS1 (Campus Nord) statt.

Die Übungen werden von Nicola Beume, Christian Bockermann, Christian Horoba, Ingo Schulz, Dirk Sudholt und Christine Zarges durchgeführt. Das Praktikum wird von Carsten Gutwenger, Jürgen Mäter, Wilfried Rupflin und Mouzhi Ge veranstaltet. Die wissenschaftlichen Mitarbeiter/innen werden von studentischen Tutor/inn/en unterstützt. Ansprechpartner bei organisatorischen Fragen zu den Übungen sind Nicola Beume und Dirk Sudholt sowie zum Praktikum Carsten Gutwenger.

Die **Modulprüfung** findet in Form einer **Klausur** am Freitag, dem 31. Juli 2009 in verschiedenen Hörsälen statt und dauert 90 Minuten. Der Nebentermin ist Montag, der 5. Oktober 2009. Voraussetzung zur Teilnahme an der Klausur ist (für Studierende im Bachelor-Studiengang) die **erfolgreiche Teilnahme** am Praktikum zu DAP1 sowie an den Übungen zu DAP2.

Detaillierte Informationen zu Übungsablauf, Scheinkriterien, Übungstests und Anmeldung finden Sie auf den Folien zur ersten Vorlesung sowie auf unseren Webseiten. Beachten Sie bitte auch die aktuellen Informationen zu DAP2 auf unserer Homepage:

- Unsere Homepage:
  `http://ls11-www.cs.uni-dortmund.de/`

- Vorlesung DAP2:
  `http://ls11-www.cs.tu-dortmund.de/people/beume/dap2-09/dap2.jsp`
- Übung zu DAP2:
  `http://ls11-www.cs.tu-dortmund.de/people/beume/dap2-09/dap2u.jsp`
- Praktikum zu DAP2:
  `http://ls11-www.cs.tu-dortmund.de/people/gutweng/DAP2-09/dap2p.jsp`

Literaturliste

Das Skript kann und soll den Besuch der Vorlesung nicht ersetzen. Es dient als begleitende Unterlage vorrangig dazu, einen Überblick über den behandelten Stoff zu geben. Über das Skript und die Vorlesung hinaus empfehlen wir, den Lehrstoff mit Büchern zu vertiefen. Die Vorlesung hält sich teilweise eng an Ausführungen der hier aufgelisteten Bücher.

T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Algorithmen - Eine Einführung *(2. Auflage, Oldenbourg Verlag 2007)*
Dieses Buch gilt als Standardwerk auf dem Gebiet Datenstrukturen und Algorithmen. Es erklärt die Themen sehr schön und ausführlich und kann auch über den Stoffumfang von DAP2 hinweg als Nachschlagewerk dienen.

R. Sedgewick: Algorithmen *(2. Auflage, Pearson 2002)*
Ein sehr anschauliches und gut zu lesendes Buch, dass sowohl auf Deutsch als auch auf Englisch erhältlich ist. Diese (aktuelle) Ausgabe beschreibt die Algorithmen – so wie wir es auch in der Vorlesung tun werden – mittels Pseudocode, und umfasst den gesamten Stoffumfang.
Es existieren vom selben Autor auch sehr ähnliche Werke für bestimmte Programmiersprachen, z.B. „Algorithmen in C++". Doch Vorsicht! Diese sind in 2 Bände (Band 1 = Teil 1–4, Band 2 = Teil 5) geteilt; in der Vorlesung behandeln wir den Stoff aus beiden Bänden!

T. Ottmann und P. Widmayer: Algorithmen und Datenstrukturen *(4. Auflage, Spektrum Akademischer Verlag 2002)*
Auch dieses Buch bietet einen sehr anschaulichen Einstieg in die Thematik, ist aber etwas mathematischer gestaltet.

Speziellere Literaturhinweise finden Sie ggf. am Ende der entsprechenden Kapitel. Darüber hinaus steht es Ihnen natürlich frei, jedes andere geeignete Buch zum behandelten Stoff auszuwählen.

Dankesworte

Das vorliegende Skript basiert auf dem Skript zu der bis 2004 an der TU Wien von Professor Dr. Petra Mutzel gehaltenen Vorlesung *Algorithmen und Datenstrukturen*.Teile des Originalskripts wurden aus einer Vorlage entnommen, die von Constantin Hellweg für die Algorithmen-Vorlesung von Professor Dr. Michael Jünger, Universität zu Köln, erstellt wurde. Bereits in Wien haben zahlreiche Mitarbeiter/innen (Gunnar Klau, Gabriele Koller, Ivana Ljubic, Günther Raidl, Martin Schönhacker und René Weiskircher, sowie der Studienassistent Georg Kraml) das Skript regelmäßig überarbeitet. Für die DAP2 Vorlesung im SS2006 in Dortmund wurden große Teile des Skripts von Petra Mutzel und den wissenschaftlichen Mitarbeitern Markus Chimani, Carsten Gutwenger und Karsten Klein überarbeitet bzw. neu geschrieben. Der Dank geht an alle diese zahlreichen Kollegen, Mitarbeiter und Mitarbeiterinnen, die so zur Entstehung des heutigen Skripts entscheidend beigetragen haben.

Sollten Sie Anmerkungen zum Skript haben, sind wir dafür dankbar. Verantwortlich für den Inhalt ist Professor Dr. Petra Mutzel.

Kapitel 1

Einführung

In diesem Kapitel klären wir zunächst wichtige Begriffe wie *Algorithmus*, *Datenstruktur* und *Programm*, die uns im Laufe des Skripts unentwegt begegnen werden. Um in weiterer Folge Algorithmen spezifizieren zu können, verwenden wir sogenannten *Pseudocode*.

Einer der wesentlichste Bereiche der Algorithmik ist die Analyse von Algorithmen, die wir in Abschnitt 1.3 einführen werden. Ohne sie könnten wir weder die Effizienz unserer Programme abschätzen, noch einen Vergleich zwischen zwei kompetitierenden Algorithmen ziehen.

1.1 Grundbegriffe

Algorithmus. Der Duden definiert einen Algorithmus als *„Rechenvorgang, der nach einem bestimmten [sich wiederholenden] Schema abläuft"*, bzw. – für den Bereich der mathematischen Logik – als *„Verfahren zur schrittweisen Umformung von Zahlenreihen"*.

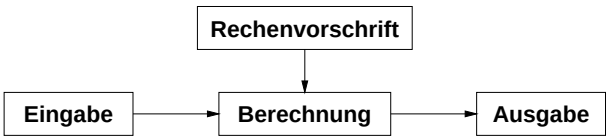


Abbildung 1.1: Definition des Begriffs Algorithmus

Genauer besitzt ein Algorithmus eine Spezifikation von gültigen Eingabedaten, führt eine Berechnung gemäß einer Rechenvorschrift durch, und liefert schließlich ein wohldefinierte Ausgabe (siehe Abb. 1.1). Weiterhin fordern wir, dass die Beschreibung eines Algorithmus endliche Größe hat, und die Berechnung nach endlich vielen Schritten terminiert.

Datenstruktur. Eine Datenstruktur ist ein Schema zur Repräsentation der durch einen Algorithmus behandelten Daten.

Programm. Nach dem bekannten Informatiker Nikolaus Wirth ist ein Programm eine *„konkrete Formulierung abstrakter Algorithmen, die sich auf bestimmte Darstellungen wie Datenstrukturen stützen"*, d.h. die Summe aus Algorithmen und Datenstrukturen. Programmerstellung und Datenstrukturierung sind untrennbare, ineinandergreifende Themen.

Schließlich bleibt noch die Frage, wie wir ein Programm nun darstellen können. Dies kann auf mehrere Arten geschehen:

- als Text (Deutsch, Englisch,...)
- als Computerprogramm (C/C++, Java, Pascal, Haskell,...)
- als Hardwaredesign

In der Vorlesung bedienen wir uns einer vereinfachten Art einer Programmiersprache:

Pseudocode. Pseudocode orientiert sich in der Regel an den weit verbreiteten *imperativen Programmiersprachen*, von deren klassischen Vertretern wie C/C++ oder Java Sie sicher schon gehört haben. In all diesen Sprachen finden sich jedoch viele syntaktische Notwendigkeiten und Langatmigkeiten, die für das Verständnis der Grundgedanken unwesentlich sind. Pseudocode trifft ausgehend von diesen Sprachen diverse Vereinfachungen, unter der Prämisse, dass die Umsetzung von Pseudocode in eine real existierende Programmiersprache eindeutig klar ist. Es existieren keine strikten Regeln nach denen Pseudocode spezifiziert werden kann, aber die in diesem Skript vorkommenden Codeteile sollten Ihnen einen Einblick in den Umgang mit ihm geben.

Anmerkung 1.1. Viele „lästige" Programmiertätigkeiten dürfen allerdings nicht ausgelassen werden. So darf man insbesondere auf *Initialisierungen* und *Spezifikation der Parameter* die bei einem Programmaufruf übergeben werden *nicht* verzichten!

Beispiel 1.1. Pseudocode dient zum Beispiel dazu, derartige Strukturen vereinfacht aufzuschreiben:

C / Java	*Pseudocode*

`int tmp = i;`
`i = j;`
`j = tmp;`
	vertausche i mit j

for`(int i=0; i<10; i++) {`
`  float f = A[i];`
`  ...`
`}`
	for $f := A[0], \ldots, A[9]$ **do**
	`  ...`
	end for

Hinweis 1.1. In der Übung sowie zur Klausur können natürlich Aufgabenstellungen vorkommen, in denen Sie einen Algorithmus in Pseudocode schreiben müssen. Wenn Sie sich unsicher sind wie stark Sie vereinfachen dürfen, schreiben Sie tendenziell eher zu genau (also sehr nahe an C, Java, etc.), als zu ungenau! Wenn Sie kompliziertere Hilfsfunktionen bzw. Hilfsdatenstrukturen benutzen dürfen, wird dies in der Angabe explizit vermerkt stehen.

1.2 Beispiel: Sortierproblem

Sortierproblem (informell)	
Gegeben:	eine Folge von n Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$
Gesucht:	eine Umordnung der Elemente der Eingabefolge, so dass für die neue Folge $\langle a'_1, a'_2, \ldots, a'_n \rangle$ gilt, dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Jede konkrete Zahlenfolge ist eine *Instanz* des Sortierproblems, z.B. soll $\langle 51, 24, 43, 66, 12, 32 \rangle$ in $\langle 12, 24, 32, 43, 51, 66 \rangle$ überführt werden. Allgemein gilt, dass eine Instanz eines Problems aus allen Eingabewerten besteht, die zur Lösung benötigt werden.

Ein Algorithmus zur Lösung eines Problems heißt *korrekt*, wenn seine Durchführung für jede mögliche Eingabeinstanz der Problems mit der korrekten Ausgabe terminiert. Ein korrekter Algorithmus löst also das gegebene Problem.

Wir werden im Laufe der Vorlesung verschiedene Algorithmen kennenlernen, die das Sortierproblem lösen. Einer der einfachsten dieser Algorithmen wird als *InsertionSort* bezeichnet, da der Grundgedanke auf einem sukzessiven Einfügen der noch nicht sortierten Elemente in eine schon sortierte Teilfolge basiert.

Der Algorithmus hat die schöne Eigenschaft, dass er unser eigenes Vorgehen im realen Leben simuliert. Nehmen wir an, wie bekommen bei einem Kartenspiel 8 Karten, die zu Beginn als Stapel vor uns liegen. Zunächst nehmen wir die erste dieser Karten auf die Hand. Da wir nur eine Karte in der Hand halten, ist diese Teilmenge der Karten sortiert. Nun nehmen wir die zweite Karte, und fügen sie vor oder nach der ersten Karte ein, so dass wir zwei Karten in sortierter Reihenfolge in der Hand halten. Danach nehmen wir die dritte Karte, und sortieren sie in die schon sortierten Karten ein, usw. bis wir schließlich nach dem Aufnehmen der achten Karte alle Karten sortiert in der Hand halten.

Als Datenstruktur für unser Sortierproblem bietet sich ein einfaches *Feld* (engl. *Array*) an, d.h. eine Folge von Daten gleichen Typs, auf die wir einfach mit einem Index zugreifen können. In diesem Skript fangen die Indizes eines Feldes – soweit nicht anders erwähnt – immer bei 1 an.

Eingabe: zu sortierende Zahlenfolge im Feld A (d.h. in $A[1], \ldots, A[n]$)
Ausgabe: sortierte Zahlenfolge im Feld A

```
 1: procedure INSERTIONSORT(ref A)
 2:     var Zahl key
 3:     var Indizes k, i
 4:     for k := 2, ..., n do                          ▷ Füge k-te Zahl ein.
 5:         key := A[k]                  ▷ Zwischenspeichern der einzufügenden Zahl
 6:         i := k
 7:         while i > 1 and A[i − 1] > key do                ▷ Suchen der Position...
 8:             A[i] := A[i − 1]          ▷ Verschieben der zu großen Zahl nach rechts
 9:             i := i − 1
10:         end while
11:         A[i] := key                                      ▷ Neue Zahl einfügen
12:     end for
13: end procedure
```

Listing 1.1: Sortieren durch Einfügen.

Im Folgenden sehen wir das beschriebene Sortierprogramm als Pseudocode[1]. Es ist darauf zu achten, dass das Einfügen einer Karte an der richtigen Position nicht ganz so einfach funktioniert wie in unserem Kartenspiel-Analogon. Wenn wir die k-te Karte einsortieren wollen, haben wir eine sortierte Teilfolge $A[1], \ldots, A[k-1]$. Soll diese neue Karte nun z.B. zwischen $A[3]$ und $A[4]$ eingefügt werden, so müssen wir zunächst „Platz schaffen", in dem wir die Teilfolge $A[4], \ldots, A[k-1]$ um einen Platz nach rechts an die Positionen $5, \ldots, k$ schieben. Danach können wir die neue Karte an der nun freien Stelle 4 einfügen.

Dieses Verschieben verbinden wir gleichzeitig mit der Suche nach der richtigen Einfügeposition: Wir verschieben, bei den größten beginnend, jede Karte immer um eine Position nach rechts, solange diese Karte größer als die neu einzufügende ist. Sobald wir eine Karte betrachten die kleiner ist als die neue, haben wir alle größeren Karten schon um eine Stelle nach rechts verschoben und können die neue Karte einfach einfügen.

Der Ablauf des Algorithmus bei Eingabe der Folge $\langle 51, 24, 43, 66, 12, 32 \rangle$ ist in Abbildung 1.2 dargestellt.

[1] Die Bezeichnung **ref** vor einem Parameter in der Parameterliste eines Algorithmus weist darauf hin, dass dieser Parameter ein *Referenzparameter* ist. Dadurch wird im Gegensatz zu einem herkömmlichen Wertparameter auch ein Ergebnis zurückgeliefert. In diesem Algorithmus wird im Referenzparameter A das unsortierte Feld als Eingabe erwartet und nach Beendigung das sortierte Feld zurückgeliefert.

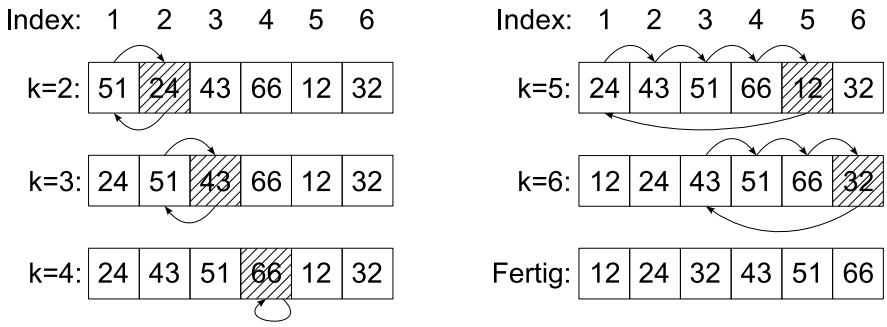


Abbildung 1.2: Illustration von Insertion-Sort

1.3 Analyse von Algorithmen

Die Analyse von Algorithmen ist sehr eng mit dem Algorithmenentwurf verbunden. Es geht darum, die Güte bzw. Effizienz eines vorliegenden Algorithmus festzustellen. Dabei können uns verschiedene Maße interessieren:

- Laufzeit,
- benötigter Speicherplatz,
- Anzahl der Vergleichsoperationen,
- Anzahl der Datenbewegungen, u.v.m.

Im Folgenden sind wir fast immer an der Laufzeit eines Algorithmus interessiert. Wir bedienen uns dabei einem sogenannten *Maschinenmodell*, also einer vereinfachten Sichtweise auf einen realen Computer. Das einfachste und gebräuchlichste Maschinenmodell ist die sogenannte *random access machine (RAM)*. Sie besitzt die folgenden Eigenschaften:

- Es gibt genau einen Prozessor, der das Programm sequentiell abarbeitet.
- Jede Zahl die wir in unserem Programm benutzen paßt in eine Speichereinheit.
- Alle Daten liegen in einem direkt zugreifbaren Speicher.
- Alle Speicherzugriffe dauern gleich lang.
- Alle *primitiven Operationen* benötigen konstante Zeit.

Unter primitiven Operationen verstehen wir

- Zuweisungen ($a := b$)
- arithmetische Operationen (Addition, Subtraktion, Multiplikation, Modulo-Berechnung, Wurzelziehen,...)
- logische Operationen (**and**, **or**, **not**,...)
- Vergleichsoperationen ($<$, $\geq$, $\neq$,...)
- Befehle zur Ablaufsteuerung (**if-then-else**, ...)

Allgemein werden wir in diesem Skript annehmen, dass eine primitive Operation jeweils eine Zeiteinheit in Anspruch nimmt.

Definition 1.1. Die *Laufzeit* eines Algorithmus ist die Anzahl der bei einer Berechnung durchgeführten primitiven Operationen.

Es ist klar, dass das Sortieren von 1000 Zahlen länger dauert als das Sortieren von 10 Zahlen. Daher beschreiben wir die Laufzeit als *Funktion der Eingabegröße*. Beim Sortieren einer n-elementigen Folge ist diese Eingabegröße n. Bei einigen Sortieralgorithmen – unter anderem InsertionSort – geht das Sortieren bei gleich langen Folgen für eine „beinahe korrekt sortierte" Folge schneller als für eine absolut unsortierte Folge. Wir müssen also bei gleicher Eingabegröße durchaus verschiedene Fälle unterscheiden.

1.3.1 Best-Case, Worst-Case und Average-Case

Analysieren wir nun die Laufzeit von Insertion-Sort. Dazu weisen wir zunächst jeder Zeile i des Algorithmus eine Laufzeit t_i zu, die sich aus der Summe der in ihr enthaltenen primitiven Operationen ergibt, und bestimmen, wie oft die Zeile ausgeführt wird.

Zeile	Zeit	Wie oft?
4	t_4	n
5	t_5	$n-1$
6	t_6	$n-1$
7	t_7	$\sum_{k=2}^{n} s_k$
8	t_8	$\sum_{k=2}^{n}(s_k-1)$
9	t_9	$\sum_{k=2}^{n}(s_k-1)$
10	t_{10}	$\sum_{k=2}^{n}(s_k-1)$
11	t_{11}	$n-1$
12	t_{12}	$n-1$

Die Zeilen der Schleifenbedingungen (4 und 7) werden immer einmal mehr ausgeführt als der Schleifenkörper selbst, da das Kriterium überprüft werden muss, um festzustellen, dass die Schleife kein weiters Mal durchlaufen wird. Die Variable s_k bezeichnet die Anzahl der Durchführungen der Zeile 7 für die k-te Zahl.

Daraus ergibt sich eine Gesamtlaufzeit von:

$$\begin{aligned} T(n) &= t_4 n + t_5(n-1) + t_6(n-1) + t_7 \sum_{k=2}^{n} s_k + t_8 \sum_{k=2}^{n} (s_k - 1) + t_9 \sum_{k=2}^{n} (s_k - 1) \\ &\quad + t_{10} \sum_{k=2}^{n} (s_k - 1) + t_{11}(n-1) + t_{12}(n-1) = \\ &= t_4 + (t_4 + t_5 + t_6 + t_7 + t_{11} + t_{12})(n-1) + (t_7 + t_8 + t_9 + t_{10}) \sum_{k=2}^{n} (s_k - 1) \end{aligned}$$

$T(n)$ wird auch als *Laufzeitfunktion in* n bezeichnet. Wir erkennen allerdings, dass wir diese Formel ohne Wissen um die Werte von s_k nicht weiter vereinfachen können, doch diese sind von den Werten der Eingabefolge abhängig. In solchen Fällen sind drei Fragestellungen besonders interessant:

- Was ist die minimale Laufzeit *(Best-Case)*?
- Was ist die maximale Laufzeit *(Worst-Case)*?
- Was ist die durchschnittliche Laufzeit *(Average-Case)*?

Best-Case-Analyse. Die Best-Case-Analyse misst die kürzeste mögliche Laufzeit über alle möglichen Eingaben der Größe n. Demnach ist der Best-Case also eine *untere Schranke* für die Laufzeit einer beliebigen Instanz dieser Größe.

Die Laufzeit ist dann am geringsten, wenn s_k für alle k möglichst klein ist. Dieses Minimum wäre, wenn wir die Zeile 7 in jedem Durchlauf der äusseren Schleife immer nur einmal auswerten müssten, um festzustellen, dass die innere Schleife nicht durchlaufen werden soll. Doch können wir uns tatsächlich eine Zahlenfolge überlegen für die diese Situation eintritt? Wenn wir den Fall betrachten, dass die Eingabefolge schon vor dem Aufruf des Algorithmus korrekt sortiert ist, stellen wir fest, dass wir Zeile 7 tatsächlich immer nur einmal auswerten müssen, da wir jedesmal feststellen, dass der Wert des neuen Elements größer ist als der größte der bisher sortierten Teilfolge. Daher gilt dann $s_k = 1$ für $k = 2, 3, \ldots, n$ und es folgt eine Laufzeit von

$$\begin{aligned} T_{\mathrm{bc}}(n) &= t_4 + (t_4 + t_5 + t_6 + t_7 + t_{11} + t_{12})(n-1) + (t_7 + t_8 + t_9 + t_{10}) \sum_{k=2}^{n} (1-1) \\ &= (t_4 + t_5 + t_6 + t_7 + t_{11} + t_{12})n - (t_5 + t_6 + t_7 + t_{11} + t_{12}) \\ &= an + b \text{ für Konstanten } a \text{ und } b, \end{aligned}$$

also eine lineare Funktion in n.

Worst-Case-Analyse. Die Worst-Case-Analyse misst die längste Laufzeit über alle möglichen Eingaben der Größe n. Demnach ist der Worst-Case also eine *obere Schranke* für die Laufzeit einer beliebigen Instanz dieser Größe.

Unsere Laufzeitfunktion liefert umso größere Resultate, umso größer die Werte s_k sind, d.h. wenn die innere Schleife immer möglichst weit durch das Array nach vorne läuft. Eine Eingabefolge die in genau umgekehrter Reihenfolge sortiert ist, bewirkt ein solches Verhalten: Jedes neue Element ist kleiner als alle vorhergehenden, und daher müssen alle anderen Elemente zunächst um eine Stelle nach rechts wandern, bevor man das neue Element an die vorderste Position setzen kann. Die innere Schleife bricht also immer erst durch die nichterfüllte Bedingung $i > 1$ ab.

Das bedeutet, dass $s_k = k$ für $k = 2, 3, \ldots, n$. Wir erinnern uns an die Gauss'sche Summenformel und wissen

$$\sum_{k=2}^{n} (k-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}.$$

Daraus folgt eine Laufzeit von

$$\begin{aligned} T_{\mathrm{wc}}(n) &= t_4 + (t_4 + t_5 + t_6 + t_7 + t_{11} + t_{12})(n-1) + (t_7 + t_8 + t_9 + t_{10}) \left(\frac{n^2}{2} - \frac{n}{2} \right) \\ &= \left(\frac{t_7 + t_8 + t_9 + t_{10}}{2} \right) n^2 + \left(t_4 + t_5 + t_6 + t_{11} + t_{12} + \frac{t_7 - t_8 - t_9 - t_{10}}{2} \right) n \\ &\quad -(t_5 + t_6 + t_7 + t_{11} + t_{12}) \\ &= an^2 + bn + c \text{ für Konstanten } a, b \text{ und } c, \end{aligned}$$

also eine quadratische Funktion in n.

Average-Case-Analyse. Die Average-Case-Analyse misst die durchschnittliche Laufzeit über alle Eingaben der Größe n. Es gibt Algorithmen, in denen der Average-Case erheblich

besser als der ,,pessimistische" Worst-Case ist; es gibt aber ebenso Algorithmen bei denen er genauso schlecht ist.

Average-Case-Analysen sind oft mathematisch weit aufwendig als eine Best- oder Worst-Case-Analyse. Das Hauptproblem besteht oft darin, dass man schliesslich keinen ,,durchschnittlichen Fall" hat, sondern den Durchschnitt über alle möglichen Fälle bilden muss. Hinzu kommt, dass es oft keinen Sinn macht, diesen Durchschnitt als arithmetisches Mittel zu berechnen! Im Allgemeinen folgen die in der Realität auftretenden Eingaben einer nichttrivialen *Verteilungsfunktion*, d.h. bestimmte Eingabemuster haben eine höhere Wahrscheinlichkeit als andere. Da die Average-Case-Analyse für verschiedene Verteilungen durchaus verschiedene Ergebnisse liefern kann, ist im Umgang mit ihr immer Vorsicht geboten.

Bei unserem Insertion-Sort beruht der Ansatz der Analyse darauf, dass jede mögliche Ausgangspermutation gleich wahrscheinlich ist. Der Einfachheit halber beschränken wir uns darauf, dass kein Schlüsselwert mehr als einmal vorkommt. Betrachten wir den Zeitpunkt, an dem wir das k-te Element a_k einsortieren möchten und analysieren, um wieviele Positionen wir a_k durchschnittlich verschieben müssen. Da alle Ausgangspermutationen unserer Folge gleich wahrscheinlich waren, wissen wir für jedes schon einsortierte Element a_j (mit $j < k$), dass die Fälle $a_j < a_k$ und $a_j > a_k$ gleich wahrscheinlich sind. Im Durchschnitt werden also die Hälfte der schon betrachteten Elemente kleiner, und die andere Hälfte größer sein als a_k.

Wir wissen aber, dass die Elemente links von a_k schon gültig sortiert wurden: Da wir beim Einsortieren von a_k genau alle Elemente abwandern, die größer als a_k sind, und beim ersten Element abbrechen, das kleiner als a_k ist, folgern wir, dass im Average-Case $s_k = \frac{k-1}{2} + 1$ gilt.

Setzen wir diesen Term in unsere allgemeine Laufzeitfunktion ein, errechnen wir zunächst

$$\sum_{k=2}^{n} \left(\frac{k-1}{2} + 1 \right) - 1 = \frac{1}{2} \sum_{k=2}^{n} (k-1).$$

Diese Summe ist also genau halb so groß wie die der Worst-Case-Analyse. Dadurch erhalten wir durch analoge Berechnungen abermals eine quadratischen Laufzeitfunktion $T_{\mathrm{ac}}(n)$, wenn auch mit kleineren Konstanten im Vergleich zu $T_{\mathrm{wc}}(n)$.

1.3.2 Asymptotische Schranken

Schon bei dem vergleichsweise sehr überschaubaren InsertionSort-Algorithmus merken wir, dass die ausführliche Laufzeitfunktion meistens zu umfangreich ist. Was uns tatsächlich interessiert, ist das Wachstumsverhalten einer Laufzeitfunktion, d.h. um wieviel verlängert sich die Laufzeit wenn sich die Eingabegröße beispielsweise verdoppelt. Desweiteren stellen wir fest, dass die Laufzeit eines Algorithmus für kleine Instanzen in der Regel recht uninteressant ist — bei kleinen Eingaben ist der Algorithmus ohnehin ,,schnell genug". Wichtig ist die Laufzeit wenn wir betrachten, dass die Eingabe immer größer wird, mathematisch gesehen: gegen unendlich geht. Dies bezeichnen wir als *asymptotische Laufzeit*.

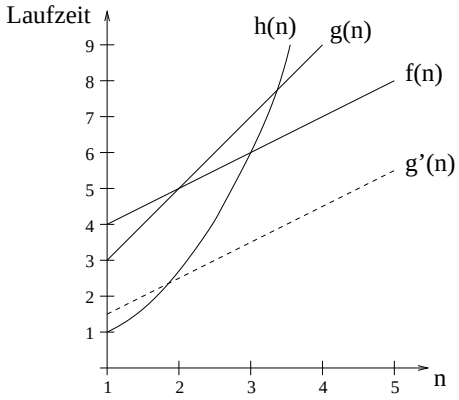


Abbildung 1.3: Laufzeitfunktion $f(n), g(n), h(n)$ und $g'(n)$

Beispiel 1.2. Nehmen wir an, dass Fridolin, Gwendoline und Harlekin jeweils einen Algorithmus entwickelt haben, deren Laufzeitfunktionen $f(n)$, $g(n)$ bzw. $h(n)$ sie in Sekunden messen, wobei $f(n) = n + 3$, $g(n) = 2n + 1$ und $h(n) = \frac{2}{3}n^2$ (vgl. Abb. 1.3):

- *Welcher Algorithmus ist der langsamste?* Intuitiv halten wir $f(n)$ und $g(n)$ für schneller als $h(n)$, obwohl für kleine Eingabegrößen n die Situation umgekehrt ist (siehe Abb. 1.3). Unsere Intuition stimmt, denn wir interessieren uns für die Laufzeitfunktion für große n — kleine n sind vernachlässigbar. Wir sagen also

  „$f(n)$ wächst langsamer als $h(n)$"

  da $f(n) \leq h(n)$ für alle n ab einem gewissen n_0 (hier z.B. 3) gilt. Analoges gilt für $g(n)$.

- *Welcher Algorithmus ist der schnellste?* Es ist leicht zu sehen, dass $f(n) \leq g(n)$ für alle $n \geq 2$. Doch nehmen wir an, dass Gwendoline sich einen schnelleren Computer kauft; die Laufzeit ist jetzt nur noch halb so lang: $g'(n) = \frac{1}{2}(g(n)) = n + \frac{1}{2}$. Während vorher Fridolin als der Sieger erschien, so ist nun Gwendolines Algorithmus immer schneller. Wir erkennen: Eine Skalierung mit einem konstanten Faktor ändert nichts an dem asymptotischem Wachstum. Daher sagen wir:

  „$f(n)$ wächst ungefähr gleich stark wie $g(n)$"

  wenn $c_1 g(n) \leq f(n) \leq c_2 g(n)$ (ab einem gewissen n_0) für zwei Konstanten c_1 und c_2 gilt.

Wir wenden nun die Erkenntnisse aus diesem Beispiel an, um die formalen Definitionen der

Θ-, O-, und Ω-Notationen einzuführen. Sie stellen die Standardbezeichnungen für Laufzeitanalysen in der Informatik dar:

Definition 1.2. Sei $g : \mathbb{N} \to \mathbb{R}^+$ eine Funktion. Wir definieren die folgenden Mengen von Funktionen:

$$\Theta(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid (\exists c_1, c_2, n_0 > 0), (\forall n \geq n_0) : c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

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

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

$\Theta(g(n))$ ist also die Menge aller Funktionen $f(n)$, für die ein c_1, c_2 und ein n_0 (alle positiv und unabhängig von n) existieren, so dass für alle $n \geq n_0$ gilt, dass $c_1 g(n) \leq f(n) \leq c_2 g(n)$. D.h. $\Theta(g(n))$ ist die Menge aller Funktionen, die asymptotisch – bis auf Skalierung mit einem konstanten Faktor – gleich stark wachsen wie $g(n)$.

$O(g(n))$ ist die Menge aller positiven Funktionen $f(n)$, für die ein c und ein n_0 (beide positiv und unabhängig von n) existieren, so dass für alle $n \geq n_0$ gilt, dass $f(n) \leq cg(n)$ ist. D.h. $f(n)$ ist durch $cg(n)$ asymptotisch von oben beschränkt.

$\Omega(g(n))$ schließlich bezeichnet die entsprechende Menge von Funktionen $f(n)$ für die $cg(n)$ eine asymptotische untere Schranke ist.

Aus der Definition ist klar, dass jede Funktion in $\Theta(g(n))$ auch ein Element aus $O(g(n))$ und $\Omega(g(n))$ ist. Es gilt $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Wir benutzen die

- O-Notation für eine *oberen Schranke*,
- Ω-Notation für eine *unteren Schranke*, und
- Θ-Notation für die „genaue" Wachstumsrate einer Laufzeitfunktion.

Oft wird der Begriff „O-Notation" auch für die Gesamtheit der obigen Notationen benutzt.

Anmerkung 1.2. Man schreibt $f(n) = \Theta(g(n))$ anstelle von $f(n) \in \Theta(g(n))$; entsprechendes gilt für die O- und Ω-Notation.

Beispiel 1.3. Wenden wir uns wieder Fridolin, Gwendoline und Harlekin zu und analysieren formal die asymptotischen Laufzeiten ihrer Algorithmen:

- Zunächst zeigen wir, dass für Gwendolines Laufzeit $g(n) = \Theta(n)$ gilt. Dazu müssen wir Werte für c_1, c_2 und n_0 finden, so dass

  $$c_1 n \leq 2n + 1 \leq c_2 n \text{ für alle } n \geq n_0.$$

  Mit Division durch $n > 0$ erhalten wir

  $$c_1 \overset{(1)}{\leq} 2 + \frac{1}{n} \overset{(2)}{\leq} c_2.$$

  Da wir aus der Analysis wissen, dass $\frac{1}{n}$ monoton fallend ist und $\lim_{n\to\infty} \frac{d}{n} = 0$ (für eine beliebige Konstante d), wissen wir auch:

  (1) gilt für alle $n \geq 1$, wenn $c_1 \leq 2$.

  (2) gilt für alle $n \geq 1$, wenn $c_2 \geq 3$.

  Also wählen wir z.B. $c_1 = 2$, $c_2 = 3$ und $n_0 = 1$, und die asymtoptisch lineare Laufzeit ist bewiesen. Analog können wir zeigen, dass $f(n) = \Theta(n)$.

  Für einen Beweis ist es also sehr wichtig, diese Konstanten c_1, c_2 und n_0 auch anzugeben. Natürlich müssen diese Konstanten nicht durch exaktes Lösen der Gleichungssysteme bestimmt werden; grobe Abschätzungen genügen bereits. Beispielsweise kann man in diesem Beispiel auch die Werte $c_1 = \frac{1}{2}$, $c_2 = 50$ und $n_0 = 100$ wählen.

- Nun zeigen wir, dass wir für Harlekins Algorithmus keine lineare obere Schranke finden können, also dass $h(n) \neq O(n)$. Wir beweisen durch Widerspruch, d.h. wir nehmen zunächst an, dass $h(n)$ eine lineare Laufzeit garantieren kann. Dann müsste für geeignete Konstanten c und n_0 gelten:

  $$0 \leq n^2 \leq cn \text{ für alle } n \geq n_0.$$

  Dividieren wir wieder durch $n > 0$ erhalten wir

  $$0 \leq n \leq c \text{ für alle } n \geq n_0.$$

  Wenn n aber nun gegen unendlich strebt, werden wir nie eine Konstante c finden können, die diese Bedingung erfüllt. Die Laufzeit $h(n)$ kann also nicht durch $O(n)$ nach oben beschräkt werden. Es lässt sich aber einfach zeigen, dass $h(n) = \Theta(n^2)$.

Asymptotische Laufzeit von InsertionSort. Zunächst zeigen wir, dass die Worst-Case Laufzeit von InsertionSort in $\Theta(n^2)$, also quadratisch ist. Wir wissen, dass $T_{wc} = an^2+bn+c$ mit $a > 0$ — wir wissen nichts über das Vorzeichen von b und c — und müssen also zeigen, dass

$$c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$$

für alle $n > n_0$ und für geeignete Konstanten c_1, c_2 und n_0 gilt. Durch Division durch $n^2 > 0$ erhalten wir

$$c_1 \leq a + \frac{b}{n} + \frac{c}{n^2} \leq c_2.$$

Da allgemein $\frac{1}{n^k}$ monoton fallend ist und $\lim_{n\to\infty} \frac{d}{n^k} = 0$ (für beliebiges d und $k > 0$ gilt), müssen wir nur noch die Konstanten finden. Wählen wir n_0 so, dass

$$a > 2\left(\frac{|b|}{n_0} + \frac{|c|}{n_0^2}\right).$$

Dann können wir c_1 einfach als $\frac{a}{2}$, und c_2 als $2a$ wählen. Da a, b und c Konstanten sind, sind auch c_1 und c_2 Konstanten, und dadurch ist die asymptotische Laufzeit bewiesen. Analog können wir die Best-Case Laufzeit mit $\Theta(n)$ abschätzen. Ganz allgemein gilt:

$$\begin{aligned} a_1 n + a_0 &= \Theta(n) \\ a_2 n^2 + a_1 n + a_0 &= \Theta(n^2) \\ &\vdots \\ a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 &= \Theta(n^k) \end{aligned}$$

Anmerkung 1.3. Aus den asymptotischen Worst- und Best-Case Laufzeiten können wir nun einige Schlussfolgerungen ziehen:

- Aus der Worst-Case Laufzeit $\Theta(n^2)$ folgt eine Laufzeit von $O(n^2)$ für beliebige Eingaben.
- Aus der Worst-Case Laufzeit $\Theta(n^2)$ folgt *nicht* eine Laufzeit von $\Theta(n^2)$ für beliebige Eingabe. So gilt ja z.B. bei Insertion-Sort eine Laufzeit von $\Theta(n)$ falls die Eingabe bereits sortiert ist.
- Aus der Best-Case Laufzeit $\Theta(n)$ folgt eine Laufzeit $\Omega(n)$ für beliebige Eingaben.

1.3.3 Weitere asymptotische Schranken

Zusätzlich zu den oben genannten drei Hauptnotationen sind noch zwei weitere Notationen interessant und gebräuchlich.

Wir wissen, dass $\Theta(g(n))$ alle Funktionen umfasst, die genauso schnell wachsen wie $g(n)$ selbst. Die Menge $O(g(n))$ umfasst nicht nur diese Funktionen, sondern zusätzlich auch alle

Funktionen die schwächer wachsen als $g(n)$. Analog umfasst $\Omega(g(n))$ alle Funktionen die mindestens so schnell wachsen wie $g(n)$.

Wenn wir dies analog zu den Vergleichsoperationen der klassischen Arithmetik betrachten, könnten wir „$=$" mit Θ, „$\leq$" mit O und „$\geq$" mit Ω vergleichen. In diesem Vergleich fällt uns das „Fehlen" von „$<$" und „$>$" auf. Deren Analoga werden wir nun einführen:

Definition 1.3. Sei $g : \mathbb{N} \to \mathbb{R}^+$ eine Funktion. Wir definieren die folgenden Mengen von Funktionen:

$$\begin{aligned} o(g(n)) &= \left\{ f : \mathbb{N} \to \mathbb{R}^+ \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \right\} \\ \omega(g(n)) &= \left\{ f : \mathbb{N} \to \mathbb{R}^+ \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \right\} \end{aligned}$$

Dies ist äquivalent zu:

$$\begin{aligned} o(g(n)) &= \{ f : \mathbb{N} \to \mathbb{R}^+ \mid (\forall c > 0), (\exists n_0 > 0), (\forall n \geq n_0) : f(n) < cg(n) \} \\ \omega(g(n)) &= \{ f : \mathbb{N} \to \mathbb{R}^+ \mid (\forall c > 0), (\exists n_0 > 0), (\forall n \geq n_0) : cg(n) < f(n) \} \end{aligned}$$

Diese formal abschreckende Definition besagt, dass alle Funktionen $f(n)$ in der Menge $o(g(n))$ (sprich „klein-o") enthalten sind, bei denen die Division von f durch g eine Nullfolge bildet, d.h. die Menge umfasst alle Funktionen die echt schwächer wachsen als g selbst (denn dann wird der Divisor schneller größer als der Dividend, und die Folge strebt gegen 0). Die Menge $\omega(g(n))$ (sprich „klein-Omega") ist analog so definiert, dass sie alle Funktionen enthält, die echt stärker wachsen als $g(n)$ selbst.

Auf Grund dieser Definitionen gilt ($\stackrel{g}{=}$ bedeutet nur für geschlossene Funktionen):

$$\begin{aligned} O(g(n)) &\stackrel{g}{=} \Theta(g(n)) \cup o(g(n)) \\ \Theta(g(n)) \cap o(g(n)) &= \emptyset \\ \Omega(g(n)) &\stackrel{g}{=} \Theta(g(n)) \cup \omega(g(n)) \\ \Theta(g(n)) \cap \omega(g(n)) &= \emptyset \\ f(n) \in O(g(n)) &\Leftrightarrow g(n) \in \Omega(f(n)) \\ f(n) \in o(g(n)) &\Leftrightarrow g(n) \in \omega(f(n)) \end{aligned}$$

1.3.4 O-Notation in Termen

Manchmal sieht man Gleichungen, in denen die obigen Notationen vorkommen, z.B.

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) \ .$$

Dies bedeutet: es gibt eine Funktion $f(n) \in \Theta(n)$ mit $2n^2 + 3n + 1 = 2n^2 + f(n)$ (hier: $f(n) = 3n + 1$). Die intuitive Bedeutung des rechten Terms der Gleichung ist, dass die Funktion im wesentlichen mit $2n^2$ wächst, zu dem „nur" ein linear wachsender Anteil hinzukommt.

Entsprechend gilt $2n^2 + \Theta(n) = \Theta(n^2)$, denn es bedeutet formal

$$(\forall f(n) \in \Theta(n))\ (\exists g(n) \in \Theta(n^2))\ (\forall n)\ 2n^2 + f(n) = g(n).$$

Hinweis 1.2. Wir könnten also die Laufzeit von Gwendolines Algorithmus formal richtig mit $2n + O(1)$ beschreiben. Allerdings ist dies *nicht* die richtige Antwort auf die Frage nach der asymptotischen oberen Schranke in O-Notation. Diese muss soweit gekürzt sein wie möglich, d.h. in diesem Fall $O(n)$. Noch viel mehr ist auch von Schreibweisen wie $O(2n + 2)$ abzusehen.

Kapitel 2

Abstrakte Datentypen und Datenstrukturen

Für den Entwurf eines guten Algorithmus ist die Verwendung geeigneter Datenstrukturen von großer Bedeutung. Häufig hat man in einem Algorithmus die Wahl, für die selbe Aufgabe unterschiedliche Datenstrukturen zu verwenden. Daher ist es sinnvoll, die Schnittstelle einer Datenstruktur (also das, was die Datenstruktur kann) von der eigentlichen Implementierung zu trennen. Datenstrukturen mit gleicher Schnittstelle können dann innerhalb eines Algorithmus problemlos ausgetauscht werden, der Algorithmus verwendet lediglich einen *abstrakten Datentyp*. Die konkrete Realisierung dieses Datentyps, also die verwendete Datenstruktur, bestimmt die Laufzeit der einzelnen Operationen, die auf dem Datentyp ausgeführt werden können. Insbesondere müssen wir also bei der Analyse eines Algorithmus immer wissen, welche Datenstrukturen verwendet werden.

Formal definieren wir die Begriffe abstrakter Datentyp und Datenstruktur wie folgt:

Definition 2.1. Ein *abstrakter Datentyp (ADT)* besteht aus einem Wertebereich (d.h. einer Menge von *Objekten*) und darauf definierten *Operationen*. Die Menge der Operationen bezeichnet man auch als *Schnittstelle* des Datentyps.

Definition 2.2. Eine *Datenstruktur* ist eine *Realisierung* bzw. *Implementierung* eines ADT mit den Mitteln einer Programmiersprache (Variablen, Funktion, Schleifen, usw.). Die Realisierung ist in der Regel nicht in einer konkreten Programmiersprache sondern in Pseudocode verfasst.

Im folgenden Abschnitt zeigen wir am Beispiel des ADT *Sequence*, dass es verschiedene Realisierungen eines ADT mit unterschiedlichem Laufzeitverhalten der Operationen geben kann.

2.1 Der ADT Sequence

Der ADT *Sequence* repräsentiert eine Folge von Elementen eines gemeinsamen Grundtyps (z.B. Zahlen, Zeichenketten, ...). Der Wertebereich und die möglichen Operationen sind wie folgt spezifiziert.

Wertebereich: Der Wertebereich umfasst die Menge aller endlichen Folgen eines gegebenen Grundtyps, den wir im Folgenden mit *ValueType* bezeichnen. Wir schreiben eine Sequence S mit n Elementen als

$$S = \langle a_1, a_2, \ldots, a_n \rangle.$$

Insbesondere bezeichnet $\langle\rangle$ eine leere Sequence.

Operationen: Sei $S = \langle a_1, a_2, \ldots, a_n \rangle$, $n \geq 0$, die Sequence vor Anwendung der Operation. Wir beschreiben die Operationen von S, indem wir für jede Operation definieren, welche Parameter sie hat, was der Rückgabewert ist, und wie S nach der Operation aussieht.

- INSERT(*ValueType* x, *PositionType* p) : *PositionType*

  Fügt ein Element x vor dem Element an Position p in S ein und gibt die Position von x in S zurück. Dabei ist x ein Element des Grundtyps und p eine Position in der Sequence S. Wie eine Position beschrieben wird, hängt von der jeweiligen Implementierung ab. Um ein Element am Ende der Liste einzufügen, wird die spezielle Position *nil* angegeben.

  Falls $p =$ *nil* ist, dann ist S nach Aufruf $\langle a_1, \ldots, a_n, x \rangle$, sonst sei a_i das Element an Position p und S ist nach Aufruf $\langle a_1, \ldots, a_{i-1}, x, a_i, \ldots, a_n \rangle$. Falls p keine gültige Position bezeichnet, dann ist das Ergebnis undefiniert.

- DELETE(*PositionType* p)

  Löscht das Element an Position p aus S. Dabei ist $p \neq$ *nil* eine Position in S. Sei a_i das Element an Position p. Dann ist S nach Aufruf $\langle a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n \rangle$. Falls p keine gültige Position bezeichnet, dann ist das Ergebnis undefiniert.

- GET(**int** i) : *ValueType*

  Gibt das Element an i-ter Stelle von S zurück, wobei i eine ganze Zahl ist. Falls $1 \leq i \leq n$ ist, dann wird a_i zurückgegeben, sonst ist das Ergebnis undefiniert.

- CONCATENATE(*Sequence* S')

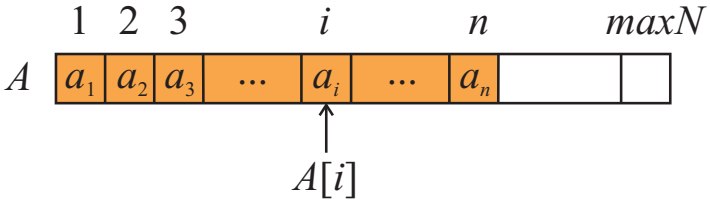
  Hängt die Sequence S' an S an, wobei S' ein Sequence vom gleichen Grundtyp wie S ist. Falls $S' = \langle a'_1, \ldots, a'_m \rangle$, dann ist S nach dem Aufruf $\langle a_1, \ldots, a_n, a'_1, \ldots, a'_m \rangle$ und S' ist leer.

Um den ADT Sequence zu realisieren, sind verschiedene Datenstrukturen möglich. Wir betrachten im Folgenden Realisierungen unter Verwendung von Feldern und verketteten Listen.

Anmerkung 2.1. Der Einfachheit halber haben wir einige Operationen weggelassen, die der Datentyp Sequence unterstützen sollte, insbesondere Funktionalität die das Iterieren über alle Elemente der Sequence erlaubt. Erweiterungen dieser Art sind meist recht einfach; wir konzentrieren uns hier bewusst auf die oben beschriebene Schnittstelle, um Unterschiede im Laufzeitverhalten der verschiedenen Realisierungen aufzuzeigen.

2.1.1 Realisierung mit Feldern

Die einfachste Möglichkeit ist es, die Sequence in einem Feld A zu speichern. Wir initialisieren das Feld für *maxN* Elemente, wobei wir davon ausgehen, dass diese Anzahl ausreichend ist. Graphisch lässt sich das wie folgt veranschaulichen:



Die Implementierung der Schnittstelle ist in Listing 2.1 dargestellt. Eine Position innerhalb der Sequence wird einfach durch den entsprechenden Feldindex repräsentiert. Wichtig bei der Implementierung eines ADTs ist es, zunächst die *interne Repräsentation* genau festzulegen, sowie anzugeben, wie diese *initialisiert* wird. In unserem Fall setzten wir die Zahl n der Elemente auf 0, wodurch wir anfangs eine leere Sequence erhalten.

Bei INSERT müssen wir zunächst überprüfen, ob wir in dem Feld überhaupt noch genügend Platz haben, da wir ja die mögliche Anzahl der Element in der Sequence mit *maxN* beschränkt haben. Danach schaffen wir durch Verschieben eines Teilbereichs des Feldes Platz für das einzufügende Element. Analog müssen wir bei DELETE verschieben. Der Einfachheit halber kodieren wir den Spezialfall $p = \mathit{nil}$ bei INSERT zum Einfügen am Ende der Sequence durch $p = n + 1$.

Hinweis 2.1. Die Festlegung der Feldgröße *maxN* ist dann sinnvoll, wenn man eine geeignete obere Schranke für die Anzahl abzuspeichernder Elemente kennt. Ist das nicht der Fall, kann man das Feld auch dynamisch vergrößern, wenn es zu klein wird, oder man verwendet gleich eine geeignetere Implementierung (siehe nächster Abschnitt).

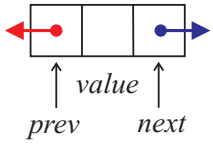
2.1.2 Realisierung mit verketteten Listen

Eine alternative Implementierung für den ADT Sequence verwendet eine *doppelt verkettete Liste*. Doppelt verkettete Listen lassen sich auf verschiedene Arten realisieren, wir betrachten hier eine ringförmig verkettete Variante, die eine sehr elegante Implementierung erlaubt.

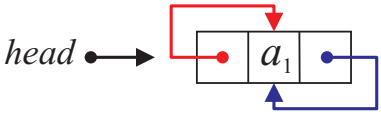
```
1:  ▷ Interne Repräsentation einer Sequence
2:  var ValueType A[1..maxN]                ▷ Feld, das Elemente der Sequence enthält
3:  var int n                               ▷ Anzahl der Elemente in der Sequence
4:  ▷ Initialisierung
5:  n := 0

6:  procedure INSERT(ValueType x, int p) : int
7:      if n = maxN then throw Overflow Exception
8:      for i := n downto p do
9:          A[i + 1] := A[i]
10:     end for
11:     A[p] := x
12:     n := n + 1
13:     return p
14: end procedure

15: procedure DELETE(int p)
16:     for i := p + 1 to n do
17:         A[i − 1] := A[i]
18:     end for
19:     n := n − 1
20: end procedure

21: function GET(int i) : ValueType
22:     return A[i]
23: end function

24: procedure CONCATENATE(Sequence S')
25:     if n + S'.n > maxN then throw Overflow Exception
26:     for i := 1 to S'.n do
27:         A[n + i] := S'.A[i]
28:     end for
29:     n := n + S'.n
30:     S'.n := 0
31: end procedure
```

Listing 2.1: Implementierung des Datentyps Sequence durch ein Feld.

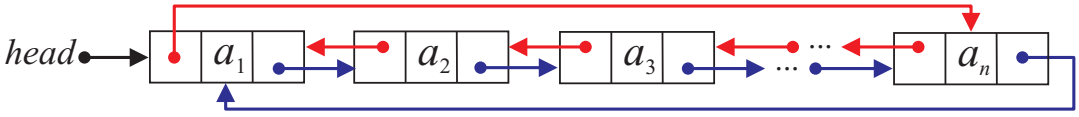
Ein *Listenelement* wir durch die Struktur *ListElement* repräsentiert:



Sie besteht aus zwei Verweisen, je einem auf den Vorgänger und einem auf den Nachfolger in der Liste, sowie dem eigentlichen Wert, der in dem Element abgespeichert ist. Wir verwenden einen Verweis auf ein Objekt vom Typ *ListElement*, um eine Position in der Liste zu bezeichnen. Die Liste selbst besteht lediglich aus einem Verweis *head* auf das erste Element in der Liste. Bei einer leeren Liste zeigt *head* auf *nil*. Eine Liste mit einem einzigen Element sieht wie folgt aus:



Sowohl der Verweis auf den Vorgänger als auch der auf den Nachfolger zeigen also auf das Element selbst. Die Liste ist somit vorwärts und rückwärts ringförmig verkettet. Insbesondere zeigen die Verweise auf Vorgänger und Nachfolger eines Elements niemals auf *nil*. Die ringförmige Verkettung wird besser deutlich, wenn wir eine Liste aus mehren Elementen $a_1, a_2, a_3, \ldots, a_n$ betrachten:



Es ist leicht zu sehen, wie eine Liste eine Sequence darstellt. Listing 2.2 zeigt die Definition der verwendeten Datenstruktur und deren Initialisierung, welche eine leere Sequence liefert.

Die Operationen INSERT und DELETE sind in Listing 2.3 dargestellt. Wenn wir ein neues Listenelement anlegen, müssen wir mit **new** Speicher für ein Objekte vom Typ *ListElement* allozieren. Entsprechend muss dieser Speicher beim Löschen eines Elements mit **delete** wieder freigegeben werden. Die Implementierungen an sich bestehen im Prinzip nur aus geschicktem Umsetzen der Vorgänger- und Nachfolger-Verweise.

Anmerkung 2.2. Programmiersprachen wie Java versuchen dem Programmierer das Leben zu erleichtern, indem auf das explizite Freigeben von mit **new** alloziertem Speicher verzichtet wird. Ein komplexer Mechanismus namens *Garbage Collection* sorgt dafür, dass nicht mehr über Verweise erreichbarer Speicher automatisch freigegeben wird. Es ist in der Regel nicht vorhersagbar, wie sich dieser Mechanismus auf das Laufzeitverhalten auswirkt. Wir gehen grundsätzlich davon aus, dass effiziente Datenstrukturen und Algorithmen selbst für die Freigabe von Speicher verantwortlich sind!

Ähnliches gilt für die Implementierung von CONCATENATE (siehe Listing 2.4). Die Elemente der Sequence S' werden einfach an das letzte Elemente der Sequence angehängt. Wichtig hierbei ist, dass der *head* der Sequence S' auf *nil* gesetzt wird, da sonst Listenelemente in zwei verschiedenen Listen vorkommen würden, was zu einer Inkonsistenz der Datenstruktur führen kann. Ungünstiger gestaltet sich allerdings die Implementierung von GET. Um auf das i-te Element der Sequence zugreifen zu können, müssen wir die Liste von vorne durchlaufen, bis wir beim i-ten Element angekommen sind.

2.1.3 Analyse der Laufzeit

Implementierung mit Feld. Bei der Initialisierung müssen wir ein Feld mit *maxN* Elementen allozieren. Der Aufwand dafür kann je nach Programmiersprache und System unterschiedlich sein, und lässt sich nicht einfach abschätzen; wie schreiben daher Alloc(*maxN*) als die Zeit, die benötigt wird einen Speicherbereich der Größe $maxN$ zu allozieren. Alle weiteren Schritte der Initialisierung verursachen nur konstanten Aufwand. Offensichtlich benötigt GET(i) konstante Laufzeit. Bei INSERT(x, p) haben wir konstanten Aufwand plus den Aufwand für die **for**-Schleife, welche $n - p + 1$ mal durchlaufen wird. Im günstigsten Fall ist $p = n + 1$ und die Schleife wird gar nicht durchlaufen, im ungünstigsten Fall ist $p = 1$ und die Schleife wird n mal durchlaufen. Daher ergibt sich eine Best-Case Laufzeit von $\Theta(1)$ und eine Worst-Case Laufzeit von $\Theta(n)$.

Analog ergeben sich bei DELETE(p) $n - p$ Schleifendurchläufe und die gleichen Laufzeitschranken wie bei INSERT. Bei CONCATENATE(S') hängt die Laufzeit einfach linear von der Anzahl der Element in S' ab, daher ist der Best- und Worst-Case hier $\Theta(1 + S'.n)$. Der Summand 1 ist erforderlich, da wir ja für $S'.n = 0$ konstanten Aufwand haben, und nicht einen Aufwand von 0.

Wir analysieren noch die Average-Case Laufzeit für INSERT(x, p). Wir nehmen an, dass alle Positionen $p = 1, \ldots, n + 1$ gleich wahrscheinlich sind, d.h. jeweils die Wahrscheinlichkeit

```
 1: ▷ Repräsentation eines Listenelements
 2: struct ListElement
 3:     var ListElement prev                    ▷ Verweis auf Vorgänger
 4:     var ListElement next                    ▷ Verweis auf Nachfolger
 5:     var ValueType value                     ▷ gespeicherter Wert
 6: end struct
 7: ▷ Interne Repräsentation einer Sequence
 8: var ListElement head
 9: ▷ Initialisierung
10: head := nil
```

Listing 2.2: Interne Repräsentation einer Sequence durch eine verkette Liste.

```
1:  procedure INSERT(ValueType x, ListElement p) : ListElement
2:      var ListElement q := new ListElement          ▷ Neues Listenelement anlegen.
3:      q.value := x
4:      if head = nil then                            ▷ War Liste vorher leer?
5:          head := q.next := q.prev := q
6:      else
7:          if p = head then head := q
8:          if p = nil then p := head                 ▷ hinten anfügen bedeutet vor head einfügen
9:          q.next := p
10:         q.prev := p.prev
11:         q.next.prev := q.prev.next := q
12:     end if
13:     return q
14: end procedure

15: procedure DELETE(ListElement p)
16:     if p.next = p then
17:         head := nil
18:     else
19:         p.prev.next := p.next
20:         p.next.prev := p.prev
21:         if p = head then head := p.next
22:     end if
23:     delete p                                      ▷ Speicher für Listenelement wieder freigeben.
24: end procedure
```

Listing 2.3: Einfügen und Löschen in einer Sequence mittels verketteter Listen.

$\frac{1}{n+1}$ haben. Ein Schleifendurchlauf bis zur Position p benötigt $n - p + 1$ Zeit. Daher ist die durchschnittliche Anzahl $S(n)$ von Schleifendurchläufen

$$\begin{aligned} S(n) &= \frac{1}{n+1}\sum_{p=1}^{n+1}(n-p+1) \\ &= \frac{1}{n+1}\sum_{i=0}^{n} i = \frac{1}{n+1}\frac{(n+1)n}{2} = \frac{n}{2}. \end{aligned}$$

Die Vereinfachungen in der zweiten Zeilen basieren auf Umstellung der Summanden und Anwenden der Gauss'schen Summenformel. Daraus folgt, dass die Average-Case Laufzeit von INSERT $\Theta(n)$ ist. Analog zeigt man, dass auch die Average-Case Laufzeit von DELETE $\Theta(n)$ ist.

```
 1: procedure CONCATENATE(Sequence S')
 2:     if head = nil then
 3:         head := S'.head
 4:     else if S'.head ≠ nil then
 5:         var ListElement first := S'.head
 6:         var ListElement last := first.prev
 7:         head.prev.next := first
 8:         last.next := head
 9:         first.prev := head.prev
10:         head.prev := last
11:     end if
12:     S'.head := nil                          ▷ Sequence S' ist jetzt leer!
13: end procedure

14: function GET(int i) : ValueType
15:     var ListElement p := head
16:     while i > 1 do
17:         p := p.next
18:         i := i − 1
19:     end while
20:     return p.value
21: end function
```

Listing 2.4: Zugriff und Konkatenieren in einer Sequence mittels verketteter Listen.

Implementierung mit doppelt verketteter Liste. Die Laufzeitanalyse ist hier deutlich einfacher. INSERT, DELETE und CONCATENATE benötigen offensichtlich nur konstante Laufzeit. Andererseits müssen wir bei GET(i) die Liste bis zum i-ten Element durchlaufen. Das ergibt im Best-Case (für $i = 1$) eine Laufzeit von $\Theta(1)$ und im Worst-Case (für $i = n$) eine Laufzeit von $\Theta(n)$. Der Average-Case hat wie der Worst-Case eine Laufzeit von $\Theta(n)$, wenn alle Positionen gleich wahrscheinlich sind (Nachrechnen!).

Diskussion. Tabelle 2.1 gibt nochmals einen Überblick über die Laufzeitenschranken. Wir sehen, dass wir die Einfachheit der Implementierung mittels Feldern teuer erkauft haben. Sowohl INSERT und DELETE wie auch CONCATENATE benötigen bei Listen nur konstante Laufzeit, während wir selbst im Average Case bei Feldern lineare Laufzeit bzw. eine Laufzeit linear in der Größe der anzuhängenden Sequence (bei CONCATENATE) benötigen. Lediglich der wahlfreie Zugriff auf ein Element (GET-Operation) wird von Feldern effizienter unterstützt. Darüber hinaus benötigt die Implementierung durch Felder (um einen konstanten Faktor) weniger Speicher.

Wir folgern daraus, dass Listen dann vorzuziehen sind, wenn wir Elemente an beliebiger

	Best-Case		*Average- und Worst-Case*	
Operation	*Felder*	*Listen*	*Felder*	*Listen*
Initialisierung	$\Theta(1)+\text{Alloc}(\textit{maxN})$	$\Theta(1)$	$\Theta(1)+\text{Alloc}(\textit{maxN})$	$\Theta(1)$
INSERT(x, p)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
DELETE(p)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
GET(i)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
CONCATENATE(S')	$\Theta(1 + S'.n)$	$\Theta(1)$	$\Theta(1 + S'.n)$	$\Theta(1)$

Tabelle 2.1: Laufzeiten der beiden Realisierungen des ADT Sequence.

Stelle in einer Folge einfügen oder löschen wollen, und wir nicht unbedingt auf Elemente über ihren Index zugreifen müssen. Wir werden im folgenden Abschnitt Datentypen kennen lernen, die ebenfalls Folgen von Elementen verwalten, jedoch nur eine eingeschränkte Schnittstelle unterstützen müssen. Dadurch wird die Implementierung mit Hilfe von Feldern wieder effizient möglich sein.

2.2 Stacks

Einer der fundamentalsten Datentypen ist der *Stack* (dt. *Stapelspeicher*, *Kellerspeicher*). Ein Stack unterstützt im Wesentlichen nur zwei Operationen: Man kann Elemente oben auf den Stack drauflegen und das oberste Elemente entfernen. Ein Stack "arbeitet" also im Prinzip wie ein Stapel Papier auf dem Schreibtisch. Aufgrund seiner Arbeitsweise wird ein Stack auch als *LIFO-Speicher* (Last-In, First-Out) bezeichnet.

Trotz der sehr eingeschränkten Funktionalität wird dieser Datentyp sehr häufig verwendet, z.B. zur Verwaltung des Prozedurstacks von Programmen oder bei der Auswertung von arithmetischen Ausdrücken. Die Schnittstelle wird wie folgt definiert.

Wertebereich: Ein Stack ist eine Folge $S = \langle a_1, \ldots, a_n \rangle$ von Elementen des gleichen Grundtyps *ValueType*.

Operationen: Sei $S = \langle a_1, \ldots, a_n \rangle$ der Stack vor Anwendung der Operation.

- ISEMPTY() : **bool**

  Testet, ob der Stack leer ist. Gibt **true** zurück, falls $S = \langle\rangle$, sonst **false**.

- PUSH(*ValueType* x)

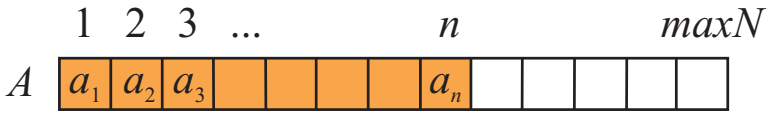
  Legt ein neues Elemente x auf den Stack. Die neue Folge ist $\langle a_1, \ldots, a_n, x \rangle$.

- POP() : *ValueType*

  Gibt das oberste Element des Stacks zurück und entfernt es. Falls $n > 0$, dann wird a_n zurückgeben und die neue Folge ist $\langle a_1, \ldots, a_{n-1} \rangle$, sonst ist das Ergebnis undefiniert.

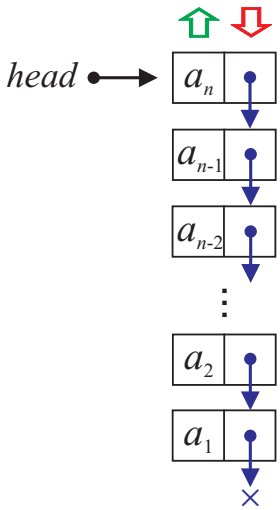
Gerade die eingeschränkte Funktionalität erlaubt es uns aber, sehr effiziente Implementierungen anzugeben. Ähnlich wie beim Datentyp Sequence betrachten wir Realisierungen mit Feldern und verketteten Listen.

Realisierung durch Felder. Genauso wie bei der Realisierung einer Sequence durch Felder speichern wir einen Stack in einem Feld ab:



Wir müssen also wiederum eine geeignete obere Schranke *maxN* bei Initialisierung des Stacks kennen. Da wir aber nicht mehr mitten im Feld Einfügen und Löschen müssen, können wir uns das ineffiziente Verschieben von Teilbereichen ersparen. Wir müssen lediglich am Ende ein Element hinzufügen (PUSH) oder wegnehmen (POP). Die vollständige Implementierung ist in Listing 2.5 angegeben.

Realisierung durch Listen. Die Realisierung durch Listen gestaltet sich ebenfalls einfacher als bei der Sequence. Wir benötigen lediglich eine einfach verkettete Liste (ein Listenelement besitzt nur einen Verweis auf seinen Nachfolger). Wir verzichten auch auf eine zyklische Verkettung, d.h. der Nachfolgerverweis des letzten Elements zeigt auf *nil*. Schematisch stellt sich das wie folgt dar:



```
 1: ▷ Interne Repräsentation des Stacks
 2: var ValueType A[1..maxN]                    ▷ Feld, das Elemente des Stacks enthält
 3: var int n                                   ▷ Anzahl Elemente im Stack
 4: ▷ Initialisierung
 5: n := 0                                      ▷ leerer Stack

 6: function IsEmpty( ) : bool
 7:     return n = 0
 8: end function

 9: procedure Push(ValueType x)
10:     if n = maxN then throw Overflow Exception
11:     n := n + 1
12:     A[n] := x
13: end procedure

14: function Pop( ) : ValueType
15:     if n = 0 then throw Underflow Exception
16:     var ValueType x := A[n]
17:     n := n − 1
18:     return x
19: end function
```

Listing 2.5: Implementierung des ADT Stack durch ein Feld.

Operation	*Felder*	*Listen*
Initialisierung	$\Theta(1)+\text{Alloc}(\textit{maxN})$	$\Theta(1)$
IsEmpty()	$\Theta(1)$	$\Theta(1)$
Push(x), Put(x)	$\Theta(1)$	$\Theta(1)$
Pop(), Get()	$\Theta(1)$	$\Theta(1)$

Tabelle 2.2: Laufzeiten der beiden Realisierungen von Stacks und Queues.

Da wir nur noch am Anfang der Liste einfügen und entfernen müssen, vereinfacht sich auch die Implementierung, welche in Listing 2.6 zu sehen ist.

Laufzeitanalyse. Die Laufzeiten der einzelnen Operationen sind im Worst- wie im Best-Case immer konstant (bis auf die Initialisierung bei Verwendung von Feldern); siehe Tabelle 2.2. Jedoch ist die Variante mit Feldern dann vorzuziehen, wenn man eine gute Maximalzahl für die Elemente kennt, da diese Implementierung natürlich auf einem realen Rechner immer schneller ist als die Variante mit Listen, welche dynamisch Speicher allozieren muss.

```
 1: struct SListElement
 2:     var SListElement next                          ▷ Verweis auf Nachfolger
 3:     var ValueType value                            ▷ abgespeicherter Wert
 4: end struct
 5: ▷ Interne Repräsentation des Stacks
 6: var SListElement head                              ▷ Anfang der Liste
 7: ▷ Initialisierung
 8: head := nil                                        ▷ leerer Stack

 9: function IsEmpty( ) : bool
10:     return head = nil
11: end function

12: procedure Push(ValueType x)
13:     var SListElement p := new SListElement
14:     p.value := x
15:     p.next := head
16:     head := p
17: end procedure

18: function Pop( ) : ValueType
19:     if head = nil then throw Underflow Exception
20:     var SListElement p := head
21:     var ValueType x := p.value
22:     head := p.next
23:     delete p
24:     return x
25: end function
```

Listing 2.6: Implementierung des ADT Stack durch eine Liste.

2.3 Queues

Im Gegensatz zu Stacks arbeiten *Queues* (dt. *Warteschlangen*) nach dem *FIFO-Prinzip* (First-In, First-Out). Das kann man sich wie eine Schlange vor der Kinokasse vorstellen. Es wird immer derjenige als nächstes bedient, der auch zuerst da war. Als Operationen stehen entsprechend Einfügen am Ende und Entfernen am Anfang der Queue zur Verfügung. Queues verwendet man also dann, wenn eingehende Daten auch in dieser Reihenfolge verarbeitet werden sollen.

Wertebereich: Eine Queue ist eine Folge $Q = \langle a_1, \ldots, a_n \rangle$ von Elementen des gleichen Grundtyps *ValueType*.

Operationen: Sei $Q = \langle a_1, \ldots, a_n \rangle$ die Queue vor Anwendung der Operation.

- ISEMPTY() : **bool**

  Testet, ob die Queue leer ist. Gibt **true** zurück, falls $Q = \langle\rangle$, sonst **false**.

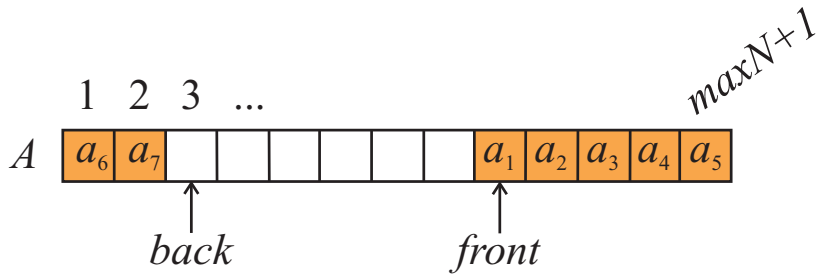
- PUT(*ValueType* x)

  Fügt ein neues Elemente x in die Queue ein. Die neue Folge ist $\langle a_1, \ldots, a_n, x \rangle$.

- GET() : *ValueType*

  Gibt das erste Element der Queue zurück und entfernt es. Falls $n > 0$, dann wird a_1 zurückgegeben und die neue Folge ist $\langle a_2, \ldots, a_n \rangle$, sonst ist das Ergebnis undefiniert.

Realisierung durch Felder. Um Queues mit Feldern effizient implementieren zu können, müssen wir etwas trickreicher vorgehen als bei Stacks. Da wir im Anfang entfernen und am Ende einfügen, jedoch nicht komplette Teilbereiche verschieben wollen, müssen wir uns die Queue zyklisch im Feld gespeichert vorstellen:

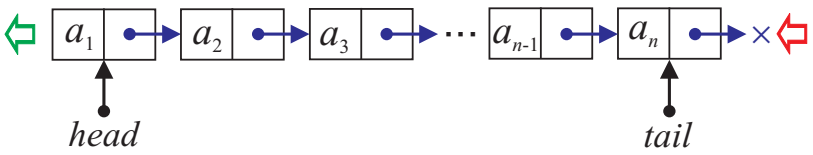


Der Anfang der Queue (*front*) kann sich an beliebiger Stelle im Feld befinden; falls die Elemente der Queue "über das Ende hinausgehen", dann wird die Folge einfach am Anfang des Feldes fortgesetzt. Das Ende der Queue wird durch *back* gekennzeichnet, was genau genommen auf ein Element hinter dem letzten Element der Queue zeigt (dort wird als nächstes eingefügt). Um korrekt auf Über- und Unterlauf testen zu können, benötigen wir ein Feld der Größe $maxN + 1$, d.h. ein Element des Feldes ist immer unbenutzt. Die komplette Implementierung befindet sich in Listing 2.7.

```
1:  ▷ Interne Repräsentation der Queue
2:  var ValueType A[1..maxN + 1]          ▷ Feld, das Elemente der Queue enthält
3:  var int front                         ▷ Index von erstem Element der Queue
4:  var int back                          ▷ Index nach letztem Element der Queue
5:  ▷ Initialisierung
6:  front := back := 1                    ▷ leere Queue

7:  function IsEmpty( ) : bool
8:      return front = back
9:  end function

10: procedure Put(ValueType x)
11:     A[back] := x
12:     if back = maxN + 1 then back := 1 else back := back + 1
13:     if front = back then throw Overflow Exception
14: end procedure

15: function Get( ) : ValueType
16:     if front = back then throw Underflow Exception
17:     var ValueType x := A[front]
18:     if front = maxN + 1 then front := 1 else front := front + 1
19:     return x
20: end function
```

Listing 2.7: Implementierung des ADT Queue durch ein Feld.

Realisierung durch Listen. Die Realisierung von Queues durch Listen gestaltet sich ähnlich wie bei Stacks, nur das wir zusätzlich einen Zeiger *tail* auf das Ende der Liste benötigen.



Interessanterweise reichen uns auch dabei einfach verkettete Listen aus und die Implementierung ist recht einfach (siehe Listing 2.8). Das Laufzeitverhalten ist dementsprechend wie bei Stacks (siehe Tabelle 2.2).

```
 1: ▷ Interne Repräsentation der Queue
 2: var SListElement head                                  ▷ Anfang der Liste
 3: var SListElement tail                                  ▷ Ende der Liste
 4: ▷ Initialisierung
 5: head := tail := nil                                    ▷ leerer Stack

 6: function IsEmpty( ) : bool
 7:     return head = nil
 8: end function

 9: procedure Put(ValueType x)
10:     var SListElement p := tail
11:     tail := new SListElement
12:     tail.value := x; tail.next := nil
13:     if head = nil then head := tail else p.next := tail
14: end procedure

15: function Get( ) : ValueType
16:     if head = nil then throw Underflow Exception
17:     var ValueType x := head.value
18:     var SListElement p := head.next
19:     delete head
20:     head := p
21:     return x
22: end function
```

Listing 2.8: Implementierung des ADT Queue durch eine Liste.

2.4 Dictionaries

Ein sehr leistungsfähiger Datentyp ist der *Dictionary* (dt. *Wörterbuch*). Ähnlich wie bei einem Englisch-Deutsch Wörterbuch werden dabei bestimmten Schlüsseln (englische Wörter) Werte (deutsche Übersetzungen) zugewiesen. Der Datentyp erlaubt es, einen neuen Schlüssel mit assoziiertem Wert in den Dictionary einzufügen, Schlüssel inklusive ihrem Wert zu entfernen, sowie zu einem Schlüssel den assoziierten Wert nachzuschlagen.

Formal definieren wir die Schnittstelle eines Dictionary folgendermaßen:

Wertebereich: Ein Dictionary D ist eine Relation $D \subseteq K \times V$, die jedem $k \in K$ höchstens ein $v \in V$ zuordnet. Die Elemente von K werden als Schlüssel und die Elemente von V als Werte bezeichnet.

Operationen: Sei Q der Dictionary vor Anwendung der Operation.

- INSERT(K k, V v)

  Fügt einen neuen Schlüssel k mit Wert v in den Dictionary D ein oder ändert den Wert eines vorhandenen Schlüssels k auf v. Falls ein $v' \in V$ existiert mit $(k, v') \in D$, dann ist der neue Dictionary $(D \setminus \{(k, v')\}) \cup \{(k, v)\}$, sonst $D \cup \{(k, v)\}$.

- DELETE(K k)

  Entfernt einen Schlüssel k aus dem Dictionary D. Falls ein $v \in V$ existiert mit $(k, v) \in D$, dann ist der neue Dictionary $D \setminus \{(k, v)\}$, sonst D.

- SEARCH(K k) : $V \cup \{$*undef*$\}$

  Gibt den beim Schlüssel k gespeicherten Wert zurück. Falls ein $v \in V$ existiert mit $(k, v) \in D$, dann wird v zurückgegeben, sonst wird *undef* zurückgegeben.

Das Problem, eine Datenstruktur mit möglichst effizienten Implementierungen für die Operationen eines Dictionary zu finden, nennt man das *Wörterbuchproblem*. Unsere Aufgabe ist es nun, effiziente Implementierung dafür zu finden.

Eine triviale mögliche Realisierung des Datentyps Wörterbuch verwendet eine doppelt verkettete Liste. Offensichtlich können wir durch lineares Durchlaufen der Liste die Operation *Search* implementieren. Das hat allerdings eine Laufzeit von $O(n)$, wenn n die Anzahl der Elemente im Dictionary ist. Damit können wir dann auch *Delete* und *Insert* in $O(n)$ Zeit implementieren (auch bei *Insert*(k, v) müssen wir zunächst nach dem Schlüssel k suchen, damit nicht zwei Werte für k im Dictionary enthalten sind). Für große Datenmengen sind diese Laufzeiten aber viel zu ineffizient

Im Kapitel 4 werden wir sehen, dass man den Aufwand für alle Operationen auf $O(\log n)$ verringern kann, indem man balancierte binäre Suchbäume verwendet (siehe Abschnitt 4.3). Ändert sich die Datenmenge fast nie, d.h. wir haben fast nur *Search* Operationen, dann kann man die Schlüssel auch sortiert in einem Feld verwalten und mit Hilfe von *Binärer Suche* in $O(\log n)$ Zeit suchen (siehe Kapitel 4.1). Unter gewissen Bedingungen kann man das Wörterbuchproblem sogar in nahezu konstanter Zeit lösen (siehe Hashing-Verfahren in Kapitel 5).

2.5 Priority Queues

Eine Priority Queue (dt. Prioritätswarteschlange) verwaltet eine Menge von Elementen, die jeweils eine Priorität haben. Die Prioritäten selbst sind aus einer linear geordneten Menge. Neben dem Einfügen und Löschen von Elementen erlaubt eine Priority Queue den effizienten Zugriff auf das Element mit minimaler Priorität. Priority Queues sind wesentliche Bestandteile von Netzwerkalgorithmen (siehe Abschnitt 6.6.1) und geometrischen Algorithmen.

Wertebereich: Ein Priority Queue S ist eine Multi-Menge von Paaren $(p, v) \in P \times V$, wobei P eine linear geordnete Menge von Prioritäten und V eine beliebige Menge von Werten ist.

Operationen: Sei S die Priority Queue. Ähnlich wie beim ADT *Sequence* verweisen wir mit einem Element vom Typ *PositionType* auf ein Element in S.

- INSERT($P\ p, V\ v$) : *PositionType*

  Fügt ein neues Element (p, v) in S ein und gibt den Verweis darauf zurück.

- DELETE(*PositionType pos*)

  Löscht das Element in S, auf das *pos* verweist.

- ISEMPTY() : **bool**

  Gibt **true** zurück, wenn $S = \emptyset$, **false** sonst.

- PRIORITY(*PositionType pos*) : P

  Gibt die Priorität des Elements zurück, auf das *pos* verweist.

- MINIMUM() : *PositionType*

  Gibt den Verweis auf ein Element mit minimaler Priorität in S zurück.

- EXTRACTMIN() : $P \times V$

  Gibt die Priorität und den den Wert eines Elementes in S mit minimaler Priorität zurück und entfernt es aus S.

- DECREASEPRIORITY(*PositionType* $pos, P\ p'$)

  Setzt die Priorität des Elementes (p, v), auf das *pos* verweist, auf p'. Dabei darf p' nicht größer sein als p.

Wir werden im Abschnitt 3.1.6 eine effiziente Implementierung von Priority-Queues kennen lernen.

Kapitel 3

Sortieren

Sortieren ist ein klassisches Problem sowohl der theoretischen als auch der angewandten Informatik. Bei der Verwaltung großer Datenmengen ist das Sortieren von Zahlen oder Objekten im Rechner eine einfach erscheinende, aber dennoch unentbehrliche Aufgabe. Das Ziel beim Sortierproblem ist es, Datensätze, die über einen (oder mehrere) Schlüssel charakterisiert sind, in eine nach den Schlüsselwerten geordnete Reihenfolge zu bringen. Häufig taucht das Sortierproblem als Teilproblem von praktischen Aufgabenstellungen auf, deshalb ist eine solide Kenntnis der grundlegenden Eigenschaften der verschiedenen Sortieralgorithmen für Informatiker unerlässlich. Da das Sortieren einen signifikanten Teil der Rechenzeit für die Lösung vieler Aufgabenstellungen beansprucht, gibt es eine Vielzahl von Ansätzen, um möglichst effiziente Sortieralgorithmen zu entwickeln. Hierbei spielen auch die verwendeten Datenstrukturen eine wichtige Rolle. Vielfach werden Mengen nur einmal sortiert und dann ständig in sortierter Reihenfolge gehalten, wobei für die effiziente Umsetzung eine entsprechende Datenstruktur benötigt wird. Neben der Rechenzeit kann, gerade bei sehr großen Datenmengen, der zusätzlich zur Eingabegröße benötigte Speicher oder die Möglichkeit der Ausführung auf externem Speicher ein wichtiges Kriterium bei der Auswahl eines geeigneten Sortieralgorithmus sein.

Formal können wir das *Sortierproblem* wie folgt beschreiben:

Sortierproblem	
Gegeben:	Folge von Datensätzen $s_1, s_2, \ldots, s_n$ mit den Schlüsseln $k_1, k_2, \ldots, k_n$, auf denen eine Ordnungsrelation „$\leq$“ definiert ist
Gesucht:	Permutation $\pi : \{1, 2, \ldots, n\} \stackrel{1:1}{\rightarrow} \{1, 2, \ldots, n\}$, so dass die Umordnung der Datensätze gemäß π die Schlüssel in aufsteigende Reihenfolge bringt: $k_{\pi(1)} \leq k_{\pi(2)} \leq \ldots \leq k_{\pi(n)}$

Mit dieser Beschreibung ist nicht festgelegt, wie die Datensätze und Schlüssel aufgebaut sind und wie wir die Effizienz eines Verfahrens messen können. Im Weiteren arbeiten wir

(sofern nichts anderes gesagt wird) auf einem Array, d.h. die Datensätze der Folge sind in einem Feld $A[1], \ldots, A[n]$ gespeichert. Die Schlüssel sind folgendermaßen ansprechbar: $A[1].\textit{key}, \ldots, A[n].\textit{key}$. Jeder Datensatz beinhaltet ein zusätzliches Informationsfeld, das über $A[1].\textit{data}, \ldots, A[n].\textit{data}$ ansprechbar ist. Dieses Informationsfeld kann im Allgemeinen sehr groß sein und kann beliebige Zeichen enthalten (z.B. string, integer, ...). Nach dem Sortieren gilt:

$$A[1].\textit{key} \leq A[2].\textit{key} \leq \ldots \leq A[n].\textit{key}.$$

Laufzeitmessung. Statt alle primitiven Operationen zu zählen, interessieren wir uns hier jeweils für zwei besondere Charakteristika:

- Anzahl der durchgeführten Schlüsselvergleiche und
- Anzahl der durchgeführten Bewegungen von Datensätzen.

Da sich durch die in der Eingabe gegebene Anordnung der Elemente signifikante Unterschiede der Laufzeit ergeben können (Algorithmen können sich im Fall einer „fast sortierten“ Folge z.B. besser oder auch schlechter verhalten), analysieren wir die Algorithmen jeweils für den Best-Case, den Worst-Case und einige Algorithmen auch für den Average-Case über alle $n!$ möglichen Anfangsanordnungen.

Die Laufzeitfunktionen für die Schlüsselvergleiche in den jeweiligen Fällen sind $C_{\text{best}}(n)$, $C_{\text{worst}}(n)$ und $C_{\text{avg}}(n)$, wobei C für *Comparisons* steht. Für die Bewegungen sind es dementsprechend $M_{\text{best}}(n)$, $M_{\text{worst}}(n)$ und $M_{\text{avg}}(n)$, wobei M für *Movements* steht. Die Laufzeit des Algorithmus, wenn wir uns weder auf Vergleiche noch auf Bewegungen beschränken, wird in der Regel dem Maximum dieser beiden Werte entsprechen.

Wir betrachten bei unseren Analysen immer den Fall, dass die Schlüssel paarweise verschieden sind. Sind die Informationsfelder der Datensätze sehr groß, dann ist jedes Kopieren der Daten (d.h. jede Bewegung) sehr aufwändig und zeitintensiv. Deshalb erscheint es manchmal günstiger, einen Algorithmus mit mehr Schlüsselvergleichen aber weniger Datenbewegungen einem anderen Algorithmus mit weniger Schlüsselvergleichen aber mehr Datenbewegungen vorzuziehen.

Für Sortierverfahren gibt es unabhängig von ihrer Effizienz einige relevante Eigenschaften.

In situ. Wenn ein Sortieralgorithmus zusätzlich zur Eingabe, also etwa einem Feld von Zahlen, höchstens konstant viel zusätzlichen Speicher benötigt, nennt man ihn auch *in situ* (für Sortieren „am Platz “, manchmal auch *in place* genannt).

Anmerkung 3.1. Es gibt auch Definitionen dieser Eigenschaft, die logarithmisch viel Platz erlauben.

Adaptiv. In der Praxis hat man selten wirklich zufällige Eingaben. Häufig sind die Eingabeinstanzen schon teilweise vorsortiert (wobei wir hier nicht auf die verschiedenen Maße eingehen wollen, mit denen man diese Vorsortierung misst). Einige Sortieralgorithmen können diese Vorsortierung derart ausnutzen, dass sich ihre Effizienz steigert. Wir nennen ein Sortierverfahren *adaptiv*, wenn es auf einer fast sortierten Eingabe asymptotisch schneller läuft als auf einer unsortierten Eingabe.

Stabil. Ein Sortierverfahren heißt *stabil*, wenn in der sortierten Reihenfolge die aus der Eingabe gegebene relative Reihenfolge der Elemente mit gleichem Schlüssel beibehalten wird, d.h., steht ein Element e_1 mit dem Schlüssel 5 in der Eingabe vor einem Element e_2 mit dem Schlüssel 5, so steht e_1 auch in der sortierten Ausgabe vor e_2.

Intern/Extern. Geht das Verfahren davon aus, dass alle Daten während des Sortierens im Hauptspeicher gehalten werden können, so spricht man von einem *internen* Sortierverfahren, werden Daten hingegen auf externen Speicher ausgelagert, spricht man von einem *externen* Sortierverfahren. Algorithmen können für diese Fälle unterschiedlich gut geeignet sein; dies wird in Abschnitt 3.3 genauer besprochen.

Im Folgenden werden die wichtigsten Sortieralgorithmen vorgestellt.

3.1 Allgemeine Sortierverfahren

Allgemeine Sortierverfahren (auch *vergleichsorientierte* Verfahren genannt) setzen ausser der Vergleichbarkeit von Schlüsseln in konstanter Zeit kein Vorwissen über die zu sortierenden Daten voraus. Insbesondere nutzen sie kein Wissen über die Struktur und die arithmetischen Eigenschaften der zu sortierenden Schlüssel, z.B. dass diese aus einem endlichen Intervall ganzer Zahlen kommen oder ähnliches.

3.1.1 Insertion-Sort

Der Algorithmus für Insertion-Sort (*Sortieren durch Einfügen*) wurde bereits in den Abschnitten 1.2 und 1.3 behandelt, wobei wir auch den Aufwand berechnet haben. Wir holen daher an dieser Stelle nur noch die Aufteilung dieses Aufwands in Schlüsselvergleiche und Datenbewegungen nach.

Der einzige Schlüsselvergleich findet sich in Zeile 7 von Listing 1.1. Aus der Analyse in Abschnitt 1.3 wissen wir, dass diese Zeile im Best-Case $\Theta(n)$ mal durchlaufen wird, während sie im Worst-Case $\Theta(n^2)$ mal bearbeiten werden muss. Daher gilt:

$$C_{\text{best}}(n) = \Theta(n) \quad \text{und} \quad C_{\text{worst}}(n) = \Theta(n^2).$$

Im Mittel ist die Hälfte der vorangehenden Elemente größer als das k-te Element, deshalb unterscheidet sich der Average-Case nur um eine Konstante vom Worst-Case und es gilt

$$C_{\text{avg}}(n) = \Theta(n^2).$$

Für die Analyse der Datenbewegungen müssen wir uns Zeile 8 des Algorithmus 1.1 betrachten, die sich in der inneren Schleife befindet (Zeilen 5 und 11 ergeben linearen Aufwand. Damit ergibt sich wie zuvor aus den bereits berechneten Laufzeitfunktionen:

$$M_{\text{best}}(n) = \Theta(n) \quad \text{und} \quad M_{\text{worst}}(n) = M_{\text{avg}}(n) = \Theta(n^2).$$

Diskussion von Insertion-Sort.

- Worst-Case Laufzeit $O(n^2)$
- Insertion-Sort ist ein stabiles Sortierverfahren, das in situ arbeitet.
- Der Algorithmus profitiert davon, wenn die Eingabe schon annähernd sortiert ist, da die Laufzeit direkt von der Zahl der Inversionen abhängt, das ist die Anzahl der Elemente links von einem Element an Stelle i, die größer sind als dieses Element. Er ist also adaptiv.
- Insertion-Sort ist einfach zu implementieren und wird, da es auf kleinen Eingaben sehr effizient arbeitet, in der Praxis als Subroutine für andere Sortierverfahren zur Bearbeitung kleiner Teilprobleme eingesetzt.
- Die Suche nach der Einfügestelle kann durch binäre Suche (siehe Kapitel 4) verbessert werden.

3.1.2 Selection-Sort

Selection-Sort (*Sortieren durch Auswahl*) ist ein weiterer einfacher Sortieralgorithmus. Wie Insertion-Sort geht er davon aus, dass die Folge links von zur aktuellen Position j bereits sortiert ist. Bei Insertion-Sort wird nun jeweils das Element an der Position j in die bereits sortierte Teilfolge eingeordnet, indem alle Elemente die größer sind, der Reihe nach einen Schritt nach rechts rücken und dadurch schließlich einen Platz an der richtigen Stelle der Teilfolge schaffen. Im Unterschied dazu durchsucht Selection-Sort alle Elemente von Position j bis Position n und wählt aus ihnen das Element mit dem kleinsten Schlüssel aus. Dieses wird dann in einer einzigen Datenbewegung an die Position j getauscht.

Abbildung 3.1 illustriert Selection-Sort an einem Beispiel. In jeder Zeile ist jenes Element durch einen Kreis markiert, das als kleinstes Element der noch unsortierten (rechten) Teilfolge ausgewählt wird.

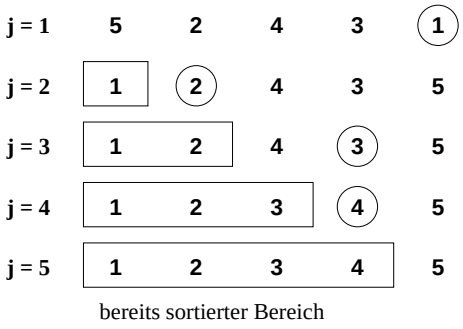


Abbildung 3.1: Beispiel für Selection-Sort.

Pseudocode des Algorithmus. In Listing 3.1 findet sich der Pseudocode von Selection-Sort. In vielen Implementierungen wird die Abfrage in Zeile 10 weggelassen und einfach immer die Vertauschung durchgeführt. Was in der Praxis effizienter ist, hängt vom Aufwand des Indexvergleichs relativ zum Aufwand der Vertauschung ab.

Eingabe: Folge in Feld A
Ausgabe: sortierte Folge in Feld A

```
 1: procedure SELECTIONSORT(ref A)
 2:     var Indizes minpos, i, j
 3:     for j := 1, 2, ..., n − 1 do  ▷ Bestimme Position des Minimums aus A[j], ..., A[n]
 4:         minpos := j
 5:         for i := j + 1, ..., n do
 6:             if A[i].key < A[minpos].key then
 7:                 minpos := i
 8:             end if
 9:         end for
10:         if minpos > j then
11:             Vertausche A[minpos] mit A[j]
12:         end if
13:     end for
14: end procedure
```

Listing 3.1: Selection-Sort

Analyse von Selection-Sort Wir zählen die Schlüsselvergleiche, das sind die Vergleiche in Zeile 6. Diese werden in jedem Durchlauf der beiden **for**-Schleifen immer ausgeführt,

deshalb ergibt sich:

$$C_{\text{best}}(n) = C_{\text{worst}}(n) = C_{\text{avg}}(n) = \sum_{j=1}^{n-1}(n-j) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \Theta(n^2)$$

Die Datenbewegungen finden in Zeile 11 statt. Im Worst-Case müssen wir in jedem Durchlauf der äusseren Schleife Daten bewegen, im Durchschnitt immerhin noch in der Hälfte der Fälle. Falls die Eingabe bereits sortiert ist, benötigen wir allerdings gar keine Vertauschung. Es gilt also:

$$M_{\text{best}}(n) = 0, \quad M_{\text{worst}}(n) = M_{\text{avg}}(n) = \Theta(n)$$

Wir werden später sehen, dass es weitaus kostengünstigere Sortieralgorithmen gibt. Allerdings bringen diese meist einen höheren Bewegungsaufwand mit sich.

Diskussion von Selection-Sort.

- Laufzeit $\Theta(n^2)$.
- Die Laufzeit des Algorithmus profitiert nicht davon, wenn die Eingabe schon teilweise sortiert ist, allerdings kann die Anzahl der Datenbewegungen sinken.
- Selection-Sort arbeitet in situ und ist nicht stabil.
- Der Einsatz von *Selection-Sort* kann sich lohnen, falls die Bewegungen von Datensätzen teuer sind, und die Vergleiche zwischen den Schlüsseln billig.

3.1.3 Merge-Sort

Merge-Sort (*Sortieren durch Verschmelzen*) ist einer der ältesten für den Computer entwickelten Sortieralgorithmen und wurde erstmals 1945 durch John von Neumann vorgestellt. Er folgt einem wichtigen Entwurfsprinzip für Algorithmen, das als *„Divide and Conquer“* (*„Teile und Erobere“*) bezeichnet wird:

- **Teile** das Problem in Teilprobleme auf.
- **Erobere** die Teilprobleme durch rekursives Lösen. Wenn sie klein genug sind, löse sie direkt.
- **Kombiniere** die Lösungen der Teilprobleme zu einer Lösung des Gesamtproblems.

Für das Sortierproblem kann man dieses Prinzip wie folgt umsetzen:

- **Teile:** Teile die n-elementige Folge in der Mitte in zwei Teilfolgen.
- **Erobere:** Sortiere beide Teilfolgen rekursiv mit *Merge-Sort*. Für einelementige Teilfolgen ist nichts zu tun.
- **Kombiniere:** Verschmelze die beiden Teilfolgen zu einer sortierten Gesamtfolge.

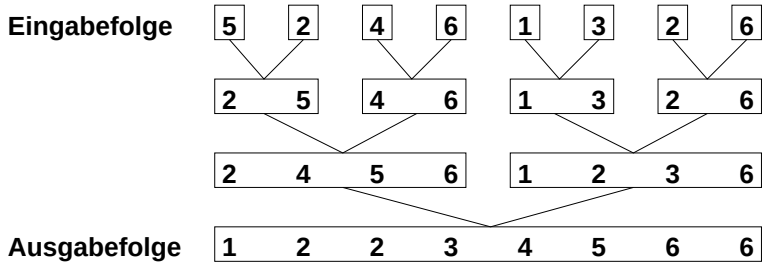


Abbildung 3.2: Illustration von Merge-Sort.

Das Aufteilen in kleinere Teilprobleme hat bei Merge-Sort also eine sehr einfache Struktur, wir teilen die Folge in der Mitte des Eingabefeldes. Die Kombination zweier bereits sortierter Teilfolgen geschieht durch Ablaufen beider Teilfolgen und elementweisen Vergleich. Das kleinere der beiden Elemente wird in das Ergebnis übernommen und wir rücken in der entsprechenden Teilfolge zum nächsten Element weiter und vergleichen wieder, bis alle Elemente abgearbeitet sind. Abbildung 3.2 illustriert die Idee von Merge-Sort an einem kleinen Beispiel.

Pseudocode des Algorithmus. In Listing 3.2 findet sich die rekursive Implementierung von Merge-Sort. Wird der Algorithmus für eine leere oder einelementige Teilfolge aufgerufen, so terminiert die Rekursion. Listing 3.3 zeigt die zugehörige Operation MERGE.

- Das Sortieren eines n-elementigen Feldes geschieht durch den Aufruf von MERGE-SORT $(A, 1, n)$.
- MERGESORT(A, l, r) sortiert $A[l, \ldots, r]$ (falls $l \geq r$ gilt, so ist nichts zu tun).
- MERGE(A, l, m, r) verschmilzt die beiden Teilfelder $A[l, \ldots, m]$ und $A[m+1, \ldots, r]$ die bereits sortiert sind, wobei $l \leq m \leq r$. Das Resultat ist ein sortiertes Feld $A[l, \ldots, r]$. Die Laufzeit von *Merge* beträgt $\Theta(r - l + 1)$.

Merge-Sort ist ein rekursiver Algorithmus, d.h. er ruft sich immer wieder selbst auf. Rekursive Algorithmen können ein bisschen schwieriger zu verstehen sein, aber sie sind manchmal wesentlich eleganter und übersichtlicher zu programmieren als iterative Varianten. Es ist wichtig, sich mit diesem rekursiven Ansatz genauer zu beschäftigen, weil gerade dieses Design-Prinzip immer wieder auftaucht.

Eingabe: Folge A; Indexgrenzen l und r (falls $l \geq r$ ist nichts zu tun)
Ausgabe: sortierte Teilfolge in $A[l]$,...,$A[r]$

```
1: procedure MERGESORT(ref A, l, r)
2:     var Index m
3:     if l < r then
4:         m := ⌊(l + r)/2⌋
5:         MERGESORT(A, l, m)
6:         MERGESORT(A, m + 1, r)
7:         MERGE(A, l, m, r)
8:     end if
9: end procedure
```

Listing 3.2: Merge-Sort

Pseudocode von Merge. Unser Pseudocode von MERGE nutzt drei Hilfsindizes, i ist ein Index in der ersten Teilfolge, j ein Index in der zweiten Teilfolge und k der Index für das Hilfsfeld B, in welchem wir das Ergebnis zwischenspeichern. Der Algorithmus läuft in einer Schleife durch die Indizes des Hilfsfeldes und muss sich entscheiden ob er das nächste Element aus der ersten oder aus der zweiten Teilfolge auswählt. Ist der Index i zu groß, oder sind beide Indizes i und j gültig und der Schlüssel an der Stelle j kleiner als der an der Stelle i, so wird das durch j spezifizierte Element an die k-te Position in B kopiert. Ansonsten wird das durch i angegebene Element ausgewählt. Da der entsprechende Index danach um 1 erhöht wird, wird jedes Element der beiden Teilfolgen genau einmal nach B kopiert. Insgesamt erhalten wir ein sortiertes Hilfsfeld B, dass wir schliesslich wieder in das Feld A zurückschreiben.

Anmerkung 3.2. Der Aufwand für das Umkopieren kann reduziert werden, indem das Feld B nicht lokal, sondern global verwaltet wird und die Felder A und B ihre Rolle zum Ende einer Rekursionsstufe wechseln. Das Ergebnis verbleibt zu Anfang in Hilfsfeld B, das als Eingabe für die nächste Stufe genommen wird, bei der das Ergebnis dann im Feld A aufgebaut wird, das Ergebnis einer Rekursionsstufe landet also jeweils alternierend in A bzw. B. Am Ende muss man dann nur einmal das Ergebnis von B nach A kopieren, wenn die Rekursionstiefe ungerade ist, da dann das letzte Ergebnis in B steht.

Anmerkung 3.3. Wir gehen im Pseudocode in Zeile 6 explizit davon aus, dass die sogenannte *short circuit* Auswertung benutzt wird (und von links nach rechts ausgewertet wird), das heißt logische Ausdrücke werden nur solange ausgewertet, bis der Wert des Ausdrucks nicht mehr geändert werden kann. Dadurch wird der Schlüsselvergleich nur ausgeführt wenn nötig.

Intuitive Herleitung der Laufzeitfunktion. Wir analysieren die Laufzeit zunächst einmal informell, um ein Verständnis für die relevanten Abläufe zu bekommen. Wir nehmen der

Eingabe: sortierte Teilfolgen $A[l], \ldots, A[m]$ und $A[m+1], \ldots, A[r]$
Ausgabe: verschmolzene, sortierte Teilfolge in $A[l], \ldots, A[r]$

```
 1: procedure MERGE(ref A, l, m, r)
 2:     var Hilfsfolge B; Indizes i, j, k
 3:     i := l;                          ▷ i läuft von A[l] bis A[m] (ersten Teilfolge)
 4:     j := m + 1;                      ▷ j läuft von A[m+1] bis A[r] (zweiten Teilfolge)
 5:     for k := l to r do               ▷ das nächste Element der Resultatfolge ist B[k]
 6:         if (i > m) or ((j ≤ r) and (A[i].key > A[j].key)) then
 7:             B[k] := A[j];    j := j + 1
 8:         else
 9:             B[k] := A[i];    i := i + 1
10:         end if
11:     end for
12:     Schreibe sortierte Folge von B zurück nach A
13: end procedure
```

Listing 3.3: Merge

Einfachheit halber an, dass $n = 2^k$ für ein beliebiges $k \in \mathbb{N}$. In jedem Aufteilungsschritt wird die Anzahl der zu lösenden Teilinstanzen verdoppelt, die Größe dieser Teilinstanzen jedoch jeweils halbiert. Der Aufwand für das Aufteilen einer Folge ist konstant, der Aufwand für die Merge-Funktion besteht aus einem Durchlauf und ist damit linear in der Größe der Folge. Damit bleibt der Gesamtaufwand in allen Teile- und Kombiniere-Schritten der gleichen Stufe gleich n. Abbildung 3.3 zeigt dies für $n = 2^3 = 8$.

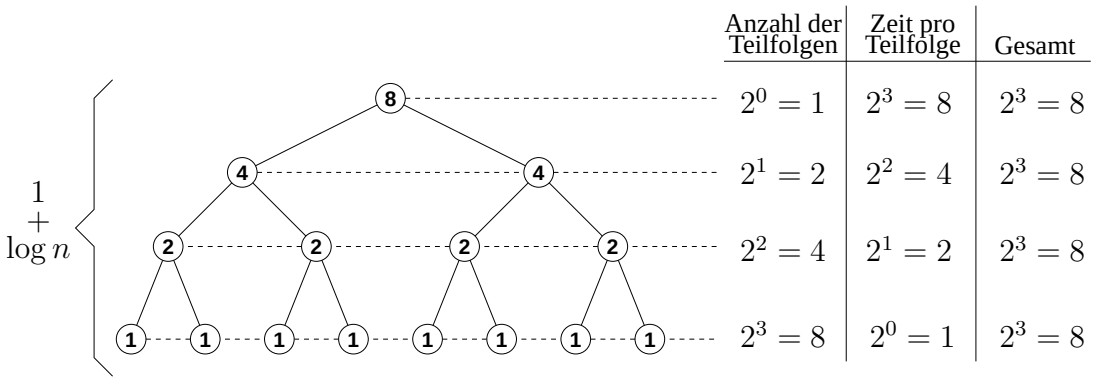


Abbildung 3.3: Visualisierung für $n = 2^3 = 8$ und $a = 1$. In den Kreisen steht die Anzahl der Elemente der zu bearbeitenden Folge.

Wieviele dieser Stufen gibt es? Die Folgenlänge n wird so lange (x mal) halbiert, bis die Teilfolgenlänge 1 ist, d.h. $\frac{n}{2^x} = 1$. Also gilt $x = \log n$. Es gibt also genau $\log n + 1$ solcher

Stufen.

Daraus ergibt sich für $n = 2^k$ eine Laufzeit von:

$$\begin{aligned} T(n) &= \text{Aufwand pro Stufe} \cdot \text{Anzahl der Stufen} \\ &= n(1 + \log n) = n + n \log n \\ &= \Theta(n \log n) \end{aligned}$$

Formale Analyse von Merge-Sort. Analysieren wir nun den Algorithmus formal und bestätigen unser bisheriges Ergebnis.

- **Teile:** Bestimme die Mitte des Teilvorgangs in konstanter Zeit: $\Theta(1)$.
- **Erobere:** Löse die zwei Teilprobleme der Größe $\lceil \frac{n}{2} \rceil$ bzw. $\lfloor \frac{n}{2} \rfloor$: $T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right)$.
- **Kombiniere:** MERGE: $\Theta(n)$.

Dies ergibt insgesamt folgende *Rekursionsgleichung* der Laufzeitfunktion:

$$T(n) = \begin{cases} \Theta(1), & \text{für } n = 1 \\ T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + \Theta(n), & \text{für } n > 1. \end{cases}$$

Das heisst, die gesamte Laufzeit zum Sortieren einer Instanz der Größe n setzt sich aus der Summe der Zeiten zum Sortieren der linken und rechten Hälfte und der Zeit für MERGE zusammen.

Die Lösung dieser Rekursionsgleichung lautet $T(n) = \Theta(n \log n)$. Wir werden nur $O(n \log n)$ formal beweisen; $\Omega(n \log n)$ geht analog, woraus schliesslich $\Theta(n \log n)$ folgt.

Beweis. Wir zeigen $T(n) = O(n \log n)$. Aus der Rekursionsgleichung folgt: Es existiert ein $a > 0$, so dass gilt

$$T(n) \leq \begin{cases} a, & \text{für } n = 1 \\ T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + a\,n, & \text{für } n > 1 \end{cases}$$

Wir zeigen mit Induktion, dass mit $c = 3a$ für alle $n \geq 3$ gilt $T(n) \leq c\,n \log(n-1)$

Induktionsanfang:

$$\begin{aligned} n = 3: \quad T(3) &\leq T(1) + T(2) + 3a \\ &\leq T(1) + 2T(1) + 2a + 3a \\ &\leq 3T(1) + 5a \\ &\leq 8a = \tfrac{8}{3}c \\ &\leq 3c = c\,3\underbrace{\log(3-1)}_{=1} \end{aligned}$$

$$\begin{aligned} n = 4: \quad T(4) &\leq T(2) + T(2) + 4a \\ &\leq 4T(1) + 8a \\ &\leq 12a = 4c \\ &\leq c\,4\underbrace{\log(4-1)}_{>1} \end{aligned}$$

$$
\begin{array}{lrcl}
n=5: & T(5) & \leq & T(2)+T(3)+5a \\
 & & \leq & 2T(1)+2a+T(2)+T(1)+3a+5a \\
 & & \leq & 2T(1)+2a+2T(1)+2a+T(1)+3a+5a \\
 & & = & 5T(1)+12a \\
 & & \leq & 17a=\frac{17}{3}c \\
 & & \leq & 10c=c\,5\log(5-1)
\end{array}
$$

Induktionsschluss: Wir nehmen an, die Behauptung gilt für alle Instanzen der Größe kleiner als n und schließen daraus auf n:

$$
\begin{array}{rcl}
T(n) & \underset{(I.V.)}{\leq} & T\left(\lceil \frac{n}{2} \rceil\right)+T\left(\lfloor \frac{n}{2} \rfloor\right)+a\,n \\
 & \leq & c\left\lceil \frac{n}{2} \right\rceil \log\left(\left\lceil \frac{n}{2} \right\rceil-1\right)+c\left\lfloor \frac{n}{2} \right\rfloor \log\left(\left\lfloor \frac{n}{2} \right\rfloor-1\right)+a\,n \\
 & \leq & c\frac{n+1}{2}\log\left(\frac{n+1}{2}-1\right)+c\frac{n}{2}\log\left(\frac{n}{2}-1\right)+a\,n \\
 & \leq & c\frac{n+1}{2}\log\left(\frac{n-1}{2}\right)+c\frac{n}{2}\log\left(\frac{n-2}{2}\right)+a\,n \\
 & \overset{(*)}{=} & \frac{1}{2}c(n+1)\log(n-1)+\frac{1}{2}c\,n\log(n-2)-c\frac{n+1}{2}-c\frac{n}{2}+a\,n \\
 & \leq & \frac{1}{2}c\,n\log(n-1)+\frac{1}{2}c\log(n-1)+\frac{1}{2}c\,n\log(n-1)-c\,n \\
 & & -\frac{c}{2}+a\,n \\
 & = & c\,n\log(n-1)+\frac{c}{2}\log(n-1)-c\,n-\frac{c}{2}+\frac{c}{3}n \\
 & \leq & c\,n\log(n-1)-\frac{c}{2}\underbrace{(2n+1-\log(n-1))}_{\text{(für } n\geq 2)}+\frac{c}{2}n \\
 & \leq & c\,n\log(n-1)-\frac{c}{2}n+\frac{c}{2}n \\
 & = & c\,n\log(n-1)
\end{array}
$$

$(*): (\log\frac{Z}{N}=\log Z-\log N)$ □

Wir wissen also:

$$
\begin{array}{rcrcrcl}
C_{\text{worst}}(n) & = & C_{\text{best}}(n) & = & C_{\text{avr}}(n) & = & \Theta(n\log n) \\
M_{\text{worst}}(n) & = & M_{\text{best}}(n) & = & M_{\text{avr}}(n) & = & \Theta(n\log n)
\end{array}
$$

Diskussion zu Merge-Sort.

- Laufzeit $\Theta(n\log n)$.

- Merge-Sort benötigt $\Theta(n)$ zusätzlichen Speicherplatz für die Hilfsfolge B, arbeitet also nicht in situ.

- Als alternative Datenstruktur eignen sich verkettete Listen, da alle Teilfolgen nur sequentiell verarbeitet werden. Dann werden keine Daten bewegt, nur Zeiger verändert. Merge-Sort ist deshalb auch gut als externes Sortierverfahren geeignet (klassisch: Sortieren auf Bändern) und wird in der Praxis häufig dafür verwendet (siehe auch Abschnitt 3.3).

- Es existieren in der Praxis eine Vielzahl von Verbesserungen von Merge-Sort, insbesondere platz- und zeitsparende Versionen von MERGE, der Algorithmus kann ausserdem auch iterativ definiert werden.

Bedeutung der asymptotischen Laufzeit. Wir sind bei Merge-Sort im Worst-Case und Average-Case besser als bei Insertion-Sort und Selection-Sort. Welche Auswirkungen hat das in der Praxis? Nehmen wir an, es ginge in einem Wettbewerb um das Sortieren von $1\,000\,000$ Zahlen. Die Teilnehmer sind ein schneller Computer mit einem langsamen Programm und ein langsamer Computer mit einem schnellen Programm, wie die folgende Tabelle zeigt.

	Algorithmus	Implementierung	Geschwindigkeit
Schneller Computer	Insertion-Sort	$2n^2$	$1\,000$ Mio. Op./sec
Langsamer Computer	Merge-Sort	$50n \log n$	10 Mio. Op./sec

Zeitbedarf:

$$\begin{aligned} \text{Schneller Computer:} \quad & \frac{2 \cdot (10^6)^2 \text{ Operationen}}{10^9 \text{ Operationen/sec}} \\ = \; & 2\,000 \text{ sec} \\ \approx \; & 33 \text{ min} \end{aligned}$$

$$\begin{aligned} \text{Langsamer Computer:} \quad & \frac{50 \cdot 10^6 \cdot \log 10^6 \text{ Operationen}}{10^7 \text{ Operationen/sec.}} \\ \approx \; & 100 \text{ sec} \approx 1,67 \text{ min} \end{aligned}$$

Dies ist doch ein immenser Unterschied. Rechnen beide Algorithmen auf dem gleichen Computer, z.B. dem schnelleren, so ist der Unterschied noch deutlicher: die Rechenzeiten betragen 1 Sekunde gegenüber 33 Minuten. Man sieht also:

Wie Computer-Hardware sind auch Algorithmen Technologie!

3.1.4 Quick-Sort

Dieser in der Praxis sehr schnelle und viel verwendete Algorithmus wurde 1960 von C.A.R. Hoare entwickelt und basiert ebenfalls auf der Strategie „Divide and Conquer“. Während jedoch bei Merge-Sort die Folgen zuerst aufgeteilt wurden, bevor sortiert wurde, ist dies bei Quick-Sort umgekehrt: hier wird zunächst Vorarbeit geleistet, bevor rekursiv aufgeteilt wird, die Eingabefolge wird anhand eines sogenannten *Pivotelementes* zerlegt (*„partitioniert”*). Der Unterschied zu Merge-Sort besteht darin, dass der Aufwand für die Aufteilung größer ist, dafür aber der Schritt des Zusammenfügens einfach durch Aneinanderhängen der sortierten Teilfolgen geschieht. Während also Merge-Sort beim Zusammenfügen in MERGE sortiert, sortiert Quick-Sort bereits beim Partitionieren.

Idee. Wähle ein Pivotelement (z.B. das letzte in der Folge) und teile das Feld anhand dieses Pivotelements in kleinere und größere Elemente auf. Man bestimmt dabei gleichzeitig die Position, die dieses Element am Ende in der vollständig sortierten Folge einnehmen wird. Dafür werden in einem einfachen Schleifendurchlauf diejenigen Elemente, die kleiner (bzw. kleiner gleich) als das Pivotelement sind, an die linke Seite des Feldes gebracht, und diejenigen, die größer (bzw. größer gleich) sind, an die rechte Seite. Das Pivotelement wird schliesslich zwischen diese beiden Teile eingesetzt und sitzt damit an seiner richtigen Endposition: alle kleineren Elemente liegen links davon und alle größeren rechts davon – jedoch noch unsortiert. Um diese Teilfolgen (die linke und die rechte) zu sortieren, wenden wir das Verfahren rekursiv an.

Der Algorithmus ist wie folgt in das Prinzip „Divide and Conquer” einzuordnen:

- **Teile:** Falls A die leere Folge ist oder nur ein Element hat, so ist A bereits fertig sortiert. Ansonsten wähle ein Pivotelement k aus A und teile A ohne k in zwei Teilfolgen A_1 und A_2, so dass gilt: A_1 enthält nur Elemente $\leq k$ und A_2 enthält nur Elemente $\geq k$.
- **Erobere:** Zwei rekursive Aufrufe QUICKSORT(A_1) und QUICKSORT(A_2). Danach sind A_1 und A_2 sortiert.
- **Kombiniere:** Bilde A durch Hintereinanderfügen in der Reihenfolge A_1, k, A_2.

Pseudocode. Die Listings 3.4 und 3.5 zeigen den Pseudocode für Quick-Sort sowei der Hilfsoperation PARTITION, die die eigentliche Aufteilung der Daten in kleinere und grössere Schlüssel vornimmt. Das Sortieren eines n-elementigen Feldes A wird mit dem Aufruf QUICKSORT$(A, 1, n)$ durchgeführt.

Der Ablauf der Operation PARTITION$(A, 4, 9, 4)$ beim Aufruf von QUICKSORT$(A, 4, 9)$ ist in Abbildung 3.4 illustriert.

Eingabe: Folge A; Indexgrenzen l und r (falls $l \geq r$ ist nichts zu tun)
Ausgabe: sortierte Teilfolge in $A[l]$,...,$A[r]$

```
1: procedure QUICKSORT(ref A, l, r)
2:     var Index p
3:     if l < r then
4:         p := Partition(A, l, r);
5:         QUICKSORT (A, l, p − 1);
6:         QUICKSORT (A, p + 1, r);
7:     end if
8: end procedure
```

Listing 3.4: Quick-Sort

Eingabe: Folge A; Indexgrenzen l und r (falls $l \geq r$ ist nichts zu tun);
Ausgabe: Pivotstelle i; partitionierte Teilfolge $A[l]$,...,$A[r]$

```
1: function PARTITION(ref A, l, r)
2:     var Indizes i, j; Schlüsselwert x
3:     x := A[r].key
4:     i := l − 1;    j := r
5:     repeat
6:         repeat                                  ▷ Suche größeres Element als x
7:             i := i + 1
8:         until A[i].key ≥ x
9:         repeat                                  ▷ Suche kleineres Element als x
10:            j := j − 1
11:        until j < l  or A[j].key < x
12:        if i < j then
13:            vertausche A[i] und A[j]
14:        end if
15:    until i ≥ j          ▷ i ist Pivotstelle: dann sind: alle links ≤ x, alle rechts ≥ x
16:    vertausche A[i] und A[r]
17:    return i
18: end function
```

Listing 3.5: Partition

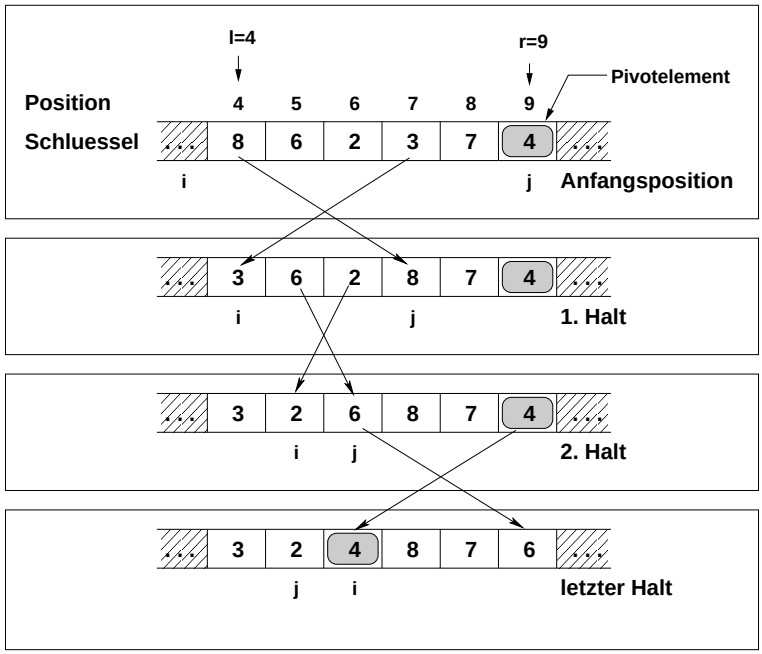


Abbildung 3.4: Illustration eines Aufrufs von QuickSort.

Korrektheit. Die Korrektheit des Algorithmus ist offensichtlich, wenn die Operation PARTITION korrekt funktioniert. Insbesondere müssen wir auch ihre Terminierung zeigen, d.h. zeigen, dass die Operation tatsächlich stoppt.

Die erste innere Repeat-Schleife (Zeile 6–8) terminiert spätestens wenn wir das gesamte Teilfeld von links nach rechts durchlaufen haben, und auf das Pivotelement stossen. Die zweite innere Repeat-Schleife (Zeile 9–11) terminiert spätestens wenn wir an den linken Rand stossen.

Die Bedingung in Zeile 12 kann nur erfüllt sein, wenn weder i noch j an die Arraygrenzen gestossen sind. In diesem Fall sind die beiden inneren Schleifen gestoppt, weil sie Elemente gefunden haben, die (im Fall von i) zu groß, und im Fall von j zu klein waren, um in der jeweiligen linken bzw. rechten Teilfolge liegen zu dürfen.

Die äussere Schleife terminiert, wenn i und j sich „überkreuzt" haben, d.h. sobald i nicht mehr den Bereich der kleineren Elemente, und j nicht mehr den Bereich der grösseren Elemente betrachtet; dies ist insbesondere immer dann der Fall, wenn i oder j an eine Arraygrenze gestossen sind.

Haben sich i und j aber überkreuzt, so bezeichnet i aber die erste Stelle des Bereichs der großen Elemente, und daher die richtige Position um das dort stehende Element mit dem Pivotelement zu vertauschen! Letzteres steht dann genau an der Grenze zwischen den kleineren und den größeren Schlüsseln.

Analyse von Quick-Sort. Die Laufzeit hängt davon ab, wie gut die Partitionierung der Eingabe in möglichst gleichgroße Teile gelingt. d.h., ob das Pivotelement möglichst in der Mitte der sortierten Folge liegt. Wir unterscheiden wieder verschiedene Fälle:

Worst-Case. Der schlechteste Fall tritt z.B. auf, wenn die Eingabeinstanz der Länge n eine bereits aufsteigend sortierte Folge ist. Dann befindet sich durch die Pivotauswahl kein Element in der zweiten Teilfolge und wir rufen Quick-Sort rekursiv auf einer $n-1$ elementigen Teilmenge auf. Der rekursive Aufrufbaum hat daher lineare Tiefe. Die Partitionierung eines n-elementigen Feldes benötigt in diesem Fall n Vergleiche über Index i und einen Vergleich über Index j, insgesamt also $n+1$ Vergleiche, für beliebige Feldlänge k benötigt man $k+1$ Vergleiche. Da wir bis zur Feldgröße 2 partitionieren, haben wir insgesamt:

$$C_{\text{worst}}(n) = \sum_{k=2}^{n}(k+1) = \sum_{k=3}^{n+1} k = \frac{(n+1)(n+2)}{2} - 3 = \Theta(n^2)$$

Die Anzahl der Bewegungen in diesem Fall ist $\Theta(n)$. Die Anzahl der Bewegungen im schlechtest möglichen Fall ist $\Theta(n \log n)$ (ohne Beweis).

Best-Case. Die durch die Aufteilung erhaltenen Folgen A_1 und A_2 haben immer ungefähr die gleiche Länge. Dann ist die Höhe des Aufrufbaumes wie schon bei Merge-Sort $\Theta(\log n)$ und auf jedem Niveau werden $\Theta(n)$ Vergleiche durchgeführt, also

$$C_{\text{best}}(n) = \Theta(n \log n)$$

Dabei ist auch die Anzahl der Bewegungen $\Theta(n \log n)$. Im besten Fall jedoch ist die Anzahl der Bewegungen (s. oben)

$$M_{\text{best}}(n) = \Theta(n).$$

Average-Case. Unsere Grundannahme hier ist, dass alle Schlüssel verschieden sind und o.B.d.A. genau der Menge $\{1, 2, \ldots, n\}$ entsprechen; alle Permutationen werden als gleich wahrscheinlich angenommen. Daher tritt jede Zahl $k \in \{1, 2, \ldots, n\}$ mit gleicher Wahrscheinlichkeit (nämlich $\frac{1}{n}$) an der letzten Position auf, und wird als Pivotelement gewählt. Mit der Auswahl von k werden zwei Folgen der Längen $(k-1)$ und $(n-k)$ erzeugt. Jede dieser Aufteilungen hat also die Wahrscheinlichkeit $\frac{1}{n}$. Die führt zu folgender Rekursionsformel für die Laufzeitfunktionen mit Konstanten a und b:

$$T_{\text{avg}}(n) \leq \begin{cases} a, & \text{für } n = 1 \\ \frac{1}{n}\sum_{k=1}^{n}(T_{\text{avg}}(k-1) + T_{\text{avg}}(n-k)) + bn, & \text{für } n \geq 2 \end{cases}$$

wobei die Summe über alle n möglichen Aufteilungen in Teilfolgen läuft und bn der Aufteilungsaufwand für eine Folge der Länge n ist.

Das Einsetzen von $T_{\text{avg}}(0) = 0$ ergibt:

$$T_{\text{avg}}(n) \leq \begin{cases} a, & \text{für } n = 1 \\ \frac{2}{n} \sum_{k=1}^{n-1} T_{\text{avg}}(k) + bn, & \text{für } n \geq 2 \end{cases}$$

Die Lösung der Rekursionsgleichung lautet:

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

was wir in der Folge beweisen werden. Durch den Best-Case wissen wir schon, dass $T_{\text{avg}}(n) = \Omega(n \log n)$, daher müssen wir nur noch $T_{\text{avg}}(n) = O(n \log n)$ formal zeigen.

Beweis. Wir zeigen mit Hilfe von vollständiger Induktion, dass gilt:

$$T_{\text{avg}}(n) = O(n \log n), \text{ d.h.}$$

$$\exists c, n_0 > 0 : \forall n \geq n_0 : 0 \leq T_{\text{avg}}(n) \leq cn \log n$$

Wir wählen $n_0 = 2$.

Induktionsanfang: $n = 2 : T_{\text{avg}}(2) = T_{\text{avg}}(1) + bn = a + 2b$
Somit ist $T_{\text{avg}}(2) \leq c\, 2 \log 2$ genau dann, wenn $c \geq \frac{a+2b}{2} = \frac{a}{2} + b$

Induktionsschluss: Es gelte nun $n \geq 3$:

$$\begin{aligned} T_{\text{avg}}(n) &\leq \frac{2}{n} \sum_{k=1}^{n-1} T_{\text{avg}}(k) + bn \\ &\overset{IV}{\leq} \frac{2}{n} \sum_{k=1}^{n-1} (ck \log k) + bn \\ &= \frac{2c}{n} \sum_{k=1}^{n-1} (k \log k) + bn \\ &\leq \frac{2c}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + bn \quad \text{(Analysis)} \\ &= cn \log n - \frac{c}{4} n + bn \\ &\leq cn \log n, \text{ falls } c \geq 4b \end{aligned}$$

Insgesamt wählen wir

$$c = \max\{\frac{a}{2} + b, 4b\}$$

Mit diesem c und $n_0 = 2$ ist die Behauptung $T_{\text{avg}}(n) = O(n \log n)$ bewiesen. □

Diskussion zu Quick-Sort.

- Quick-Sort ist ein sehr gutes Verfahren in der Praxis und wird deshalb sehr häufig verwendet. Trotz dieses guten praktischen Verhaltens sollte man allerdings bedenken, dass es beim Einsatz von Quick-Sort zu quadratischer Laufzeit kommen kann, was nicht bei allen Anwendungen tolerabel ist.
- Quick-Sort arbeitet nicht stabil.

Anmerkung 3.4. Bei jedem rekursiven Aufruf werden die aktuellen Werte der Parameter und der lokalen Variablen auf den Stack gelegt, und nach Beendigung des rekursiven Aufrufs wieder vom Stack geholt. Wir können deshalb folgern dass der Stack für Quick-Sort im Worst-Case $\Theta(n)$ Platz benötigt. Man jedoch den benötigten Speicherplatz auf $\Theta(\log n)$ drücken, indem man an Stelle von zwei rekursiven Aufrufen, nur den durchführt, der die kleinere Teilfolge bearbeitet; die zweite Teilfolge kann dann durch einen iterativen Ansatz ohne zusätzlichen Speicheraufwand realisiert werden.

Bei der Analyse des Quick-Sort-Verfahrens fällt auf, dass die Laufzeit stark von der Struktur der Eingabe abhängt. Die Worst-Case Analyse hat dabei gezeigt, dass unsere Strategie, das Pivotelement immer vom Ende des zu sortierenden Bereichs zu nehmen, problematisch ist. Man kann sehr einfach eine Klasse von Eingabeinstanzen entwerfen, bei denen die Worst-Case Laufzeit immer erreicht wird, nämlich sortierte Folgen. Um diese Entartung zu verhindern und das Verfahren weniger anfällig gegen die Struktur der Eingabeinstanz zu machen, wurde eine Reihe von Änderungen entwickelt.

Ein einfaches Mittel besteht darin, statt eines einzelnen Elements z.B. drei Elemente (vom Anfang, der Mitte und dem Ende) der Folge zu betrachten und davon das der Größe nach mittlere (den *Median*) als Pivotelement zu wählen. Dies verringert zwar die Gefahr einer Entartung abhängig von der Eingabe, beseitigt sie aber nicht vollständig. Eine bessere Strategie werden wir im folgenden Abschnitt kennenlernen.

Randomisierter Quick-Sort. Um die Laufzeit von der Qualität der Eingabe zu entkoppeln, wurde das randomisierte Quick-Sort-Verfahren entwickelt. Hierbei wird das Pivotelement nicht mehr deterministisch an einer bestimmten Position entnommen, sondern zufällig aus dem gesamten Bereich der (Teil-)Folge gewählt. Dabei ist für jedes Element die Wahrscheinlichkeit, gewählt zu werden, gleich gross. Es ergibt sich damit ein *Erwartungswert* für die Laufzeit von

$$E[T(n)] = \Theta(n \log n)$$

wie beim Average-Case im deterministischen Fall. Wir sind jetzt nicht mehr von einer Annahme über die Eingabe abhängig, so dass bei einer Eingabe zwar weiterhin der Worst-Case eintreten kann, bei der nächsten Sortierung derselben Eingabe wird sich das Laufzeitverhalten jedoch in der Regel verbessern. Es gibt also keine in dem Sinne schlechten Eingaben mehr.

Alternativ zum Ansatz der randomisierten Auswahl des Pivotelements kann man auch weiterhin das deterministische Verfahren nutzen, wobei allerdings die Eingabefolge vor der Pivot-Auswahl zufällig permutiert wird. Dadurch hat jedes Element die gleiche Chance, als Pivotelement gewählt zu werden. Dieser Ansatz wird auch *Input-Randomisierung* genannt und führt zum gleichen Erwartungswert wie die Randomisierung der Pivot-Auswahl.

3.1.5 Heap-Sort

Die Analyse von Selection-Sort hat gezeigt, dass die Auswahl des minimalen Elementes in der noch unsortierten Teilfolge durch Durchlauf der inneren **for** Schleife zu quadratischer Laufzeit führt. Durch Beschleunigung dieser Auswahl können wir auch die Laufzeit für das Sortieren durch Auswahl verbessern. Der Algorithmus Heap-Sort folgt dem gleichen Prinzip wie Selection-Sort, nämlich dem Prinzip des Sortierens durch Auswahl, organisiert diese Auswahl jedoch geschickter. Dazu wird eine spezielle Datenstruktur, der sogenannte *Heap* (Halde, Haufen), verwendet, die die Extraktion des Maximums aus einer Menge von Werten effizient unterstützt.

Definition 3.1. Wir nennen eine Folge $H = \langle k_1, k_2, \ldots, k_n \rangle$ von Schlüsseln einen *MaxHeap*, wenn für jede Position i die folgende Heapeigenschaft erfüllt ist: falls $2i \leq n$ so gilt $k_i \geq k_{2i}$ und falls $2i + 1 \leq n$ so gilt $k_i \geq k_{2i+1}$.

Ein MinHeap ist analog definiert (kleiner-gleich Relation zwischen den Schlüsseln).

Der Zugriff auf das maximale (bzw. minimale beim MinHeap) Element ist jetzt sehr einfach möglich, da es sich immer an der ersten Stelle der Folge befindet.

Die in der Definition des Heaps geforderte Anordnung von Schlüsseln kann einfach in einem Feld realisiert werden, für die Betrachtung der Funktionsweise nutzen wir jedoch die intuitivere Veranschaulichung eines Heaps als binärer Baum (Ein binärer Baum ist ein gewurzelter Baum in welchem jeder Knoten maximal zwei Kindknoten besitzt).

Dazu tragen wir einfach die Elemente des Heaps der Reihe nach beginnend bei der Wurzel in einen leeren Baum ein und füllen jeweils die einzelnen Stufen des Baumes auf. Abbildung 3.5 veranschaulicht den durch die Folge H gegebenen Heap für

$$\begin{array}{rccccccccc} & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \\ H = & \langle & 8, & 6, & 7, & 3, & 4, & 5, & 2, & 1 & \rangle \end{array}$$

Bei der Veranschaulichung des Heaps H als Binärbaum T können wir folgende Beobachtungen machen:

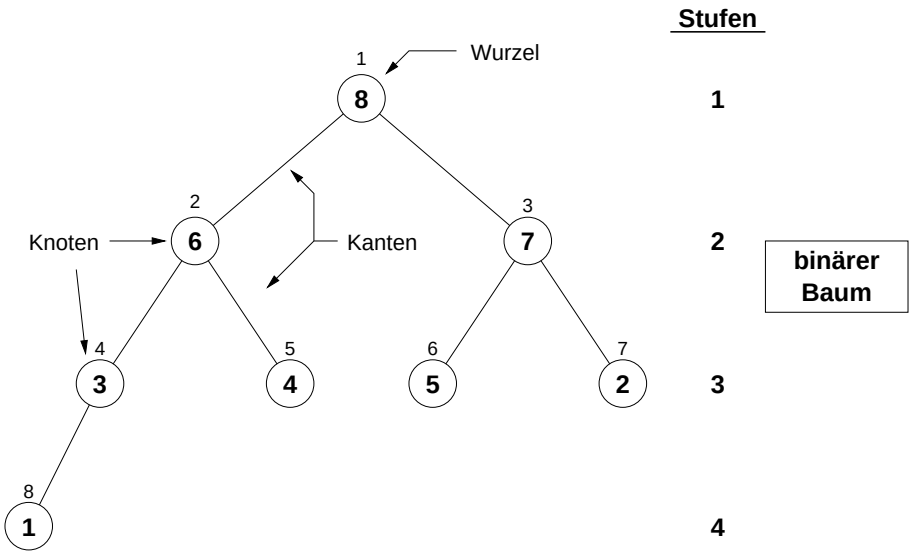


Abbildung 3.5: Der Heap H veranschaulicht als binärer Baum

- Die Schlüssel k_i entsprechen den Knoten des Binärbaums T.
- Die Paare (k_i, k_{2i}) bzw. (k_i, k_{2i+1}) entsprechen den Kanten in T, die die Elter-Kind Beziehung im Baum darstellen.
- k_{2i} in T ist linkes Kind von k_i.
- k_{2i+1} in T ist rechtes Kind von k_i.
- k_i in T ist Elter von k_{2i} und k_{2i+1}.

Ein binärer Baum ist also ein MaxHeap, wenn der Schlüssel jedes Knotens mindestens so groß ist wie die Schlüssel seiner beiden Kinder (falls diese existieren). Aus dieser Beobachtung erhält man: Das größte Element im Heap steht an der ersten Position der Folge bzw. ist die Wurzel des dazugehörigen Binärbaumes. Einen solchen, auf der Idee eines binären Baums basierenden, Heap nennt man auch *Binary Heap* (dt. *Binärer Heap*).

Die Implementierung eines Heaps geschieht am einfachsten mittels eines Feldes. Die Daten werden in der gegebenen Reihenfolge in das Feld geschrieben, der Zugriff zu den Kindern eines Elements an der Stelle i erfolgt über die Indizes $2i$ für das linke und $2i + 1$ für das rechte Kind, das Elterelement befindet sich am Index $\lfloor \frac{i}{2} \rfloor$.

Sortieren von Feldern. Wenn wir unsere Eingabedaten in einem Heap, d.h. in einem Feld mit einer die Heapeigenschaft erfüllenden Anordnung, gegeben haben, können wir sie mit dem folgenden Schema sortieren.

1. Initialisiere ein Ausgabefeld B und setze einen Index i auf die letzte Stelle in B.

2. Solange der Heap nicht leer ist

   (a) Extrahiere das erste Element (das Maximum) aus dem Heap und schreibe es an Stelle i in B.

   (b) Stelle die Heapeigenschaft in den verbliebenen Elementen wieder her, so dass das Maximum der verblieben Elemente wieder an erster Stelle steht.

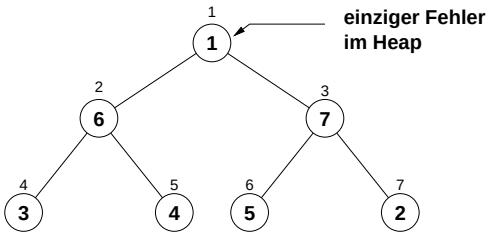
Wenn der Heap komplett abgearbeitet ist, steht in Feld B die aufsteigend sortierte Folge unserer Daten. Wir können allerdings auf die Benutzung eines Hilfsfeldes verzichten und die Ausgabe innerhalb unseres Eingabefeldes erledigen, also in situ arbeiten, wenn wir wie folgt vorgehen:

Bei der Extraktion des Maximums schreiben wir das Element nicht in ein Hilfsfeld, sondern in unser Eingabefeld. Damit das dort stehende Eingabeelement nicht überschrieben wird, tauschen wir es mit dem Maximalelement aus, schreiben es also an die erste Stelle des Feldes. Da wir damit aber die Heapeigenschaft verletzt haben können, müssen wir diese wieder herstellen, indem wir das getauschte Element an eine geeignete Position verschieben. Diese Operation, die in unserer Interpretation des Heaps einem Weg von der Wurzel des Baums nach unten entspricht, nennen wir *SiftDown* (*Versickern*). Nach der Vertauschung des größten Elements brauchen wir es nicht mehr zu betrachten und bearbeiten im Folgenden nur noch den Bereich an den Indizes 1 bis $n - 1$.

In unserem Beispiel vertauschen wir also den Wert 8 von der ersten Position im Feld mit der 1 an der letzten Position. Den größten Wert 8 in unserem Heap haben wir mit der Vertauschung ausgegeben und betrachten ab jetzt nur noch den Bereich der Positionen 1 bis 7, in denen unsere restlichen Schlüssel gespeichert sind. Wir müssen nun dafür sorgen, dass der Wert 1, der jetzt an der ersten Position steht, an eine Stelle verschoben wird, so dass das Feld an den Positionen 1 bis 7 die Heapeigenschaft wieder erfüllt.

Da wir ausser der Vertauschung von zwei Elementen nichts geändert haben, wissen wir aber, dass die zwei Teilbäume, die Kinder des Wertes an der ersten Position sind, die Heapeigenschaft noch erfüllen. Es reicht also, beim Versickern des Wertes an der Wurzel des Baumes einen Pfad für diesen Wert in Richtung der Blätter zu betrachten. Ist die Heapeigenschaft verletzt, so ist der Wert an der Wurzel kleiner als mindestens eines seiner Kinder. Um dies zu ändern, führen wir auch hier eine Vertauschung durch. Damit anschliessend die Heapeigenschaft an dieser Position garantiert erfüllt ist, vertauschen wir den Wert an der Wurzel mit dem größeren seiner Kinder, so dass nachher der Wert an der Wurzel größer oder gleich als seine Kinder ist.

Im Beispiel vertauschen wir den Wert 1 mit dem Wert 7 und haben an der Wurzel wieder die Heapeigenschaft erfüllt.



Die Heapbedingung kann natürlich weiterhin an dem Wert verletzt sein, den wir gerade von der Wurzelposition verschoben haben. Wir müssen also den Vergleich dieses Wertes mit seinen Kindern und gegebenenfalls eine Vertauschung so lange durchführen, bis der Wert an eine Position versickert ist, an der er nicht mehr kleiner als eines seiner Kinder ist. Dies kann auch eine Blattposition sein, z.B. falls wir das kleinste Element im Heap an die Wurzelposition getauscht hatten.

In unserem Beispielheap müssen wir Wert 1 auch noch mit der 5 tauschen, um die Heapbedingung wieder herzustellen. Danach haben wir für die jetzt noch betrachteten Heapelemente wieder einen korrekten Heap und können in unserem Schema fortfahren, das größte Element 7 durch Vertauschung mit dem letzten Element auszugeben.

Haben wir die Heapeigenschaft in unserem Restheap durch vollständiges Versickern des Wertes wieder hergestellt, fahren wir mit der Ausgabe des nächsten Elements fort, d.h. wir vertauschen wieder das jetzt größte Element mit dem letzten Element, versickern dieses, falls nötig, usw.

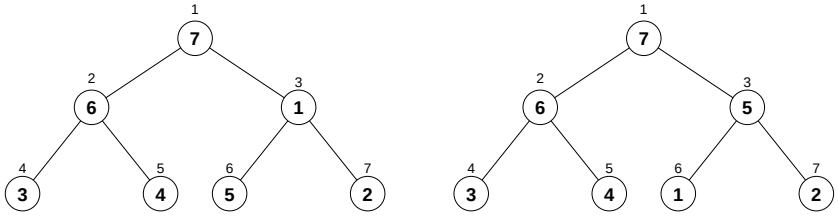


Abbildung 3.6 illustriert das aufsteigende Sortieren an unserem Beispiel. Man beachte dabei die Analogie zu Selection-Sort!

In Listing 3.6 ist der Pseudocode für Heap-Sort angeben. Wir benötigen dazu zwei Hilfsoperationen: SIFTDOWN (Listing 3.7) kümmert sich um das Versickern eines Elements wie oben beschreiben. Die Operation CREATEHEAP (Listing 3.8) wollen wir nun noch etwas genauer besprechen:

CreateHeap. Wir interpretieren die Eingabefolge wieder als Binärbaum. Wir gehen „von rechts nach links" vor und lassen jeweils Schlüssel versickern, deren beide „Unterbäume"

Eingabe: Folge A
Ausgabe: sortierte Folge A

```
 1: procedure HeapSort(ref A)
 2:     var Index i
 3:     CreateHeap(A)
 4:     for i := n, ..., 2 do
 5:         Vertausche A[1] mit A[i]
 6:         SiftDown(A, 1, i − 1)
 7:     end for
 8: end procedure
```

Listing 3.6: Heap-Sort

Eingabe: Folge A; Indexgrenzen i und m
Ausgabe: Eingangswert $A[i]$ ist bis maximal $A[m]$ versickert

```
 1: procedure SiftDown(ref A, i, m)
 2:     var Index j
 3:     while 2i ≤ m do                          ▷ A[i] hat linkes Kind
 4:         j := 2i                              ▷ A[j] ist linkes Kind
 5:         if j < m then                        ▷ A[i] hat rechtes Kind
 6:             if A[j].key < A[j + 1].key then
 7:                 j := j + 1                   ▷ A[j] ist größtes Kind
 8:             end if
 9:         end if
10:         if A[i].key < A[j].key then
11:             Vertausche A[i] mit A[j]
12:             i := j                           ▷ Weiter versickern
13:         else
14:             return                           ▷ Heap-Bedingung erfüllt
15:         end if
16:     end while
17: end procedure
```

Listing 3.7: Versickern

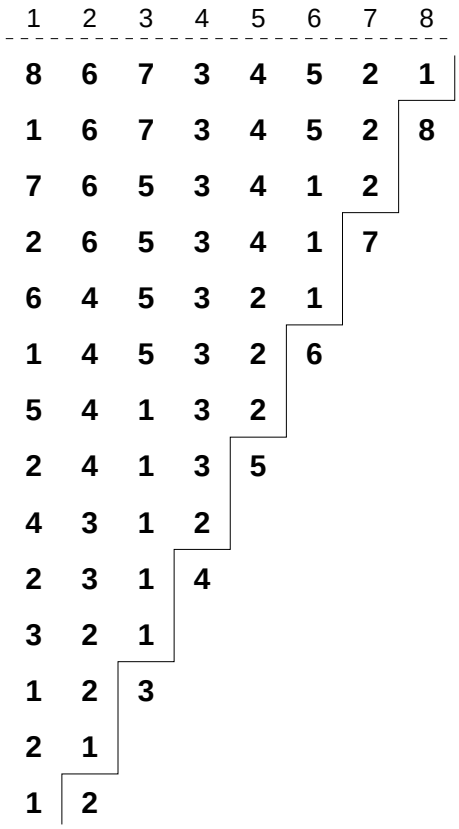


Abbildung 3.6: Beispiel für Sortieren mit Heap-Sort.

Eingabe: Folge A; maximaler Index n
Ausgabe: Heap in A

```
1: procedure CreateHeap(ref A, n)
2:     var Index i
3:     for i := ⌊n/2⌋, ..., 1 do
4:         SiftDown(A, i, n)
5:     end for
6: end procedure
```

Listing 3.8: Erstellen eines Heaps

bereits die Heapeigenschaft erfüllen. Da die Blätter des Baums keine Kinder haben, ist für sie nichts zu tun und wir beginnen beim Index $\lfloor n/2 \rfloor$. Versickern bedeutet dabei wieder, dass geprüft wird, ob das Wurzelelement eines Teilbaumes kleiner ist als eines seiner Kinder. Ist dies der Fall, so ist die Heapeigenschaft nicht erfüllt und wir vertauschen die Wurzel mit dem größeren Kind. Dies wird solange wiederholt, bis das ursprüngliche Wurzelelement

eine Position erreicht hat, an er beide Kinder nicht größer sind. Abbildung 3.7 zeigt die Konstruktion eines Heaps an einem Beispiel.

Analyse von SiftDown(A, i, m). Wir versickern ein Element in einem Heap mit Wurzel i und Maximalindex m. Die Schleife in Zeile 3 wird höchstens so oft durchlaufen, wie es der Anzahl der Stufen des Heaps entspricht. Als sehr pessimistische Abschätzung ist dies maximal die Höhe der n-elementigen Heaps, also $O(\log n)$.

Analyse von CreateHeap. Ein vollständiger Binärbaum mit j Stufen besitzt genau $2^j - 1$ Knoten, daher hat ein Heap mit n Schlüsseln genau $j = \lceil \log(n+1) \rceil$ Stufen.

Sei j die Anzahl der Stufen des Binärbaums, d.h. $2^{j-1} \leq n \leq 2^j - 1$. Auf Stufe k sind höchstens 2^{k-1} Schlüssel. Die Anzahl der Vergleiche und Bewegungen zum Versickern eines Elements von Stufe k ist im Worst-Case proportional zu $j - k$, insgesamt ist die Anzahl proportional zu

$$\begin{aligned}
\sum_{k=1}^{j-1} 2^{k-1}(j-k) &= 2^0(j-1) + 2^1(j-2) + 2^2(j-3) + \ldots + 2^{j-2} \cdot 1 \\
&= \sum_{k=1}^{j-1} k \cdot 2^{j-k-1} = 2^{j-1} \sum_{k=1}^{j-1} \frac{k}{2^k} \\
&\overset{(*)}{\leq} 2^{j-1} \cdot 2 \leq 2n = O(n).
\end{aligned}$$

(*) folgt aus $\sum_{k=1}^{\infty} \frac{k}{2^k} = 2$ (siehe beliebiges Grundlagenbuch zur Analysis). Daraus folgt, dass wir einen Heap aus einer unsortierten Folge in Linearzeit aufbauen können.

Analyse von Heap-Sort Der Algorithmus *SiftDown* wird $n-1$ Mal aufgerufen. Das ergibt insgesamt $O(n \log n)$ Vergleiche und Bewegungen im Worst-Case. Die lineare Laufzeit von CREATEHEAP ist daher nicht von Interesse, und wir erhalten:

$$C_{\text{worst}} = \Theta(n \log n) \quad \text{und} \quad M_{\text{worst}}(n) = \Theta(n \log n).$$

Die Vergleiche in Zeile 5 und 10 von SIFTDOWN werden sowohl im Worst-Case als auch im Best-Case in jedem Durchlauf der While-Schleife benötigt, unabhängig z.B. von einer Vorsortierung der Eingabefolge. Für die Analyse des Best-Case müssen wir also feststellen, wie oft die While-Schleife durchlaufen wird. Während im Worst-Case die Abbruchbedingung $2i \leq m$ erfüllt wird, nachdem i entsprechend oft verdoppelt wurde, könnten wir im

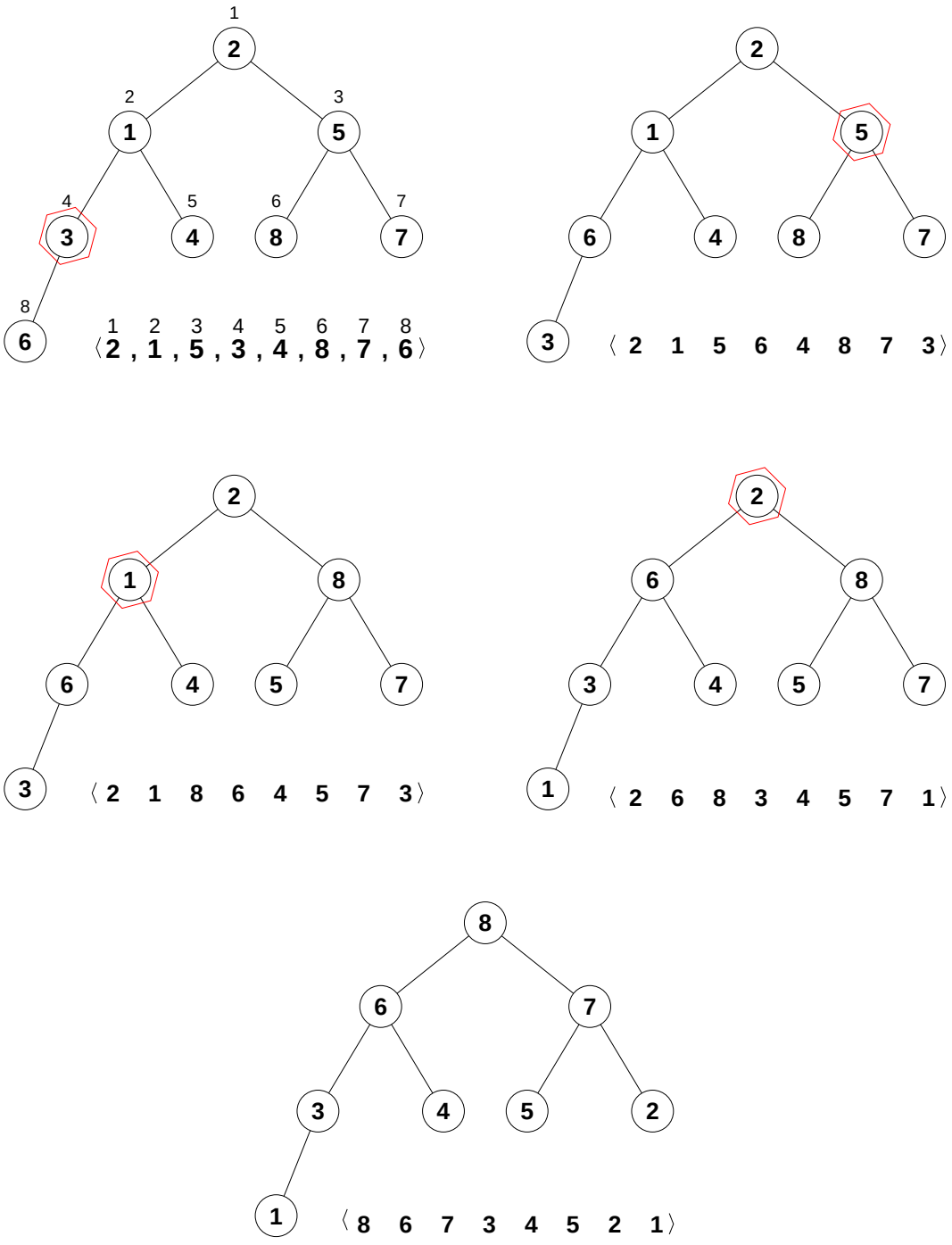


Abbildung 3.7: Konstruktion eines Heaps

Best-Case darauf hoffen, dass die Heapbedingung frühzeitig erfüllt ist und deshalb das Return in Zeile 14 ausgeführt wird. Dazu muss der Schlüssel an Stelle i größer oder gleich dem Schlüssel an Stelle j sein, welcher der größere Schlüssel der beiden Kinder ist. Das jeweils betrachtete Element wurde aber gerade erst mit der Wurzel getauscht und wir zeigen, dass nicht alle getauschten Elemente „oben“ auf dem Heap liegen bleiben können, sondern nach unten absinken müssen. Dazu betrachten wir die obere Hälfte unseres Heaps, d.h. die Elemente mit Tiefe kleiner oder gleich $\frac{\lceil \log n \rceil}{2}$ sowie die Menge U der $n/2$ größten Elemente. In der oberen Hälfte des Heaps haben maximal $2^{\frac{\lceil \log n \rceil}{2}} = \sqrt{n}$ Elemente Platz.

Wenn von den Elementen in U im initialen Heap mindestens die Hälfte, also $n/4$, keine Blätter sind, dann befinden sich mindestens $n/4 - \sqrt{n}$ unterhalb der Tiefe $\leq \frac{\lceil \log n \rceil}{2}$, ohne Blätter zu sein. Dann haben wir mindestens genauso viele kleinere Kinder, die mindestens bis Tiefe $\frac{\lceil \log n \rceil}{2}$ absinken müssen und damit ergibt sich eine Laufzeit von mindestens $(n/4 - \sqrt{n})\frac{\lceil \log n \rceil}{2} = \Omega(n \log n)$. Sind umgekehrt mindestens die Hälfte der Elemente in U Blätter, so sind deren Vorfahren (mindestens $n/8$) ebenfalls aus U und keine Blätter, womit die Laufzeit mit dem Argument von oben wieder mindestens $(n/8 - \sqrt{n})\frac{\lceil \log n \rceil}{2} = \Omega(n \log n)$ ist.

Dadurch ergibt sich:

$$\begin{array}{llll} C_{\text{worst}}(n) & = C_{\text{avg}}(n) & = C_{\text{best}}(n) & = \Theta(n \log n) \\ M_{\text{worst}}(n) & = M_{\text{avg}}(n) & = M_{\text{best}}(n) & = \Theta(n \log n) \end{array}$$

Diskussion von Heap-Sort.

- Laufzeit $\Theta(n \log n)$.
- Quicksort ist im Durchschnitt schneller als Heapsort, aber Heap-Sort benötigt nur konstant viel zusätzlichen Speicherplatz, arbeitet also in situ.
- Für das Versickern eines Elements gibt es verschiedene Variationen, die effizienter sein können als die Standardmethode.
- Da Heap-Sort auf die Elemente im Feld nicht der Reihe nach zugreift, kann es auf modernen Rechnern eine schlechtere Cache-Performance als z.B. Merge-Sort aufweisen.
- Heap-Sort arbeitet nicht stabil.

3.1.6 Exkurs: Realisierung von Priority Queues durch Heaps

Ein MinHeap erfüllt mit der effizienten Rückgabe des minimalen Elements die charakteristische Anforderung an den ADT Priority Queue. Wir beschreiben im Folgenden die Realisierung einer Priority Queue durch einen binären MinHeap. Wir gehen dabei davon aus, dass wir von vorneherein wissen wieviele Elemente maximal gleichzeitig gespeichert werden, so dass wir die Feldgröße n des Heaps zu Beginn entsprechend wählen.

Anmerkung 3.5. Für den Fall, dass der Heap dynamisch verwaltet werden muss, da die Größe unbekannt ist, kann man statt mittels eines Feldes die implizite Baumstruktur auch über entsprechende Verlinkung realisieren und muß dann für ein neues Element jeweils nur den entsprechenden Platz bereitstellen.

Unser Heap H besitzt außer dem Feld A der fixen Länge n nun eine Variable *heapSize*, in der die aktuelle Anzahl der Elemente im Heap gespeichert ist. Wir müssen für eine Priority Queue auch Verweise auf Elemente (der *PositionType* aus der Definition des ADT Priority Queue) bereitstellen, da z.B. für die effiziente Änderung der Priorität ein solcher Verweis auf das zu ändernde Element der Funktion DECREASEPRIORITY als Parameter übergeben werden muss. Wir können dafür allerdings nicht die Feld-Indexwerte selbst benutzen, da diese sich während der dynamischen Operationen auf der Priority Queue ändern können. Deshalb speichern wir in dem Feld statt der Elemente selbst nur Verweise auf die Elemente und geben diese beim Einfügen zurück. Zusätzlich merken wir uns den jeweiligen tatsächlichen Feld-Index in jedem Element, so dass wir bei Übergabe eines Elementverweises die zugehörige Position im Feld finden können. Die Indexwerte können dann in den Update-Operationen geändert werden, ohne dass die Element-Verweise ihre Gültigkeit verlieren. Dazu kapseln wir die Priorität und den zu speichernden Datensatz zusammen mit dem Index in einem *HeapElement*, ein Zeiger auf ein solches HeapElement realisiert dann den *PositionType* aus der Beschreibung des ADT Priority Queue. Wir müssen dann die Funktionen SIFTUP und SIFTDOWN nur so ergänzen, dass bei einer Vertauschung zweier Elementverweise auch die in den Elementen gespeicherten Indizes getauscht werden.

- INSERT(*PriorityType p, Value v*): Für die Insert-Operation müssen wir zunächst die Heapgröße um 1 vergrößern. Das neue Element x wird nun an das Ende des Heaps angefügt, so dass es in unserer intuitiven Vorstellung des Heaps als Baum das am weitesten rechts stehende Blatt auf der untersten Ebene ist. Da das neue Element unter Umständen eine kleinere Priorität hat als sein Elter, kann die Heapeigenschaft jetzt verletzt sein. Wir stellen die Heapeigenschaft wieder her, indem wir x solange nach oben wandern lassen, bis es entweder an der Wurzel steht oder die Priorität des Elterelements nicht größer ist als p (Listing 3.10). Dazu nutzen wir die Funktion SIFTUP$(heapSize)$ (Listing 3.13). Die Laufzeit wird im Worst-Case durch die Laufzeit von SIFTUP bestimmt, die asymptotisch durch die Höhe des Baums beschränkt ist, also $O(\log n)$. Dieser Fall tritt auf, wenn p kleiner ist als jede Priorität im Heap.

- DELETE(*HeapElement pos*): Das Löschen eines Elements verläuft ähnlich wie die Entfernung des maximalen Elements in einem MaxHeap, die bereits beim Sortieren besprochen wurde. Um ein Element zu löschen, tauschen wir es mit dem letzten Element im Heap, passen die Indexwerte an und verringern die Heapgröße um 1. Dadurch kann an Position pos die Heapeigenschaft verletzt sein, da das vertauschte Element eventuell eine größere Priorität hat als eines seiner Kinder oder eine kleinere Priorität hat als sein Elter. Beide Verletzungen der Heap-Eigenschaft können wir beheben, indem wir entweder dieses Elemente nach unten oder nach oben versickern lassen. Daher

```
 1: ▷ Repräsentation eines Heap-Elements
 2: struct HeapElement
 3:     var PriorityType priority                          ▷ Priorität des Elements
 4:     var ValueType value                                ▷ Gespeicherter Wert
 5:     var int index                  ▷ Index des gespeicherten Elements im Feld
 6: end struct
 7: ▷ Interne Repräsentation einer binären MinHeap-PQ
 8: var int heapSize
 9: var HeapElement A[1...n]
10: ▷ Initialisierung
11: heapSize := 0
12: function IsEmpty( ) : bool
13:     return heapSize = 0
14: end function

15: function Minimum( ) : HeapElement
16:     return A[1]
17: end function

18: function Priority(pos) : PriorityType
19:     return pos.priority
20: end function
```

Listing 3.9: Priority Queue-Implementierung durch einen binären MinHeap.

rufen wir für den Index $i := pos.index$ des aktuellen Elementes sowohl SIFTDOWN und SIFTUP auf, wobei nur höchstens eine der Prozeduren wirklich etwas tun wird. Der Pseudocode ist in den Listings 3.11, 3.13 und 3.14 angegeben. Der Parameter der Funktion SIFTDOWN, der angibt, wie tief ein Element maximal versickert werden kann, wird hier implizit auf *heapSize* gesetzt, da wir in einer Priority Queue Elemente immer bis zur aktuellen Größe des Heaps versickern lassen. Die Laufzeit ist damit durch die Höhe des Baums beschränkt, also $O(\log n)$.

- ISEMPTY(): Da wir über den Wert von $heapSize$ direkt die Anzahl der gespeicherten Elemente kennen, können wir hier den Vergleich von $heapSize$ mit 0 zurückgeben.

- PRIORITY(*HeapElement pos*): Der übergebene Verweis auf ein HeapElement ermöglicht es uns, direkt die dort gespeicherte Priorität zurückzugeben.

- MINIMUM(): Die Abfrage des Elements mit minimaler Priorität kann wie schon bei der Beschreibung des binären Heaps erläutert einfach durch Rückgabe eines Verweises auf das erste Element im Heap erfolgen.

- EXTRACTMIN(): Wir löschen das minimale Heapelement an Position 1 und geben das Paar aus Priorität und Wert des Elements zurück. Die Laufzeit ist hier durch die Lösch-Operation bestimmt, also $O(\log n)$.

- DECREASEPRIORITY(*HeapElement pos, PriorityType p*): Wir verringern die Priorität im Heapelement, auf das pos verweist und rufen SIFTUP$(pos.index)$ auf, um die eventuell verletzte Heapeigenschaft wieder herzustellen. Da wir ein Element im Worst-Case von einem Blatt bis zur Wurzel verschieben müssen, ist die Laufzeit $O(\log n)$.

Eingabe: Priorität p, Wert v
Ausgabe: Verweis auf neues Heapelement

```
1:  function INSERT(p, v) : HeapElement
2:      var HeapElement x
3:      heapSize := heapSize + 1
4:      x := new HeapElement
5:      x.priority := p
6:      x.value := v
7:      x.index := heapSize
8:      A[heapSize] := x
9:      SIFTUP(heapSize)
10:     return x
11: end function
```

Listing 3.10: Einfügen eines Elements in eine Heap-Priority Queue

Eingabe: Löschposition pos

```
1:  procedure DELETE(pos)
2:      A[heapSize].index := pos.index
3:      Tausche A[heapSize] und A[pos.index]
4:      heapSize := heapSize − 1
5:      i := pos.index
6:      SIFTUP(i)
7:      SIFTDOWN(i)
8:      delete pos
9:  end procedure
```

Listing 3.11: Löschen eines Elements in einer Heap-Priority Queue

Eingabe: Position pos, neue Priorität p

```
1: procedure DecreasePriority(pos, p)
2:     pos.priority := p
3:     SiftUp(pos.index)
4: end procedure
```

Listing 3.12: Verkleinern der Priorität eines Elements in einer Heap-Priority Queue

Eingabe: Feld-Index i

```
1: procedure SiftUp(i)
2:     var int parent
3:     parent := ⌊i/2⌋
4:     while parent > 0 do
5:         if A[i].priority < A[parent].priority then
6:             A[parent].index := i                    ▷ Indizes anpassen
7:             A[i].index := parent
8:             Vertausche A[i] mit A[parent]           ▷ Weiterlaufen
9:             i := parent
10:            parent := ⌊i/2⌋
11:        else
12:            return
13:        end if
14:    end while
15: end procedure
```

Listing 3.13: SiftUp in einer Heap-Priority Queue

Eingabe: Feld-Index i

```
 1: procedure SiftDown(i)
 2:     var Index j
 3:     while 2i ≤ heapSize do                          ▷ A[i] hat linkes Kind
 4:         j := 2i                                     ▷ A[j] ist linkes Kind
 5:         if j < heapSize then                        ▷ A[i] hat rechtes Kind
 6:             if A[j].priority > A[j + 1].priority then
 7:                 j := j + 1                          ▷ A[j] ist kleinstes Kind
 8:             end if
 9:         end if
10:         if A[i].priority > A[j].priority then
11:             A[j].index := i
12:             A[i].index := j
13:             Vertausche A[i] mit A[j]
14:             i := j                                  ▷ Weiter versickern
15:         else
16:             return                                  ▷ Heap-Bedingung erfüllt
17:         end if
18:     end while
19: end procedure
```

Listing 3.14: Versickern in einer Heap-Priority Queue

3.1.7 Eine untere Laufzeit-Schranke für allgemeine Sortierverfahren

Wir haben nun verschiedene Algorithmen zum Sortieren von n Objekten kennengelernt, bei denen die Worst-Case Laufzeit zwischen $O(n \log n)$ und $O(n^2)$ lag. Nun stellt sich die Frage, ob man auch in Worst-Case Zeit $O(n)$ sortieren kann.

Wieviele Schritte mindestens zum Sortieren von n Datensätzen benötigt werden, hängt davon ab, was wir über die zu sortierenden Datensätze wissen, und welche Operationen wir zulassen. Wir betrachten wie bisher das folgende Szenario für allgemeine Sortierverfahren.

> Wir haben kein Wissen über die Datensätze, ausser dass alle Schlüssel verschieden sind. Zur Bestimmung einer Sortierung sind lediglich *elementare Vergleiche* zwischen den Schlüsseln erlaubt, d.h. die Frage „Gilt $a_i \leq a_j$? “.

Um obige Frage zu beantworten, führen wir einen *Entscheidungsbaum* ein. Die Knoten des Baumes entsprechen einem Vergleich zwischen a_i und a_j (siehe Abbildung 3.8). Die Kinder repräsentieren dann einen positiven bzw. negativen Ausgang des Vergleichs und schränken damit die möglichen Sortierreihenfolgen ein. Die Blätter entsprechen dann jeweils einer der Permutationen der Folge der zu sortierenden Elemente.

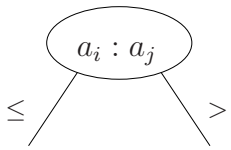


Abbildung 3.8: Ein elementarer Vergleich a_i mit a_j im Entscheidungsbaum

Abbildung 3.9 zeigt als Beispiel den Entscheidungsbaum des Insertion-Sort Algorithmus für $\langle a_1, a_2, a_3 \rangle$.

Die Anzahl der Schlüsselvergleiche im Worst-Case C_{worst} entspricht genau der Anzahl der Knoten auf dem längsten Pfad von der Wurzel bis zu einem Blatt minus 1, also der *Tiefe* des Baums. Um die Frage nach der minimal möglichen Schrittanzhl in einem Worst-Case Szenario zu beantworten, suchen wir also nach einer unteren Schranke für die Höhe eines Entscheidungsbaums.

Lemma 3.1. *Jeder Entscheidungsbaum für die Sortierung von n paarweise verschiedenen Schlüsseln hat die Tiefe $\Omega(n \log n)$.*

Beweis. Wir betrachten einen Entscheidungsbaum der Tiefe t, der n disjunkte Schlüssel sortiert. Der Baum hat mindestens $n!$ Blätter, die ja alle möglichen Permutationen darstellen müssen. Ein Binärbaum der Tiefe t hat höchstens 2^t Blätter, also

$$2^t \geq n! \quad \Longleftrightarrow \quad t \geq \log(n!).$$

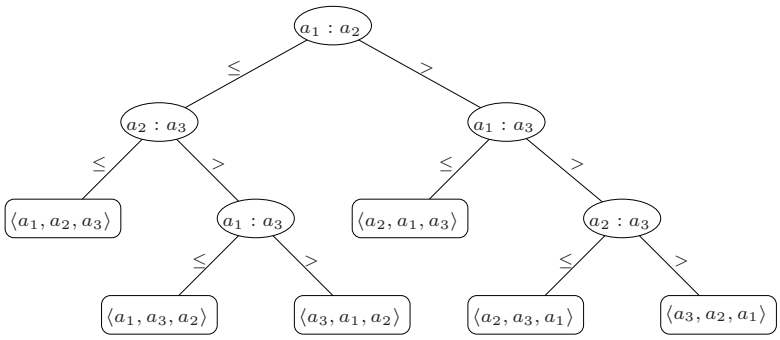


Abbildung 3.9: Entscheidungsbaum für Insertion-Sort von drei Elementen

Damit gilt

$$t \geq \log(n!) = \sum_{i=1}^{n} \log i \geq \frac{n}{2} \log \frac{n}{2} = \frac{n}{2}(\log n - 1) = \Omega(n \log n). \quad \square$$

Aus diesem Lemma folgt aber direkt:

Theorem 3.1. *Jedes allgemeine Sortierverfahren benötigt zum Sortieren von n paarweise verschiedenen Schlüsseln mindestens $\Omega(n \log n)$ im Worst-Case.*

Heap-Sort und Merge-Sort sind also asymptotisch zeitoptimale Sortieralgorithmen!

3.2 Lineare Sortierverfahren

Während allgemeine Sortierverfahren kein Vorwissen über die zu sortierenden Daten voraussetzen, hat man in der Praxis oft Informationen über die Art oder den Wertebereich der Schlüsselwerte. Zum Beispiel wird man häufig nur Zahlen aus einem bestimmten Bereich oder Zeichen aus einem endlichen Alphabet sortieren müssen. Diese Informationen kann man nutzen, um die untere Schranke für die Laufzeit der allgemeinen Sortierverfahren zu umgehen und auch Sortierverfahren mit linearer Laufzeit zu ermöglichen. Im Folgenden werden einige davon vorgestellt.

3.2.1 Bucket-Sort

Bucket-Sort ist ein Sortierverfahren, das eingesetzt werden kann, wenn die Schlüssel der Eingabeelemente aus einem endlichen Alphabet der Länge k kommen, z.B. ganze Zahlen

aus dem Bereich $[0, .., k-1]$ sind. Das Verfahren wechselt zwischen *Verteilungsphase* und *Sammelphase*.

Die grundlegende Idee der Verteilungsphase ist, dass die Eingabeelemente in k *Buckets* (Fächer) verteilt werden, so dass das i-te Fach F_i alle Elemente enthält, deren Schlüssel den Wert i haben. Das jeweils nächste Element wird stets nach den in seinem Fach bereits vorhandenen Elementen gereiht. Dies kann z.B. dadurch erreicht werden, dass die Elemente in jedem Fach in einer Queue gespeichert werden, und das nächste Element dann an diese Queue angehängt wird.

Im zweiten Schritt, der Sammelphase, sammelt man die Schlüssel der Reihe nach aus den Buckets wieder auf, wobei die Buckets in aufsteigender Indexreihenfolge durchlaufen werden. In der Sammelphase werden die Elemente in den Fächern $F_0, .., F_{k-1}$ so eingesammelt, dass die Elemente im Fach F_{i+1} als Ganzes hinter die Sätze im Fach F_i kommen. Dafür werden die Fächer nach aufsteigenden Werten abgearbeitet und die Queues geleert.

Durch das Anhängen des jeweils nächsten betrachteten Elements an eine Queue in der Verteilungsphase wird auch die Eingabereihenfolge von Elementen mit gleichem Schlüssel erhalten, d.h. BucketSort sortiert stabil.

Die Laufzeit von Bucket-Sort ergibt sich wie folgt: Zunächst erfolgt die Verteilung durch Ablauf aller Eingabeelemente, also mit Aufwand $\Theta(n)$. Dann werden die Buckets zum Aufsammeln durchlaufen, was Aufwand $\Theta(k)$ erfordert. Dabei werden auch die Queues geleert, die die n Elemente speichern, womit dieser Schritt Aufwand $\Theta(n+k)$ hat. Insgesamt hat Bucket-Sort also eine Laufzeit in $\Theta(n+k)$.

Wie man leicht sieht, ist der Aufwand unabhängig von der Sortierung der Eingabe, womit Worst-Case, Average-Case und Best-Case zusammenfallen.

Beispiel Bucket-Sort Die Folge (C, B', C', A, B) mit dem Alphabet $\{A, B, C\}$, $m = 3$, soll sortiert werden (Die Anführungszeichen sind nur zur Unterscheidung von Elementen mit denselben Werten angefügt). Verteilungsphase:

Fach	Datensätze
F_0	$\mathbf{A}$
F_1	$\mathbf{B}', \mathbf{B}$
F_2	$\mathbf{C}, \mathbf{C}'$

Sammelphase: Durchlaufen der Fächer F_0, F_1 und F_2 mit dem Ergebnis (A, B', B, C, C')

Diskussion von Bucket-Sort.

- Bucket-Sort sortiert stabil.
- Da die Effizienz von Bucket-Sort von der Anzahl der möglichen Werte k abhängig ist, lohnt sich dieses Verfahren nur, wenn k nicht allzu groß ist.

Eingabe: Zu sortierende Zahlenfolge in Feld A, Feld $Bucket$ von Listen
Ausgabe: sortierte Folge in Feld A

```
 1: procedure BUCKETSORT(ref A)
 2:     var Liste Bucket[0..k − 1]
 3:     Initialisiere Bucket
 4:     for i := 1, .., n do                                  ▷ Verteilen
 5:         Bucket[A[i].key].put(A[i])
 6:     end for
 7:     i=1
 8:     for j := 1, .., k − 1 do                              ▷ Sammeln
 9:         while not Bucket[j].isEmpty() do
10:             A[i] := Bucket[j].get()
11:             i := i + 1
12:         end while
13:     end for
14: end procedure
```

Listing 3.15: Bucket-Sort

- Bucket-Sort ist einfach zu implementieren.
- Bucket-Sort kann auch zum Sortieren genutzt werden, wenn die Eingabeelemente (annähernd) uniform über das Intervall $[0, .., 1)$ verteilt sind (z.B. bei Fließkommazahlen). Dann wird das Intervall einfach in n gleichgroße Buckets aufgeteilt. Die entstehenden Listen sind aufgrund der Verteilung hoffentlich relativ klein und werden dann z.B. mit Insertion-Sort sortiert.

3.2.2 Counting-Sort

Counting-Sort geht davon aus, dass die zu sortierenden Elemente ganze Zahlen aus dem Intervall $[0..k-1]$ sind. Für k in $O(n)$ erfolgt die Sortierung dann in $\Theta(n)$ Laufzeit. Die grundlegende Idee bei Counting-Sort ist es, für jedes Element x die Anzahl der Elemente zu ermitteln, die kleiner sind als x, und damit direkt die Position von x zu bestimmen. Aufpassen muss man dabei auf den Fall, dass einige Elemente gleich gross sind, damit diese nicht an dieselbe Position gesetzt werden. Counting-Sort erzeugt eine sortierte Version der Eingabe aus Feld A in einem Resultatsfeld B.

Der Pseudocode von Counting-Sort ist in Abbildung 3.16 dargestellt. Der Algorithmus geht wie folgt vor: In der ersten **for**-Schleife wird das temporäre Zählfeld C auf den Wert 0 initialisiert. Dann wird in der zweiten Schleife jedes Eingabeelement angeschaut und für seinen Wert l der Eintrag $C[l]$ um eins erhöht. Dadurch enthält C nach der Schleife an Index l die

Anzahl der Eingabeelemente mit Wert l. Um die Anzahl der Elemente, die kleiner oder gleich einem Wert l sind, festzustellen, werden dann in der dritten Schleife einfach die Werte in C sukzessive aufsummiert. Damit haben wir im Prinzip schon für jedes Element $A[j] = i$ sein Position festgelegt, es kommt an die Stelle $C[A[j]]$, die dieser Anzahl entspricht. Wir laufen also in der letzten Schleife rückwärts über das Feld A und schreiben den Wert $A[j]$ in das Ergebnisfeld B an die Stelle $C[A[j]]$. Da wir auch gleiche Werte in der Eingabe zulassen, verringern wir jedesmal die Anzahl $C[A[j]]$ um eins, so dass das nächste Eingabeelement, das denselben Wert hat wie $A[j]$, eine Position vor dem Element aus $A[j]$ in B eingetragen wird. Dies hat den Nebeneffekt, dass Counting-Sort stabil sortiert, d.h., Elemente mit gleichen Schlüsseln werden in der Ausgabe so angeordnet, wie sie in der Eingabe vorkommen.

Eingabe: zu sortierende Zahlenfolge in Feld $A[1...n]$ mit Elementen aus dem Bereich $[0..k-1]$
Ausgabe: sortierte Zahlenfolge in Feld A

```
 1: procedure CountingSort(ref A, int k)
 2:     var Zahl C[0...k − 1]
 3:     var Hilfsfolge B[1...n]
 4:     for l := 0, .., k − 1 do                          ▷ Initialisiere Zählfeld
 5:         C[l] := 0
 6:     end for
 7:     for j := 1, .., n do
 8:         C[A[j]] := C[A[j]] + 1                ▷ C[j] ist die Anzahl der l mit A[l] = j
 9:     end for
10:     for l := 1, .., k − 1 do
11:         C[l] := C[l] + C[l − 1]                ▷ C[l] ist Anzahl der l mit A[l] ≤ l
12:     end for
13:     for j := n, .., 1 do
14:         B[C[A[j]]] := A[j]
15:         C[A[j]] := C[A[j]] − 1
16:     end for
17:     Schreibe sortierte Folge zurück von B nach A
18: end procedure
```

Listing 3.16: Counting-Sort

Analyse von Counting-Sort. Die Laufzeit von Counting-Sort lässt sich durch eine einfache Betrachtung des Pseudocode ableiten. Die erste Schleife wird genau k mal durchlaufen, die zweite Schleife n mal, die dritte Schleife wieder k mal und die letzte Schleife n mal. Dabei werden jeweils nur einfache Anweisungen bzw. Operationen ausgeführt. Unsere Laufzeit ist also durch das Maximum von n und k bestimmt, in O-Notation erhalten wir $\Theta(n + k)$.

Für den Fall $k = O(n)$ erhalten wir eine Laufzeit von $\Theta(n)$, womit Counting-Sort dann in der Praxis als lineares Sortierverfahren angewandt werden kann.

Diskussion von Counting-Sort.

- Counting-Sort ist ein stabiles Sortierverfahren
- Counting-Sort benötigt zusätzlichen Platz der Größe $\Theta(k)$
- Counting-Sort kann aufgrund seiner Eigenschaften als Subroutine in anderen Sortierverfahren, wie z.B. RadixSort (siehe Abschnitt 3.2.3), eingesetzt werden.
- Während Bucket-Sort auch bei kontinuierlichen Werten (entsprechende Verteilung der Werte vorausgesetzt) eingesetzt werden kann, ist Counting-Sort nur im Spezialfall ganzer Zahlen sinnvoll einsetzbar.

3.2.3 Radix-Sort

Radix-Sort kann als Sortierverfahren angewandt werden, wenn die Eingabeschlüssel Zahlen im Bereich $\left[0..k^d - 1\right]$ sind. Man fasst die Zahlen dann als d-stellige Zahlen zur Basis k auf und sortiert die Stellen jeweils einzeln mit einem stabilen Sortierverfahren, beginnend mit der niedrigstwertigen Stelle. Beispielsweise ist $k = 10$, wenn die Schlüssel Dezimalzahlen sind, und $k = 26$, wenn die Schlüssel Buchstaben aus dem lateinischen Alphabet sind. Der Pseudocode zu Radix-Sort ist sehr einfach aufgebaut, siehe Alg. 3.17.

Eingabe: zu sortierende Zahlenfolge in Feld A, Stellenzahl d
Ausgabe: sortierte Zahlenfolge

```
1: procedure RADIXSORT(ref A, int d)
2:     for i := 0, .., d − 1 do                    ▷ Für alle Stellen
3:         Benutze stabiles Sortierverfahren um A nach Stelle i zu sortieren
4:     end for
5: end procedure
```

Listing 3.17: RadixSort: Stellenweises Sortieren.

Auf den ersten Blick erscheint es ungewöhnlich, nach der niedrigstwertigen Ziffer zuerst zu sortieren, aber die Sortierung in den ersten Runden wird durch die Benutzung eines stabilen Sortierverfahrens in den nachfolgenden Sortierrunden jeweils stabil gehalten und die abschliessende Sortierung erfolgt dann nach der höchstwertigen Ziffer jedes Eingabewertes, wodurch das Ergebnis dann eine korrekte Sortierung ergibt.

Laufzeitanalyse von Radixsort. Die Laufzeit von Radix-Sort ist natürlich abhängig vom verwendeten stabilen Sortierverfahren. Bei der oben genannten Interpretation der Eingabewerte ist jede Stelle eine Ziffer aus dem Wertebereich $[0..k-1]$, womit z.B. CountingSort eingesetzt werden kann. Jeder Durchlauf über eine Stelle der n d-stelligen Zahlen benötigt dann $\Theta(n+k)$, bei d Durchläufen erhalten wir somit eine Gesamtlaufzeit von $\Theta(d(n+k))$. Hat man eine feste Maximalzahl von Stellen, also konstantes d und einen Ziffernbereich $k = O(n)$, dann sortiert RadixSort in Linearzeit.

Welche genauen Eigenschaften Radix-Sort, z.B. auch bzgl. des Speicherbedarfs, hat, hängt von dem verwendeten stabilen Sortierverfahren ab.

Beispiel Radix-Sort.

Eingabe	Erste Runde Sortiere Einser stabil	Zweite Runde Sortiere Zehner stabil	Dritte Runde Sortiere Hunderter stabil
237	422	512	119
422	512	213	213
427	213	119	237
512	237	422	422
119	427	427	427
213	119	237	512

3.3 Externe Sortierverfahren

Wir gehen in dieser Vorlesung nicht im Detail auf externe Sortierverfahren ein, wollen aber dennoch die Problematik kurz erläutern und ein Verfahren beispielhaft vorstellen. Die klassischen Sortierverfahren wie Insertion-Sort etc. berücksichtigen eventuell problematische Aspekte des Speicherzugriffs überhaupt nicht. Ihr Einsatz in der Praxis, also eine Implementierung auf konkreter Hardware, erfordert die implizite Annahme, dass alle Daten gleichzeitig in den Hauptspeicher passen.

In den letzten Jahren stieg die Menge an Daten, die verarbeitet werden müssen, rasant an. Daher und zusätzlich haben sich eine Vielzahl von Techniken etabliert, die schnellen Speicherzugriff erlauben, ohne dass der gesamte Speicherbereich aus sehr schnellen und damit teuren Speicherbausteinen bestehen muss; ein Beispiel dafür sind die Caching-Strategien in modernen PCs. Konkrete Hardwarearchitekturen bauen also auf einer Speicherhierarchie auf, die von Prozessorregistern über First-, Second-, ggf. Third-Level Cache, und Hauptspeicher bis zur Festplatte und noch langsameren (z.B. optischen) Massenspeichern reicht. Daten werden dann jeweils in *Blöcken* einer festen Größe aus nachgeordneten Speicherebenen geladen. Da die Zugriffszeiten auf die Speicherebenen jeweils um Grössenordnungen auseinanderliegen können, müssen die Zugriffe möglichst effizient gestaltet werden. Wir können also nicht

mehr wie bisher davon ausgehen, dass z.B. der Zugriff auf eine Position in einem Feld in konstanter Zeit möglich ist und müssen versuchen, die Anzahl der Externspeicher-Zugriffe (*I/Os*) so gering wie möglich zu halten. Ein geladener Block sollte deshalb möglichst viele nützliche Daten enthalten (*örtliche Lokalität*), also z.B. Feldelemente, die hintereinander verarbeitet werden, und es sollen möglichst wenige Blöcke geladen werden müssen.

Beim Entwurf und der Analyse von externen Sortierverfahren geht man nun vom abstrakten RAM Maschinenmodell ab und berücksichtigt stattdessen die unterschiedliche Wertigkeit von externen und internen Speicherzugriffen in einem hierarchischen Speichermodell. Die Laufzeiten dieser Algorithmen werden deshalb nicht nur abhängig von der Eingabegröße n angegeben, sondern hängen auch von Hardwareparametern ab wie:

- Hauptspeichergröße M
- Speicher-Blockgröße B

sowie der Anzahl an verfügbaren Platten usw.

Die Analyse und Verbesserung externer Verfahren ist ein Gebiet der aktuellen Forschung; ein Ziel ist es z.B. aufgrund der Heterogenität der heutigen Hardware, auf möglichst verschiedenen Architekturen eine gute Laufzeit zu erreichen. Ein Ansatz für externe Sortierverfahren ist es, das externe Sortieren auf das interne Sortieren zurückzuführen, indem man die Daten zunächst in kleinere Teile aufteilt, die klein genug sind, um intern sortiert zu werden. Diese Teile müssen dann noch kombiniert werden. Wir haben bereits einen Algorithmus kennengelernt, der das Sortieren während der Zusammenführung von Teilmengen durchführt, nämlich Merge-Sort. Wir beschreiben als Beispiel für externe Verfahren im Folgenden den externen Merge-Sort Algorithmus, ein Standardverfahren zum externen Sortieren.

3.3.1 Algorithmische Aspekte

Die Idee beim externen Merge-Sort ist es, die Eingabe zunächst in m Teilmengen aufzuteilen, welche dann intern sortiert werden können. Damit hat man nach dieser Phase (*Run-Formation-Phase*) kleinere, sortierte Teilmengen der Eingabe, die sogenannten *Runs*. Diese Runs werden bei der Erzeugung in den externen Speicher ausgelagert. In einem weiteren Schritt werden die Runs dann wieder zur Gesamtmenge zusammengefügt (*Merging-Phase*), indem Blöcke der Runs in den Hauptspeicher geladen werden und deren Inhalt in die sortierte Ausgabereihenfolge eingefügt werden.

Wir betrachten zunächst die I/O-Komplexität von normalem Merge-Sort. Das interne Sortieren verursacht keine I/O-Operationen, das Einlesen und wieder Herausschreiben in der Run-Formation-Phase kann mit $\Theta(n/B)$ I/Os erfolgen. Für die Merge-Phase haben wir:

- Verschmelzen der Teilfolgen S_1 und S_2: $O(1 + (|S_1| + |S_2|)/B)$ I/Os.

- Auf jeder Rekursionsstufe: $O(n + n/B)$ I/Os.
- Auf allen Stufen zusammen: $O((n + n/B) \log n)$ I/Os.

Wir können dieses Verhalten allerdings einfach verbessern:

1. Wir verhindern in der Run-Formation-Phase 1-elementige Mengen. Dazu beenden wir die Aufteilung bereits bei Teilmengen der Länge M, da wir wissen, dass diese komplett in den Speicher passen. Wir laden dann n/M Teile nacheinander in den Hauptspeicher und sortieren sie dort. Danach können wir die sortierte Teilfolge in den externen Speicher zurückschreiben. Der Aufwand ist dann wie folgt:
   - Wir brauchen M/B I/Os, um eine Folge hin und her zu kopieren, zusammen also $O((n/M)(M/B + 1)) = O(n/B)$ I/Os, das Sortieren selbst benötigt keine I/Os.
   - Auf jeder Stufe brauchen wir $O(n/M + n/B)$ I/Os.
   - Da wir jetzt nicht mehr $\log n$ Stufen haben, sondern nur noch $\log n/M$, ergibt sich auf allen Stufen zusammen: $O((n/M + n/B) \log(n/M))$ I/Os. Dies ist gleich
   $$O(n/B(1 + \log(n/B)))$$
   da $n/M \leq n/B$.
2. Wir verschmelzen statt zwei jeweils $p = M/(2B)$ Runs (***Multiway-Merging***). Dazu kopieren wir zunächst die jeweils kleinsten Elemente $x_1, \ldots, x_p$ der Runs $S_1, \ldots, S_p$ in den Hauptspeicher (Blockzugriff) und kopieren dann das Minimum x_i dieser Elemente in den Run, der herausgeschrieben wird. Dann liest man das nächste Element von S_i ein usw. Wir haben damit für das Verschmelzen von Teilfolgen $S_1, \ldots, S_p$ mindestens Kosten p und $(|S_1| + \ldots + |S_p|)/B$ Blöcke, und damit:
   - Auf jeder Stufe $O(n/M + n/B)$ I/Os.
   - Auf allen Stufen zusammen: $O((n/M+n/B) \log_{M/B}(n/B))$ I/Os. Dies ist gleich
   $$O(n/B(1 + \log_{M/B}(n/B))$$

Nun betrachten wir die interne Laufzeit des Algorithmus mit unseren Verbesserungen. Wir können das Minimum der Elemente aus den verschiedenen Runs effizient ermitteln, indem wir die jeweils kleinsten Elemente $x_1, \ldots, x_p$ (mit ihren Blockzugriffspartnern) in einer Priority Queue im Hauptspeicher halten. Damit haben wir für die benötigten Operationen DELETEMIN und INSERT Laufzeit $\log p$. Der Speicherplatz hierfür ist $Bp = BM/(2B) = M/2$.

Für die Gesamtlaufzeit mit unseren Verbesserungen ergibt sich nun

- In Phase 1 sortieren wir n/M Stücke der Länge M:
$$O((n/M)(M \log M)) = O(n \log M).$$

- Auf jeder Stufe verschmelzen wir n Elemente und suchen jeweils das Minimum:

$$O(n \log p).$$

- Die Tiefe des Baums ist $O(\log_p(n/B))$.

Insgesamt haben wir damit

$$O((n \log M) + (n \log p) \log_p(n/B)) =$$
$$= O((n \log M) + (n \log(M/B)) \log_{M/B}(N/B)) =$$
$$= O(n \log n).$$

Theorem 3.2. *Eine Menge von n Elementen kann mit Hilfe von Multiway Merging in interner Zeit $O(n \log n)$ mit $O(n/B(1 + \log_{M/B}(n/B)))$ I/Os sortiert werden.*

Anmerkung 3.6. Man kann zeigen, dass dies die bestmögliche Komplexität ist.

Kapitel 4

Suchen

Das Suchen in Datenmengen ist eine der grundlegenden Aufgaben, die heutzutage mit dem Computer gelöst werden, und damit auch ein wichtiges Thema der Algorithmentheorie.

Die Suche nach Stichwörtern im Internet oder die Suche nach Datensätzen mit bestimmten Eigenschaften in einer Datenbank sind bekannte Beispiele dafür. Auch beim Suchen sind wie beim Sortieren die betrachteten Objekte durch einen (Such-)Schlüssel aus einer linear geordneten Menge (dem *Universum*) U charakterisiert. Diese Schlüssel können z.B. bestimmte Attribute der Objekte sein, etwa die Versicherungsnummer von Personen in einer Kundendatenbank. Wir beschreiben zunächst einige elementare Suchalgorithmen, die auf sequentiell gespeicherten Elementfolgen (Listen und Feldern) arbeiten, und stellen dann dynamische Datenstrukturen vor, die die Suche in den von ihnen verwalteten Elementen effizient unterstützen. Diese Datenstrukturen werden zur Verwaltung von *dynamischen Datenmengen*, d.h. Datenmengen, deren Inhalt sich durch Einfügen oder Entfernen von Elementen ändern kann, benutzt, für die wir bereits den abstrakten Datentyp Dictionary kennengelernt haben.

Bei der Betrachtung der verschiedenen Suchmethoden gehen wir wie beim Sortieren davon aus, dass kein Schlüssel mehrfach in der Eingabe vorkommt. Nach diesen vergleichsorientierten Suchverfahren werden in Kapitel 5 Suchverfahren behandelt, bei denen zusätzlich arithmetische Operationen erlaubt sind.

4.1 Suchen in sequentiell gespeicherten Folgen

Wir nehmen in diesem Abschnitt an, dass die Datenelemente in einem Feld $A[1], \ldots, A[n]$ gegeben sind und wir in A ein Element mit Schlüssel $s \in U$ suchen.

Wir betrachten drei Suchverfahren: ein einfaches lineares Verfahren, die binäre Suche sowie die geometrische Suche.

4.1.1 Lineare Suche

Die lineare Suche setzt die einfachste Suchidee um: Man durchläuft alle Elemente des Feldes von vorne nach hinten und vergleicht jeden Schlüssel mit dem Suchschlüssel, bis der Schlüssel gefunden wird.

Analyse.

- Best-Case (sofortiger Treffer): $C_{\min}(n) = 1 = O(1)$
- Worst-Case (erfolglose Suche): $C_{\max}(n) = n = O(n)$
- Average-Case (für erfolgreiche Suche unter der Annahme, dass jede Anordnung gleich wahrscheinlich ist):

$$C_{\text{avg}}(n) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2} = O(n)$$

Diskussion.

- Diese einfache Methode eignet sich auch für einfach verkettete Listen.
- Lineare Suche ist nur für kleine n empfehlenswert.
- Lineare Suche ist bei unsortierten Daten die einzige Möglichkeit.

4.1.2 Binäre Suche

Wenn in einer Folge häufig gesucht wird, ohne dass sie sich ständig ändert, lohnt es sich, die Elemente bereits sortiert zu verwalten und dann in der sortierten Folge zu suchen, was den Suchaufwand erheblich verringern kann. Die Idee der binären Suche ist, den Suchbereich in jedem Schritt zu halbieren, indem das gesuchte Element mit dem Element an der mittleren Position in der sortierten Folge verglichen wird. Ist das gesuchte Element kleiner, so arbeitet man auf der linken Hälfte weiter, ist es größer, auf der rechten Seite. Bei Gleichheit hat man das Objekt gefunden. Diese Methode funktioniert nur, wenn man einen indexbasierten Zugriff auf die Elemente der Folge hat, also nicht auf Listen.

Schema. In einem ersten Schritt wird die Folge sortiert; jetzt gilt:

$$A[1].key \leq A[2].key \leq \ldots \leq A[n].key$$

Nun folgt man der *Divide and Conquer Methode*: Vergleiche den Suchschlüssel s mit dem Schlüssel des Elementes in der Mitte. Falls

Eingabe: Feld $A[1..n]$; Schlüssel s
Ausgabe: Position von s oder 0, falls s nicht vorhanden

```
 1: procedure BINARYSEARCH(A, s)
 2:     var Index m, l = 1, r = n
 3:     repeat
 4:         m := ⌊(l + r)/2⌋
 5:         if s < A[m].key then
 6:             r := m − 1
 7:         else
 8:             l := m + 1
 9:         end if
10:     until s = A[m].key ∨ l > r
11:     if s = A[m].key then
12:         return m
13:     else
14:         return 0
15:     end if
16: end procedure
```

Listing 4.1: Binäre Suche

- s gefunden: Stop, Rückgabe des Elements
- s ist kleiner: Durchsuche Elemente „links der Mitte“
- s ist größer: Durchsuche Elemente „rechts der Mitte“

Listing 4.1 zeigt eine nicht-rekursive Implementierung der binären Suche.

Analyse. Um die Analyse zu vereinfachen nehmen wir an, dass $n = 2^k - 1$ gilt für ein geeignetes k. Wir zählen die Anzahl der Vergleiche bei Zeile 3 und Zeile 9 in Algorithmus 4.1.

- Best-Case (d.h. sofortiger Treffer): $C_{\min}(n) = \Theta(1)$.
- Worst-Case: Wie lange dauert es, das Element an Position $i \in \{1, 2, \ldots, n\}$ zu finden? Bei erfolgloser Suche müssen wir den Suchbereich so lange halbieren, bis er nur noch ein Element enthält, d.h. wir müssen logarithmisch oft halbieren. Im schlechtesten Fall für die erfolgreiche Suche ist das gesuchte Element auch das letzte Element im Suchbereich, d.h. wir müssen ebenso oft halbieren.

  Sowohl für erfolgreiches als auch für erfolgloses Suchen gilt damit:
  $C_{\max}(n) = \log(n + 1) = \Theta(\log n)$.

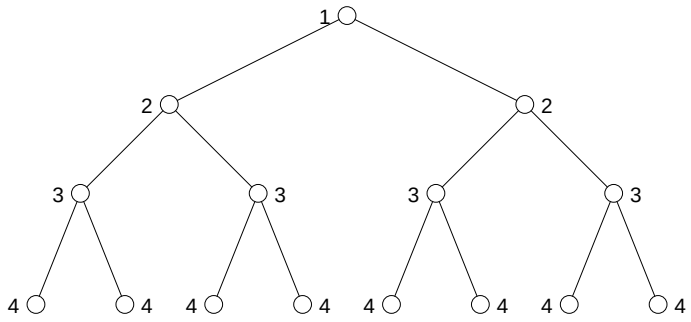


Abbildung 4.1: Illustration der Schrittzahl im Average-Case

- Average-Case: Beim Average-Case benutzen wir die Tatsache, dass sich die Anzahl der Elemente, die wir bei einer bestimmten Schrittzahl finden, bei jeder Vergrößerung der Schrittzahl verdoppelt: Wir finden mit einem Schritt nur ein einziges Element, nämlich das in der Mitte des Feldes, mit zwei Schritten schon zwei und die Hälfte der Elemente finden wir mit der Worst-Case Suchzeit $\log(n+1)$. Abbildung 4.1 zeigt dieses Schema.

  Damit ergibt sich als Gesamtaufwand $\sum_{i=1}^{\log(n+1)} i \cdot 2^{i-1}$ und deshalb als durchschnittlicher Aufwand (bei erfolgreicher Suche) von

$$C_{\text{avg}}(n) = \frac{1}{n} \sum_{i=1}^{\log(n+1)} i \cdot 2^{i-1} = \frac{1}{n}(\log(n+1) - 1)(n+1) + 1$$

  Durch Einsetzen von $k = \log(n+1)$ ergibt sich schließlich:

$$\begin{aligned} &= (\log(n+1) - 1)(n+1) + 1 \\ &= (n+1)\log(n+1) - n \end{aligned}$$

  Daraus berechnet sich die durchschnittliche Zeit für die Suche:

$$\begin{aligned} \frac{1}{n}\left[(n+1)\log(n+1) - n\right] &= \frac{n+1}{n}\log(n+1) - 1 \\ &= \log(n+1) + \frac{\log(n+1)}{n} - 1 \\ &\underset{n\to\infty}{\to} \log(n+1) - 1 \end{aligned}$$

  Das bedeutet, es wird eine Zeiteinheit weniger als im Worst-Case benötigt. Also gilt: $C_{avg}(n) = \Theta(\log n)$.

Diskussion.

- Binäre Suche ist für große n sehr empfehlenswert. Eine lineare Suche bei ca. 2 Millionen Elementen würde schlimmstenfalls 2 Millionen Vergleiche benötigen, binäre Suche hingegen nur $\log(2 \cdot 10^6) \approx 20$ Vergleiche.
- In der Praxis macht man meist *Interpolationssuche* (auch *Fingersuche* genannt), die wie Binärsuche funktioniert. Es wird jedoch m durch die erwartete Position des Suchschlüssels ersetzt. Die Interpolationssuche benötigt im Durchschnitt $\Theta(\log \log n)$ Vergleiche, aber im Worst-Case $\Theta(n)$ (!) bei einer unregelmäßigen Verteilung der Schlüssel, wie z.B.: „$AAAAAAABZ$“.
- Binäre Suche ist nur sinnvoll, wenn sich die Daten nicht allzu oft ändern; sonst sind binäre Suchbäume besser geeignet (siehe Abschnitt 4.2).
- Die Analyse zeigt, daß wir im Durchschnitt kaum weniger Vergleiche brauchen als im Worst-Case. Statt in jedem Schritt als Abbruchbedingung auf Gleichheit mit dem Suchelement zu testen, können wir deshalb auch für jede Suche logarithmisch oft halbieren, bis unser Suchbereich nur noch ein Element enthält und dabei dann einen Vergleich pro Schritt sparen.

Exkurs: Binäre Suche bei Insertion-Sort

Beim Sortieralgorithmus Insertion-Sort haben wir bereits ein Suchproblem kennengelernt: Wir suchten für ein aktuelles Element der Eingabe die richtige Position in der bereits sortierten Teilfolge. Dies haben wir mit einem einfachen Durchlauf, d.h. in Begriffen aus diesem Kapitel mit linearer Suche erreicht. Stattdessen können wir auch die binäre Suche anwenden, da die Teilfolge, in der wir suchen, sortiert ist und in einem Feld gespeichert vorliegt. Die Suche ist erfolglos und findet damit die „Lücke“, in die wir unser aktuelles Element einfügen können. Dadurch haben wir im Worst-Case von Insertion-Sort nur noch $O(n \log n)$ Vergleiche, allerdings ändert sich nichts an der Anzahl der nötigen Verschiebungen, wir haben also weiterhin quadratische Worst-Case Laufzeit.

4.1.3 Geometrische Suche

Die geometrische Suche verkleinert wie die binäre Suche den Suchbereich in jedem Schritt um einen ganzen Bereich (d.h. um mehr als ein Element wie in der linearen Suche), allerdings nicht mit einem konstanten Faktor, sondern in immer größeren Schritten. Der gesuchte Schlüssel wird mit den Schlüsseln an Position $2^0, 2^1, 2^2, \ldots, 2^k$ vergleichen, wobei 2^k die größte Zweierpotenz mit $2^k \leq n$ ist, und durch das Ergebnis der Vergleichsoperation wird der neue Suchbereich bestimmt. Wird an einer Position 2^i ein Schlüssel $x > s$ gefunden, so wird im Bereich $2^{i-1}, \ldots 2^i - 1$ mittels binärer Suche weitergesucht. Ist hingegen auch der

Schlüssel an 2^k noch kleiner als s, wird in den restlichen Positionen $2^k, \ldots, n$ binär gesucht. Da sowohl die Binärsuche als auch der Aufteilungsschritt nie mehr als $\lceil \log n \rceil + 1$ Schritte benötigt, brauchen wir nie mehr als doppelt so viele Schritte wie binäre Suche, allerdings sind wir für Elemente, die relativ klein (Position an Index $< \sqrt{n}$) sind, sehr effizient. Damit bietet die geometrische Suche einen Kompromiss zwischen linearer Suche und binärer Suche und ist gut geeignet, wenn der Suchschlüssel wahrscheinlich relativ klein zu den gespeicherten Daten ist.

Schema. In einem ersten Schritt wird die Liste sortiert; jetzt gilt:

$$A[1].key \leq A[2].key \leq \ldots \leq A[n].key$$

Nun folgt man der *Divide and Conquer Methode*: Vergleiche den Suchschlüssel s in Schritt i mit dem Schlüssel des Elementes an der Position 2^i des Suchbereichs. Falls

- s gefunden: Stop, Rückgabe des Elements
- s ist kleiner: Durchsuche Elemente im letzten Teilbereich links von der aktuellen Position.
- s ist größer: Gehe eine Suchposition weiter durch Verdoppeln des Index, falls noch kleiner als n, sonst durchsuche den Rest mit binärer Suche.

Parametersuche. In dem hier besprochenen Fall der Suche in einem Array hat die geometrische Suche nur beschränkten Nutzen in speziellen Anwendungsfällen. Der klassische Bereich in dem dieses Suchverfahren jedoch glänzen kann ist die *Parametersuche*. Hier suchen wir nicht einen Wert in einem Array, sondern wollen einen „optimalen" Parameter n für eine Funktion $f(n)$ finden. Allerdings können wir die Güte eines Parameters nur durch Berechnen dieser Funktion erfahren.

Wir starten in diesem Fall also z.B. in dem wir $f(1)$ ausrechnen, und (problemspezifisch!) feststellen ob „1" der optimale Parameter war, bzw. falls nicht, ob der optimale Parameter kleiner oder größer ist[1]. Ist er größer, so verdoppeln wir nun jeweils den Parameter und prüfen die Funktion für $f(2)$. Ist dieser Parameter noch immer zu klein, so testen wir $f(4)$, danach $f(8)$, $f(16)$, $f(32)$, usw. Sobald wir einen Parameter testen der zu groß war, haben wir ein eingeschränktes Intervall zwischen 2^k und 2^{k+1} (für irgendein k), und können dieses Intervall dann z.B. mit binärer Suche genauer untersuchen.

Die Besonderheit der geometrischen Suche liegt also vorallem darin, dass wir einen Bereich absuchen können, von dem wir seine obere Schranke nicht kennen.

[1] In vielen Anwendungsfällen kann man dies zum Beispiel an dem Vorzeichen des Resultats erkennen.

4.2 Binäre Suchbäume

Binäre Suchbäume sind binäre Bäume mit zusätzlichen speziellen Eigenschaften, die sich zum Suchen in dynamischen Datenmengen eignen, d.h. sie unterstützen die Operationen des abstrakten Datentyps Dictionary. Wir haben binäre Bäume bereits bei Heapsort und der Diskussion der unteren Schranke für das Sortierproblem kennengelernt, und führen zunächst einige Begriffe formal ein.

Bezeichnungen. Binäre Suchbäume basieren auf sogenannten *gewurzelten Bäumen*, die wie folgt rekursiv definiert sind: Ein gewurzelter Baum ist entweder leer oder er besteht aus einem Knoten (genannt *Wurzel*), der eine Menge von *Zeigern* besitzt, die selbst wieder auf gewurzelte Bäume zeigen. Zeigt der Zeiger eines Knotens v auf einen nichtleeren gewurzelten Baum, so nennen wir die Wurzel w dieses Baums auch *Kind* von v und v den Elter von w.

Die Wurzel ist somit der einzige Knoten in einem gewurzelten Baum ohne Elter. Alle anderen Knoten können durch Verfolgung von Elter-Kind Beziehungen über genau einen Weg von der Wurzel aus erreicht werden. Die *Tiefe* $t_w(v)$ eines Knotens v ist die Anzahl der Elter-Knoten auf diesem Weg einschliesslich der Wurzel w (Ist klar, welche Wurzel gemeint ist, lassen wir den Index auch weg und schreiben $t(v)$). Sie ist eindeutig, da nur ein solcher Weg existiert. Die Tiefe der Wurzel ist 0. Die Menge aller Knoten mit gleicher Tiefe im Baum wird auch *Ebene* genannt. Knoten, die keine Kinder besitzen, heißen *Blätter*, alle anderen Knoten heißen *innere Knoten*.

Jeder Knoten v in einem gewurzelten Baum T ist selbst Wurzel eines *Unterbaums*, der aus ihm selbst sowie den Knoten in den an seinen Kindern gewurzelten Unterbäumen besteht. Die Knoten im Unterbaum mit Wurzel v nennen wir auch *Nachfolger* von v in T. Die *Höhe* $h(T_r)$ eines (Unter-)Baumes T_r mit dem Wurzelknoten r ist definiert als $h(T_r) = max\{t_r(v) : v \text{ ist Knoten in } T_r\}$. Die Höhe eines leeren Baums ist als -1 definiert. In Abbildung 4.2 gilt z.B. $h(T_{w_1}) = 2$ und $h(T_{w_2}) = 4$. Die *Höhe* eines Knotens v ist analog die Höhe des Unterbaums der an v gewurzelt ist, also die Länge eines längsten Pfades in T von v zu einem Nachkommen von v. Abbildung 4.3 gibt ein Beispiel für die Höhe und die Tiefe von Knoten in einem gewurzelten Baum.

Ein gewurzelter Baum heißt *geordnet*, wenn die Reihenfolge der Kinder an jedem Knoten festgelegt ist, die Kinder also eine Folge $c_1, \ldots, c_l$ bilden. Ein *binärer gewurzelter Baum* ist ein gewurzelter Baum, bei dem jeder Knoten genau zwei Kindzeiger hat. Ist ein binärer gewurzelter Baum geordnet, dann können wir zwischen dem linken und dem rechten Kind unterscheiden. Die an den Kindern gewurzelten Unterbäume werden entsprechend als linker bzw. rechter Unterbaum bezeichnet. Ein *vollständig balancierter* binärer Baum ist ein binärer Baum, in welchem jede Ebene, außer eventuell die unterste, völlig mit Knoten gefüllt ist. Ein binärer Baum heißt *vollständig*, wenn alle seine Ebenen komplett gefüllt sind.

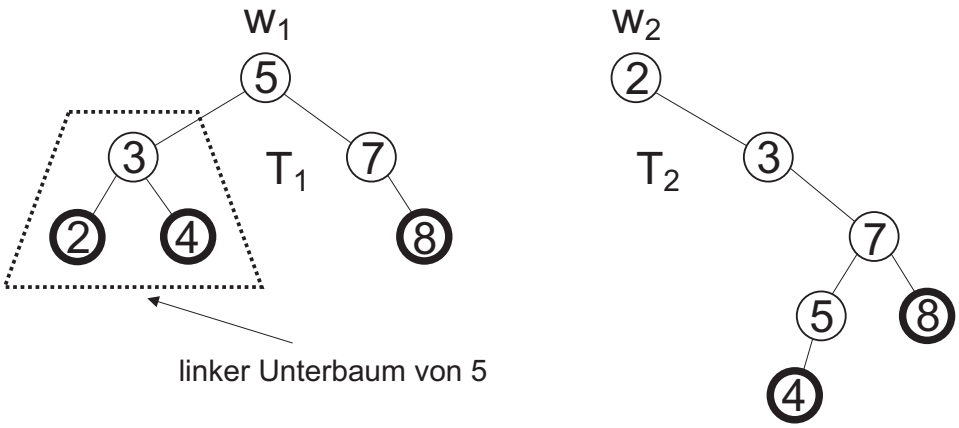


Abbildung 4.2: Zwei binäre Suchbäume, Blätter haben fette Umrandung, Knoten w_1 mit Schlüssel 5 ist Wurzel von T_1, Knoten w_2 mit Schlüssel 2 Wurzel von T_2

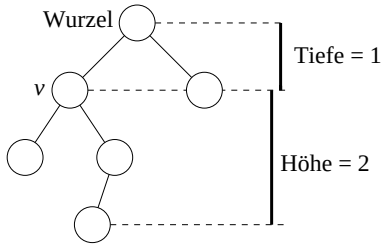


Abbildung 4.3: Höhe und Tiefe eines Knotens v

Definition 4.1. Ein *binärer Suchbaum* ist ein binärer geordneter Baum, dessen Knoten einen Suchschlüssel enthalten, und der die folgende *Suchbaumeigenschaft* erfüllt:

> Enthält ein Knoten in einem binären Suchbaum den Schlüssel x, so sind alle Schlüssel in seinem linken Unterbaum kleiner als x und alle Schlüssel in seinem rechten Unterbaum größer als x.

Implementierung von binären Bäumen. Wir realisieren binäre Suchbäume als verallgemeinerte Listen mit bis zu zwei Nachfolgern, siehe Listing 4.14. Der Zugriff auf den Baum erfolgt über seinen Wurzelknoten *root*. Den Verweis auf das Elterelement fügen wir nur ein, um die Pseudocode-Beschreibungen in diesem Kapitel einfach zu gestalten.

Traversierungen für geordnete binäre Bäume. Die Traversierung eines binären Baums besucht alle Knoten des Baums in einer bestimmten Reihenfolge. Nach der unterschiedlichen Position der Wurzel und der beiden Unterbäume in einem solchen Durchlauf unterscheidet man die folgenden Methoden:

- Inorder-Traversierung: Traversiere rekursiv zunächst den linken Unterbaum, besuche dann die Wurzel, traversiere danach den rechten Unterbaum. Die Inorder-Traversierung ist die wichtigste Traversierung für binäre Suchbäume, weil dabei die Schlüssel aufsteigend sortiert durchlaufen werden.

- Preorder-Traversierung: Besuche zunächst die Wurzel, traversiere dann rekursiv zuerst den linken, dann den rechten Unterbaum.

- Postorder-Traversierung: Traversiere zunächst rekursiv den linken und den rechten Unterbaum, besuche danach die Wurzel.

Listing 4.3 zeigt den Pseudocode der Traversierungen, wobei als Aufrufparameter jeweils die Wurzel eines binären Baums übergeben wird. Wenn man die Rolle des linken und rechten Unterbaums jeweils austauscht, erhält man entsprechende symmetrische Methoden.

Anmerkung 4.1. Gelegentlich wird auch die sogenannte *Level-Order-Traversierung* beschrieben, bei der alle Ebenen von links nach rechts und die Ebenen von oben nach unten durchlaufen werden.

```
 1: ▷ Repräsentation eines Knotens
 2: struct TreeNode
 3:     var TreeNode left                          ▷ Linker Zeiger
 4:     var TreeNode right                         ▷ Rechter Zeiger
 5:     var KeyType key                            ▷ Schlüssel des Knotens
 6:     var ValueType info                         ▷ Gespeicherter Wert
 7:     var TreeNode parent                        ▷ Zeiger auf Elter
 8: end struct
 9: ▷ Interne Repräsentation eines Baums
10: var TreeNode root
11: ▷ Initialisierung
12: left := nil
13: right := nil
14: parent := nil
```

Listing 4.2: Repräsentation eines binären Baums durch Knotenelemente.

```
procedure INORDER(p)
    if p ≠ nil then
        INORDER(p.left)
        Ausgabe von p
        INORDER(p.right)
    end if
end procedure
```

```
procedure PREORDER(p)
    if p ≠ nil then
        Ausgabe von p
        PREORDER(p.left)
        PREORDER(p.right)
    end if
end procedure
```

```
procedure POSTORDER(p)
    if p ≠ nil then
        POSTORDER(p.left)
        POSTORDER(p.right)
        Ausgabe von p
    end if
end procedure
```

Listing 4.3: Baum-Traversierungen

Beispiel 4.1. Inorder-Traversierung von T_1 aus Abb. 4.2, angegeben ist die rekursive Aufrufreihenfolge und die Ausgabe:

```
Inorder (5)
  Inorder (3)
    Inorder (2)
      Ausgabe 2
    Ausgabe 3
    Inorder (4)
      Ausgabe 4
  Ausgabe 5
  Inorder (7)
    Ausgabe 7
    Inorder (8)
      Ausgabe 8
```

Es ergibt sich die folgende Ausgabereihenfolge: 2, 3, 4, 5, 7, 8. Für T_2 ergibt sich die gleiche Ausgabereihenfolge. Man kann also ausgehend von einer gegebenen Inorder-Reihenfolge den ursprünglichen Baum nicht eindeutig rekonstruieren.

Bei gegebener Inorder- und Preorder-Traversierungsreihenfolge eines beliebigen binären Baums kann dieser eindeutig rekonstruiert werden. Dies ist nicht möglich, wenn lediglich die Preorder- und Postorder-Reihenfolge gegeben sind, wie das Beispiel in Abbildung 4.4 zeigt.

Implementierung der Dictionary-Operationen. Wir besprechen nun eine Implementierung der Dictionary-Operationen mit Laufzeit $O(h(T))$ für binäre Suchbäume, wobei $h(T)$ die Höhe des gegebenen Suchbaumes T ist.

Für SEARCH, DELETE und INSERT mit gegebenem Schlüssel s ist es wichtig, die Position in T zu finden, an der s zur Einhaltung der Suchbaumeigenschaft in T gespeichert sein

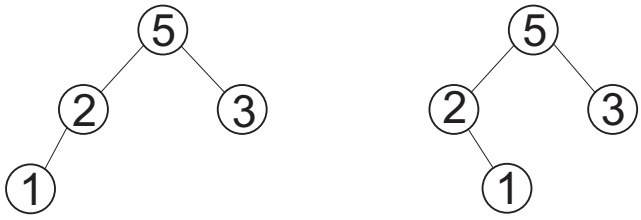


Abbildung 4.4: Zwei Bäume mit den gleichen Preorder (5, 2, 1, 3) und Postorder (1, 2, 3, 5) Traversierungsreihenfolgen

müßte. Dazu nutzen wir die Suchbaumeigenschaft aus, die kleinere Schlüssel als den der Wurzel stets im linken Unterbaum und größere Schlüssel im rechten Unterbaum speichert. Wir vergleichen s mit dem in der Wurzel gespeicherten Schlüssel s'. Falls $s \neq s'$, setzen wir die Suche für $s < s'$ im linken Unterbaum und für $s > s'$ im rechten Unterbaum fort. Bei Gleichheit haben wir Schlüssel s im Baum gefunden und können aufhören. Ansonsten endet unser Suchverfahren wenn wir in einen leeren Unterbaum verzweigen, die Suche war dann erfolglos. Damit haben wir nun schon die SEARCH-Operation komplett beschrieben, deren Laufzeit von der Länge des Suchpfads abhängt und damit im Worst-Case linear in der Höhe des Baums ist.

- SEARCH: Wir suchen nach einem Element mit dem gegebenen Schlüssel s, indem wir rekursiv Schlüsselvergleiche mit der Wurzel des Baums durchführen und je nach Ergebnis im linken oder rechten Unterbaum weitersuchen. Die Suche endet erfolgreich, wenn wir auf den gesuchten Schlüssel treffen und nicht erfolgreich, wenn wir in einen leeren Unterbaum verzweigen. Die Laufzeit ist abhängig von der Länge des Suchpfades und damit im Worst-Case und im Average-Case linear in der Höhe des Baums, im Best-Case ist sie konstant. Listing 4.4 zeigt den Pseudocode für die Suchoperation.

- INSERT: Wollen wir in einen binären Suchbaum T ein Element x mit Schlüssel s einfügen, so suchen wir wie bei SEARCH die richtige Position (erfolglos), und fügen bei Erreichen eines leeren Unterbaums an dessen Stelle einen neuen Knoten mit den Werten von x in den Baum ein. Listing 4.5 zeigt den Pseudocode für die Einfügeoperation.

- DELETE: Auch beim Löschen eines Elements mit gegebenem Schlüssel s suchen wir zunächst nach s im Baum. Haben wir s in einem Knoten v gefunden, unterscheiden wir zwei Fälle:

  (1) v hat höchstens ein Kind. Dann können wir v einfach löschen und falls v ein Kind hat, rückt dieses an seine Stelle.
  (2) v hat zwei Kinder. Um unsere Suchbaumeigenschaft nach dem Löschen zu erhalten, brauchen wir einen Ersatzknoten, der an die Stelle von v treten kann. Dies können die beiden Nachbarn von v in der sortierten Liste der Schlüssel sein, also die

```
Eingabe: Baum mit Wurzel p; Schlüssel s
Ausgabe: Knoten mit Schlüssel s oder nil, falls s nicht vorhanden
 1: function SEARCH(p, s) : TreeNode
 2:     while p ≠ nil ∧ p.key ≠ s do
 3:         if s < p.key then
 4:             p := p.left
 5:         else
 6:             p := p.right
 7:         end if
 8:     end while
 9:     return p
10: end function
```

Listing 4.4: Suche in binären Suchbäumen

Knoten mit dem nächstgrößeren bzw. nächstkleineren Schlüssel. Wir beschreiben die Ersetzung durch den nächstkleineren Schlüssel. Dies ist der größte Wert im linken Unterbaum von v. Wir laufen deshalb zunächst zum linken Kind von v und dann solange zum rechten Kind, bis ein leerer Kindzeiger erreicht wird. Der letzte durchlaufene Knoten w enthält dann den gesuchten Schlüssel. Knoten w wird nun mit v ausgetauscht, danach kann v gelöscht werden, da wir nun in Fall 1) sind. Für den Fall der Suche nach dem nächstgrößeren Schlüssel hätten wir analog den kleinsten Schlüssel im rechten Unterbaum für die Ersetzung benutzt. Listing 4.6 zeigt den Pseudocode für das Entfernen eines Knotens, Abbildung 4.5 zeigt ein Beispiel.

Zusätzlich zu den Dictionary-Operationen werden oft auch die folgenden einfachen Hilfsfunktionen benötigt:

- MINIMUM (MAXIMUM): Wir suchen das Element mit dem kleinsten (größten) Schlüssel im Baum. Dazu laufen wir von der Wurzel aus rekursiv in den linken (rechten) Unterbaum, bis wir auf ein leeres Kind treffen. Der letzte durchlaufene Knoten enthält dann den kleinsten (größten) Schlüssel.

- PREDECESSOR (SUCCESSOR): Ausgehend von einem gegebenen Element suchen wir das Element mit dem nächstkleineren (nächstgrößeren) Schlüssel im Baum. Die Suche nach dem nächstkleineren Schlüssel haben wir bereits bei DELETE für den Fall beschrieben, dass das Element im Baum ein linkes Kind besitzt. Ist der linke Unterbaum leer, kann das gesuchte Element, falls vorhanden, nur auf dem Pfad zur Wurzel liegen. Wir folgen solange von v aus den Elter-Einträgen, bis wir entweder aus dem rechten Unterbaum zum Elter gelangen (dann ist der Schlüssel in diesem Elter der gesuchte), oder an der Wurzel in einen leeren Eintrag laufen, dann gibt es keinen kleineren Wert. Den Ablauf der Suche nach dem nächstgrößeren Schlüssel (SUCCESSOR) erhalten wir analog zu obiger Beschreibung durch Vertauschen der Richtungen links und rechts. Listing 4.7 gibt den Pseudocode der Operation an.

```
Eingabe: Baum mit Wurzel root; Schlüssel k und Wert v
 1: procedure INSERT(ref root, k, v)   ▷ Fügt Wert v mit Schlüssel k in Baum mit Wurzel
    root ein
 2:     var TreeNode r, p
 3:     r := nil                       ▷ r wird Elter von des neuen Knotens
 4:     p := root
 5:     while p ≠ nil do
 6:         r := p                     ▷ r ist der zuletzt besuchte Knoten
 7:         if k < p.key then
 8:             p := p.left
 9:         else if k > p.key then
10:             p := p.right
11:         else                       ▷ Schlüssel schon vorhanden
12:             p.info := v
13:             return
14:         end if
15:     end while
16:     q := new TreeNode
17:     q.parent := r
18:     q.key := k
19:     q.info := v
20:     if r = nil then
21:         root := q                  ▷ Neuen Knoten in leeren Baum einfügen
22:     else if q.key < r.key then
23:         r.left := q
24:     else
25:         r.right := q
26:     end if
27: end procedure
```

Listing 4.5: Einfügen in binären Suchbäumen

Analyse. Die Laufzeit aller Operationen ist offensichtlich $O(h(T))$, da wir jeweils nur Pfade im Baum verfolgen, die asymptotisch durch die Baumhöhe beschränkt sind. Die Struktur des binären Suchbaums, der durch Einfügen (und Löschen) von Elementen entsteht, und damit auch die Höhe des Baums, ist aber von der Einfügereihenfolge der Elemente abhängig. Abbildung 4.6 zeigt zwei Extrembeispiele die bei unterschiedlicher Einfügereihenfolge derselben Schlüssel entstehen können. Im ersten Fall ist der Baum T_1 zu einer linearen Liste degeneriert, im anderen Fall sind die Knoten so verteilt, dass alle Ebenen vollständig besetzt sind bis auf eine Position auf der untersten Ebene; der Baum T_2 ist also vollständig balanciert.

Der Extremfall, dass ein Baum zu einer linearen Liste ausartet, entsteht z.B., wenn die Ein-

```
Eingabe: Baum mit Wurzel root; Schlüssel s
 1: procedure DELETE(ref root, s)
 2:     ▷ Entfernt Knoten mit Schlüssel s im Baum mit Wurzel root
 3:     var TreeNode r, p, q                ▷ Bestimme einen Knoten r zum Herausschneiden
 4:     q := SEARCH(s)                                  ▷ Suche s in Baum (erfolgreich)
 5:     if (q.left = nil) or (q.right = nil) then       ▷ Fall 1:q hat max. 1 Kind
 6:         r := q
 7:     else          ▷ q hat 2 Kinder, wird durch Predecessor ersetzt, dieser wird entfernt
 8:         r := PREDECESSOR(q)
 9:         q.key := r.key; q.info := r.info            ▷ Umhängen der Daten von r nach q
10:     end if
11:     ▷ Jetzt löschen wir Knoten r mit max. einem Kind
12:     ▷ Lasse p auf Kind von r verweisen (p = nil, falls r keine Kinder hat)
13:     if r.left ≠ nil then p := r.left else p := r.right end if
14:     if p ≠ nil then
15:         p.parent := r.parent                    ▷ Neuer Elter von p ist jetzt Elter von r
16:     end if                                      ▷ Hänge p anstelle von r in den Baum ein
17:     if r.parent = nil then                      ▷ r war Wurzel: neue Wurzel ist p
18:         root := p
19:     else if r = r.parent.left then      ▷ Hänge p an der richtigen Seite des Elter von r
20:         r.parent.left := p                      ▷ p sei linker Nachfolger
21:     else
22:         r.parent.right := p                     ▷ p sei rechter Nachfolger
23:     end if
24: end procedure
```

Listing 4.6: Entfernen in binären Suchbäumen

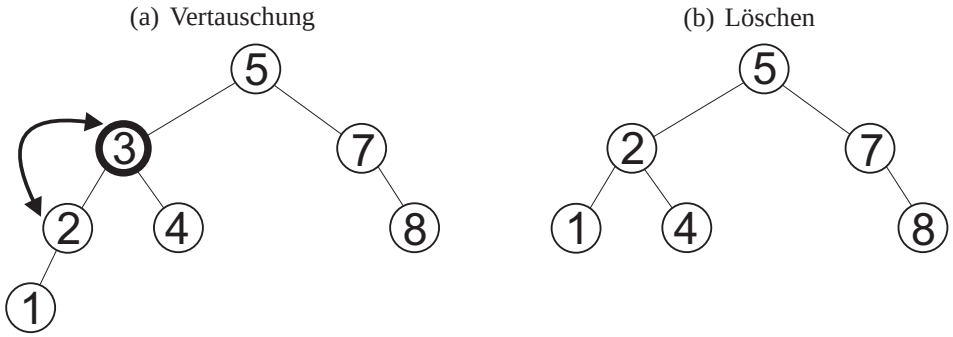


Abbildung 4.5: Löschen eines Knotens mit zwei Kindern: Der Knoten mit Schlüssel 3 wird zunächst mit seinem Vorgänger 2 getauscht, dieser wird dann gelöscht

```
Eingabe: Knoten p
Ausgabe: Vorgänger von Knoten p ≠ nil in der Inorder-Traversierungsreihenfolge
 1: function PREDECESSOR(p)
 2:     var TreeNode q
 3:     if p.left ≠ nil then
 4:         return MAXIMUM(p.left)
 5:     else
 6:         q := p.parent                ▷ Ist p rechtes Kind von q, dann ist q Predecessor
 7:         while q ≠ nil and p = q.left do
 8:             p := q
 9:             q := q.parent
10:         end while
11:         return q
12:     end if
13: end function
```

Listing 4.7: Suche des Vorgänger-Schlüssels

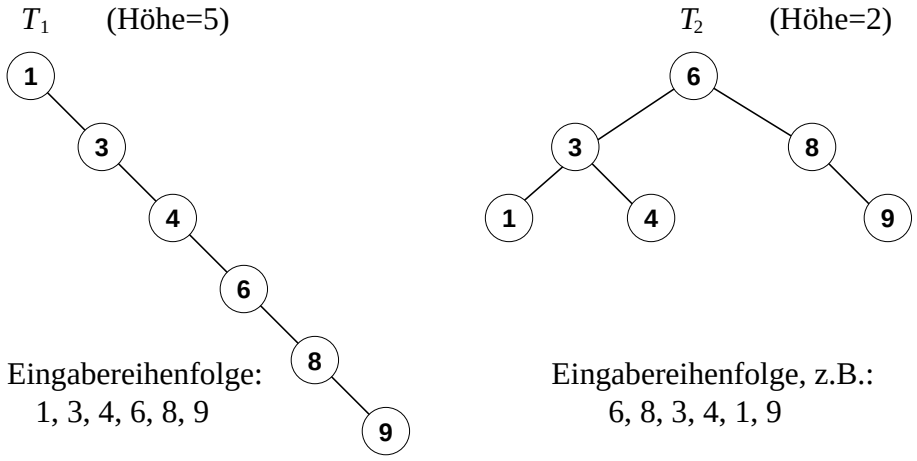


Abbildung 4.6: Zwei Bäume mit denselben Datenelementen: Baum T_1 ist zur linearen Liste entartet, Baum T_2 dagegen ist vollständig balanciert

träge mit aufsteigenden Schlüsseln in einen anfangs leeren Baum eingefügt werden. Der Baum verhält sich dann bei der Suche wie eine lineare Liste, d.h. die Suche benötigt lineare Zeit, da $h(T) = \Theta(n)$.

Vollständig balancierte Bäume T dagegen haben die Höhe $h(T) = \Theta(\log n)$, d.h. alle Operationen benötigen Zeit $O(\log n)$. Für dieses Zeitverhalten genügen aber auch „hinreichend balancierte“ Bäume, die wir im nächsten Abschnitt besprechen werden.

Einen binären Suchbaum, der durch eine beliebige Folge der bisher beschriebenen Opera-

tionen ausgehend von einem leeren Baum entstehen, nennt man auch *natürlichen* binären Suchbaum. Damit ist hier eher die Entstehung als die Baumstruktur festgelegt, da die Gestalt des Baums entscheidend von der Reihenfolge abhängt, in der die Schlüssel eingefügt werden. Man kann zeigen: die erwartete durchschnittliche Suchpfadlänge (über alle möglichen Permutationen von n Einfügeoperationen) für einen natürlichen binären Suchbaum mit n Knoten ist:

$$I(n) = 2 \ln n - O(1) = 1.38629... \cdot \log_2 n - O(1),$$

d.h. für große n ist die durchschnittliche Suchpfadlänge nur ca. 40% länger als im Idealfall. Das heißt natürlich nicht, dass natürliche binäre Suchbäume für die Praxis eine gute Wahl darstellen.

Aufgrund der Abhängigkeit von der Einfügereihenfolge kann man nicht garantieren, dass die wichtigsten Basisoperationen für Bäume (Suchen, Einfügen und Entfernen von Schlüsseln) einen logarithmisch mit der Anzahl der im Baum gespeicherten Elemente wachsenden Aufwand haben. Es gibt jedoch Techniken, die es erlauben, einen Baum, der nach dem Einfügen oder Entfernen eines Elements aus der Balance geraten ist, wieder so zu rebalancieren, dass die Basisoperationen mit logarithmischem Aufwand ausführbar sind. Ein Beispiel für diese sogenannten *balancierten Suchbäume* wird nun vorgestellt.

4.3 AVL-Bäume

Binäre Suchbäume lösen das Wörterbuchproblem effizient, wenn die Höhe der Bäume logarithmisch beschränkt bleibt. Balancierte Suchbäume vermeiden die Extremfälle, wie sie für degenerierte binäre Suchbäume auftauchen, indem der Baum nach Änderungen, die die Balance zerstören, wieder ausbalanciert wird. Man erkauft sich also die Beschränkung der Laufzeit mit den durch das Ausbalancieren etwas komplizierteren Operationen. Für die Balancierung gibt es verschiedene Möglichkeiten:

- Höhenbalancierte Bäume: Die Höhen der Unterbäume jedes Knotens unterscheiden sich um höchstens eins voneinander.
- Gewichtsbalancierte Bäume: Die Anzahl der Knoten in den Unterbäumen jedes Knotens unterscheidet sich höchstens um einen konstanten Faktor.
- (a, b)-Bäume $(2 \leq a \leq b)$: Jeder innere Knoten (außer der Wurzel) hat zwischen a und b Kinder und alle Blätter haben den gleichen Abstand zur Wurzel.

Wir betrachten in diesem Abschnitt ein Beispiel zu höhenbalancierten Bäumen (AVL-Bäume) und im nächsten ein Beispiel zu (a,b)-Bäumen (B-Bäume, vgl. Abschnitt 4.4). AVL-Bäume wurden 1962 von G. M. Adelson-Velskii und Y. M. Landis eingeführt. Wir führen die folgenden Begriffe ein:

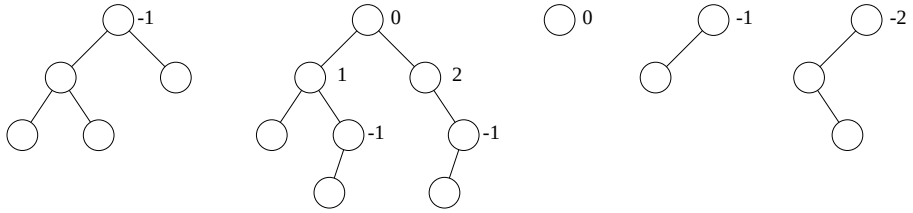


Abbildung 4.7: Beispiele für die Balance von Knoten in einem binären Baum

- Die *Balance eines Knotens* v in einem binären Baum ist definiert als $bal(v) = h_2 - h_1$, wobei h_1 und h_2 die Höhe des linken bzw. rechten Unterbaumes von v bezeichnen (siehe Abb. 4.7).
- Ein Knoten v heißt *(höhen)balanciert*, falls $bal(v) \in \{-1, 0, +1\}$, sonst unbalanciert.
- Ein Baum heißt *(höhen)balanciert*, wenn sich die Höhen der Unterbäume an jedem Knoten höchstens um 1 voneinander unterscheiden, d.h. jeder Knoten balanciert ist.

Definition 4.2. Ein *AVL-Baum* ist ein binärer Suchbaum, in dem alle Knoten höhenbalanciert sind.

Ein binärer Suchbaum ist also ein AVL-Baum, wenn für jeden Knoten v des Baumes gilt, dass sich die Höhe des linken Unterbaums von der Höhe des rechten Unterbaums von v höchstens um 1 unterscheidet.

(a) Nicht höhenbalanciert (b) Höhenbalanciert

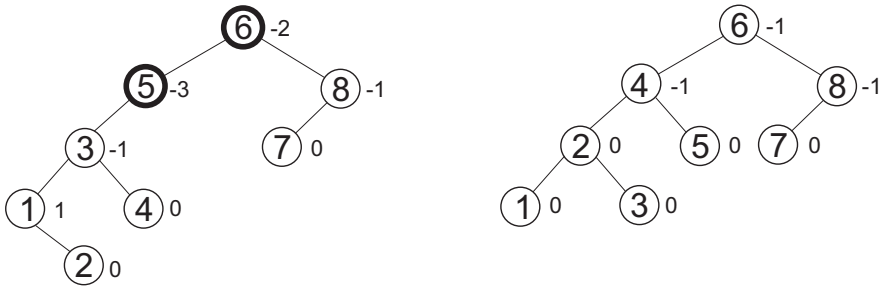


Abbildung 4.8: Balance bei binären Suchbäumen, Balancewerte sind als Label angegeben, fett gezeichnete Knoten sind nicht balanciert

Das folgende wichtige Theorem gibt eine obere Schranke für die Höhe von AVL-Bäumen und damit auch für deren Suchtiefe. Da der Beweis aufwändig ist, wird es hier nur zitiert.

Theorem 4.1. *Ein AVL-Baum mit n Knoten hat Höhe $O(\log n)$.*

Die Operationen SEARCH, MINIMUM, MAXIMUM, SUCCESSOR und PREDECESSOR bei AVL-Bäumen werden genauso wie bei natürlichen binären Suchbäumen ausgeführt. Da die Suchtiefe $O(\log n)$ beträgt, können diese Operationen in Zeit $O(\log n)$ ausgeführt werden. Aufpassen müssen wir hingegen bei den Operationen INSERT und DELETE, da hier eventuell die AVL-Baum-Eigenschaft verloren geht.

Zunächst werden die Operationen INSERT bzw. DELETE genauso wie bei natürlichen binären Suchbäumen durchgeführt. Danach wird untersucht, ob nun eventuell ein Knoten nicht mehr balanciert ist. Da wir nur einen Knoten entfernt oder eingefügt haben, wissen wir in diesem Fall, dass ein Knoten u existiert mit $bal(u) \in (-2, 2)$. Dieser Knoten kann nur auf den bei der Operation abgelaufenen Suchpfaden liegen. Im Folgenden beschreiben wir, wie der Baum geändert wird, so dass er danach wieder balanciert ist (*Rebalancierung*).

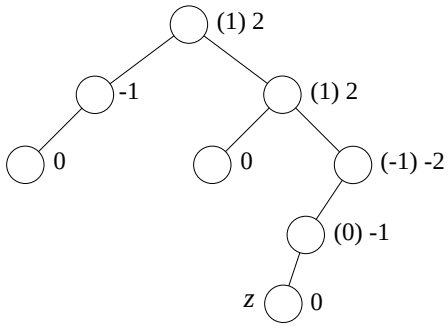


Abbildung 4.9: Ein Baum, der durch Einfügen eines Knotens z seine Balance-Eigenschaft verloren hat. Die alten Balancen sind geklammert, falls sie verändert wurden.

Rebalancierung. Um bei AVL-Bäumen nach einer Operation die AVL-Baum-Eigenschaft wieder herzustellen, nutzen wir das Konzept der *Rotation*, bei der eine lokale Änderung der Elter-Kind-Beziehung zwischen Knoten und eventuell ein Umhängen von Unterbäumen erfolgt.

Sei T_0 ein AVL-Baum vor der Zerstörung der Balance (z.B. durch Einfügen oder Entfernen eines Knotens), und T_1 der unbalancierte Baum hinterher. Sei u der unbalancierte Knoten maximaler Tiefe, d.h. $bal(u) \in \{-2, +2\}$. Seien T und T^* die Unterbäume mit Wurzel u in T_0 bzw. T_1.

Fall 1: Sei $bal(u) = -2$, d.h. der linke Unterbaum ist um 2 Ebenen höher als der rechte. Dann sei v das linke Kind von u (dieses existiert, weil $bal(u) < 0$).

Fall 1.1: $bal(v) \in \{-1, 0\}$, d.h. der linke Unterbaum des linken Kindes von u ist höher als oder gleich hoch wie der rechte. Wir erreichen die Rebalancierung von u durch *Rechts-Rotation* an u, d.h. u wird rechtes Kind von v. Das rechte Kind

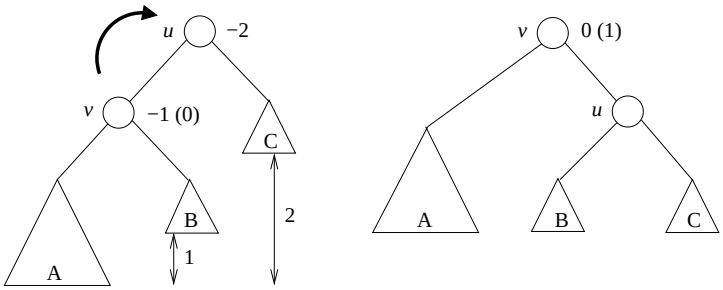


Abbildung 4.10: Rebalancierung durch Rechts-Rotation an Knoten u: der Pfeil im linken, unbalancierten Baum deutet die Richtung der Rotation an

von v wird als linkes Kind an u abgegeben (vgl. Abb. 4.10). Der Name Rotation ist passend, weil man sich gut vorstellen kann, den Baum an u im Uhrzeigersinn zu drehen. (Dadurch wandert v nach oben und u nach unten. Der rechte Unterbaum von v fällt dabei von v nach u).
Durch die Rotation nach rechts bleibt die Inorder-Reihenfolge (hier $A\,v\,B\,u\,C$) erhalten. Für alle Knoten unterhalb hat sich nichts geändert. Wir wissen, dass die Höhen der Unterbäume B und C gleich der Höhe von A minus eins (bzw. gleich für $bal(v) = 0$) sind. Der neue Unterbaum T' mit Wurzel v ist also nach der Rotation balanciert (siehe Abb. 4.10).

Fall 1.2: $bal(v) = 1$, d.h. der rechte Unterbaum des linken Kindes von u ist höher als der linke. Dann existiert das rechte Kind w von v. Wir rebalancieren durch eine *Links-Rotation* an v und eine anschließende Rechts-Rotation an u. Dies nennt man auch eine *Links-Rechts-Rotation* (s. Abb. 4.11). Auch hier ändert sich durch die Doppelrotation die Inorder-Reihenfolge nicht. Weiterhin wissen wir, dass die Höhen von B oder C sowie D gleich der Höhe von A sind. Der Teilbaum T'' mit Wurzel w ist demnach hinterher balanciert.

Fall 2: $bal(u) = 2$. Dieser Fall ist genau symmetrisch zu Fall 1. Man muss nur den Baum gespiegelt betrachten. Wir nutzen analog zu Fall 1 die Konzepte *Links-Rotation* und *Rechts-Links-Rotation,* die sich nur durch ein Vertauschen der jeweiligen Seiten von den Operationen in Fall 1 unterscheiden.

Abbildung 4.12 zeigt je ein Beispiel für jeden der vier Fälle.

Für die einfache Rotation gilt: $h(T') - h(T) \in \{0, 1\}$, denn: entweder wurde die Höhe des niedrigeren Unterbaumes um 1 kleiner oder die Höhe des höheren Unterbaumes um 1 größer. Im Falle der Doppelrotation gilt $|h(T'') - h(T)| \leq 1$, weil $h(T') - h(T^*) \in \{0, 1\}$ und $h(T'') - h(T') \in \{-1, 0\}$.

Daraus folgt, dass alle Transformationen von T nach T' bzw. T'' die Höhe eines Unterbaums an einem Knoten und damit die Balance um höchstens 1 verändern.

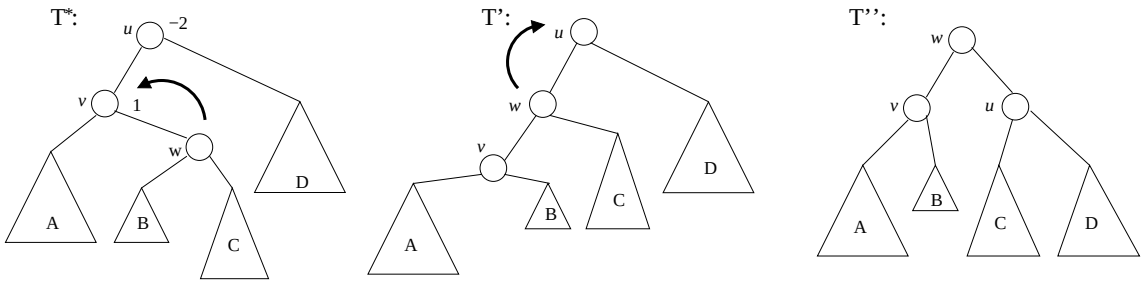


Abbildung 4.11: Rebalancierung durch Links-Rechts-Rotation an Knoten u: die beiden Pfeile in den unbalancierten Bäumen kennzeichnen die Richtung der Rotationen

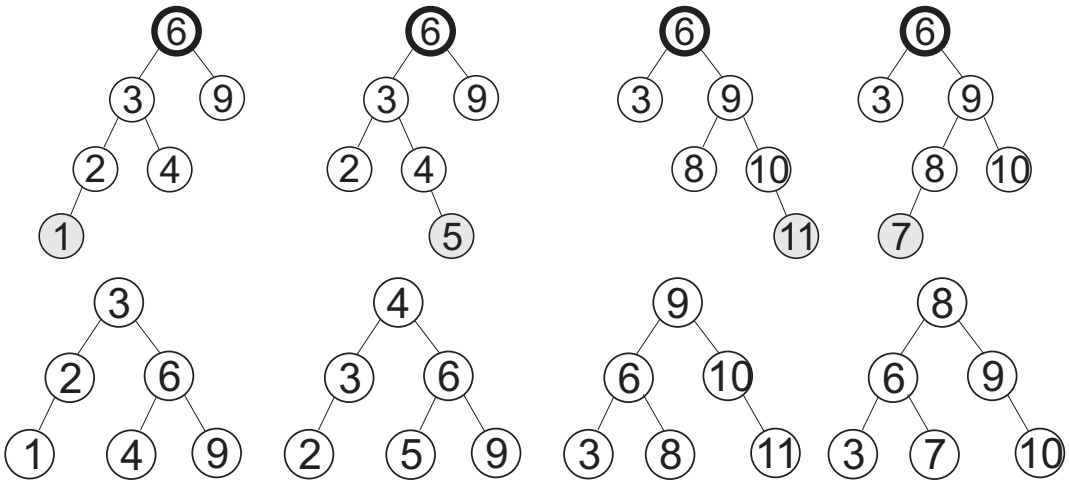


Abbildung 4.12: Obere Reihe: Vier Beispiele für AVL-Bäume, die durch das Einfügen eines Knotens (grau hinterlegt) aus der Balance geraten sind (fette Knoten sind nicht balanciert). Untere Reihe: Die vier Bäume nach der Rebalancierung

Für das Einfügen eines Knoten reicht eine Rotation, um die Balance wieder herzustellen. Beim Löschen eines Knotens kann es allerdings nötig sein, mehrere Rotationen durchzuführen, da sich durch die Rotation in einem Unterbaum die Balance des Elter geändert haben kann. Deshalb wird die Rebalancierung im gesamten Baum wiederholt fortgesetzt: Wir betrachten wieder einen Knoten u mit $bal(u) \in \{-2, +2\}$ und maximaler Tiefe. Diese Tiefe ist nun kleiner als vorher, d.h. das Verfahren terminiert nach $O(h(T_1))$ Iterationen zu einem gültigen AVL-Baum.

Für die nachfolgenden Algorithmen kommt nun in jedem Knoten das Attribut *height* hinzu, das die Höhe des Knotens speichert. Wir nutzen die in Algorithmus 4.8 gezeigte Hilfsfunktion zur Rückgabe der Höhe eines Baums, um den Pseudocode möglichst einfach zu halten, aber eventuelle Zugriffe auf nicht vorhandene Unterbäume zu verhindern.

```
Eingabe: Teilbaum mit Wurzel u
Ausgabe: Höhe des Teilbaums
 1: procedure HEIGHT(u)
 2:     if u = nil then
 3:         return −1                          ▷ Teilbaum ist leer
 4:     else
 5:         return u.height                    ▷ gespeicherte Höhe zurückliefern
 6:     end if
 7: end procedure
```

Listing 4.8: Height

Algorithmus 4.9 zeigt den Pseudocode für das Einfügen eines Knotens in einen AVL-Baum. In Algorithmus 4.10 bzw. 4.11 werden die *Rechts-Rotation* bzw. die *Links-Rechts-Rotation* in Pseudocode dargestellt.

Anmerkung 4.2. Der beim natürlichen Suchbaum eingeführte Verweis *parent* auf den Elternknoten eines Knoten ist hier aus Gründen der Übersichtlichkeit nicht berücksichtigt.

Beispiel für das Entfernen eines Elementes aus einem AVL-Baum. Wir betrachten den AVL-Baum in Abb. 4.13(a). Wir wollen Knoten 9 entfernen. Dabei soll die AVL-Eigenschaft erhalten bleiben.

1. Schritt: Entferne Knoten 9 wie bei natürlichen binären Suchbäumen.
2. Schritt: Rebalancierung. Diese wird im Folgenden ausgeführt.

Der nicht balancierte Knoten maximaler Tiefe ist u mit Schlüssel 10. Da $bal(u) = 2$, trifft Fall 2 zu. Sei v das rechte Kind von u. Da $bal(v) = -1$, trifft Fall 2.2 ein. Sei w linkes Kind von v. Wir führen eine Rechts-Links-Rotation aus, d.h. zunächst eine einfache Rotation nach rechts an v, dann eine Rotation nach links an u aus (s. Abb. 4.13(b)).

Es ergibt sich der in Abb. 4.13(c) dargestellte Baum.

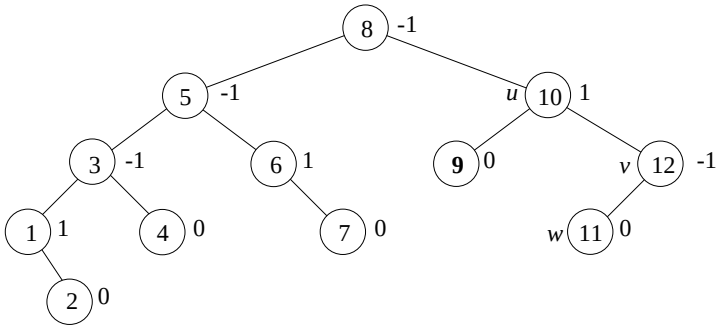
Obwohl der Knoten 8 (u' in Abb. 4.13(c)) vor der Rebalancierung balanciert war, ist er nun, nach der Rebalancierung von u, nicht mehr balanciert. Die Rebalancierung geht also weiter. Auf u' trifft Fall 1 zu, denn $bal(u') = -2$. Sei v' das linke Kind von u'. Da $bal(v') = -1$, trifft Fall 1.1 zu. Zur Rebalancierung von u' genügt eine Rotation nach rechts an u' (siehe Abb. 4.13(d)).

Analyse. Rotationen und Doppelrotationen können in konstanter Zeit ausgeführt werden, da nur konstant viele Zeiger umgehängt werden müssen. Wir führen eine Rebalancierung höchstens einmal an jedem Knoten auf dem Pfad vom eingefügten oder gelöschten Knoten zur Wurzel durch. Daraus folgt:

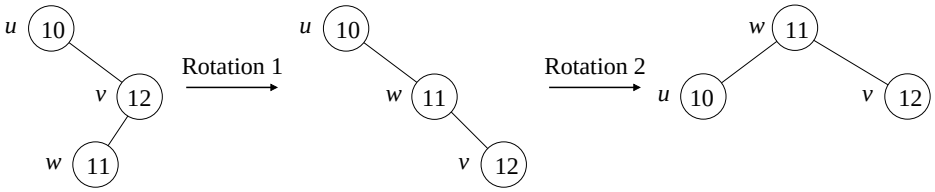
Eingabe: Baum mit Wurzel p; Schlüssel k und Wert v

```
1:  procedure INSERTAVL(ref p, k, v)
2:      ▷ Fügt Wert v mit Schlüssel k in AVL-Baum mit Wurzel p ein und führt Rebalancierung durch
3:      var TreeNode q
4:      if p = nil then                                          ▷ Füge neuen Knoten ein
5:          p := new TreeNode
6:          p.key := k
7:          p.info := v
8:          p.height := 0
9:      else
10:         if k < p.key then
11:             INSERTAVL(p.left, k, v)
12:             if HEIGHT(p.right) − HEIGHT(p.left) = −2 then
13:                 if HEIGHT(p.left.left) > HEIGHT(p.left.right) then
14:                     ▷ linker Unterbaum von p.left ist höher
15:                     p := ROTATERIGHT (p)                     ▷ Fall 1.1
16:                 else
17:                     p := ROTATELEFTRIGHT (p)                 ▷ Fall 1.2
18:                 end if
19:             end if
20:         else
21:             if k > p.key then
22:                 INSERTAVL(p.right, k, v)
23:                 if HEIGHT(p.right) − HEIGHT(p.left) = 2 then
24:                     if HEIGHT(p.right.right) > HEIGHT(p.right.left) then
25:                         p := ROTATELEFT (p)
26:                     else
27:                         p := ROTATERIGHTLEFT (p)
28:                     end if
29:                 end if
30:             else
31:                 ▷ Überschreibe Wert an Knoten p
32:                 p.info := v
33:             end if
34:         end if
35:         p.height := max (HEIGHT(p.left), HEIGHT(p.right)) + 1
36:     end if
37: end procedure
```

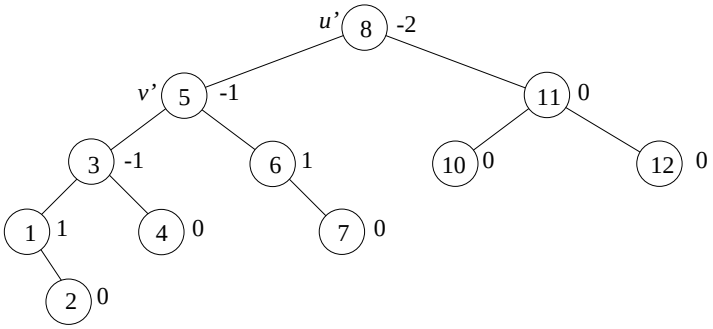
Listing 4.9: Einfügen in einen AVL-Baum



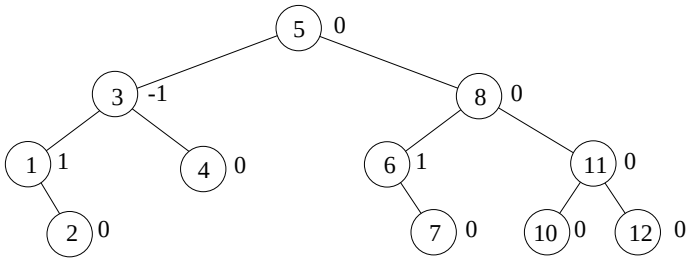
(a) Ein AVL-Baum, aus dem wir Knoten 9 entfernen wollen



(b) Rechts-Links-Rotation an u



(c) Nach der ersten Rotation ist der Baum immer noch nicht balanciert.



(d) Nach der Rotation nach rechts ist der Baum wieder balanciert.

Abbildung 4.13: Löschen in einem AVL-Baum an einem Beispiel

```
Eingabe: Unterbaum mit Wurzel u
Ausgabe: Neue Wurzel v des rebalancierten Unterbaums
 1: function RotateRight(TreeNode u) : TreeNode
 2:     v := u.left
 3:     u.left := v.right                          ▷ u bekommt das rechte Kind von v
 4:     v.right := u                               ▷ u wird zum neuen rechten Kind von v
 5:     ▷ Berechne die Höhen der betroffenen Knoten neu
 6:     u.height := max (Height(u.left), Height(u.right)) + 1
 7:     v.height := max (Height(v.left), Height(u)) + 1
 8:     return v                                   ▷ Liefere die neue Wurzel des Unterbaums, v, zurück
 9: end function
```

Listing 4.10: Rechts-Rotation in AVL-Bäumen

```
Eingabe: Unterbaum mit Wurzel u
Ausgabe: Neue Wurzel des rebalancierten Teilbaums
 1: function RotateLeftRight(u) : TreeNode
 2:     u.left := RotateLeft(u.left)
 3:     return RotateRight(u)
 4: end function
```

Listing 4.11: Links-Rechts-Rotation

Das Einfügen und Entfernen ist in einem AVL-Baum in Zeit $O(\log n)$ möglich.

AVL-Bäume unterstützen also die Operationen Einfügen, Entfernen, Minimum, Maximum, Successor, Predecessor, Suchen in Zeit $O(\log n)$.

Diskussion. Wenn oft gesucht und selten umgeordnet wird, sind AVL-Bäume günstiger als natürliche binäre Suchbäume. Falls jedoch fast jeder Zugriff ein Einfügen oder Entfernen ist, und man davon ausgehen kann, dass die Daten meist gut verteilt und nicht teilweise vorsortiert sind, dann sind natürliche binäre Suchbäumevorzuziehen.

4.4 B-Bäume

Mit den AVL-Bäumen haben wir eine Methode kennengelernt, die Entartung von binären Suchbäumen zu verhindern. Außer dieser Methode gibt es noch weitere Ansätze Suchbäume mit garantierter Balacierung zu generieren; dazu werden wir uns von der Idee des binären Suchbaums trennen und mehrere Schlüssel pro Knoten zulassen.

Dieser Grundgedanke führte 1970 zur Entwicklung der *2-3-Bäumen* durch J.E. Hopcroft, bei denen jeder innere Knoten entweder 2 oder 3 Kinder besitzt. Teilweise findet man auch den

analog definierten *2-3-4-Baum* in der Literatur als eigenständiges Konzept. Interessanterweise kann man diese Knoten mit 2 bis 4 Kindern ihrerseits wieder durch binäre Teilbäume simulieren, was zu den sogenannten *Rot-Schwarz-Bäumen* führt. Der Vorteil all dieser Abarten gegenüber den AVL-Bäumen besteht darin, dass sie nicht nur eine lokale, sondern sogar eine globale Balancierung garantieren, d.h. alle Blätter haben die gleiche Tiefe.

Wir werden uns in Folge mit den B-Bäumen beschäftigen, die eine Verallgemeinerung der 2-3-, und 2-3-4-Bäume darstellen. Sie sind vorallem deswegen so interessant, weil sie in der Praxis sehr weit verbreitet sind und beispielsweise bei Datenbanken und Dateisystemen zum Einsatz kommen.

Es gibt zahlreiche Variationen dieser Baumstrukturen, von denen der B^+-Baum wohl eine der interessantesten ist, so dass wir diesen in Abschnitt 4.4.2 etwas näher betrachten werden.

Extern. Bevor wir uns der formalen Definition des B-Baums zuwenden, überlegen wir uns, was es eigentlich bedeutet, einen balancierten Baum zu haben, dessen Knoten viele Schlüssel bzw. Kinder besitzen.

Die in den vorigen Abschnitten behandelten Suchmethoden waren für ,,internes" Suchen gut geeignet, d.h. die Daten, in denen gesucht wird, finden vollständig im Hauptspeicher Platz. Oft, z.B. bei Datenbanken und Dateisystemen, ist dies jedoch nicht der Fall. Wenn wir also in einem einfachen binären Baum suchen und dabei von einem Knoten zu einem seiner Kinder springen, kann es uns passieren, dass wir für jeden dieser Schritte einen Zugriff auf langsame Festplatten durchführen müssen. Wie schon im Abschnitt über externes Sortieren ausgeführt, ist ein solcher Zugriff um mehrere Größenordnungen langsamer als ein Zugriff auf Elemente im Hauptspeicher. Auf der anderen Seite wird bei einem Lesezugriff auf die Festplatte nicht nur ein einzelner Datensatz geladen, sondern immer ein gesamter Block, auch Speicherseite oder *page* genannt.

Im Falle eines binären Baums hilft uns das Laden einer solchen kompletten Seite in der Regel wenig, da das nächste Kind das wir besuchen sehr gut auf einer ganz anderen Speicherseite liegen kann. Der Trick des B-Baums ist es nun, die Knotengrössen gerade so zu wählen, dass sie gerade noch in genau eine Speicherseite hineinpassen; so sind zum Beispiel, abhängig von der Schlüssel- und Datensatzgrösse, B-Bäume mit 100 bis 5000 Schlüsseln pro Knoten keine Seltenheit.

Wenn wir nun in einem B-Baum absteigen, können wir tatsächlich die gesamte geladene Speicherseite nutzen und intern in diesem Knoten arbeiten, d.h. z.B. den nächsten zu untersuchenden Unterbaum finden. Und da der Baum viel breiter wird, sinkt natürlich auch seine Höhe! Wir werden die obere und untere Schranke der Höhe eines B-Baums später noch formal beweisen, doch nehmen wir zunächst für einen groben Vergleich an, dass ein vollständig balancierter(!) binärer Suchbaum im Wesentlichen die Höhe $\log_2 n$ hat, wohingegen jeder B-Baum eine Höhe von ca. $\log_m n$ garantiert. Wollen wir nun ein Element aus 1 Million

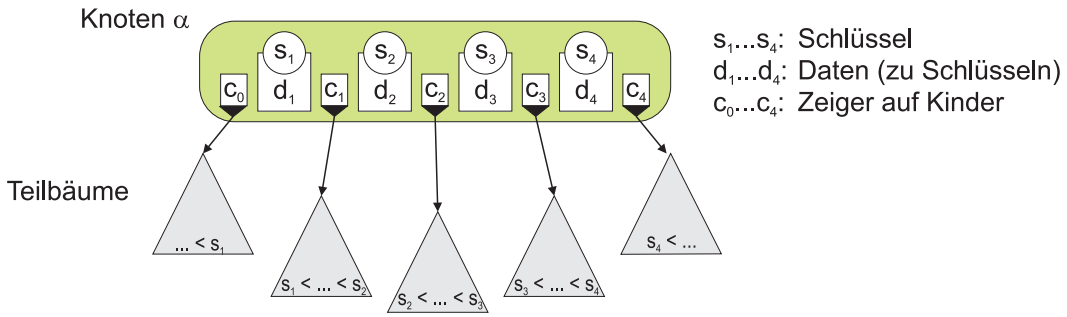


Abbildung 4.14: Ein B-Baum Knoten mit 4 Schlüsseln hat 5 Kinder.

Datensätzen suchen, so bedeutet das bei einem binären Baum eine Höhe von ca. 20; ein B-Baum der Ordnung 100 wird hingegen nur 2 externe Speicherzugriffe benötigen (da man animmt, dass der Wurzelknoten immer im Hauptspeicher gehalten wird).

4.4.1 B-Bäume

B-Bäume wurden 1972 von Bayer und McCreight entwickelt, ohne allerdings die Namenswahl zu erklären. Die Kernidee besteht darin, dass ein Knoten eines B-Baums der Ordnung m immer zwischen $\lceil m/2 \rceil$ und m Kinder besitzt. Da diese Ordnung m eine Konstante ist, kann man den Aufwand der Algorithmenteile, die ausschließlich innerhalb eines Knotens arbeiten als konstant betrachten. Die oben kurz erwähnten 2-3- und 2-3-4-Bäume entsprechen genau B-Bäumen der Ordnung 3 bzw. 4.

Überlegen wir uns nun zunächst, was es bedeutet, einen Knoten α mit $k+1$ Kindern zu haben: Innerhalb unseres Knotens müssen wir dazu k Schlüssel in sortierter Reihenfolge speichern (wir gehen im Weiteren einfachheitshalber davon aus, dass kein Schlüssel mehrfach vorkommt). Jedes Kind ist wie in einem binären Baum die Wurzel eines Unterbaums; wir müssen sicherstellen, dass die Schlüssel von α die verschiedenen Schlüssel der jeweiligen Unterbäume voneinander trennen (vergl. Abb. 4.14). Abb. 4.15 zeigt einen B-Baum der Ordnung 4.

Formal definieren wir einen B-Baum wie folgt, vergl. Abb. 4.14 und 4.15:

Definition 4.3. Ein B-Baum der Ordnung $m > 2$ ist ein Baum mit folgenden Eigenschaften:

B1: Kein Schlüsselwert ist mehrfach im Baum enthalten.

B2: Für jeden Knoten α mit $k+1$ Zeigern auf Unterbäume gilt (seien $c_0, c_1, \ldots, c_k$ die Wurzeln der nicht-leeren Unterbäume):

a) α hat k Schlüssel $s_1, s_2, \ldots, s_k$, wobei gilt: $s_1 < s_2 < \cdots < s_k$.

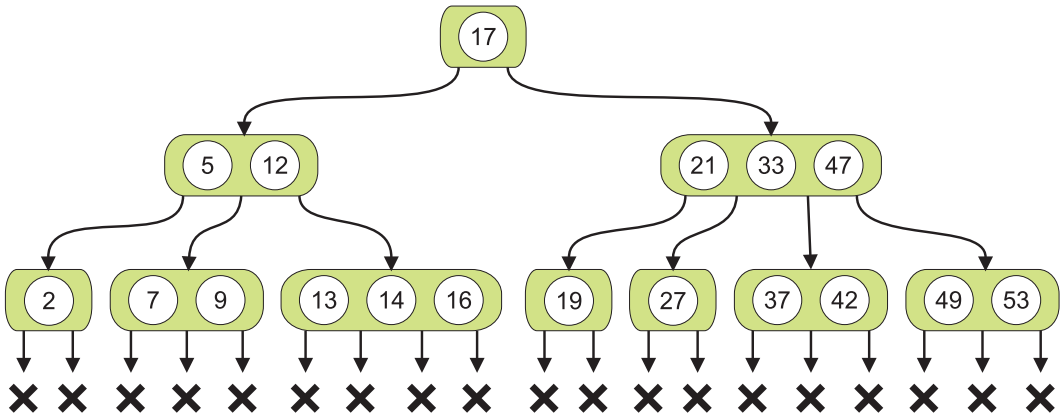


Abbildung 4.15: Beispiel eines B-Baums der Ordnung 4.

b) Für jeden Schlüssel s im Unterbaum mit Wurzel c_i gilt $s_i < s < s_{i+1}$, wobei $s_0 = -\infty$ und $s_{k+1} = +\infty$.

c) Ein Schlüssel s_i wird als *Trennschlüssel* der Unterbäume mit Wurzel c_{i-1} und c_i bezeichnet.

B3: Der Baum ist leer, oder die Wurzel hat mindestens 2 Zeiger.

B4: Jeder Knoten mit Ausnahme der Wurzel enthält mindestens $\lceil \frac{m}{2} \rceil$ Zeiger.

B5: Jeder Knoten hat höchstens m Zeiger.

B6: Alle Blätter haben die gleiche Tiefe.

Anmerkung 4.3. In der Realität werden zumeist zwei verschiedene Typen von Knoten benutzt: Innere Knoten mit Zeigern auf Kinder, und Blattknoten, ohne jegliche Zeiger. Dadurch erspart man sich die vielen recht nutzlosen *nil*-Zeiger in den Blättern. Der Übersichtlichkeit halber verzichten wir in diesem Skript auf diese Unterscheidung.

Bevor wir uns der Erklärung zuwenden, wie sich ein solcher Baum realisieren lässt und wie wir in ihm effizient Einfügen und Entfernen können, betrachten wir die Eigenschaften eines Baums, der obigen Anforderungen entspricht.

Zunächst einmal wollen wir zeigen, dass überhaupt für jede Schlüsselmenge ein Baum mit den geforderten Eigenschaften existiert. Bevor wir dies zeigen können (Theorem 4.2), zeigen wir zunächst eine schwächere Aussage: Wir wollen zunächst zeigen, dass wir aus einer Menge von Schlüsseln eine Menge von Trennschlüsseln finden können, so dass alle anderen Schlüssel („die S-Mengen") jeweils in einen B-Baum-Knoten passen.

Lemma 4.1. *Gegeben sei eine Menge M aus n Schlüsseln und eine natürliche Zahl $m \geq 3$, mit $n \geq m$. Wir setzen $\ell := \lfloor \frac{n}{m} \rfloor$. Wir können ℓ Trennschlüssel aus M so auswählen, dass die*

übrigen Schlüssel in $\ell+1$ Teilmengen $S_0, \ldots, S_\ell$ zerfallen, die jeweils eine gültige Befüllung eines Knotens in einem B-Baum der Ordnung m darstellen.

Beweis. Der zu beweisende Kern dieses Lemmas besteht eigentlich nur darin, zu zeigen, dass die Kardinalität der auftretenden Mengen weder zu klein, noch zu groß für einen B-Baum-Knoten sind. Die tatsächliche Auswahl der Trennschlüssel gemäß ihrem Wert ist dann trivial.

Auf jeden Fall muss für jede Menge S_i gelten, dass

$$\left\lceil \frac{m}{2} \right\rceil - 1 \leq |S_i| \leq m - 1.$$

Entsprechend ist klar, dass wir versuchen werden, alle S-Mengen möglichst gleich voll zu befüllen. Berechnen wir die durchschnittlich notwendige Größen der S-Mengen, so ist dies $\frac{n+1}{l+1} - 1$; wenn dieser Term keine natürliche Zahl liefert, haben wir Mengen, bei denen dieser Wert aufgerundet interpretiert werden muss, und andere bei denen wir abrunden.

- Wenn wir Aufrunden, muss gelten:

$$\left\lceil \frac{n+1}{\ell+1} - 1 \right\rceil \leq m - 1 \iff \left\lceil \frac{n+1}{\ell+1} \right\rceil \leq m.$$

  Diese Ungleichung ist am *kritischsten*, wenn wir für relativ viele Elemente relativ wenige Trennschlüssel benutzen: Wenn unser n so groß gewählt ist, dass wir bei der Berechnung von ℓ *gerade noch* abrunden müssen, so bedeutet dies, dass wir sehr wenige Trennschlüssel auswählen dürfen, und damit die durchschnittliche Grösse der S-Mengen ihr relatives Maximum erreicht. Wenn wir zeigen können, dass selbst in diesem ungünstigsten Fall die S-Mengen nicht zu groß für einen B-Baum-Knoten werden können, so ist gezeigt, dass sie nie zu groß werden.

  Dieses größtmögliche n läßt sich auch so beschreiben, dass, sobald wir auch nur einen Schlüssel mehr besitzen würden, wir schon einen weiteren Trennschlüssel auswählen dürften, also wenn $\ell + 1 = \frac{n+1}{m} \in \mathbb{N}$.

  Dann können wir den linken Teil der Ungleichung aber umformen zu

$$\left\lceil \frac{n+1}{\frac{n+1}{m}} \right\rceil = \left\lceil m \frac{n+1}{n+1} \right\rceil = m;$$

  die Ungleichung stimmt also offensichtlich.

- Wenn wir abrunden muss gelten:

$$\left\lceil \frac{m}{2} \right\rceil - 1 \leq \left\lfloor \frac{n+1}{\ell+1} - 1 \right\rfloor \iff \left\lceil \frac{m}{2} \right\rceil \leq \left\lfloor \frac{n+1}{\ell+1} \right\rfloor,$$

Diese Ungleichung ist am kritischsten, wenn wir relative viele Trennschlüssel benutzen, und dadurch unsere S-Mengen im Durchschnitt klein sein werden. Dies ist genau dann der Fall, wenn wir bei der Berechnung von ℓ nicht abrunden müssen, also wenn $\ell = \frac{n}{m} \in \mathbb{N}$, bzw. $n = \ell m$. Wenn wir also zeigen dass die Ungleichung

$$\left\lceil \frac{m}{2} \right\rceil \leq \left\lfloor \frac{\ell m + 1}{\ell + 1} \right\rfloor \tag{4.1}$$

stimmt, so werden die Teilmengen nie kleiner als für einen B-Baum-Knoten zulässig. Diese Ungleichung ist auf jeden Fall dann gültig, wenn

$$\frac{m}{2} + 1 \leq \frac{\ell m + 1}{\ell + 1} \iff 2\ell + m \leq m\ell \iff \ell \geq \frac{m}{m-2}.$$

Da $m \geq 3$, gilt dies für alle $\ell \geq 3$; ist $m \geq 4$ gilt es auch für alle $\ell \geq 2$.

Für den Fall, dass $m = 3$ und $\ell = 2$ lässt sich die Gültigkeit der Ungleichung 4.1 durch einfaches Einsetzen validieren.

Betrachten wir den Fall, dass $\ell = 1$, so führt die Ungleichung 4.1 zu

$$\left\lceil \frac{m}{2} \right\rceil \leq \left\lfloor \frac{m+1}{2} \right\rfloor.$$

Dies ist natürlich immer erfüllt, da entweder der linke oder der rechte Ausdruck auch ohne Rundung eine natürliche Zahl liefert. □

Mit diesem Lemma ausgerüstet können wir nun zeigen:

Theorem 4.2 (Existenz)**.** *Für jede beliebige Menge von Schlüsseln existiert ein gültiger B-Baum der Ordnung m.*

Beweis. Wir zeigen dies schrittweise: Besteht unsere Schlüsselmenge aus weniger als m Schlüsseln, so besteht der B-Baum einfach aus einem einzigen Wurzelknoten, der alle Schlüssel enthält.

Besteht unsere Schlüsselmenge aus mindestens m Schlüsseln, so zerlegen wir die Schlüsselmenge wie in Lemma 4.1 beschreiben. Die $\ell + 1$ S-Teilmengen bilden die Blätter unseres B-Baums, und wir verfügen über die korrekte Anzahl (nämlich ℓ) an Trennschlüsseln. Nun müssen wir für diese Trennschlüssel ihrerseits einen B-Baum erzeugen: ist $\ell < m$ so sind wir fertig, da wir alle Trennschlüssel in einen Wurzelknoten packen können. Ansonsten müssen wir wieder eine Aufteilung dieser – nun viel kleineren – Menge vornehmen, etc. □

Lemma 4.2 (Größe)**.** *Die Größe eines B-Baums ist linear in n, der Anzahl der Schlüssel.*

Beweis. Da jeder Schlüssel im Baum vorkommt, gilt offensichlich $\Omega(n)$ für seine Größe. Sei t die Anzahl der Knoten im Baum. Da jeder Knoten mindestens einen Schlüssel enthält, gilt $t = O(n)$ Knoten. Da in jedem Knoten mit k Schlüsseln gilt, dass er $k + 1$ Zeiger enthält, haben wir insgesamt $n + t = O(n)$ viele Zeiger. Aus $\Omega(n)$ und $O(n)$ folgt also, dass der Baum linear viel Speicher (in der Anzahl der Schlüssel) benötigt. □

Lemma 4.3 (Minimale Höhe)**.** *Die minimale Höhe eines B-Baums mit n Schlüsseln ist*

$$\lceil \log_m(n+1) \rceil - 1.$$

Beweis. Ein B-Baum hat die geringste Höhe, wenn alle Knoten möglichst gefüllt sind, d.h. $m-1$ Schlüssel enthalten; bei einer Höhe h enthält er dann m^h Blätter. Wenn ein B-Baum $\ell+1$ Blätter besitzt, so enthält er ℓ Schlüssel in inneren Knoten (vergl. Beweis zu Theorem 4.2). Die Anzahl der Schlüssel in einem solchen Baum (mit gegebener Höhe $h \geq 1$) ist dann

$$n_{max} = \underbrace{m^h(m-1)}_{\text{Schlüssel in Blättern}} + \underbrace{(m^h - 1)}_{\text{Schlüssel in inneren Knoten}} = m^{h+1} - 1.$$

Wenn wir dies nach h umformen, erhalten wir die Aussage des Lemmas. □

Lemma 4.4 (Maximale Höhe)**.** *Die maximale Höhe eines B-Baums mit n Schlüsseln ist*

$$\left\lfloor \log_{\lceil \frac{m}{2} \rceil} \left(\frac{n+1}{2} \right) \right\rfloor.$$

Beweis. Ein B-Baum hat die größte Höhe, wenn alle Knoten möglichst wenig gefüllt sind, d.h. die Wurzel enthält nur einen einzigen Schlüssel, die zwei Unterbäume der Kindknoten sind vollständige Bäume mit jeweils nur $\left\lceil \frac{m}{2} \right\rceil - 1$ Schlüsseln pro Knoten. Die Anzahl der Schlüssel in einem solchen Baum (mit gegebener Höhe $h \geq 1$) ist dann

$$n_{min} = 1 + \underbrace{2\left(\left\lceil \frac{m}{2} \right\rceil^h - 1 \right)}_{\text{Schlüssel in den 2 Unterbäumen}} = 2\left\lceil \frac{m}{2} \right\rceil^h - 1.$$

Wenn wir dies nach h umformen, erhalten wir die Aussage des Lemmas. □

Aus den beiden letzten Lemmata ergibt sich die für balancierte Suchbäume typische Eigenschaft, nämlich dass ihre Höhe logarithmisch in der Anzahl der Schlüssel beschränkt ist.

Datenstruktur. Wenden wir uns im Weiteren nun den Operationen zu, die wir auf einem solchen B-Baum durchführen möchten. Zunächst definieren wir die Datenstruktur für einen Knoten wie in Listing 4.12. Als Alternative zu den Arrays sind natürlich auch lineare Listen vorstellbar.

```
 1: struct BTreeNode
 2:     var int k                                          ▷ Anzahl der Schlüssel
 3:     var KeyType key[1 ... k]                           ▷ Die Schlüssel im Knoten (sortiert)
 4:     var DataType data[1 ... k]                         ▷ Die mit den Schlüsseln verknüpften Daten
 5:     var BTreeNode child[0 ... k]                       ▷ Die Kinder des Knotens
 6: end struct
 7: ▷ Interne Repräsentation eines B-Baums
 8: var BTreeNode root                                     ▷ Wurzel des B-Baums
 9: ▷ Initialisierung
10: root = nil
```

Listing 4.12: Datenstruktur eines B-Baums.

Suchen. Die Suche in einem B-Baum (vergl. Listing 4.13) funktioniert im Wesentlichen genau so, wie in einem binären Baum: Wir starten bei der Wurzel und überprüfen die Schlüssel des Knotens. Sollten wir den Eintrag nicht gefunden haben, folgen wir dem Zeiger der sich zwischen den beiden im Wurzelknoten benachbarten Schlüsseln befindet, die den gesuchten Wert umgrenzen. Dies führen wir solange fort, bis wir entweder das gesuchte Element gefunden haben, oder auf einen *nil*-Zeiger stoßen.

In der im Listing 4.13 gezeigten Implementierung wird innerhalb eines Knotens eine lineare Suche durchgeführt. Bei B-Bäumen hoher Ordnung wird man in der Realität natürlich

```
Eingabe: B-Baum mit Wurzel p; zu suchender Schlüssel x
Ausgabe: Die Daten die zum Schlüssel x gespeichert sind oder nil, falls x nicht existiert
 1: procedure SEARCH(BTreeNode p, KeyType x)
 2:     if p = nil then
 3:         return nil                                     ▷ Suche erfolglos
 4:     else
 5:         var int i := 1
 6:         while i ≤ p.k and x > p.key[i] do
 7:             i := i + 1
 8:         end while
 9:         if i ≤ p.k and x = p.key[i] then
10:             return p.data[i]                           ▷ Schlüssel gefunden
11:         else
12:             return SEARCH(p.child[i − 1],x)            ▷ rekursiv weitersuchen
13:         end if
14:     end if
15: end procedure
```

Listing 4.13: Suche in einem B-Baum.

auf eine binäre Suche zurückgreifen. Es ist interessant anzumerken, dass sich, da m eine Konstante ist, beide Ansätze nicht in der O-Notation unterscheiden!

Insgesamt garantiert uns ein B-Baum logarithmische Höhe. Da wir den Baum nur maximal einmal von oben bis unten durchlaufen, und sich der Aufwand innerhalb eines Knotens mit einer Konstante abschätzen lässt (die Ordnung des Baums ist ja konstant), führt dies zu einer Laufzeitabschätzung von $O(\log n)$.

Einfügen. Ein Element in einen B-Baum einzufügen ist etwas aufwendiger als in einem einfachen binären Suchbaum. Klarerweise ist es nicht einfach möglich einen neuen Blattknoten zu erzeugen und unter ein schon bestehendes Blatt zu hängen, da dies die globale Balancierung (Eigenschaft B6:) zerstören würde.

Wir lösen dieses Problem wie im Folgenden beschrieben; Abb. 4.16 zeigt beispielhaft einen solchen Einfügevorgang in einem B-Baum der Ordnung 3.

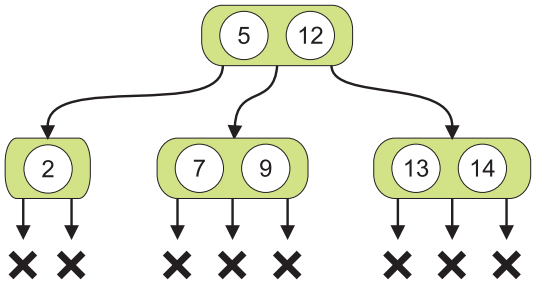
1. **Einfügeposition suchen.** Zunächst suchen wir das richtige Blatt, in dem der neue Wert x zu finden wäre, würde er schon im Baum enthalten sein. Dies entspricht genau unserer Funktion SEARCH, wobei wir das letzte besuchte Blatt zurückgeben, anstelle des *nil*-Zeigers. Sollten wir wider Erwarten das Element dabei schon im Baum finden, so ist kein Einfügen notwendig (Eigenschaft B1:).

2. **Einfügen in Blatt.** Nun fügen wir zunächst den neuen Schlüssel einfach in dieses Blatt ein. Damit ist das Element zwar eingefügt und Eigenschaft B6: erfüllt, doch könnte es dadurch geschehen sein, dass der Knoten ein Element zu viel enthält (B5:)[2].

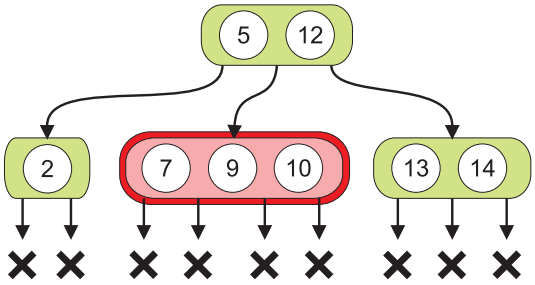
3. **Aufspalten.** Der nächste Schritt liegt nun also darin, den Baum zu „reparieren". Die Idee besteht immer darin, einen zu grossen Knoten in zwei kleinere aufzubrechen, und den „mittleren" Schlüssel als Trennschlüssel dieser beiden neuen Knoten zu benutzen (dass eine solche Aufteilung möglich ist, folgt z.B. aus Lemma 4.1). Doch was machen wir mit diesem Trennschlüssel? Er kann sicherlich keinen einzelnen Knoten bilden, da in B-Bäumen mit Ordnung 5 und höher immer mindestens zwei Schlüssel pro Knoten existieren müssen. Und selbst wenn wir einen einelementigen Knoten erzeugen dürften, wäre der resultierende B-Baum dann wieder nicht global balanciert (Eigenschaft B6:, vergl. Abb. 4.16(c))!

   Es bietet sich aber einfach an, diesen trennenden Schlüssel einfach in den Elterknoten zu verschieben. Da dieser dann einen Schlüssel mehr besitzt, bietet er auch automatisch genug Platz für die zwei neuen Kindzeiger (vergl. Abb. 4.16(d)). Allerdings muss uns klar sein, dass wir damit das Problem ggf. nur verschoben haben: Es kann durch diesen zusätzlichen Schlüssel nun passiert sein, dass der Elterknoten seinerseits zu viele Schlüssel enthält.

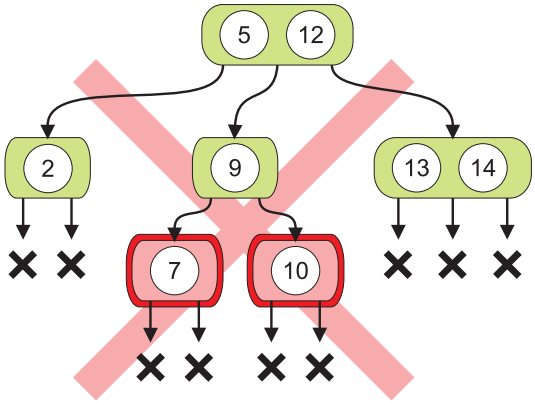
[2] In einer realen Implementierung wird man den Schlüssel aus Platzgründen gar nicht tatsächlich einfügen können, sondern getrennt gespeichert halten bis die Situation wie im Folgenden beschrieben gelöst ist. Für die Vorstellung genügt es aber, den Knoten kurzzeitig „zu voll" zu befüllen.



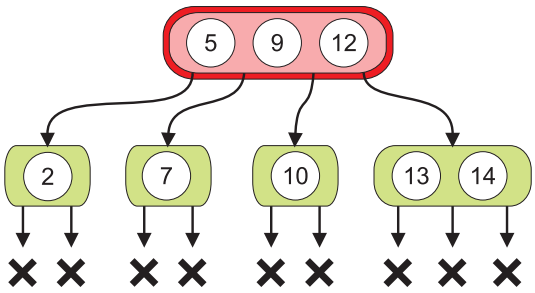
(a) Der Baum vor dem Einfügen.



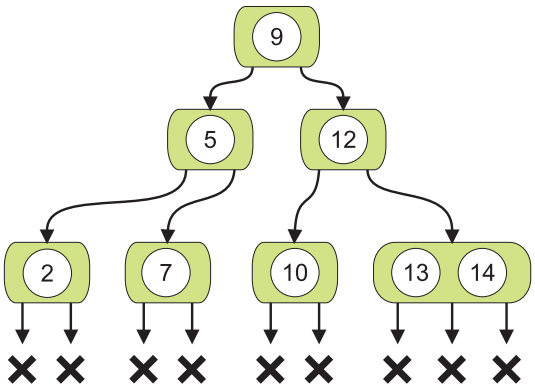
(b) Neuen Schlüssel „10” in richtiges Blatt einfügen → Blatt wird zu groß.



(c) Falsches Aufspalten des Knotens: Die Blätter hätten nicht mehr alle die gleiche Tiefe.



(d) Richtiges Aufspalten: der mittlere Schlüssel („9”) wandert in den Elterknoten. Nun ist dieser Elterknoten zu voll.



(e) Aufspalten der bisherigen Wurzel, erzeugt eine neue Wurzel. Dadurch wächst der Baum um eine Ebene.

Abbildung 4.16: Einfügen des Schlüssels „10” in einen B-Baum der Ordnung 3.

4. **Aufspaltung rekursiv fortsetzen.** Was wir also tun, ist nun ggf. diesen Elterknoten wieder in zwei Teilknoten mit einem Trennschlüssel zu zerlegen, und diesen neuen Trennschlüssel in dessen Elterknoten zu verschieben, usw. Wir müssen dies unter Umständen solange fortsetzen, bis schließlich kein Vorgängerknoten mehr existiert — also bis wir im Wurzelknoten angekommen sind — und der Wurzelknoten zu viele Elemente enthält.

5. **Sonderfall Wurzel.** Genau an dieser Stelle kommt uns aber die Eigenschaft B3: zur Hilfe: ein B-Baum darf durchaus eine Wurzel mit nur einem Schlüssel, und damit zwei Kindzeigern haben. Also ist es kein Problem, den Wurzelknoten in zwei Teilknoten aufzuspalten und den trennenden Schlüssel in einen neu erzeugten Wurzelknoten zu verschieben. Diese neue Wurzel hält dann die Zeiger auf die beiden eben aufgespaltenen Teilknoten der ehemaligen Wurzel.

Nach diesem Vorgang, ist der neue Schlüssel x eingefügt, und alle Bedingungen des B-Baums sind weiterhin erfüllt. So bleibt nun noch die Laufzeit der Operation zu untersuchen: Zunächst sind wir mit einer einfachen Suche von der Wurzel bis zu einem Blatt abgestiegen, und haben das Element dort eingefügt, also $\Theta(\log(n))$. Danach mussten wir die Eigenschaft B5: wiederherstellen, und haben dazu im ungünstigsten Fall wieder von diesem Blatt bis zur Wurzel aufsteigen müssen, also nochmals $O(\log(n))$. Die Arbeit die wir an einem einzelnen Knoten zu verrichten hatten (Untersuchen, Aufspalten, etc.) lässt sich immer durch eine Konstante abschätzen. Dadurch erhalten wir in Summe die Laufzeitgarantie $\Theta(\log(n))$ für die gesamte Einfügeoperation.

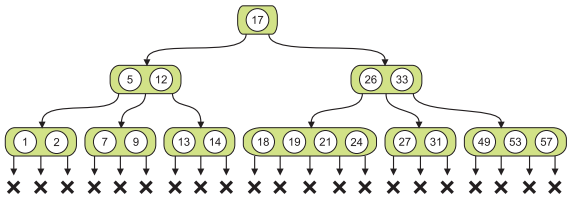
Anmerkung 4.4. Wir erkennen: Ein B-Baum wächst nicht wie ein traditioneller binärer Baum nach unten (durch neue Blätter), sondern *nach oben*, durch neue Wurzelknoten.

Entfernen. Beim Entfernen eines Elements aus einem B-Baum haben wir genau die umgekehrten Probleme im Vergleich zum Einfügen: Würden wir einen Schlüssel einfach löschen, wäre nicht sichergestellt, dass jeder Knoten noch genügend Schlüssel enthält, um Eigenschaft B4: zu erfüllen. Darüberhinaus hätten wir, wenn sich der Schlüssel in einem inneren Knoten befand, das Problem, dass nun zu viele Kindzeiger existieren würden.

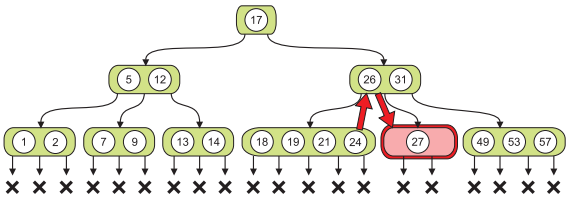
Die nun folgende Beschreibung der korrekten Entfern-Funktion kann auch in Abb. 4.17 nachvollzogen werden.

1. **Schlüssel suchen.** Zunächst müssen wir den zu löschenden Schlüssel x wieder einfach suchen. Wir unterscheiden, ob wir ihn in einem Blatt oder in einem inneren Knoten finden:

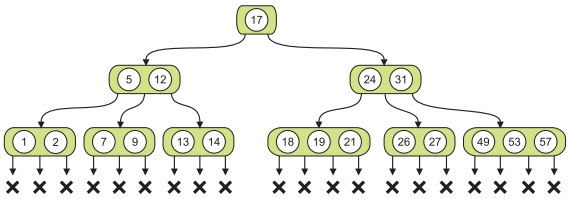
   a) **Schlüssel in Blatt.** Befindet sich der Schlüssel in einem Blatt, so werden wir ihn einfach — zusammen mit einem dann überzähligen *nil*-Zeiger — aus diesem entfernen.



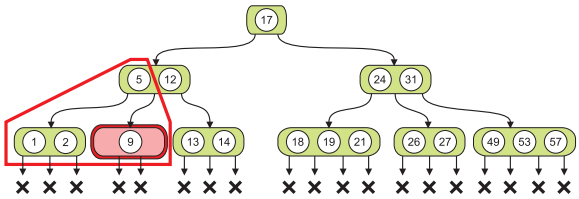
(a) Der Baum vor dem Entfernen.



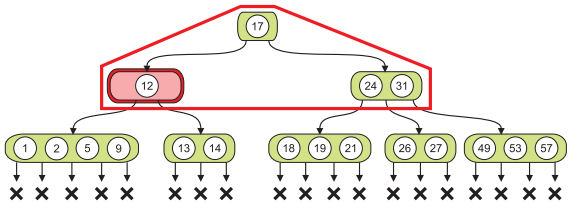
(b) Entfernen des Schlüssels „33". Wir setzen seinen Vorgänger „31" an dessen Position. → Blatt wird zu klein. Beide direkten Nachbarn sind groß genug für eine Rotation.



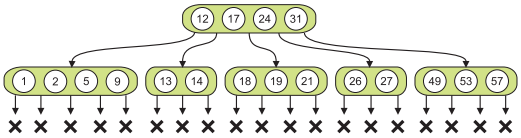
(c) B-Baum nach der Rotation. Das Enfernen ist abgeschlossen.



(d) Enfernen des Schlüssels „7". Blatt wird zu klein. Beide Nachbarn sind zu klein für eine Rotation. Wir müssen verschmelzen.



(e) Nach dem Verschmelzen ist nun der Elterknoten zu klein geworden. Sein Nachbar ist zu klein für eine Rotation. → Abermals verschmelzen.



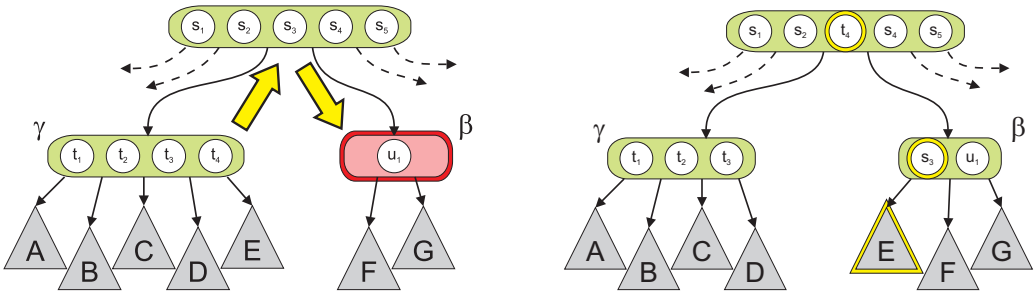
(f) Das Verschmelzen reduziert den Baum um seine bisherige Wurzel. Der Baum wird um eine Ebene kleiner. Das Entfernen ist abgeschlossen.

Abbildung 4.17: Entfernen des Schlüssels „33" und „7" in einen B-Baum der Ordnung 5.

b) **Schlüssel in innerem Knoten.** Befindet sich der Schlüssel in einem inneren Knoten, erinnern wir uns an das Entfernen aus einem einfachen binären Suchbaum: wir konnten einen inneren Knoten nicht einfach löschen, sondern haben zunächst seinen Vorgänger gesucht und diesen dann an die Stelle des zu löschenden Elements verschoben.

Ganz analog gehen wir auch in einem B-Baum vor: Wir suchen nach seinem direkten Vorgänger indem wir einfach zu dem Kind „direkt links" des Schlüssels absteigen und dann immer das rechteste Kind nehmen, bis wir auf ein Blatt stoßen: der größte Schlüssel in diesem Blatt ist der direkte Vorgänger (gemäß der Sortierung) von x. Wir entfernen nun diesen Vorgänger aus diesem Blatt und setzen ihn an die Position von x. Dadurch ist der innere Knoten weiterhin korrekt.

2. **Ein Blatt hat einen Schlüssel weniger.** Unabhängig an welcher Stelle wir den Schlüssel gefunden haben, haben wir schlussendlich immer den Zustand, dass ein Schlüssel aus einem Blatt entnommen wurde. Dadurch könnte die Eigenschaft B4:



(a) Vor der Rotation: Knoten β hat einen Schlüssel zu wenig. Linkes Geschwisterchen γ enthält ausreichend viele Schlüssel für eine Rotation.

(b) Nach der Rotation.

Abbildung 4.18: Reparatur A: Rotation. (B-Baum der Ordnung 6.)

verletzt worden sein, d.h. dieses Blatt hat nun vielleicht einen Schlüssel zu wenig. In diesem Fall müssen wir nun versuchen, die B-Baum-Eigenschaften wiederherzustellen. Ansonsten ist das Entfernen abgeschlossen.

Sei β das Blatt mit zu wenigen Schlüsseln. Wenn β der Knoten ist, auf den der Eintrag $child[i]$ im Elterknoten zeigt, so sind die Knoten $child[i-1]$ und $child[i+1]$ (sofern sie existieren) die sogenannten *direkten Geschwister* von β.

3. **Reparatur A: Rotation.** (vergl. Abb. 4.18.) Nehmen wir an β habe ein direktes linkes Geschwisterchen γ. Wir überprüfen, ob γ um einen Schlüssel mehr besitzt als für einen gültigen B-Baum-Knoten unbedingt notwendig. Ist dies der Fall, so können wir uns aus diesem Knoten sozusagen „bedienen" um β aufzufüllen. Dazu transferieren wir den Trennschlüssel im Elterknoten von β und γ in den zu kleinen Knoten β, und verschieben dafür den größten Schlüssel in γ an die Stelle des Trennschlüssels. Dieser Vorgang wird *Rotation* genannt.

   Analog können wir das direkte rechte Geschwisterchen von β untersuchen. Sind wir bei einem der beiden Nachbarn erfolgreich, so ist das Entfernen abgeschlossen.

4. **Reparatur B: Verschmelzen.** Konnten wir keine Rotation durchführen, so wissen wir, dass die direkten Nachbaren (bzw. der einzige direkte Nachbar, falls nur einer existiert) nur über $\lceil \frac{m}{2} \rceil - 1$ Schlüssel verfügen. Sei γ einer der direkten Nachbarn. Wir *verschmelzen* nun γ, β und ihren Trennschlüssel in einen neuen großen Knoten. Wir wissen, dass dieser Knoten dann $\left(\lceil \frac{m}{2} \rceil - 1\right) + \left(\lceil \frac{m}{2} \rceil - 2\right) + 1 \leq m - 1$ Knoten besitzt, also nicht so groß ist. Dadurch vermindert sich natürlich die Anzahl der Schlüssel im Elterknoten, da wir den Trennschlüssel aus diesem entfernen. Auch die Anzahl der Zeiger sinkt um 1, da wir statt γ und β nur mehr einen großen Knoten referenzieren müssen.

5. **Reparatur rekursiv fortsetzen.** Haben wir die Operation des Verschmelzens angewandt, so kann es dazu kommen, dass nun der Elterknoten zu wenige Schlüssel enthält.

D.h. wir haben das Problem nur vom Blatt um eine Ebene noch oben verschoben. So wie schon beim Einfügen müssen wir also nun versuchen, diesen Elterknoten zu reparieren, in dem wir wieder die beiden Reparaturmethoden anwenden, solange bis wir schliesslich bei der Wurzel ankommen.

6. **Sonderfall Wurzel.** Auf Grund der Eigenschaft B3:, ist es kein Problem wenn wir schliesslich in der Wurzel zu wenige Schlüssel vorfinden, solange mindestens ein Schlüssel erhalten bleibt. Sollte durch eine Verschmelz-Operation die Wurzel keinen Schlüssel mehr enthalten und nur einen einzigen Zeiger auf den neu verschmolzenen Knoten δ halten, so ernennen wir einfach δ zur Wurzel des Baums, und entfernen die alte, nun leere, Wurzel.

Nach diesem Vorgang, haben wir also den Schlüssel x entfernt und alle Bedingungen des B-Baums sind weiterhin erfüllt. So bleibt nun noch die Laufzeit der Operation zu untersuchen: Zunächst sind wir mit einer einfachen Suche von der Wurzel bis zum zu löschenden Element und dort ggf. weiter bis zu einem Blatt abgestiegen, also $\Theta(\log(n))$. Danach mussten wir die Eigenschaft B4: wiederherstellen, und haben dazu im ungünstigsten Fall wieder von diesem Blatt bis zur Wurzel aufsteigen müssen, also nochmals $O(\log(n))$. Die Arbeit die wir an einem einzelnen Knoten zu verrichten hatten (Untersuchen, Rotieren, Verschmelzen, etc.) lässt sich immer durch eine Konstante abschätzen. Dadurch erhalten wir in Summe die Laufzeitgarantie $\Theta(\log(n))$ für die gesamte Entfernoperation.

Anmerkung 4.5. Wir erkennen: Ein B-Baum schrumpft nicht wie ein traditioneller binärer Baum von unten (durch Abschneiden der Blätter), sondern *von oben*, durch Abschneiden von Wurzelknoten.

Anmerkung 4.6. Das Reparaturschema der Rotation lässt sich dahingehend erweitern, dass man, sollte ein direkter Nachbar γ zu klein für eine Rotation sein, dessen direkten Nachbarn γ' betrachtet. Ist dieser groß genug, kann man eine doppelte Rotation durchführen: einmal von γ' nach γ, und danach von γ nach β. Man kann also im Wesentlichen der Reihe nach alle Nachbarknoten durchsuchen und die Eigenschaft dadurch öfter ohne rekursiven Aufruf wiederherstellen. Da die Anzahl der Geschwister durch m beschränkt ist, bedeutet dies asymptotisch sogar nur konstanten Aufwand! Dennoch wird von diesem Vorgehen zumeist Abstand genommen: Da ein B-Baum mit richtig gewählten Parametern in der Regel so aufgebaut ist, dass $\log n < m$, ist es in der Realität sinnvoller den Baum nach oben abzuwandern.

4.4.2 B^+-Bäume und andere Varianten

Wie schon erwähnt, sind die — zwei Jahre früher entwickelten — *2-3-Bäume* normale B-Bäume der Ordnung 3. Ebenso sind die sogenannten *2-3-4-Bäume* normale B-Bäume der Ordnung 4.

Da der B-Baum vor allem auch für Anwendungen auf Extern-Speicher interessant ist, kann es unangenehm sein, dass wir den B-Baum beim Einfügen und Entfernen je zweimal (einmal

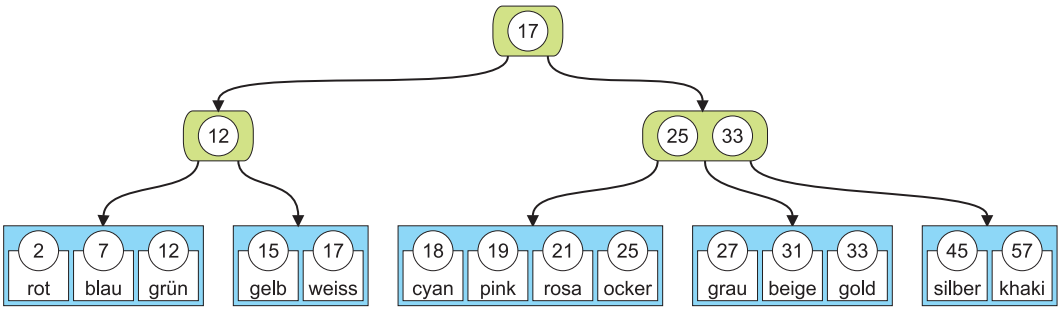


Abbildung 4.19: Ein B$^+$-Baum der Ordnung $(3, 5)$

von oben nach unten, und einmal von unten nach oben) ablaufen müssen. Daher gibt es auch eine etwas trickreichere Abwandlung des beschriebenen Algorithmus für B-Bäume gerader Ordnung, bei dem wir nur einmal von der Wurzel bis zum Blatt absteigen, und währenddessen quasi ,,auf Verdacht" große Knoten aufspalten, bzw. kleine Knoten verschmelzen. Dies führt zu einer Halbierung der Worst-Case Anzahl von externen Speicherzugriffen. Dies geht allerdings über den Umfang dieser Vorlesung hinaus. Sie können Details dazu beispielsweise im Buch ,,Introduction to Algorithms" von Cormen et al. (siehe Literaturliste) nachlesen.

In vielen Beschreibungen der 2-3-4-Bäume in der Literatur (so z.B. im Buch von R. Sedgewick, siehe Literaturliste) findet sich ebenfalls diese erwähnte Top-Down-Methode.

Darüberhinaus gibt es zahlreiche Varianten der B-Bäume bei denen z.B. vermieden wird, dass die Knoten nur zur Hälfte gefüllt sind, und statt dessen eine 2/3 Füllung garantiert wird. Andere Varianten erweitern den Baum durch Zeiger zwischen den Geschwistern oder akkumulieren ,,falsche" Balancierungen auf, bevor sie zu einem späteren Zeitpunkt gemeinsam repariert werden. Wie bei so vielem in der Informatik, ist auch bei diesen Datenstrukturen das letzte Wort noch nicht gesprochen; gerade die Anwendung für neue, effizientere Dateisysteme führt auch heute noch zu immer neuen Entwicklungen.

Zum Abschluss möchten wir uns noch einer bestimmten Spielart der B-Bäume zuwenden, den sogenannten *B$^+$-Bäumen*. In der Literatur ist die Verwendung dieses Begriffs nicht eindeutig, so taucht die selbe Datenstruktur auch teilweise als *B*-Baum* auf, bzw. bezeichnen beide Begriffe andere Varianten der B-Baum Idee. Die hier benutzte Wortwahl stellt aber wahrscheinlich die am weitesten verbreitete dar.

B$^+$-Bäume (vergl. Abb. 4.19) unterscheiden sich von B-Bäumen im Wesentlichen dadurch, dass die eigentlichen Datensätze hier nur in den Blättern stehen. Die inneren Knoten enthalten ausschliesslich Schlüssel (und Zeiger auf die Kinder), und dienen nur der Steuerung des Suchvorgangs. Dies hat den Vorteil, dass ein innerer Knoten bei gleichem Speicheraufwand (im Vergleich zu B-Bäumen) eine größere Zahl an Schlüsseln und Kindern aufnehmen kann. Dadurch wird der Verzweigungsgrad des Baums erhöht und seine Höhe nimmt (bei gleicher Gesamtzahl an Elementen) ab. In einem B$^+$-Baum unterscheiden sich die inneren Knoten von den Blättern — die dann schließlich die Schlüssel gemeinsam mit den Daten gespeichert halten — so stark, dass man sogar eine unterschiedliche Anzahl an Elementen zulässt.

So bedeutet ein B$^+$-Baum der Ordnung (m, m^+) folgendes: die inneren Knoten besitzen zwischen $\lceil \frac{m}{2} \rceil - 1$ und $m - 1$ viele Schlüssel, in jedem Blatt sind zwischen $\left\lceil \frac{m^+}{2} \right\rceil - 1$ und $m^+ - 1$ viele Schlüssel-Daten-Paare gespeichert. Durch diese Trennung der Parameter kann man weiterhin sicherstellen, dass unter Extern-Speicher-Bedingungen ein einzelner Knoten jeweils eine Speicherseite möglichst effizient ausnutzt.

Wie in Abb. 4.19 zu sehen, entstehen die Trennschlüssel in den inneren Knoten des Baums in der Regel durch Kopieren von in den Blättern vorkommenden Schlüsselwerten.

Das Suchen eines Elements in einem solchen Baum läuft im Wesentlichen identisch zu einer B-Baum-Suche ab, wobei wir auf jeden Fall bis zum Blatt absteigen müssen. Dies führt in der Realität aber kaum zu Geschwindigkeitseinbußen, da in einem B-Baum im Durchschnitt ohnehin die Hälfte aller Schlüssel in den Blättern gelagert sind und die Höhe sehr gering ist.

Die Kernidee beim Einfügen und Entfernen eines Elements ist ebenfalls analog zu den B-Bäumen. Es ergeben sich jedoch natürlich kleinere Unterschiede dadurch, dass ggf. neue trennende Schlüssel generiert, bzw. alte, überflüssige Trennschlüssel automatisch entfernt werden müssen.

4.5 Skiplisten

Bei der Analyse der Datenstrukturen in Kapitel 2 haben wir gesehen, dass verkettete Listen für die Suche auf ein Element aufgrund der linearen Suche selbst im Average-Case Laufzeit $\Theta(n)$ benötigen. Skiplisten sind eine randomisierte Datenstruktur, welche die Idee der binären Suche auf einfach verketteten Listen simuliert. Wie wir sehen werden, kann man damit alle Wörterbuch-Operationen auf Skiplisten in erwarteter logarithmischer Laufzeit ausführen.

Skiplisten basieren auf verketteten Listen, sind in der Praxis schnell und dennoch einfach zu implementieren. Die Grundidee ist, die Liste durch eine Menge von Hilfszeigern zu ergänzen. Hierzu wird jedem Element in der Liste eine bestimmte Höhe zugewiesen. Die Elemente auf jeder Höhenebene werden dann zu einer Hilfsliste verlinkt. Dabei werden die Höhen so zugewiesen, dass verschieden lange Hilfslisten entstehen. Im Idealfall halbiert sich mit jeder Ebene die Anzahl der Elemente, die auf dieser Höhe noch vertreten sind. Die Hilfsliste auf Ebene 0 enthält also alle Elemente, auf Ebene 1 ist nur noch jedes zweite Element in der Hilfsliste, und auf der i-ten Ebene ist nur noch jedes 2^i-te Element in der Hilfsliste, siehe Abbildung 4.20.

Damit haben Elemente in Skiplisten im Gegensatz zu einfach verketteten Listen nicht nur Zeiger auf ihre direkt benachbarten Elemente, sondern in höheren Ebenen auch Zeiger auf Elemente, die weiter hinten in der Liste stehen als der direkte Nachfolger. Deshalb kann man bei der Suche nach einem Schlüssel Elemente überspringen (= engl. “to skip”). Dies erlaubt es, hierarchisch von oben nach unten die Hilfslisten zu durchsuchen und dabei einen der

binären Suche auf Feldern vergleichbaren logarithmischen Aufwand zu erreichen. Skiplisten mit der oben genannten idealen Höhenverteilung nennt man auch „perfekte“ Skiplisten.

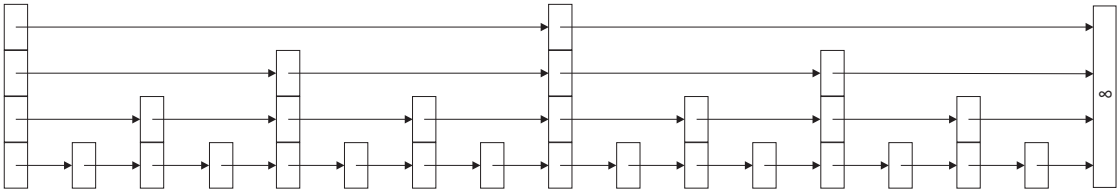


Abbildung 4.20: Schema einer perfekten Skipliste mit 15 Datenelementen sowie einem Anfangs- und einem Endelement

Ein Problem bei dieser Idee ist, dass die Struktur, welche die logarithmische Laufzeit garantiert, durch Insert- und Delete-Operationen zerstört werden kann. Damit zur Erhaltung der perfekten Struktur jedes Element immer seine korrekte Höhe hat, müsste man bei jeder Operation die Höhen neu zuweisen. Damit wir trotzdem einem Element seine Höhe unabhängig von den anderen Elementen zuweisen können, erfolgt eine randomisierte Zuweisung der Höhe. Wir weisen jedem Element eine zufällige Höhe H zu mit $Prob(H = h) = \frac{1}{2^{h+1}}$, somit kommt es in den Hilfslisten auf den Ebenen $0, \ldots, H$ vor. Damit erreichen wir im Durchschnitt die von uns gewünschte Verteilung, da wir 50% der Elemente nur auf Ebene 0, 25% auf Ebene 0 und 1 und $100/2^{i+1}\%$ Daten auf den Ebenen $0, \ldots, i$ erwarten können, womit wir eine erwartete logarithmische Laufzeit erreichen können. Skiplisten, die so entstanden sind, nennt man auch *randomisierte* Skiplisten.

Aufbau. Der Aufbau der Datenstruktur Skiplist ist damit wie folgt: Jedes Element x hat eine Höhe $height(x)$, die eine nicht-negative ganze Zahl ist. Diese Höhe wird beim Einfügen des Elements in die Liste zugewiesen und gibt die Zahl der zusätzlichen Hilfszeigerebenen an, in denen das Element verlinkt wurde. Man kann sich die zufällige Höhenzuweisung durch das folgende Zufallsexperiment vorstellen: Erst setzt man die Höhe auf 0. Dann wirft man eine Münze und vergrößert die Höhe um 1, falls „Kopf“ geworfen wird. Dies tut man so lange, bis zum ersten Mal „Zahl“ erscheint. Damit erhält man die gewünschte Verteilung. Zusätzlich zu den Datenelementen enthält die Skipliste ein (Pseudo-)Anfangs- und ein (Pseudo-)Endelement, deren Höhe dem Maximum der Höhen der Datenelemente entspricht. Die Elemente selbst bestehen aus einem Datenteil und einem Feld *next* von Nachfolger-Zeigern der Größe $height(x) + 1$. Jedes Element besitzt für jede Höhenebene, in der es vorkommt, einen Nachfolger-Zeiger auf das nächste Element in dieser Höhenebene, wodurch auf jeder Ebene eine verkettete Hilfsliste entsteht, die beim Anfangselement beginnt und beim Endelement endet. Das Anfangselement einer Liste ist über den *head*-Zeiger zugreifbar. Die Datenelemente sind in der Liste nach aufsteigenden Schlüsselwerten geordnet. Das Endelement besitzt einen Schlüssel, der größer als alle Schlüssel der Datenelemente in der Liste ist. Dieser Aufbau benötigt für perfekte Skiplisten $O(n)$ Platz, für randomisierte Skiplisten werden wir eine genaue Analyse des Erwartungswertes durchführen.

Dictionary-Operationen.

- SEARCH: Um in einer Skipliste ein Element mit Schlüssel s zu suchen, starten wir am Anfangselement in der obersten belegten Ebene und betrachten das Element, auf das der Nachfolger-Zeiger zeigt. Treffen wir auf ein Element mit einem kleineren Schlüssel als s, so betrachten wir dessen Nachfolger-Zeiger usw., bis wir auf ein Element mit größerem Schlüssel treffen (dies ist spätestens beim Endelement der Fall). Dann steigen wir in der Höhe um eine Ebene herab und suchen dort weiter. Treffen wir auf ein Element mit Schlüssel s, ist die Suche erfolgreich beendet, die erfolglose Suche endet, wenn wir auf Ebene 0 auf ein Element mit größerem Schlüssel als s treffen. Listing 4.15 zeigt den Pseudocode für die Suchoperation.

- INSERT: Zum Einfügen eines Elements x mit Schlüssel s suchen wir zunächst nach diesem Schlüssel in der Skipliste und merken uns dabei die betrachteten Nachfolger-Zeiger. Bei erfolgreicher Suche müssen wir nur die Daten im gefundenen Element überschreiben. Ist die Suche erfolglos, so landen wir bei dem größten Element in der Liste, das kleiner ist als s und haben uns alle Zeiger von Elementen mit Schlüssel $s' < s$ zu Elementen mit Schlüssel $s'' > s$ gemerkt. Wir berechnen nun zufällig eine Höhe $height(x)$, ist sie höher als die bisherige Maximalhöhe, so erhöhen wir das Anfangs- und Endelement und fügen für die neuen Ebenen entsprechend viele Zeiger zwischen diesen Elementen ein. Diese zusätzlichen Zeiger fügen wir zu den gespeicherten Zeigern hinzu. Jeden dieser gespeicherten Zeiger zwischen zwei Elementen y und z ersetzen wir nun durch Zeiger $y \rightarrow x$ und $x \rightarrow z$ auf derselben Ebene. Listing 4.16 zeigt den Pseudocode für die Einfügeoperation.

- DELETE: Für das Löschen eines Elements x mit Schlüssel s suchen wir diesen Schlüssel in der Skipliste, steigen dabei aber garantiert bis auf Ebene 0 herab, auch wenn wir das Element schon auf einer höheren Ebene finden. Dabei speichern wir auf allen Ebenen die betrachteten Zeiger auf x. Dies ist notwendig, da wir beim Entfernen von x auf jeder Ebene zwei Zeiger $y \rightarrow x$ und $x \rightarrow z$ durch einen Zeiger $y \rightarrow z$ ersetzen müssen. Wir laufen also durch unsere gespeicherten Zeiger und hängen die

```
1: ▷ Repräsentation eines Elements
2: struct SkipElement
3:     var KeyType key                          ▷ Schlüssel des Elements
4:     var ValueType info                       ▷ Gespeicherter Wert
5:     var SkipElement next[0 ... height]       ▷ Nachfolgerfeld
6: end struct
7: ▷ Interne Repräsentation einer Skipliste
8: var SkipElement head                         ▷ Anfangselement
9: var int height                               ▷ Höhe der Skipliste
```

Listing 4.14: Repräsentation einer Skipliste durch Listenelemente.

Zeiger von x auf den entsprechenden Nachfolger von x auf dieser Ebene um. Danach kann x entfernt werden. Falls dadurch die Gesamthöhe der Skipliste sinkt, müssen wir noch das Anfangs- und das Endelement anpassen. Listing 4.17 gibt den Pseudocode für die Löschoperation an.

Eine besondere Eigenschaft, die Skiplisten von anderen Datenstrukturen, die das Wörterbuchproblem lösen (z.B. Suchbäumen) unterscheidet, ist dass die Liste nach Entfernen von Elementen dieselbe Struktur hat, als wäre das Element nie in der Liste gewesen. Also kann eine Skipliste nie durch Entfernen eines Elements entarten und die "Zufälligkeit" der Struktur bleibt erhalten. Das Aussehen einer Skipliste hängt also nicht davon ab, in welcher Reihenfolge die Elemente eingefügt oder entfernt wurden. Man nennt eine solche Struktur auch *gedächtnislos*.

Abbildung 4.21 zeigt ein Beispiel für die Suche nach dem Schlüssel mit dem Wert 7. Der erste Zeiger auf der obersten Ebene zeigt auf das Element mit dem Schlüssel 45, wir setzen die Suche deshalb eine Ebene tiefer fort. Dort zeigt der Nachfolger-Zeiger auf das Element mit dem Schlüssel 5, wir bleiben deshalb auf dieser Ebene und folgen dem Zeiger zu diesem Element. Dessen Zeiger zeigt auf die 13, wir steigen also um eine Ebene herab. Der Nachfolger-Zeiger auf dieser Ebene zeigt immer noch auf die 13, deshalb steigen wir noch einmal herab und befinden uns damit bereits auf Ebene 0. Hier zeigt der Zeiger auf das Element mit dem Schlüssel 8, wir haben also erfolglos nach der 7 gesucht.

Die gestrichelten Zeiger geben den Suchpfad in der Skipliste an. Sie werden bei der Suche nach einem Wert im Intervall $(5, 8]$ betrachtet.

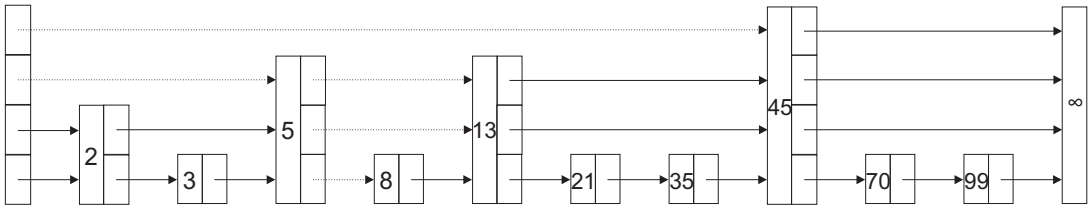


Abbildung 4.21: Randomisierte Skipliste mit zehn Datenelementen

Der Suchalgorithmus hat die Eigenschaft, dass man bei einer perfekten Skipliste für jede Ebene i höchstens einem Zeiger folgen muss d. h. die Anweisung $p := p.next[i]$ wird für jedes i höchstens ein Mal durchgeführt.

Weitere Operationen.

- CONCATENATE: Zwei Skiplisten L_1 und L_2 können bei entsprechender Verteilung der Schlüssel auch einfach konkateniert werden. Dazu paßt man, falls nötig, die Höhen an, indem man die Liste mit der kleineren Höhe auf die Höhe der größeren anpaßt. Man

```
Eingabe: Skipliste L, Schlüssel s
Ausgabe: Zeiger auf Element mit Schlüssel s falls existent und nil sonst
  function SEARCH(L, s)
    p := L.head
    for i := L.height,..., 0 do                    ▷ Folge Zeigern auf Ebene i
      while p.next[i].key < s do
        p := p.next[i]
      end while
    end for
    p := p.next[0]
    if p.key = s then
      ▷ s kommt an Position p in L vor
      return p
    else
      ▷ s kommt nicht in L vor
      return nil
    end if
  end function
```

Listing 4.15: Suchalgorithmus für Skiplisten

speichert alle Zeiger in L_1 auf das Endelement (durch Suche nach einem Schlüssel, der größer ist als alle gespeicherten, aber kleiner als ∞). Für jede Ebene ersetzt man nun einen Zeiger in L_1 von einem Element x auf das Endelement und den Zeiger in L_2 vom Anfangselement auf ein Element y durch einen Zeiger $x \rightarrow y$. Jetzt kann man das Endelement von L_1 und das Anfangselement von L_2 entfernen, die entstandene Skipliste enthält die Daten aus den beiden alten Listen L_1 und L_2.

- SPLIT: Eine Skipliste L kann an einem Element x mit Schlüssel s in zwei Skiplisten L_1 und L_2 aufgeteilt werden, indem man zunächst nach x sucht und alle Zeiger von $y < x$ nach $z > x$ speichert. Dann konstruiert man ein Enddatum E_1 für die neue Liste L_1 und ein Anfangsdatum A_2 für die neue Liste L_2. Die gespeicherten Zeiger $y \rightarrow z$ werden ersetzt durch zwei Zeiger $y \rightarrow E_1$ und $A_2 \rightarrow z$. Alle x verlassenden Zeiger $x \rightarrow z$ werden ersetzt durch $x \rightarrow E_1$ und $A_2 \rightarrow z$. L_1 erhält als Anfangsdatum das Anfangsdatum von L und L_2 als Enddatum das Enddatum von L. Gibt es nun in einer der beiden Listen Zeiger vom Anfangs- auf das Endelement, so können diese Ebenen entfernt werden.

Analyse. Zunächst machen wir einige Aussagen über die Höhe $H(n)$ einer Skipliste mit n Elementen und die Anzahl der Zeiger $Z(n)$. Dabei betrachten wir die Wahrscheinlichkeit $Prob(H(n) \geq h)$, dass die Liste eine bestimmte Höhe überschreitet, den Erwartungswert

```
Eingabe: Skipliste L, einzufügender Schlüssel s und Wert v
Ausgabe: Ein neues Element mit Schlüssel s wird in L eingefügt
  procedure INSERT(ref L, s, v)
    var SkipElement update[]
    p := L.head
    for i := L.height,..., 0 do
      while p.next[i].key < s do
        p := p.next[i]
      end while
      update[i] := p
    end for
    p := p.next[0]
    if p.key = s then
      p.info := v                          ▷ Schlüssel s kommt schon vor
    else
      ▷ Einfügen
      newHeight := randomHeight()
      if newHeight > L.height then
        ▷ Direkt mit Kopfelement verknüpfen und Listenhöhe aktualisieren
        for i := L.height + 1, ..., newHeight do
          update[i] := L.head
        end for
        L.height = newHeight
      end if
      ▷ Erzeuge neues Element mit Höhe newHeight und Schlüssel s
      p := new SkipElement(s, v, newHeight)
      for i := 0, ... newHeight do
        ▷ Füge p in die Listen auf Ebene i jeweils unmittelbar nach dem Element
  update[i] ein
        p.next[i] := update[i].next[i]
        update[i].next[i] := p
      end for
    end if
  end procedure
```

Listing 4.16: Einfügen in eine randomisierte Skipliste

```
Eingabe: Skipliste L und Schlüssel s
Ausgabe: Das Element mit Schlüssel s wird aus L entfernt
  procedure DELETE(L, s)
      p := L.head
      for i := L.height, ..., 0 do
          while p.next[i].key < s do
              p := p.next[i]
          end while
          update[i] := p
      end for
      p := p.next[0]
      if p.key = s then
          ▷ Element p entfernen und ggfs. Listenhöhe aktualisieren
          for i := 0, ..., p.height do
              ▷ Entferne p aus Liste auf Ebene i
              update[i].next[i] := p.next[i]
          end for
          while L.height ≥ 1 and L.head.next[L.height].key = ∞ do
              L.height = L.height − 1
          end while
      end if
  end procedure
```

Listing 4.17: Entfernen eines Elements aus einer Skipliste

$E(H(n))$, der uns angibt, welche Höhe wir im Durchschnitt erwarten können, und die erwartete Höhe eines Elements.

Theorem 4.3. *i) Die erwartete Höhe eines Elements beträgt* 1

ii) $Prob(H(n) \geq h) \leq min\{1, n/2^h\}$

iii) $E(H(n)) \leq \lfloor \log n \rfloor + 2$

iv) $E(Z(n)) \leq 2n + \lfloor \log n \rfloor + 3$

v) Der erwartete Platzbedarf für eine Skipliste mit n *Daten beträgt O(n)*

Beweis. i) Unser Experiment mit dem Münzwurf zeigt uns, dass wir es hier mit geometrischer Verteilung zu tun haben, da wir ein Experiment mit der Erfolgswahrscheinlichkeit $1/2$ so lange wiederholen, bis es erfolgreich ist. Wir können den Erwartungswert also direkt berechnen, da die erwartete Höhe die erwartete Anzahl an Würfen minus 1 ist:

$$E(H) = \sum_{1 \leq i < \infty} h \cdot \left(\frac{1}{2}\right)^h - 1 = 2 - 1 = 1$$

ii) Die obere Schranke 1 ist für einen Wahrscheinlichkeitswert trivialerweise erfüllt. Die Wahrscheinlichkeit, dass ein Element mindestens Höhe h bekommt, beträgt $\left(\frac{1}{2}\right)^h$, denn dafür muss bei unserem Münzexperiment für die Höhenzuweisung h-mal nicht „Zahl "fallen. Sei A_i das Ereignis, dass das i-te Element eine Höhe von mindestens h hat. Da es für die Gesamthöhe bereits ausreicht, wenn mindestens eines der Elemente Höhe h hat, ist das Ereignis „Gesamthöhe mindestens h "die Vereinigung der Ereignisse A_i. Dann gilt:

$$Prob(\bigcup_{1\leq i\leq n} A_i) \leq \sum_{1\leq i\leq n} Prob(A_i) = n(1/2)^h$$

iii) Wir leiten aus der Definition des Erwartungswertes eine neue Formel für den Erwartungswert von Zufallsvariablen, die nur nichtnegative ganze Zahlen annehmen, her.

$$\begin{aligned} E(H(n)) &= \sum_{0\leq h<\infty} h \cdot Prob(H(n)=h) \\ &= Prob(H(n)=1) + Prob(H(n)=2) + Prob(H(n)=3) + \ldots \\ &\qquad + Prob(H(n)=2) + Prob(H(n)=3) + \ldots \\ &\qquad + Prob(H(n)=3) + \ldots \\ &\qquad + \ldots \\ &= \sum_{1\leq h<\infty} Prob(H(n)\geq h) \end{aligned}$$

Nun schätzen wir die ersten $\lfloor \log n \rfloor + 1$ Summanden durch 1 ab, und den $(\lfloor \log n \rfloor + 1 + i)$-ten Summanden mittels ii) durch

$$n \cdot \frac{1}{2^{\lfloor \log n \rfloor + 1 + i}} = n \cdot \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor + 1} \cdot \left(\frac{1}{2}\right)^i \leq \left(\frac{1}{2}\right)^i$$

Damit kann die Summe abgeschätzt werden durch

$$\sum_{1\leq \lfloor \log n \rfloor + 1} 1 + \sum_{1\leq i\leq \infty} \left(\frac{1}{2}\right)^i = \lfloor \log n \rfloor + 1 + 1 = \lfloor \log n \rfloor + 2$$

iv) Der Erwartungswert ist linear. Das i-te Objekt hat eine durchschnittliche Höhe von 1, also durchschnittlich 2 Zeiger. Hinzu kommen $H(n)$ Zeiger für das Anfangsobjekt. Nun folgt das Resultat aus $iii)$.

v) folgt direkt aus $iv)$

□

Theorem 4.4. *Die erwartete Rechenzeit für jede der Operationen* SEARCH, INSERT *und* DELETE *beträgt* $O(\log n)$.

Beweis. Wir können uns auf die Analyse der erfolglosen Suche beschränken, bei der wir in jedem Fall bis zur Ebene 0 absteigen müssen. Sie ist höchstens teurer als die entsprechende erfolgreiche Suche.

Auf dem Suchpfad nach einem Schlüssel s finden wir Zeiger, die auf Elemente mit Schlüsseln größer als s zeigen. Die Anzahl dieser Zeiger ist genau die Anzahl der Ebenen. Aus Theorem 4.3 kennen wir deren erwartete Anzahl. Die Anzahl der restlichen Zeiger, an denen wir nicht absteigen, die wir also von links nach rechts verfolgen, hängt vom Aussehen der Skipliste ab. Um die erwartete Anzahl von horizontalen Schritten zwischen den Abstiegen zu analysieren, könnten wir versuchen abzuschätzen, wieviele Elemente zwischen dem erreichten und dem gesuchten Element in der Liste enthalten sind. Stattdessen betrachten wir den Suchpfad rückwärts (*backward analysis*). Wir nehmen also an, dass wir in einem Element auf Ebene l sind. Auf Suchpfaden werden Elemente stets auf ihren höchsten Ebenen erreicht. Die Wahrscheinlichkeit, dass ein Datum, das auf Ebene l existiert, auchauf Ebene $l+1$ existiert, beträgt bei unserer Konstruktion genau $1/2$. Wir wollen die erwarteten Schritte auf dem Rückwärtspfad untersuchen, bis wir die Ebene h oder das Anfangselement erreichen. Wir sind an oberen Schranken interessiert, daher nehmen wir pessimistisch an, dass die Liste nach links unbeschränkt ist, d.h. wir können das Anfangselement nicht erreichen. Jeder Schritt geht dann mit Wahrscheinlichkeit $1/2$ nach oben und mit Wahrscheinlichkeit $1/2$ nach links. Wieviele Schritte nach links erwarten wir nun vor dem ersten Schritt nach oben? Die Wahrscheinlichkeit, dass es genau i Schritte sind, beträgt $(1/2)^{i+1}$. Der Erwartungswert ist damit

$$\sum_{1\le i\le\infty} i\cdot(1/2)^{i+1} = \frac{1}{2}\sum_{1\le i\le\infty} i\cdot(1/2)^{i} = 1$$

Dies entspricht der intuitiven Vorstellung, dass es im Durchschnitt gleich viele Schritte der beiden Arten geben sollte. Da der Erwartungswert linear ist, machen wir im Durchschnitt insgesamt h Schritte nach links, bevor wir den h-ten Schritt nach oben machen. Wir erreichen Ebene h also im Durchschnitt nach nur $2h$ Rückwärtsschritten. Diese Abschätzung benutzen wir bis zum Aufstieg auf die Ebene $\lceil \log n \rceil$. Danach werden nur noch Daten erreicht, deren Höhe mindestens $\lceil \log n \rceil + 1$ ist. Die Wahrscheinlichkeit, dass ein Datum eine Höhe von mindestens $\lceil \log n \rceil + 1$ hat, ist durch $1/n$ beschränkt. Sei $X_i = 1$, falls das i-te Datum eine Höhe von mindestens $\lceil \log n \rceil + 1$ hat, und $X_i = 0$ sonst. Dann ist

$$\begin{aligned} E(X_i) &= Prob(X_i = 0)\cdot 0 + Prob(X_i = 1)\cdot 1 \\ &= Prob(X_i = 1) \le \frac{1}{n} \end{aligned}$$

und

$$E(X_1 + \ldots + X_n) = \sum_{1\le i\le n} E(X_i) \le n\cdot\frac{1}{n} = 1$$

Dabei steht $X_1 + \ldots + X_n$ für die zufällige Anzahl von Daten, deren Höhe mindestens $\lceil \log n \rceil + 1$ beträgt. Jedes Datum, das auf Ebene $\lceil \log n \rceil + 1$ noch existiert, führt zu maximal einem Schritt nach links. Dies ist also im Durchschnitt ein weiterer Schritt nach links. Insgesamt ist die erwartete Anzahl von Schritten nach links höchstens $\lceil \log n \rceil + 2$. Die erwartete Anzahl von Schritten nach oben ist durch die erwartete Höhe und damit $\lceil \log n \rceil + 3$ nach oben beschränkt. Damit ist das Theorem bewiesen. □

Diskussion. Wie beim randomisierten Quicksort vertrauen wir bei randomisierten Skiplisten nicht auf die „gute "Verteilung der Daten, sondern auf die von der Eingabe unabhängige interne Nutzung von Zufallszahlen. Durch das mit hoher Wahrscheinlichkeit garantierte Verhalten haben die Skiplisten damit in der Praxis eine gute Laufzeit der Operationen, wobei die Anzahl der Zeiger im Vergleich zu linearen Listen nur auf rund das Doppelte ansteigt.

Kapitel 5

Hashing

Die Idee des Hashverfahrens besteht darin, statt in einer Menge von Datensätzen durch Schlüsselvergleiche zu suchen, die Adresse eines Elements durch eine arithmetische Berechnung zu ermitteln. Hashverfahren unterstützen die Operationen: SUCHEN, EINFÜGEN und ENTFERNEN auf einer Menge von Elementen und lösen somit das Wörterbuchproblem. Eine klassische Anwendung für Hashverfahren ist das Suchen von Schlüsselwörtern in den Symboltabellen für Compiler. Sie sind immer dann nützlich, wenn die Anzahl der Schlüssel nicht zu groß ist und wir die erwartete Anzahl der Datensätze kennen.

Sei U das Universum aus dem die Schlüssel stammen können. Wir legen eine Tabellengröße m fest, die typischerweise etwas kleiner ist als die erwartete Anzahl der Schlüssel, die sich gleichzeitig in der Hashtabelle befinden werden, und wählen eine Hashfunktion $h : U \to \{0, \ldots, m-1\}$. Ein einzufügender Schlüssel $k \in U$ wird in $T[h(k)]$ gespeichert. Abbildung 5.1 illustriert die Idee der Hashverfahren.

Der große Vorteil dieser Methode im Vergleich zu den bisherigen Suchmethoden ist, dass hier idealerweise die Laufzeit für die Operationen SUCHEN, EINFÜGEN und ENTFERNEN nur $\Theta(1)$ plus die Zeit für die Auswertung der Hashfunktion h beträgt. Und diese ist unabhängig von der Anzahl der eingefügten Schlüssel. Dies gilt allerdings nur unter der Annahme, dass keine zwei Werte $h(k_1)$ und $h(k_2)$ *kollidieren*, das bedeutet $h(k_1) \neq h(k_2)$ für alle $k_1 \neq k_2$. Da jedoch das Universum U im Normalfall viel größer ist als die zur Verfügung stehenden Adressen, können Kollisionen im allgemeinen nicht verhindert werden.

Definition 5.1. Sei n die Anzahl der Schlüssel in einer Hashtabelle der Größe m. Wir nennen die durchschnittliche Anzahl von Schlüsseln (in einer Hashtabelle) $\alpha = \frac{n}{m}$ den *Auslastungsfaktor* (*Belegungsfaktor*).

Wenn der Auslastungsfaktor einer Hashtabelle größer als 1 ist, dann entstehen zwangsweise Kollisionen. Wir werden sehen, dass die Laufzeit der Wörterbuch-Operationen bei „vernünftiger Belegung“ der Hashtabelle praktisch konstant ist.

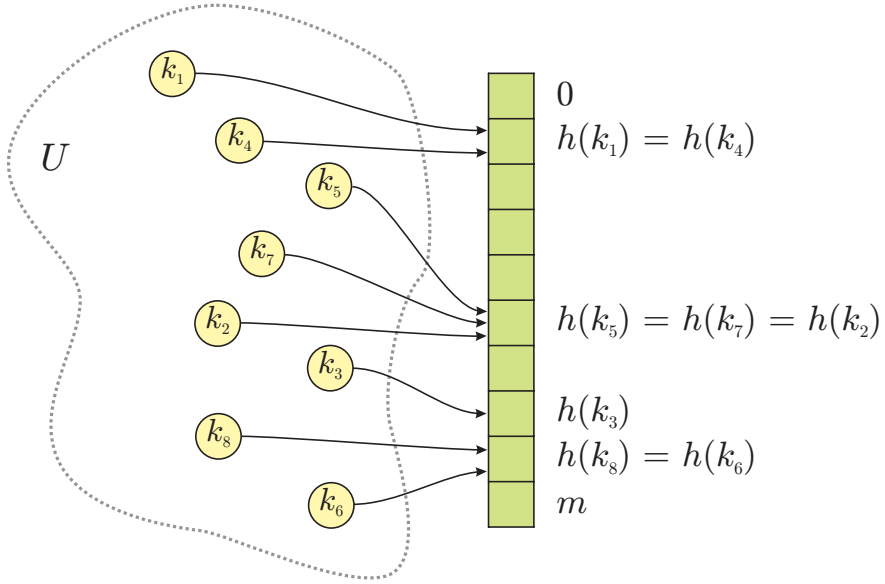


Abbildung 5.1: Die Idee von Hashing

Datenstruktur. Die sogenannte *Hashtabelle* wird als ein Array $T[0 \ldots m-1]$ realisiert. Die Hashtabelle speichert m Einträge mit Hashadressen $0, \ldots, m-1$.

Generelle Annahme. Für die folgenden Abschnitte gilt: Die Schlüssel k sind nicht-negative ganze Zahlen.

Anmerkung 5.1. Es ist (fast) immer möglich, Schlüssel als nicht-negative ganze Zahlen zu interpretieren. Üblicherweise interpretiert man z.B. *character strings* als 256-adische Zahlen (i.e., Zahlen zur Basis 256), denn jeder *character* besitzt im ASCII-Code eine Nummer *ord* im Bereich $\{0, \ldots, 255\}$. Diese ist z.B. $ord(p) = 112$ und $ord(t) = 116$. Die Verbindung der beiden ergibt

$$k(\text{`}pt\text{`}) = 112 \cdot 256 + 116 = 28788.$$

Allgemein berechnet man den Schlüssel für ASCII-Strings $s = (s_1, \ldots, s_l)$ der Länge l als:

$$k(s) = \sum_{i=1}^{l} 256^{l-i} \cdot ord(s_i) \tag{5.1}$$

Allerdings ist zu beachten, dass diese Methode für lange Strings zu sehr großen Zahlen führen kann!

Die Güte eines Hashverfahrens hängt von der gewählten Hashfunktion h und den einzufügenden Schlüsseln ab. Im folgenden Abschnitt werden wir zwei verschiedene Möglichkeiten für gute Hashfunktionen kennenlernen. Ein weiteres zentrales Problem bei Hashverfahren ist die Kollisionsbehandlung. Damit beschäftigen wir uns in den Abschnitten 5.2 und 5.3.

5.1 Zur Wahl der Hashfunktion

Eine gute Hashfunktion sollte die berechneten Hashadressen möglichst gleichmäßig auf $\{0, 1, \ldots, m-1\}$ verteilen. Insbesondere sollten Häufungen sehr ähnlicher Schlüssel möglichst gleichmäßig auf den Adressbereich streuen. Natürlich sollte die Auswertung einer Hashfunktion schnell und einfach sein. In den folgenden beiden Abschnitten werden zwei mögliche Kandidaten vorgestellt.

5.1.1 Die Divisions-Rest-Methode

Definition 5.2. Die Hashfunktion der *Divisions-Rest-Methode* ist gegeben durch:

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

Der Vorteil dieser einfachen Hashfunktion ist, dass sie in konstanter Zeit berechnet werden kann, und die Berechnung praktisch sehr schnell ist. Jedoch ist hier die **richtige Wahl von** m (Tabellengröße) sehr wichtig. Folgendes sollte man z.B. vermeiden:

- $m = 2^i$: Alle bis auf die letzten Binärziffern werden ignoriert.
- $m = 10^i$: analog bei Dezimalzahlen;
- $m = r^i$: analog bei r-adischen Zahlen;
- $m = r^i \pm j$ für kleines j: z.B.: $m = 2^8 - 1 = 255$:

$$h(k(\text{`}pt\text{`})) = (112 \cdot 256 + 116) \bmod 255 = 28788 \bmod 255 = 228$$
$$h(k(\text{`}tp\text{`})) = (116 \cdot 256 + 112) \bmod 255 = 29808 \bmod 255 = 228$$

  Dasselbe passiert, wenn in einem längeren String zwei Buchstaben vertauscht werden. Letztendlich wird in diesem Fall die Summe der beiden Buchstaben modulo m berechnet.

Die folgende Wahl von m hat sich in der Praxis gut bewährt.

Beobachtung 5.1. *Eine gute Wahl für m ist eine Primzahl, die kein $r^i \pm j$ (für kleines j) teilt und weit weg von einer Zweierpotenz ist.*

Beispiel 5.1. Eine Hashtabelle soll ca. 600 Einträge aufnehmen, die Schlüssel sind *character strings*, interpretiert als 128-adische Zahlen. Eine gute Wahl wäre hier z.B. $m = 701$, da $2^9 = 512$ und $2^{10} = 1024$.

Anmerkung 5.2. Bei einer Implementierung der Divisions-Rest-Methode für Strings, die mit Gleichung (5.1) als numerische Schlüssel interpretiert werden, kann man das *Horner-Schema* verwenden, um die explizite Berechnung von k und damit Überläufe von Standard-Integertypen durch zu große Zahlen zu vermeiden. Diese Methode beruht auf einer anderen Schreibweise von Gleichung (5.1):

$$k = (\ldots(s_1 \cdot 256 + s_2) \cdot 256 + s_3) \cdot 256 + \ldots + s_{l-1}) \cdot 256 + s_l$$

Es gilt:

$$\begin{aligned} k \bmod m \;=\; & (\ldots(s_1 \cdot 256 + s_2) \bmod m) \cdot 256 + s_3) \bmod m) \cdot 256 + \ldots + \\ & + s_{l-1}) \bmod m) \cdot 256 + s_l) \bmod m \end{aligned}$$

Durch die wiederholt vorgezogene modulo-Rechnung wird der Bereich immer möglichst rasch wieder eingeschränkt, und es treten keine Werte größer als $(m-1) \cdot 256 + 255$ auf.

5.1.2 Die Multiplikationsmethode

Bei der Multiplikationsmethode wird das Produkt des Schlüssels k mit einer Zahl A ($0 < A < 1$) gebildet. Das Produkt besitzt einen ganzzahligen und einen gebrochenen Teil. Die Hashadresse berechnet sich aus dem gebrochenen Teil, der mit der Tabellengröße m multipliziert und abgerundet wird.

Definition 5.3. Die Hashfunktion der *Multiplikationsmethode* lautet:

$$h(k) = \lfloor m(k \cdot A \bmod 1) \rfloor = \lfloor m \underbrace{(k \cdot A - \lfloor k \cdot A \rfloor)}_{\in [0,1)} \rfloor$$

mit $0 < A < 1$.

Der Term $(k \cdot A - \lfloor k \cdot A \rfloor)$ heißt auch der „gebrochene Teil“ von $k \cdot A$.

Der Vorteil dieser Methode liegt darin, dass hier die Wahl von m unkritisch ist. Bei guter Wahl von A kann man eine gleichmäßige Verteilung für $U = \{1, 2, \ldots, n\}$ auf den Adressbereich erhalten. Um eine gleichmäßige Verteilung der Hashadressen zu erhalten, muss man bei der Wahl von A darauf achten, dass der gebrochene Teil von $k \cdot A$ für alle Schlüssel

des Universums U gleichmäßig auf das Intervall $[0, 1)$ verteilt wird. Dies wird sicherlich mit $A = 0,5$ nicht erreicht. Dasselbe gilt für Zahlen mit wenigen gebrochenen Stellen. Deswegen bieten sich als gute Wahl von A die irrationalen Zahlen an. Das folgende Theorem bestätigt diese Wahl.

Theorem 5.1 (Satz von Vera Turan Sos 1957). *Sei ξ eine irrationale Zahl. Platziert man die Punkte*

$$\xi - \lfloor \xi \rfloor, 2\xi - \lfloor 2\xi \rfloor, \ldots, n\xi - \lfloor n\xi \rfloor$$

in das Intervall $[0, 1)$, dann haben die $n + 1$ Intervallteile höchstens drei verschiedene Längen. Außerdem fällt der nächste Punkt

$$(n + 1)\xi - \lfloor (n + 1)\xi \rfloor$$

in einen der größeren Intervallteile (ohne Beweis).

Beobachtung 5.2. *Eine gute Wahl für A sind irrationale Zahlen.*

Knuth empfiehlt in seinem Buch *The Art of Computer Programming, Volume 3* den *goldenen Schnitt*

$$\Phi^{-1} = \frac{\sqrt{5} - 1}{2} = 0,6180339887...$$

als die beste Wahl von A .

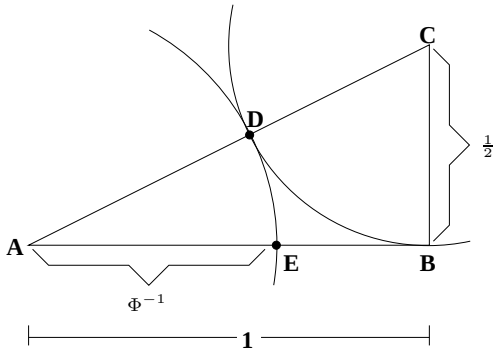


Abbildung 5.2: Der goldene Schnitt

Der goldene Schnitt ergibt sich durch die Zerlegung einer Strecke a in zwei positive Summanden x und $a - x$, so dass x geometrisches Mittel von a und $a - x$ ist, d.h. $x^2 = a(a - x)$ (siehe Abb. 5.2). Die größere Teilstrecke steht dann zur Gesamtstrecke im gleichen Verhältnis wie die kleinere Teilstrecke zur größeren:

$$\frac{x}{a} = \frac{a - x}{x}.$$

Der goldene Schnitt tauchte bereits in Euklids „Elementen“ auf. Auch Kepler spricht in seinen Werken von der „göttlichen Teilung“. Bereits in der antiken Architektur und in Kunstwerken wurde der goldene Schnitt als Maßverhältnis eingesetzt.

Beispiel 5.2. (Dezimalrechnung) $k = 123\,456$, $m = 10\,000$, $A = \Phi^{-1}$

$$\begin{aligned} h(k) &= \lfloor 10\,000 \cdot (123\,456 \cdot 0,61803... \bmod 1) \rfloor \\ &= \lfloor 10\,000 \cdot (76\,300,0041151... \bmod 1) \rfloor \\ &= \lfloor 10\,000 \cdot 0,0041151... \rfloor \\ &= \lfloor 41,151... \rfloor \\ &= 41 \end{aligned}$$

Eine gute Wahl für m wäre hier $m = 2^i$. Dann kann $h(k)$ effizient berechnet werden (eine einfache Multiplikation und ein Shift).

Es gibt noch viele weitere Hashfunktionen. Jedoch zeigen empirische Untersuchungen, dass die Divisions-Rest-Methode mindestens so gut ist wie ihre Konkurrenten. Deswegen verwenden wir diese Methode in den folgenden Abschnitten, in denen verschiedene Möglichkeiten der Kollisionsbehandlung behandelt werden. Eine Kollision tritt dann auf, wenn zwei verschiedene Schlüssel auf die gleiche Adresse in der Hashtabelle abgebildet werden.

5.2 Hashing mit Verkettung

Die Idee des Hashing mit Verkettung ist es, alle Elemente mit gleicher Adresse in der Hashtabelle in einer verketteten Liste zu speichern. Jedes Element der Hashtabelle ist also ein Zeiger auf eine einfach verkettete lineare Liste (s. Abb. 5.3).

Datenstruktur. Wir wählen als Datenstruktur ein Array von einfach verketteten Listen L, wie wir sie z.B. bei ADT Stack kennengelernt haben.

Eine mögliche Implementierung von Hashing mit Verkettung wird durch die Algorithmen in Abb. 5.1 gegeben. INSERT() hat als Eingabe einen Schlüssel k (und den dazugehörigen Eintrag v), berechnet die Hashadresse pos des einzufügenden Elements (Schlüssel gegeben durch $p.key$) und hängt p vorne an die Liste von $T.pos$ an.

Die Funktion SEARCH() benötigt als Eingabe den zu suchenden Schlüssel k. Danach wird die Liste der zu k korrespondierenden Hashadresse solange durchsucht, bis ein Element mit Schlüssel gleich k gefunden ist oder das Ende der Liste erreicht ist. Im ersteren Fall wird p zurückgegeben, sonst *undef*.

```
 1: ▷ Repräsentation eines Hash-Elementes
 2: struct HashElement
 3:     var HashElement next                    ▷ Verweis auf Nachfolger in der Liste
 4:     var K key                                ▷ Schlüssel des Elementes
 5:     var V value                              ▷ Wert des Elementes
 6: end struct

 7: ▷ Interne Repräsentation von Hashing mit Verkettung
 8: var HashElement T[0..m − 1]

 9: ▷ Initialisierung
10: for i := 0, ..., m − 1 do
11:     T[i] := nil
12: end for

13: procedure INSERT(K k, V v)
14:     pos := h(k)
15:     var HashElement p := new HashElement
16:     p.key := k;   p.value := v
17:     p.next := T[pos]
18:     T[pos] := p
19: end procedure

20: function SEARCH(K k) : V ∪ {undef}
21:     var HashElement p := T[h(k)]
22:     while p ≠ nil and p.key ≠ k do
23:         p := p.next
24:     end while
25:     if p = nil then
26:         return undef                         ▷ Kein Element mit Schlüssel k vorhanden
27:     else
28:         return p.value
29:     end if
30: end function
```

Listing 5.1: Interne Repräsentation und Implementierung von SEARCH und INSERT für Hashing mit Verkettung.

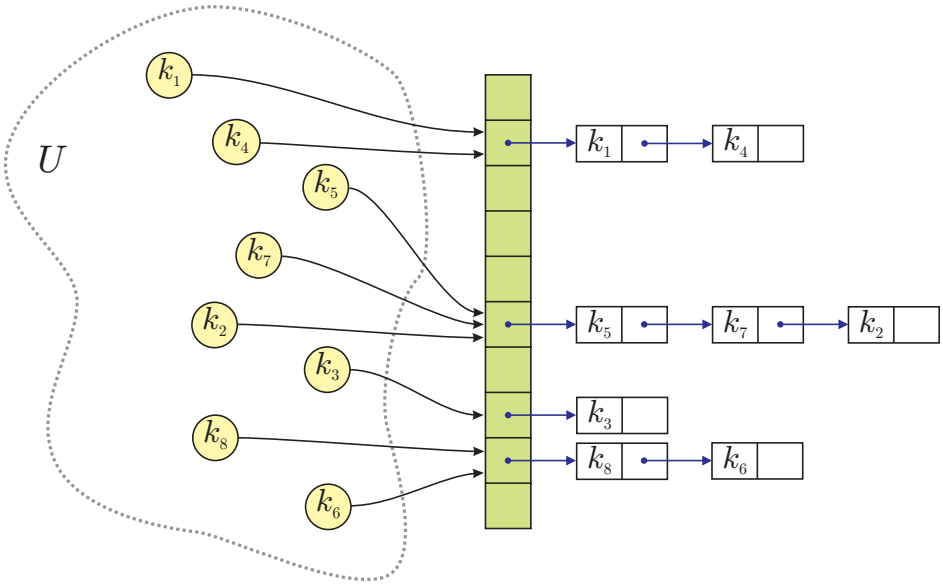


Abbildung 5.3: Hashing mit Verkettung

```
 1: procedure DELETE(K k)
 2:     pos := h(k)
 3:     var HashElement p := T[pos]
 4:     var HashElement q := nil                    ▷ Vorgänger von p in der Liste
 5:     while p ≠ nil and p.key ≠ k do
 6:         q := p
 7:         p := p.next
 8:     end while
 9:     if p ≠ nil then                             ▷ Schlüssel k überhaupt gefunden?
10:         if q = nil then
11:             T[pos] := p.next
12:         else
13:             q.next := p.next
14:         end if
15:         delete p
16:     end if
17: end procedure
```

Listing 5.2: Implementierungen von DELETE für Hashing mit Verkettung

DELETE() nimmt als Eingabe den Schlüssel k des zu entfernenden Elementes. Zunächst wird die Liste der zu k korrespondierenden Hashadresse nach einem Element mit Schlüssel gleich k durchsucht. Dabei wandert die Variable p die Listenelemente der Reihe nach ab. Das Hilfselement q zeigt dabei immer auf den Listenvorgänger von p. Am Ende der While-Scheife zeigt p auf das zu entfernende Listenelement. Mit Hilfe von q (Vorgängerelement) wird dieses aus der Liste entfernt. Falls q gleich *nil* ist, dann zeigt p auf das erste Listenelement. In diesem Fall muss einfach nur der Zeiger von $T.pos$ umgesetzt werden.

Analyse. Wir analysieren zunächst die Operation SEARCH(). Wir nehmen an, dass sich in der Hashtabelle n Schlüssel befinden. Zur Analyse zählen wir die Anzahl der benötigten Schlüsselvergleiche.

Best-Case. Im besten Fall erhalten alle Schlüssel unterschiedliche Hashwerte. Dann ist $C_{best}(n) = \Theta(1)$.

Worst-Case. Im schlimmsten Fall erhalten alle Schlüssel den gleichen Hashwert. Dann ist $C_{worst}(n) = \Theta(n)$.

Average-Case. Wir mitteln über die durchschnittliche Suchzeit aller in unserer Hashtabelle enthaltenen Elemente. Zur Analyse sind einige Annahmen notwendig:

1. Ein gegebenes Element wird auf jeden der m Plätze mit gleicher Wahrscheinlichkeit $\frac{1}{m}$ abgebildet, unabhängig von den anderen Elementen.

2. Jeder der n gespeicherten Schlüssel ist mit gleicher Wahrscheinlichkeit der gesuchte.

3. Die Prozedur INSERT() fügt neue Elemente am Ende der Liste ein. Dies erleichtert uns die Analyse. Es wird zwar in Wahrheit am Anfang der Liste eingefügt, aber es gilt: Die durchschnittliche erfolgreiche Suchzeit über alle Elemente in der Hashtabelle ist gleich, egal, ob vorne oder hinten eingefügt wird.

4. Die Berechnung von $h(k)$ benötigt konstante Zeit.

Erfolglose Suche. Wenn wir nach einem Element suchen, das nicht in unserer Hashtabelle enthalten ist, dann müssen wir alle Elemente einer Liste durchwandern. In jeder Liste sind durchschnittlich α Elemente enthalten. Es gilt also $C_{avg}(n) = \alpha$.

Erfolgreiche Suche. Gemäß der Annahme $3.$ wird beim Suchen jeweils ein Schritt mehr gemacht als beim Einfügen des gesuchten Elements. Die durchschnittliche Listenlänge beim

Einfügen des i-ten Elements ist $\frac{i-1}{m}$. Laufzeit:

$$\begin{aligned} C_{avg}(n) &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) \\ &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1) \\ &= 1+\frac{1}{nm}\cdot\frac{n(n-1)}{2} \\ &= 1+\frac{n-1}{2m} \\ &= 1+\frac{n}{2m}-\frac{1}{2m} \\ &= 1+\frac{\alpha}{2}-\frac{1}{2m} \end{aligned}$$

Es gilt also $C_{avg}(n) = O(1+\alpha)$. Daraus kann man erkennen, dass die Operation SEARCH() im Durchschnitt konstante Zeit benötigt, sofern die Anzahl der Plätze proportional zur Anzahl der Elemente ist (d.h. $n = O(m)$ bzw. $\alpha = O(1)$).

Die analoge Analyse der Operation DELETE() bringt das gleiche Ergebnis. Die Operation INSERT() hingegen ist immer in Zeit $O(1)$ möglich (wenn wir annehmen, dass die Hashfunktion in konstanter Zeit berechnet werden kann).

Anmerkung 5.3. Ein Belegungsfaktor von mehr als 1 ist hier möglich. Dies ist insofern wichtig, weil viele Hashtabellen zunächst für kleinere Schlüsselmengen angelegt werden, aber dann im Laufe der Zeit die Datenmengen viel größer werden als ursprünglich erwartet. Bei fehlender oder ungenügender Wartung wird die Hashtabellengröße nicht angepasst. Dieses Szenario führt bei Hashing mit Verkettung *nur* zu einer Verschlechterung der Suchzeiten.

Diskussion.

+ Echte Entfernungen von Einträgen sind möglich. Wir werden sehen, dass es bei den Hashing-Verfahren mit offener Adressierung (vgl. Kap.5.3) keine echten Entfernungen gibt.

+ Eignet sich für den Einsatz mit Externspeicher: man kann die Hashtabelle selbst im internen Speicher halten und die Listen teilweise im externen Speicher.

– Man hat einen relativ hohen Speicherplatzbedarf, denn zu den Nutzdaten kommt der Speicherplatzbedarf für die Zeiger.

– Der Speicherplatz der Hashtabelle wird nicht genutzt. Dies ist umso schlimmer, wenn in der Tabelle viele Adressen unbenutzt bleiben. In diesem Fall wird trotzdem zusätzlicher Speicherplatz für Listenelemente außerhalb der Hashtabelle benötigt.

5.3 Hashing mit offener Adressierung

Bei Hashing mit offener Adressierung werden alle Elemente – im Gegensatz zum vorigen Verfahren – innerhalb der Hashtabelle gespeichert. Wenn ein Platz belegt ist, so werden in einer bestimmten Reihenfolge weitere Plätze ausprobiert. Diese Reihenfolge nennt man *Sondierungsreihenfolge*. Diese sollte eine Permutation von $\langle 0, 1, \ldots, m-1 \rangle$ sein.

Definition 5.4. Bei Hashing mit offener Adressierung wird die Hashfunktion auf zwei Argumente erweitert:

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Die *Sondierungsreihenfolge* ergibt sich dann durch

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle.$$

Damit das Konzept der offenen Adressierung funktioniert, gilt die folgende Voraussetzung.

Annahme. Die Hashtabelle enthält immer wenigstens einen unbelegten Platz.

Beim Einfügen wird solange ein Platz gemäß der Sondierungsreihenfolge ausgewählt, bis ein freier Platz in der Hashtabelle gefunden wird. Genauso funktioniert auch die Suche: man berechnet zunächst den ersten Platz gemäß der Hashfunktion (also $h(k, 0)$). Falls dieser Platz belegt ist, sucht man gemäß der Sondierungsreihenfolge weiter. Allerdings kann man die Suche abbrechen, sobald man auf einen unbelegten Platz stößt. Denn wenn das Element in T enthalten wäre, dann wäre es spätestens an diesem Platz eingefügt worden. Um diese Strategie beim Suchen beibehalten zu können, dürfen wir beim Entfernen eines Elements nicht wirklich „entfernen“: Es wird lediglich als „entfernt“ markiert. Beim Einfügen wird ein solcher Platz als „frei“, beim Suchen als „früher belegt“ (d.h. verfügbar, aber früher schon einmal belegt) betrachtet. Dies führt jedoch zu einem Nachteil bezüglich der Suchzeit. Diese ist nun nicht mehr proportional zu $\Theta(1 + \alpha)$. Deshalb sollte man, sofern man viele Entfernungen vornehmen muss, die Methode der Verkettung vorziehen.

Eine ideale Sondierungsreihenfolge wäre das *uniforme Sondieren*: Hier erhält jeder Schlüssel mit gleicher Wahrscheinlichkeit eine bestimmte der $m!$ Permutationen von $\{0, 1, \ldots, m-1\}$ als Sondierungsreihenfolge zugeordnet. Da dies schwierig zu implementieren ist, versucht man in der Praxis dieses Verhalten zu approximieren. Wir lernen in den folgenden Abschnitten drei verschiedene Sondierungsverfahren kennen, nämlich das *lineare Sondieren* (vgl. Abschnitt 5.3.1), das *quadratische Sondieren* (vgl. Abschnitt 5.3.2), und das *Double Hashing* (vgl. Abschnitt 5.3.3).

5.3.1 Lineares Sondieren

Das lineare Sondieren ist die einfachste Sondierungsmethode: Ist der von der originalen Hashfunktion h' berechnete Platz bereits belegt, so wird einfach der nächste Platz in der Hashtabelle getestet. Ist man beim letzten Platz $m-1$ angekommen, dann macht man beim ersten Platz 0 wieder weiter.

Definition 5.5. Gegeben ist eine einfache Hashfunktion $h' : U \to \{0, 1, \ldots, m-1\}$. Die Sondierungsreihenfolge beim Verfahren *lineares Sondieren* ist gegeben durch

$$h(k, i) = (h'(k) + i) \bmod m,$$

wobei $i = 0, 1, \ldots, m-1$.

Beispiel 5.3. Lineares Sondieren mit $h'(k) = k \bmod m$ für $m = 8$.

Schlüssel und Wert der Hashfunktion:

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

Die Hashtabelle ist dann wie folgt belegt:

0	1	2	3	4	5	6	7
14	16	10	19			22	31

Die durchschnittliche Zeit für eine erfolgreiche Suche ist $\frac{9}{6} = 1,5$ (siehe Tabelle).

k	10		19		31		22		14		16		
	1	+	1	+	1	+	1	+	3	+	2	=	9

Diskussion. Lange belegte Teilstücke tendieren dazu schneller zu wachsen als kurze. Dieser unangenehme Effekt wird *Primäre Häufungen* genannt. Auch gibt es statt der gewünschten $m!$ nur m verschiedene Sondierungsfolgen, da die erste berechnete Position die gesamte Sequenz festlegt: $h'(k), h'(k)+1, \ldots, m-1, 0, 1, \ldots, h'(k)-1$.

Analyseergebnisse. Für die Anzahl der Sondierungen im Durchschnitt gilt (ohne Beweis, mit $0 < \alpha = \frac{n}{m} < 1$):

Erfolglose Suche $\approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$

Erfolgreiche Suche $\approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$

5.3.2 Quadratisches Sondieren

Beim quadratischen Sondieren hängt die Sondierungsreihenfolge von einer quadratischen Funktion ab.

Definition 5.6. Gegeben ist eine einfache Hashfunktion $h' : U \to \{0, 1, \ldots, m-1\}$. Die Sondierungsreihenfolge bei *quadratischem Sondieren* ist gegeben durch

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

wobei $i = 0, 1, \ldots, m-1$ und c_1 und c_2 jeweils kleine Konstanten sind. Man muss jedoch bei der Wahl von c_1 und c_2 darauf achten, dass der Wert $c_1 i + c_2 i^2$ ganzzahlig ist.

Beispiel 5.4. Quadratisches Sondieren mit $h'(k) = k \bmod m$ für $m = 8$ und $c_1 = c_2 = \frac{1}{2}$, und den gleichen Schlüsseln wie vorhin. Schlüssel und Wert der Hashfunktion:

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

0	1	2	3	4	5	6	7
		10	19			22	31

$$\begin{aligned} 14 \to 6 &\to 6 + \frac{1}{2}(1 + 1^2) \bmod 8 = 7 \\ &\to 6 + \frac{1}{2}(2 + 2^2) \bmod 8 = 1 \end{aligned}$$

0	1	2	3	4	5	6	7
16	14	10	19			22	31

Die durchschnittliche Zeit für eine erfolgreiche Suche beträgt $\frac{8}{6} = 1,33$ (siehe Tabelle).

k	10		19		31		22		14		16		
	1	+	1	+	1	+	1	+	3	+	1	=	8

Anmerkung 5.4. Eine beliebte Wahl ist $c_1 = 0$ und $c_2 = 1$.

Diskussion. Wie beim linearen Sondieren gibt es nur m verschiedene Sondierungsfolgen, denn auch hier legt die erste berechnete Position die gesamte Sondierungsfolge fest. Dieser Effekt wird *Sekundäre Häufungen* genannt.

Analyseergebnisse. Die Anzahl der Sondierungen im Durchschnitt beträgt (ohne Beweis):

Erfolglose Suche $\approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$

Erfolgreiche Suche $\approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$

5.3.3 Double Hashing

Bei Double Hashing hängt die Sondierungsreihenfolge von einer zweiten Hashfunktion ab, die unabhängig von der ersten Hashfunktion ist. Die Sondierungsreihenfolge hängt in zweifacher Weise vom Schlüssel ab: Die erste Sondierungsposition wird wie bisher berechnet, die Schrittweite der Sondierungsreihenfolge hingegen wird durch die zweite Hashfunktion bestimmt. Es ergeben sich $\Theta(m^2)$ verschiedene Sondierungsfolgen.

Definition 5.7. Gegeben sind zwei Hashfunktionen $h_1, h_2 : U \to \{0, 1, \ldots, m-1\}$. Die Sondierungsreihenfolge bei *Double Hashing* ist gegeben durch

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m,$$

wobei $i = 0, 1, \ldots, m-1$.

Damit die Hashtabelle vollständig durchsucht werden kann, muss für alle Schlüssel k die zweite Hashfunktion $h_2(k)$ relativ prim zu m sein :

$$ggT(h_2(k), m) = 1.$$

Falls nämlich $ggT(h_2(k), m) = d > 1$ ist, dann wird nur $\frac{1}{d}$-tel der Hashtabelle durchsucht bzw. beim Einfügen ausgenutzt.

Zwei Vorschläge zur Wahl von $h_1(k)$ und $h_2(k)$:

(1) $m = 2^p$, $p \in \mathbb{N} \wedge p > 1$ (schlecht für Divisionsmethode), $h_2(k)$ immer ungerade

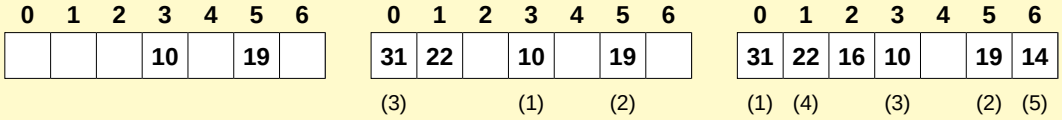
(2) m Primzahl, $0 < h_2(k) < m$, z.B.

$$\begin{aligned} h_1(k) &= k \bmod m \\ h_2(k) &= 1 + (k \bmod m') \end{aligned}$$

mit $m' = m - 1$ oder $m' = m - 2$.

Beispiel 5.5. Double Hashing für Tabellengröße $m = 7$, $h_1(k) = k \bmod 7$ und $h_2(k) = 1 + (k \bmod 5)$

k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2



Die durchschnittliche Zeit für eine erfolgreiche Suche ist $\frac{12}{6} = 2.00$ (siehe Tabelle). Dies ist jedoch ein untypisch schlechtes Beispiel für Double Hashing.

k	10		19		31		22		14		16		
	1	+	1	+	3	+	1	+	5	+	1	=	12

Diskussion. Double Hashing funktioniert in der Praxis sehr gut. Es ist eine gute Approximation an uniformes Hashing, denn man kann zeigen, dass der theoretische Unterschied zwischen beiden Verfahren klein ist, wenn $h_2(k)$ unabhängig von $h_1(k)$ ist. Ein weiterer Vorteil liegt darin, dass Double Hashing sehr leicht zu implementieren ist.

Verbesserung nach Brent [1973]. Für Fälle, bei denen häufiger gesucht als eingefügt wird (z.B. Einträge in Symboltabellen für Compiler), ist es von Vorteil, die Schlüssel beim Einfügen so zu reorganisieren, dass die Suchzeit verkürzt wird. Brent schlug das folgende Verfahren vor. Wenn beim Einfügen eines Schlüssels k ein sondierter Platz j belegt ist und $k' = T[j].key$ der dazugehörige Schlüssel ist, dann berechne

$$\begin{aligned} j_1 &= j + h_2(k) \bmod m \\ j_2 &= j + h_2(k') \bmod m. \end{aligned}$$

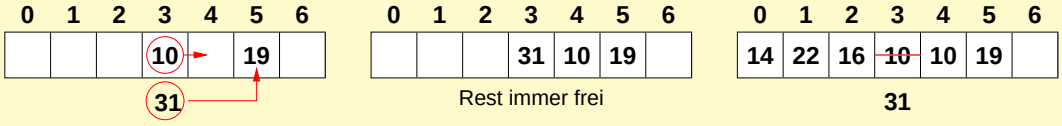
Ist Platz j_1 frei oder Platz j_2 belegt, so fährt man mit j_1 fort wie in der ursprünglichen Double-Hashing-Methode. Andernfalls (j_1 belegt aber j_2 frei) trägt man k' in $T[j_2]$ ein und k in $T[j]$ (nicht in $T[j_1]$!).

Statt also noch lange nach einem freien Platz für k zu suchen, trägt man k an Platz j ein, denn für den Schlüssel k', der vorher dort war, konnte leicht ein freier Platz (gemäß der Sondierungsreihenfolge für k') gefunden werden. Der Pseudocode zum Einfügen nach Brent ist in Algorithmus 5.3 angegeben.

```
 1: ▷ Repräsentation eines Hash-Elementes
 2: struct HashElement
 3:     var K key                                  ▷ Schlüssel des Elementes
 4:     var V value                                ▷ Wert des Elementes
 5:     var bool used                              ▷ Ist das Element benutzt oder noch frei
 6: end struct

 7: ▷ Interne Repräsentation
 8: var HashElement T[0..m − 1]

 9: ▷ Initialisierung
10: for i := 0, ..., m − 1 do
11:     T[i].used := false
12: end for

13: ▷ Brents Variation von doppeltem Hashing
14: procedure INSERT(K k, V v)
15:     j := h_1(k)
16:     while T[j].used do
17:         k' := T[j].key
18:         j_1 := (j + h_2(k)) mod m
19:         j_2 := (j + h_2(k')) mod m
20:         if not T[j_1].used or T[j_2].used then
21:             j := j_1
22:         else
23:             v' := T[j].value
24:             T[j].key := k;   T[j].value := v
25:             k := k';   v := v'
26:             j := j_2
27:         end if
28:     end while
29:     T[j].key := k:   T[j].value := v
30:     T[j].used := true
31: end procedure
```

Listing 5.3: Einfügen nach Brent $(\mathbf{var}\ T, k)$

Beispiel 5.6. Angewendet auf unser Beispiel:



Die durchschnittliche Zeit für eine erfolgreiche Suche ist $\frac{7}{6} = 1.17$ (siehe Tabelle).

k	10		19		31		22		14		16		
	2	+	1	+	1	+	1	+	1	+	1	=	7

Anmerkung 5.5. Bei Anwendung von Brent für das Einfügen von Elementen kann der Algorithmus zum Suchen oder Entfernen von Elementen ganz normal angewendet werden. Denn selbst wenn nach einem Element gesucht wird, das mit Hilfe von Brent einen Platz weiter gemäß seiner Soniderungsreihenfolge verschoben worden ist, wird dieses gefunden. Der Platz an dem es vor der Verschiebung war, wurde ja gleichzeitig mit der Verschiebung von einem anderen Element besetzt.

Analyseergebnisse für Brent. Die Anzahl der Sondierungen beträgt im Durchschnitt:

Erfolglose Suche $\approx \frac{1}{1-\alpha}$ (wie uniform)

Erfolgreiche Suche < 2.5 (unabhängig von α für $\alpha < 1$).

5.4 Übersicht über die Güte der Kollisionsstrategien

In diesem abschließenden Abschnitt wollen wir uns einen Überblick über die verschiedenen Kollisionsstrategien verschaffen. Tabelle 5.1 zeigt einige Werte der theoretischen Analyseergebnisse für verschiedene Auslastungsfaktoren α für das *Hashingverfahren mit Verkettung* sowie die Hashingverfahren *lineares Sondieren* und *quadratisches Sondieren*.

Zum Vergleich sind jeweils die Werte für das *uniforme Sondieren* aufgelistet. Hierbei wird das Ziel erreicht, alle $m!$ möglichen Permutationen als Sondierungsreihenfolge mit gleicher Wahrscheinlichkeit zu erhalten. Man kann zeigen, dass uniformes Sondieren asymptotisch optimal ist. Da es jedoch praktisch sehr aufwändig zu realisieren ist, wird hier auf eine genauere Diskussion verzichtet. Wir können jedoch die folgenden Analysewerte als Vergleichswerte für den „Idealfall“ verwenden.

Analyseergebnisse für uniformes Sondieren. Die Anzahl der Vergleiche ist im Durchschnitt für $\alpha = \frac{n}{m} < 1$:

Erfolglose Suche $\approx \frac{1}{1-\alpha}$

Einfügen $\approx \frac{1}{1-\alpha}$

Erfolgreiche Suche $\approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

α	Verkettung		Hashing mit off. Adr.					
			lineares S.		quadr. S.		uniformes S.	
	erfolgreich	erfolglos	erf.reich	erf.los	erf.reich	erf.los	erf.reich	erf.los
0.5	1.250	0.50	1.5	2.5	1.44	2.19	1.39	2
0.9	1.450	0.90	5.5	50.5	2.85	11.40	2.56	10
0.95	1.475	0.95	10.5	200.5	3.52	22.05	3.15	20
1.0	1.500	1.00	—	—	—	—	—	—

Tabelle 5.1: Einige Werte der Analyseergebnisse für verschiedene Auslastungsfaktoren α

Die Tabelle zeigt, dass Hashing mit Verkettung deutlich schnellere Suchzeiten besitzt als die Hashverfahren mit offener Adressierung. Es besitzt jedoch den Nachteil des relativ hohen Speicherbedarfs. Bei den offenen Hashverfahren sieht man, dass das quadratische Sondierungsverfahren – selbst für hohe Auslastungsfaktoren – relativ nah am Idealfall „uniform hashing“ ist. Das Verfahren der linearen Sondierung fällt deutlich ab und ist nicht empfehlenswert.

Die Tabelle enthält keine Daten zu Double Hashing nach Brent. Wir wissen jedoch, dass die erfolglose Suche bei Double Hashing nach Brent mit der für uniform Hashing identisch ist, und erfolgreiche Suche immer kleiner als 2.5 ist. Somit ist Double Hashing auf jeden Fall ein guter Kandidat für Hashingverfahren mit offener Adressierung.

Aus dieser Tabelle kann man ablesen, wie hoch die Unterschiede bei den erwarteten durchschnittlichen Suchzeiten je nach Füllgrad der Hashtabelle sind. Bei einer Hashtabelle, die zu 90% gefüllt ist, ist die Suchzeit bei quadratischem und uniformem Hashing ungefähr fünf Mal so hoch wie diejenige einer Tabelle, die nur zu 50% gefüllt ist. Übersteigt der Belegungsfaktor jedoch die kritische Zahl 90%, dann steigt die Suchzeit drastisch an. Deswegen sollte der Belegunsfaktor nicht größer als 90% sein.

Die Werte für Hashing mit Verkettung hingegen sind deutlich besser als diejenigen für Hashingverfahren mit offener Adressierung. Allerdings benötigt die Verfolgung der Zeiger durch die verketteten Listen deutlich höheren Aufwand als das einfache Durchlaufen eines Arrays.

Kapitel 6

Graphen

Viele Aufgabenstellungen lassen sich mit Hilfe graphentheoretischer Konzepte modellieren. *Graphen* beschreiben Objekte und deren Beziehungen zueinander. Sie modellieren also diskrete Strukturen, wie z.B. Netzwerke oder Prozesse. Betrachten wir als Beispiel das Bahnnetz von Deutschland. Die Objekte (so genannte *Knoten*) können hier die Bahnhöfe (und eventuell Weichen) sein, die Beziehungen (so genannte *Kanten*) die Schienen des Bahnnetzes. Belegt man die Kanten mit zusätzlichen Gewichten oder Kosten, so kann man damit verschiedene Probleme formulieren. Bedeutet das Gewicht einer Kante zwischen zwei Knoten X und Y die Fahrzeit eines Zuges von X nach Y, so besteht das Problem des *Kürzesten Weges* darin, die minimale Fahrzeit von einem beliebigen Bahnhof A zu einem beliebigen Bahnhof B zu ermitteln. Andere Beispiele, die mit Hilfe von graphentheoretischen Konzepten gelöst werden können, sind *Routenplanungsprobleme*. Hier sollen beispielsweise Güter von mehreren Produzenten zu den Verbrauchern transportiert werden.

Ein klassisches Graphenproblem ist das „Königsberger Brückenproblem“, das Euler 1736 gelöst und damit die Theorie der Durchlaufbarkeit von Graphen gegründet hat. Am konkreten Beispiel der Stadt Königsberg sollte die Frage beantwortet werden, ob es einen Rundweg gibt, bei dem man alle sieben Brücken der Stadt genau einmal überquert und wieder zum Ausgangspunkt zurückgelangt. Euler konnte zeigen, dass es keinen solchen Rundweg geben kann. Da es dabei nicht auf die genaue Lage der Brücken ankommt — man muss nur wissen, welche Bereiche der Stadt durch die Brücken verbunden werden —, spricht man hierbei von einem *topologischen Problem*.

Ein einflussreiches Forschungsgebiet der topologischen Graphentheorie ist das *Vierfarbenproblem*. Die ursprüngliche Aufgabe besteht darin, eine Landkarte so zu färben, dass jeweils benachbarte Länder unterschiedliche Farben bekommen. Dabei sollen möglichst wenige Farben verwendet werden. Während ein Beweis, dass dies immer mit maximal fünf Farben möglich ist, relativ schnell gefunden wurde, existiert bis heute nur ein sogenannter Computerbeweis (erstmals 1976 von Appel und Haken) dafür, dass auch vier Farben immer ausreichen.

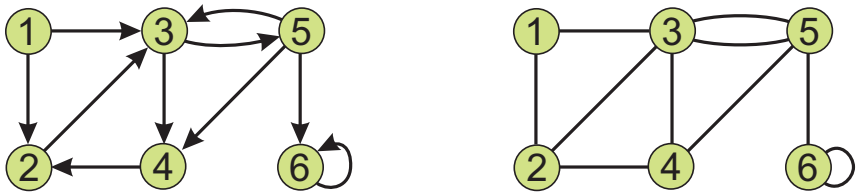


Abbildung 6.1: Beispiele eines (a) gerichteten und (b) ungerichteten Graphen

Im Bereich der chemischen Isomere (Moleküle, die die gleichen Atome enthalten, aber unterschiedliche Struktur haben) ist es für die Industrie im Rahmen der Entwicklung neuer künstlicher Stoffe und Materialien von großer Bedeutung, folgende bereits 1890 von Caley gestellte Frage zu beantworten: Wie viele verschiedene Isomere einer bestimmten Zusammensetzung gibt es? Caley hat damit die Abzähltheorie von Graphen begründet. Oftmals werden auch Beziehungsstrukturen und soziale Netzwerke mit Hilfe von Graphen dargestellt. Bei der Umorganisation von Betrieben kann man damit z.B. relativ leicht feststellen, wer die „wichtigsten“ bzw. „einflussreichsten“ Mitarbeiter sind. Zunehmend werden auch Betriebsabläufe und Geschäftsprozesse als Graphen modelliert.

6.1 Definition von Graphen

Definition 6.1. Ein *Graph* ist ein Tupel (V, E), wobei V eine endliche Menge von *Knoten* (engl. *vertices, nodes*) und E eine endliche (Multi-)Menge von Paaren von Knoten ist. Sind die Paare in E geordnet $(E \subseteq V \times V)$, spricht man von einem *gerichteten* Graphen oder *Digraphen*; sind sie ungeordnet, dann spricht man von einem *ungerichteten* Graphen.

Die Elemente in E heißen gerichtete bzw. ungerichtete *Kanten* (engl. *edges*); gerichtete Kanten nennt man auch *Bögen* (engl. *arcs*). Eine Kante (v, v) heißt *Schleife* (engl. *self-loop*).

Abbildung 6.1 zeigt je ein Beispiel für einen gerichteten und einen ungerichteten Graphen.

Anmerkung 6.1. Meistens betrachtet man in der Graphentheorie nur Graphen, deren Kanten eine Menge (und nicht Multi-Menge) sind. In diesem Fall kann eine Kante (v, w) nur höchstens einmal im Graphen vorkommen. Gerade in praktischen Anwendungen ist es jedoch sinnvoll, dass (v, w) auch mehrfach als Kante vorkommen darf; in diesem Fall spricht man von einer *Mehrfachkante*. Aus diesem Grund definieren wir die Kanten als Multi-Menge. Die meisten Graphenalgorithmen verarbeiten problemlos auch Graphen mit Mehrfachkanten.

Ist $e = (v, w)$ eine Kante in E, dann sagen wir:

- v und w sind *adjazent*.
- v (bzw. w) und e sind *inzident*; v und w sind die *Endpunkte* von e.
- v und w sind *Nachbarn*.
- e ist eine *ausgehende* Kante von v und eine *eingehende* Kante von w (falls der Graph gerichtet ist).

Für einen gerichteten Graphen $G = (V, A)$ ist die *eingehende Nachbarmenge* eines Knotens $v \in V$ definiert als

$$N^+(v) := \{u \in V \mid (u, v) \in A\}$$

und die *ausgehende Nachbarmenge* analog als $N^-(v) := \{w \in V \mid (v, w) \in A\}$. Außerdem bezeichnen wir mit $A^+(v)$ die Menge der eingehenden Kanten eines Knotens v und mit $A^-(v)$ die Menge der ausgehenden Kanten, d.h.

$$\begin{aligned} A^+(v) &:= \{(u, v) \mid (u, v) \in A\} \\ A^-(v) &:= \{(v, u) \mid (v, u) \in A\}. \end{aligned}$$

Zusätzlichen setzen wir $A(v) := A^+(v) \cup A^-(v)$. Der *Eingangsgrad* $d^+(v) := |A^+(v)|$ eines Knotens v ist die Anzahl seiner eingehenden Kanten; entsprechend ist der *Ausgangsgrad* $d^-(v) := |A^-(v)|$ die Anzahl seiner ausgehenden Kanten. Außerdem setzen wir den *Knotengrad* $d(v) := d^+(v) + d^-(v)$.

Für einen ungerichteten Graphen $G = (V, E)$ bezeichnen wir mit

$$N(v) := \{u \in V \mid (u, v) \in E\}$$

die *Nachbarmenge* eines Knotens $v \in V$ und mit $E(v) := \{(u, v) \mid (u, v) \in E\}$ die Menge der zu v inzidenten Kanten; der Knotengrad $d(v)$ ist die Anzahl der zu v inzidenten Kanten, wobei eine Schleife (v, v) zweimal gezählt wird.

Für ungerichtete Graphen gilt folgendes Lemma.

Lemma 6.1. *In einem ungerichteten Graphen $G = (V, E)$ ist die Anzahl der Knoten mit ungeradem Knotengrad gerade.*

Beweis. Summiert man über alle Knotengrade, so zählt man jede Kante genau zweimal:

$$\sum_{v \in V} d(v) = \underbrace{2 \cdot |E|}_{\text{gerade Zahl}}$$

Da die rechte Seite gerade ist, folgt daraus, dass auch die linke Seite gerade sein muss. Also ist die Anzahl der ungeraden Summanden auch gerade. □

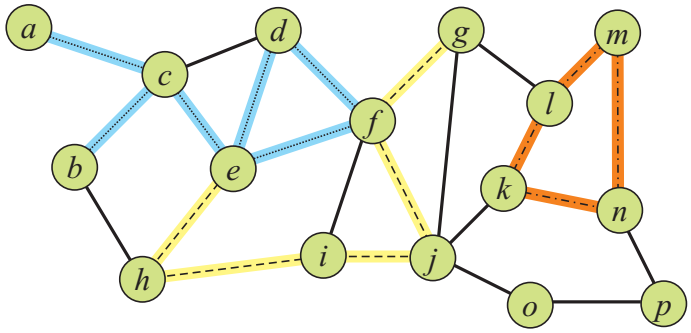


Abbildung 6.2: Ein Kantenzug a, c, e, f, d, e, c, b (blau) der Länge 7, ein Weg g, f, j, i, h, e (gelb) der Länge 5 und ein Kreis k, l, m, n, k (orange) der Länge 4.

Wenn wir den Kanten des Graphen folgen, so können wir Kantenzüge oder Wege im Graphen finden. Formal definieren wir diese Begriffe wie folgt:

Definition 6.2. Sei G ein (gerichteter oder ungerichteter) Graph.

- Ein *Kantenzug* (engl. *walk*) der *Länge* k ist eine nicht-leere Folge

  $$v_0, e_1, v_1, e_2, \ldots, e_k, v_k$$

  von abwechselnd Knoten und Kanten aus G mit $e_i = (v_{i-1}, v_i)$ für $i = 1, \ldots, k$. Wir schreiben einen Kantenzug auch kurz als $v_0, \ldots, v_k$, wenn wir die Kanten nicht näher spezifizieren wollen.
- Ein *Weg* (engl. *path*) ist ein Kantenzug, in dem alle Knoten verschieden sind.
- Ist $v_0, e_1, \ldots, e_{k-1}, v_{k-1}$ ein Weg mit $k \geq 3$ und $e_k = (v_{k-1}, v_0)$ eine Kante aus G, dann ist $v_0, e_1, \ldots, e_{k-1}, v_{k-1}, e_k, v_0$ ein *Kreis* (engl. *cycle*) der *Länge* k in G.

Abbildung 6.2 zeigt je ein Beispiel für einen Kantenzug, einen Weg und einen Kreis.

6.2 Darstellung von Graphen im Rechner

Bei der Darstellung eines Graphen im Rechner ist zunächst zu berücksichtigen, ob wir einen gegebenen Graphen nur intern speichern und nicht weiter modifizieren wollen (*statische Graphen*), oder ob die interne Repräsentation auch Update-Operationen wie z.B. Einfügen eines neuen Knotens unterstützen soll (*dynamische Graphen*).

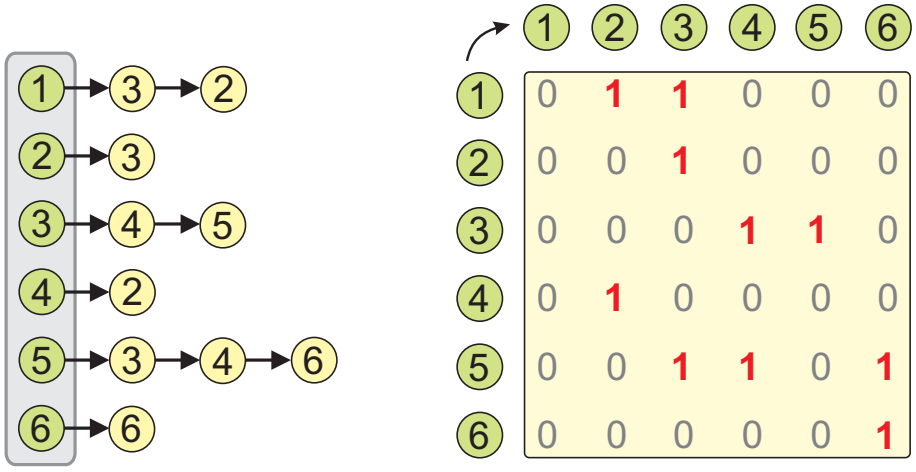


Abbildung 6.3: Repräsentation des gerichteten Graphen aus Abbildung 6.1(a) mit Adjazenzlisten (links) und einer Adjazenzmatrix (rechts).

Wir konzentrieren uns zunächst auf den statischen Fall und betrachten zwei klassische Varianten, um einen Graphen $G = (V, E)$ im Rechner zu speichern. Sie unterscheiden sich in Hinblick auf den benötigten Speicherplatz und die benötigte Laufzeit für typische Anfragen. Wir nehmen im Folgenden an, dass die Knotenmenge $V = \{v_1, \ldots, v_n\}$ ist.

6.2.1 Statische Graphen

Adjazenzlisten. Die am häufigsten verwendete Variante ist die Speicherung des Graphen mittels *Adjazenzlisten*. Dabei wird für jeden Knoten seine ausgehende Nachbarmenge gespeichert. Man kann diese Darstellung mit Hilfe von einfach verketteten Listen implementieren. Die Adjazenzlistendarstellung des gerichteten Graphen aus Abbildung 6.1(a) ist in Abbildung 6.3 (links) skizziert. Bei einem ungerichteten Graphen gibt es für jede Kante (u, v) zwei Listeneinträge, je einen in der zu u gehörenden Adjazenzliste, und einen in der zu v gehörenden; siehe Abbildung 6.4 (links).

Adjazenzmatrizen. Bei der Speicherung des Graphen als *Adjazenzmatrix* wird der Graph G durch eine $n \times n$-Matrix $M = (m_{i,j})$ mit

$$m_{i,j} := \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$

repräsentiert. Wollen wir einen Graphen mit Mehrfachkanten speichern, dann können wir den Eintrag $m_{i,j}$ auf die Anzahl der Kanten (v_i, v_j) setzen. Die Adjazenzmatrizen zu den Graphen aus Abbildung 6.1 sind in den Abbildungen 6.3 und 6.4 jeweils rechts dargestellt.

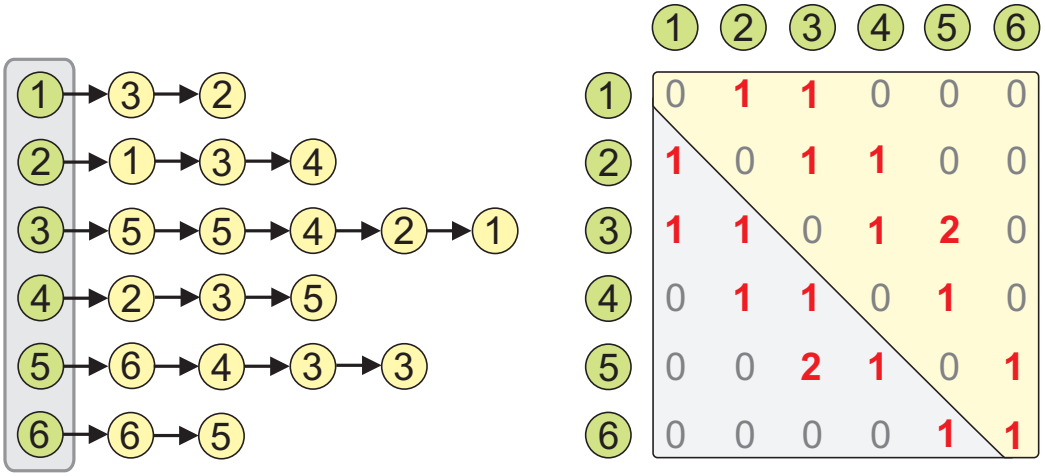


Abbildung 6.4: Repräsentation des ungerichteten Graphen aus Abbildung 6.1(b) mit Adjazenzlisten (links) und einer Adjazenzmatrix (rechts).

Da eine Kante (u, v) im ungerichteten Fall ein ungeordnetes Paar ist, ist die Adjazenzmatrix eines ungerichteten Graphen immer symmetrisch, d.h. $M = M^{\mathrm{T}}$. Daher genügt es in diesem Fall, nur die obere Hälfte der Matrix (inklusive Hauptdiagonale) zu speichern.

Diskussion. Wir überlegen uns zunächst, wie viele Kanten ein Graph mit n Knoten haben kann. Ist der Graph gerichtet und hat weder Mehrfachkanten noch Schleifen, dann kann er maximal $n(n-1)$ viele Kanten haben; im ungerichteten Fall sind es maximal $\frac{1}{2}n(n-1)$ viele. Sind jedoch Mehrfachkanten oder Schleifen erlaubt, dann kann der Graph natürlich beliebig viele Kanten haben. Wir definieren das Verhältnis der Kantenanzahl zur Knotenanzahl als die Dichte des Graphen:

Definition 6.3. Die *Dichte* (engl. *density*) eines Graphen G mit n Knoten und m Kanten ist das Verhältnis m/n.

Eine (unendliche) Familie $\mathcal{G}$ von Graphen $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \ldots$ heißt *dünn*, falls es eine Konstante $c \in \mathbb{R}^+$ gibt, so dass die Dichte jedes Graphen G_i höchstens c ist; $\mathcal{G}$ heißt *dicht*, falls es eine Konstante $c' \in \mathbb{R}^+$ gibt, so dass die Dichte jedes G_i mindestens $c' \cdot |V_i|$ ist.

Anmerkung 6.2. Wir müssen bei der Definition von dünnen und dichten Graphen den etwas umständlichen Weg über eine unendliche Familie von Graphen gehen, da wir für einen einzelnen Graphen $G = (V, E)$ (und sogar eine endliche Familie von Graphen) immer Konstanten c und c' wählen können, so dass $c' \cdot |V| \leq |E|/|V| \leq c$ gilt. Es ist jedoch üblich,

	Adjazenzliste	*Adjazenzmatrix*
Existiert Kante (v, w)?	$\Theta(d(v))$	$\Theta(1)$
Iteration über alle Nachbarn eines Knotens v	$\Theta(d(v))$	$\Theta(\|V\|)$
Iteration über alle ausgehenden Kanten eines Knotens v	$\Theta(d^-(v))$	$\Theta(\|V\|)$
Iteration über alle eingehenden Kanten eines Knotens v	$\Theta(d^+(v))$	$\Theta(\|V\|)$
Platzverbrauch	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\|^2)$

Tabelle 6.1: Laufzeiten und Speicherplatzverbrauch bei Adjazenzlisten und -matrizen.

einfach von dichten und dünnen Graphen zu reden, wobei man davon ausgeht, dass die Konstanten sinnvoll gewählt sind, also z.B. $c \leq 4$ und $c' \geq 1/4$.

Eine wesentliche Eigenschaft der Adjazenzlistendarstellung ist der lineare Speicherverbrauch in der Anzahl der Knoten plus der Anzahl der Kanten. Im Gegensatz dazu benötigt die Adjazenzmatrix immer quadratisch viel Platz in der Anzahl der Knoten, unabhängig davon wie dicht der Graph ist. Insbesondere ist zu beachten, dass alleine der Aufbau der Adjazenzmatrix quadratische Laufzeit in $|V|$ benötigt! Ist der Graph *dünn*, dann benötigt die Adjazenzmatrix also um einen Faktor $\Omega(|V|)$ mehr Platz als die Adjazenzliste. Das ist einer der wichtigsten Gründe, warum man in der Regel die Adjazenzlistendarstellung bevorzugt. Im sehr unrealistischen Fall, dass wir so viele Mehrfachkanten haben, dass die Dichte des Graphen $\omega(|V|)$ ist, kann die Adjazenzliste zwar mehr Platz verbrauchen als die Adjazenzmatrix, allerdings könnte man dann auch die Adjazenzliste anpassen und für Mehrfachkanten einfach jeweils nur einen Repräsentanten mit einer Multiplizität speichern.

Der wichtigste Vorteil der Adjazenzmatrix ist, dass wir in konstanter Zeit entscheiden können, ob eine Kante (u, v) im Graphen enthalten ist. Bei der Adjazenzliste bleibt uns nichts anderes übrig, als die Adjazenzliste von u zu durchlaufen, was $O(d(u))$ Zeit benötigt. Andererseits ist das Iterieren über alle Nachbarn eines Knotens v bei der Adjazenzlistendarstellung günstiger, da lediglich die Adjazenzliste von v durchlaufen werden muss. Bei der Adjazenzmatrix müssen wir eine komplette Zeile betrachten, was $O(|V|)$ Zeit kostet, unabhängig davon wie viele Nachbarn v hat.

Tabelle 6.1 fasst unsere Beobachtungen nochmals zusammen. Für die Effizienz eines Graphenalgorithmus ist es also wichtig, jeweils die richtige Datenstruktur zur Speicherung des Graphen zu wählen. Wir gehen im Folgenden davon aus, dass der Graph durch Adjazenzlisten repräsentiert wird. Außerdem sind die Knotengrade in konstanter Zeit abrufbar. Falls eine Darstellung als Adjazenzmatrix günstiger ist, werden wir explizit darauf hinweisen.

6.2.2 Dynamische Graphen

Wenn wir eine Graph-Datenstruktur in einem Programm verwenden, erwarten wir meist mehr Funktionalität, als lediglich für einen gegebenen Graphen Abfrage-Operationen

ausführen zu können. Es soll darüber hinaus möglich sein, neue Knoten und Kanten hinzuzufügen, sowie vorhandene zu löschen. Natürlich sollen diese Update-Operationen auch möglichst effizient sein. Wir skizzieren im Folgenden eine derartige Datenstruktur, welche einen gerichteten Graphen $G = (V, A)$ mittels Inzidenzlisten repräsentiert und die Update-Operationen *Einfügen von Knoten bzw. Kanten* und *Löschen von Kanten* in konstanter Zeit unterstützt. Beim Löschen eines Knotens v müssen natürlich auch alle zu v inzidenten Kanten gelöscht werden, daher ist die Laufzeit dieser Operation $\Theta(d(v))$.

Anmerkung 6.3. Wir sprechen hier von *Inzidenzlisten*, da wir in den Adjazenzlisten die Listeneinträge als Kanten interpretieren (was sinnvoll ist, da wir bei einem gerichteten Graphen für jede Kante genau einen korrespondierenden Listeneintrag haben), und somit für jeden Knoten die Liste seiner ein- und ausgehenden *inzidenten* Kanten speichern.

Da wir insbesondere das Löschen von Knoten und Kanten effizient realisieren wollen, bietet es sich an, die Knoten in einer doppelt verketteten Liste zu halten, und auch jede Inzidenzliste selbst doppelt zu verketten. Wir haben bereits in Kapitel 2 gesehen, dass doppelt verkettete Listen sowohl Einfügen wie auch Löschen an beliebiger Position in der Liste in konstanter Zeit unterstützen. Für Knoten und Kanten definieren wir daher folgende Strukturen:

```
struct Node
    var Node prev          ▷ Vorgänger in Knotenliste
    var Node next          ▷ Nachfolger in Knotenliste
    var Edge outHead       ▷ Anfang der Liste der ausgehenden Kanten
    var Edge inHead        ▷ Anfang der Liste der eingehenden Kanten
    var int index          ▷ fortlaufender Index
end struct

struct Edge
    var Edge prevOut       ▷ Vorgänger (Liste der ausgehenden Kanten)
    var Edge nextOut       ▷ Nachfolger (Liste der ausgehenden Kanten)
    var Edge prevIn        ▷ Vorgänger (Liste der eingehenden Kanten)
    var Edge nextIn        ▷ Nachfolger (Liste der eingehenden Kanten)
    var Node source        ▷ Anfangsknoten der Kante
    var Node target        ▷ Endknoten der Kante
end struct
```

Trickreich ist dabei die Implementierung der Listen der ein- und ausgehenden Kanten: Jede Kante ist genau einmal in jeder dieser Listen enthalten, daher können wir die Vorgänger- und Nachfolgerverweise für *beide* Listen in der Kantenstruktur verwalten.

Abbildung 6.5 veranschaulicht die Datenstruktur am Beispiel des gerichteten Graphen aus Abbildung 6.1(a). Rechts neben jedem Knoten befindet sich die Liste der ausgehenden Kanten; die Liste der eingehenden Kanten ist durch die blauen Pfeile dargestellt (wir verzichten

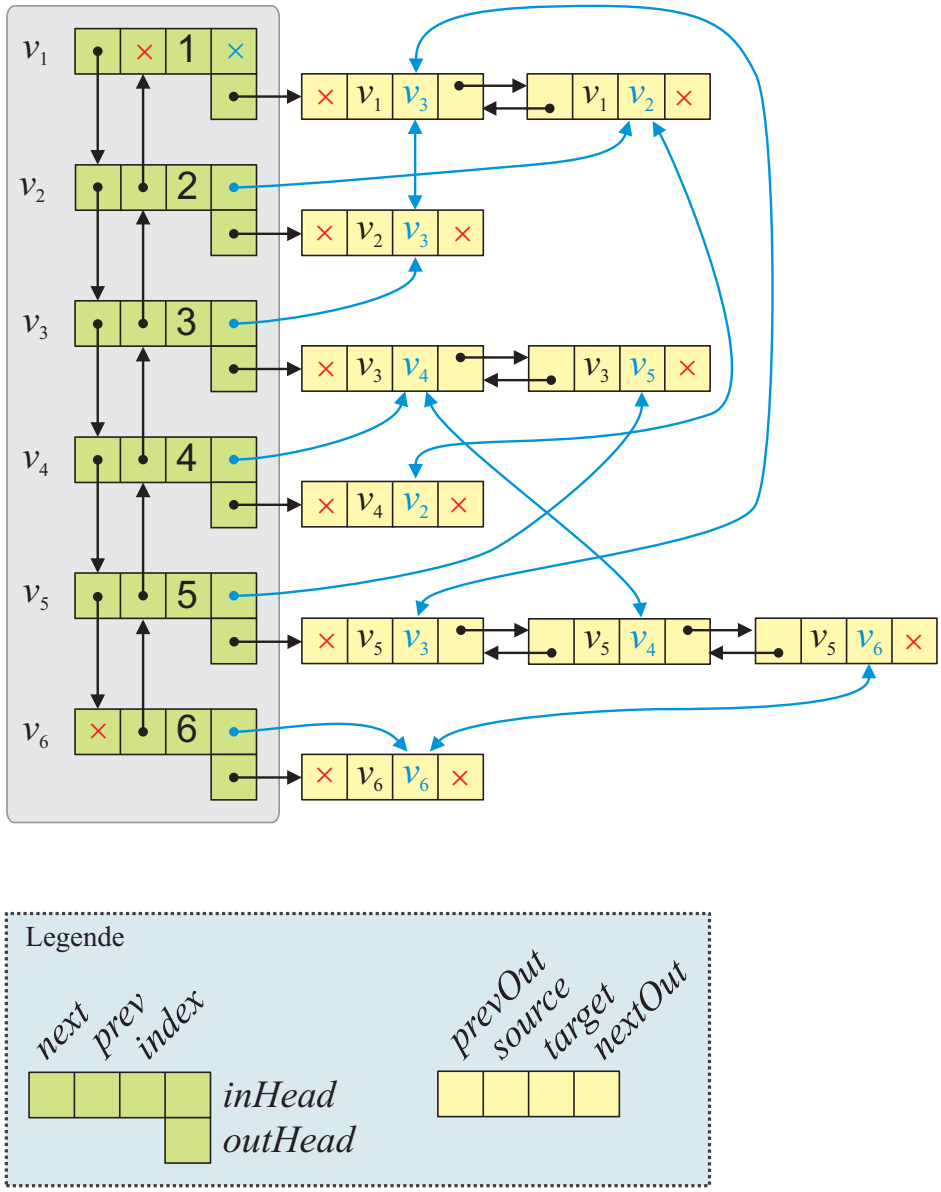


Abbildung 6.5: Interne Inzidenzlistendarstellung des gerichteten Graphen aus Abbildung 6.1(a), die dynamische Updates effizient unterstützt.

Operation	
Existiert Kante (v, w)?	$\Theta(d(v))$
Iteration über alle Nachbarn eines Knotens v	$\Theta(d(v))$
Iteration über alle ausgehenden Kanten eines Knotens v	$\Theta(d^-(v))$
Iteration über alle eingehenden Kanten eines Knotens v	$\Theta(d^+(v))$
Einfügen eines Knotens	$\Theta(1)$
Löschen eines Knotens	$\Theta(d(v))$
Einfügen einer Kante	$\Theta(1)$
Löschen einer Kante	$\Theta(1)$
Platzverbrauch	$\Theta(\|V\| + \|A\|)$

Tabelle 6.2: Laufzeiten und Speicherplatzverbrauch bei dynamischen Graphen.

aus Gründen der Übersichtlichkeit in der Zeichnung auf die explizite Darstellung der Variablen *prevIn* und *nextIn* in den Kantenstrukturen und zeichnen diese mit einem Doppelpfeil). Für Knoten vergeben wir zusätzlich einen fortlaufenden Index $1, 2, \ldots$. Dieser ist nützlich, wenn wir Felder mit Knoten indizieren wollen; der Index im Feld ist dann der beim Knoten abgespeicherte Index.[1] Jede Kante speichert einen Verweis auf ihren Anfangs- und Endknoten. Der auf den ersten Blick redundant erscheinende Verweis auf den Anfangsknoten ist notwendig, da beispielsweise die Operation zum Löschen einer Kante wissen muss, wo der Zeiger auf den Anfang der Inzidenzliste gespeichert ist. Tabelle 6.2 fasst die Laufzeiten typischer Operationen auf dieser Darstellung nochmals zusammen. Die Implementierung der Update-Operationen überlassen wir dem Leser als Übungsaufgabe.

6.3 Traversieren von Graphen

In diesem Abschnitt betrachten wir die wichtigsten Verfahren zum *Traversieren* bzw. Durchsuchen eines *ungerichteten* Graphen: Breitensuche (BFS) und Tiefensuche (DFS). Traversieren bedeutet dabei, systematisch den Kanten des Graphen zu folgen, so dass alle Knoten des Graphen besucht werden. Trotz ihrer Einfachheit bilden diese Traversierungsverfahren das Grundgerüst vieler Graphenalgorithmen, die wir später kennen lernen werden.

6.3.1 Breitensuche (BFS)

Breitensuche (engl. *breadth-first search, BFS*) besucht die Knoten des Graphen nach aufsteigendem, graphentheoretischem Abstand zu einem vorher festgelegten Startknoten.

[1] Analog können wir Indizes für Kanten speichern um Kanten-Felder zu realisieren. Der Übersichtlichkeit halber verzichten wir darauf an dieser Stelle.

Definition 6.4. Der *graphentheoretische Abstand* zweier Knoten u, v eines ungerichteten Graphen G ist die Länge des kürzesten Weges von u nach v, falls ein solcher existiert, sonst ∞.

Zu beachten ist dabei, dass der graphentheoretische Abstand keine Kantenlängen berücksichtigt. Kürzeste Wege bei vorgegebenen Kantenlängen bzw. Kantengewichten werden wir in Abschnitt 6.6 betrachten! Mit Hilfe von Breitensuche kann man also das kürzeste Wege Problem für ungewichtete Graphen lösen:

Unweighted Single-Source Shortest Paths (USSSP)	
Gegeben:	ungerichteter Graph $G = (V, E)$ Startknoten $s \in V$
Gesucht:	ein kürzester Weg von s nach v für jeden Knoten $v \in V$

Die Breitensuche arbeitet nach einem sehr einfachen Prinzip: Wir starten am Knoten s. Wenn wir einen Knoten u besuchen, dann betrachten wir jede zu u inzidente Kante (u, v) — wir sagen auch die Kante (u, v) wird *erforscht*. Für den Knoten v können nun zwei Fälle auftreten:

1. Wir sehen v zum ersten Mal; da wir die Knoten nach aufsteigendem graphentheoretischem Abstand besuchen, muss der Abstand von v zu s um eins größer sein als der Abstand von u zu s. Wir können u besuchen, nachdem wir alle bisher gesehenen Knoten besucht haben.

2. Wir haben v vorher schon einmal gesehen, dann ist nichts zu tun.

Wir müssen also einerseits Knoten markieren, wenn wir sie zum ersten Mal sehen, und andererseits Knoten, die zum ersten Mal gesehen werden, in einer Datenstruktur zwischenspeichern, so dass die Knoten, die zuerst zwischengespeichert wurden, auch zuerst wieder abgerufen werden können. Diese Operationen unterstützt gerade eine Queue (siehe Abschnitt 2.3) effizient.

Der Algorithmus zur Breitensuche und Bestimmung der kürzesten Wege (in ungewichteten Graphen!) ist in Listing 6.1 dargestellt. Zur Unterscheidung des zur Traversierung des Graphen notwendigen Codes von dem Code, der die kürzesten Wege bestimmt, ist letzterer grau hinterlegt. Das Markieren der Knoten erfolgt mit einem Knotenfeld *marked*. Ein Knotenfeld können wir einfach implementieren, wenn wir jedem Knoten einen eindeutigen Index zuordnen und diesen dann zur Indizierung in einem gewöhnlichen Feld verwenden. Im Pseudocode deklarieren wir ein Knotenfeld, indem wir als Indexmenge die Menge der Knoten V angeben, also z.B.:

Eingabe: ungerichteter Graph $G = (V, E)$; Startknoten $s \in V$
Ausgabe: Distanz der Knoten zu s in Feld *dist*, BFS-Baum in Feld π

```
1:  var int dist[V]                                          ▷ Distanz zu s
2:  var Node π[V]                                            ▷ Vorgänger im BFS-Baum
3:  var bool marked[V]
4:  var Node v
5:  for all v ∈ V \ {s} do                                   ▷ Initialisierung
6:      marked[v] := false; dist[v] := ∞; π[v] := nil
7:  end for
8:  var Queue Q
9:  Q.Put(s)                                                 ▷ Beginne mit Startknoten
10: marked[s] := true; dist[s] := 0; π[s] := nil
11: while not Q.IsEmpty() do
12:     var Node u := Q.Get()                                ▷ Knoten u wird besucht
13:     for all v ∈ N(u) do                                  ▷ erforsche Kante (u, v)
14:         if not marked[v] then
15:             Q.Put(v)
16:             marked[v] := true; dist[v] := dist[u] + 1; π[v] := u
17:         end if
18:     end for
19: end while
```

Listing 6.1: Breitensuche (BFS) zur Lösung des USSSP Problems.

var bool *marked*$[V]$

Die kürzesten Wege werden mit Hilfe der folgenden Datenstrukturen dargestellt:

- Der Abstand eines Knotens v zu s wird in *dist*$[v]$ gespeichert; s hat von sich selbst natürlich Abstand 0, noch nicht gesehene bzw. nicht erreichbare Knoten haben Abstand ∞.
- Um den eigentlichen kürzesten Weg von s zu einem Knoten zu speichern, genügt es, wenn wir uns für jeden Knoten v den Vorgänger $\pi[v]$ auf einem kürzesten Weg von s nach v merken. Das Feld π codiert also alle kürzesten Wege nach s in Form eines *BFS-Baums*: s ist die Wurzel des Baumes und $\pi[v]$ der Elter von v.

Beispiel 6.1. Abbildung 6.6 zeigt den Ablauf der Breitensuche an einem Beispiel. Der Startknoten ist dabei der Knoten a. Die einzelnen Schritte visualisieren jeweils den Zustand zu Beginn der **while**-Schleife. Die Knoten werden in der Reihenfolge

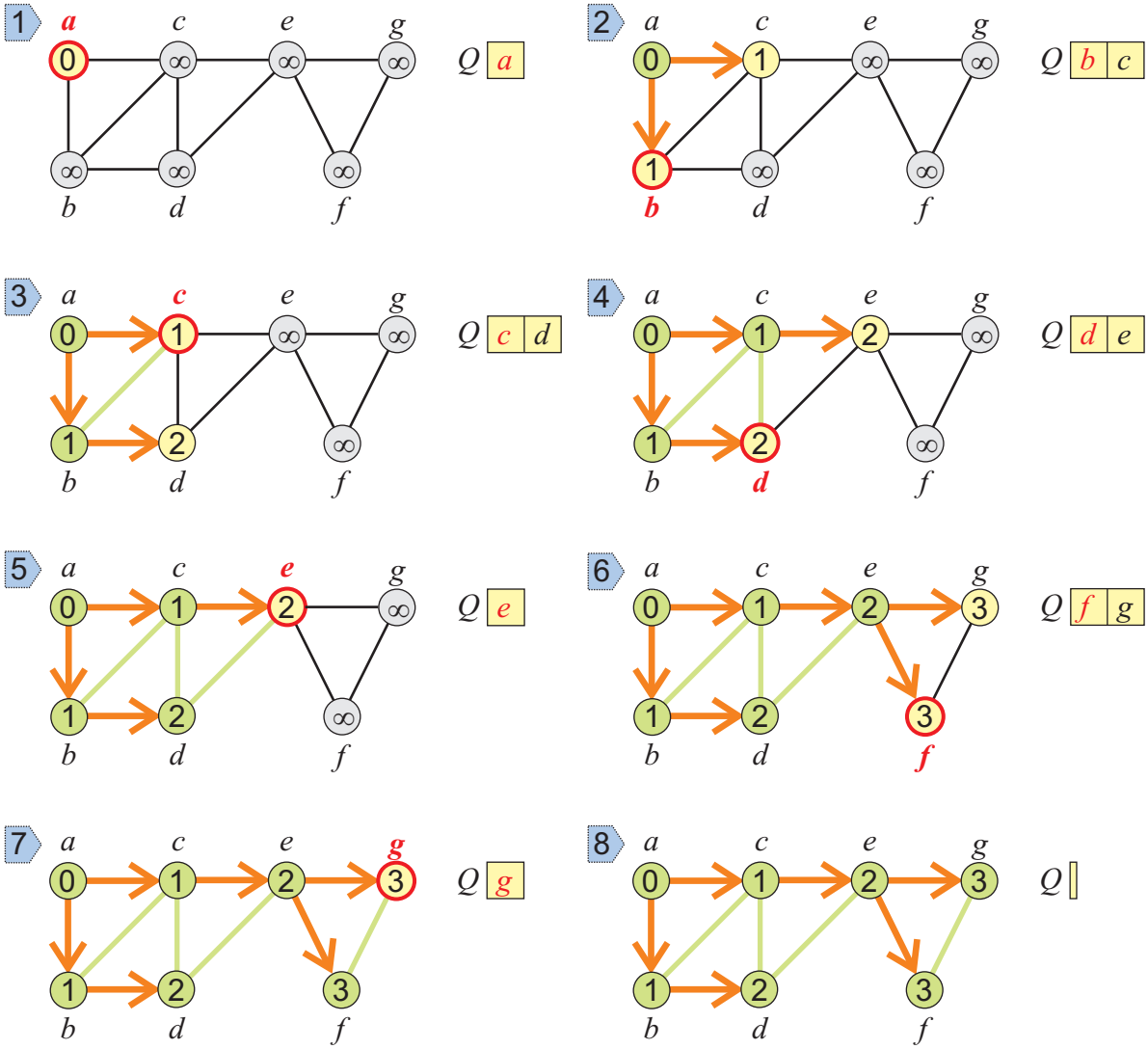


Abbildung 6.6: Ablauf des BFS-Algorithmus zur Lösung des USSSP Problems. Der Zustand der Queue Q wird zu Beginn jeder Iteration der **while**-Schleife gezeigt; die Knoten sind mit $dist[v]$ beschriftet und der durch $\pi[v]$ definierte BFS-Baum wird durch die orangefarbenen Pfeile dargestellt. Grüne Knoten sind bereits besucht, gelbe Knoten sind in der Queue, graue Knoten wurden noch nicht gesehen; der in der aktuellen Iteration besuchte Knoten ist dick rot umrandet.

a, b, c, d, e, f, g besucht. Gemäß dem graphentheoretischen Abstand $0, 1, 2$ und 3 der Knoten zu a erhalten wir 4 Teilfolgen: $\langle a \rangle, \langle b, c \rangle, \langle d, e \rangle, \langle f, g \rangle$

Analyse der Laufzeit von BFS.

- Die Initialisierung (Zeilen 1–10) benötigt $\Theta(|V|)$ Zeit, da die **for all**-Schleife $(|V|-1)$-mal durchlaufen wird und die Knotenfelder in $O(|V|)$ Zeit initialisiert werden können; der Rest benötigt nur konstante Zeit.
- Die **while**-Schleife wird für jeden von s aus erreichbaren Knoten genau einmal durchlaufen, da nur noch nicht markierte Knoten in die Queue kommen, d.h. jeder Knoten kommt höchstens einmal in die Queue hinein. Die **for all**-Schleife in Zeile 13 durchläuft für jeden Knoten v die Liste seiner Nachbarn $N(v)$, verursacht also jeweils $\Theta(d(v))$ Aufwand
- Den Gesamtaufwand können wir somit abschätzen mit

$$\Theta(|V|) + \sum_{v \in V} \Theta(d(v)) = \Theta(|V| + |E|),$$

  da im Worst-Case alle Knoten von s aus erreichbar sind.

Da wir mit Hilfe von BFS insbesondere auch das USSSP lösen können, erhalten wir das folgende Theorem:

Theorem 6.1. *Sei $G = (V, E)$ ein ungerichteter Graph und $s \in V$. Dann löst der in Listing 6.1 dargestellte Algorithmus das Unweighted Single-Source Shortest Paths Problem für G mit Startknoten s in Zeit $O(|V| + |E|)$.*

Anmerkung 6.4. Das hier vorgestellte Verfahren zur Breitensuche besucht nur die Knoten, die auch von s aus erreichbar sind. Natürlich kann man das Verfahren sehr einfach erweitern, so dass alle Knoten besucht werden: Man wählt einfach solange wie möglich einen noch nicht besuchten Knoten und startet von dort aus die Breitensuche. Bei Algorithmen, die die Breitensuche verwenden, ist man aber meist gar nicht an unerreichbaren Knoten interessiert, wie wir es bei der Bestimmung der kürzesten Wege gesehen haben.

6.3.2 Tiefensuche (DFS)

Tiefensuche (engl. *depth-first search*, *DFS*) ist die klassische, rekursive Methode, einen Graphen zu durchlaufen. Wenn wir einen Knoten besuchen, dann markieren wir ihn als besucht und besuchen rekursiv alle zu ihm adjazenten Knoten, die noch nicht markiert wurden.

Der wesentliche Unterschied zu BFS ist dabei, dass wir nicht zuerst alle noch nicht markierten Nachbarn eines Knotens besuchen, bevor ein anderer Knoten besucht wird; stattdessen wird die Suche beim nächsten, noch nicht markierten Nachbarn rekursiv fortgesetzt. Das führt dazu, dass die Suche “tiefer” in den Graphen vordringt, wann immer das möglich ist.

Eingabe: ungerichteter Graph $G = (V, E)$
Ausgabe: DFS-Baum in Feld π

```
 1: var Node π[V]                                   ▷ Vorgänger im DFS-Baum
 2: var bool marked[V]
 3: for all v ∈ V do
 4:     marked[v] := false; π[v] := nil
 5: end for
 6: for all v ∈ V do
 7:     if not marked[v] then DFS-Visit(v)
 8: end for

 9: ▷ rekursiver Aufruf von DFS für alle noch nicht markierten Nachbarknoten von v
10: procedure DFS-Visit(Node v)
11:     marked[v] := true
12:     for all w ∈ N(v) do
13:         if not marked[w] then
14:             π[w] := v
15:             DFS-Visit(w)
16:         end if
17:     end for
18: end procedure
```

Listing 6.2: Tiefensuche (DFS).

Der Basisalgorithmus zur Tiefensuche ist in Listing 6.2 dargestellt. Zunächst werden alle Knoten als nicht-markiert gekennzeichnet und danach wird die Suche (Prozedur DFS-Visit) für den ersten Knoten des Graphen gestartet. Die rekursive Prozedur DFS-Visit(v) arbeitet dann genau wie oben beschrieben: Sie markiert den Knoten und ruft sich selbst für alle noch nicht markierten Nachbarn von v auf.

Das Starten der Suche wird für jeden noch nicht markierten Knoten wiederholt; das ist notwendig, wenn vom Startknoten aus nicht alle Knoten erreicht werden können. Analog zur Breitensuche können wir den Vorgängerknoten im Feld π speichern, welcher im Falle von DFS eine Menge von Bäumen, einen so genannten Wald, ergibt: den *DFS-Wald*. Da das für das Grundschema der Suche nicht notwendig ist, ist der entsprechende Code im Listing wiederum grau hinterlegt. Sind alle Knoten vom ersten Startknoten aus erreichbar, dann ist der Vorgängergraph auch ein Baum, dessen Wurzel der Startknoten ist. In diesem Fall sprechen wir auch von einem *DFS-Baum*.

Beispiel 6.2. Die Arbeitsweise von DFS wird in Abbildung 6.7 an Hand eines Beispiels illustriert. Im Gegensatz zur Queue bei BFS verwaltet DFS die Knoten implizit in einem

Stack, nämlich dem Prozeduraufrufstack. Der aktuell besuchte Knoten (in rot dargestellt) liegt oben auf dem Stack, die Wurzel des Baumes ist das unterste Element. Verdeutlichen Sie sich, dass die Elemente des Stacks einen Weg im Graphen von der Wurzel zum aktuellen Knoten repräsentieren! Jedes Bild stellt den Zustand zu Beginn der Prozedur DFS-VISIT dar. Die Knoten werden in der Reihenfolge a, c, e, f, g, d, b besucht; die Zahlen in den Knoten spiegeln diese Reihenfolge wieder; noch nicht besuchte Knoten sind grau. Die orangefarbenen Pfeile stellen den DFS-Baum dar.

Analyse der Laufzeit von DFS.

- Die Initialisierung der Felder (Zeilen 1–5) benötigt offensichtlich Zeit $\Theta(|V|)$.
- Die **for all**-Schleife in den Zeilen 6–8 benötigt ohne die Aufrufe von DFS-VISIT Zeit $O(|V|)$.
- DFS-VISIT wird für jeden Knoten genau einmal aufgerufen, da DFS-VISIT nur für noch nicht markierte Knoten aufgerufen wird.
- Jeder Aufruf von DFS-VISIT(v) (ohne der Kosten der rekursiven Aufrufe) benötigt $\Theta(d(v))$ Zeit, da die **for all**-Schleife $d(v)$-mal durchlaufen wird.
- Insgesamt ergibt das einen Aufwand von

$$\Theta(|V|) + \sum_{v \in V} \Theta(d(v)) = \Theta(|V|) + \Theta(|E|) = \Theta(|V| + |E|).$$

Der DFS-Wald (oder -Baum) partitioniert die Kanten des Graphen in zwei Klassen:

1. *Baumkanten* (engl. *tree edges*, *T-Kanten*). Das sind die Kanten des Graphen, die den Kanten im DFS-Wald entsprechen, also Kanten der Form $(\pi[v], v)$.
2. *Rückwärtskanten* (engl. *back edges*, *B-Kanten*). Das sind die restlichen Kanten. Für jede Rückwärtskante (u, v) gibt es einen Weg $r, \ldots, v, \ldots, u$ im DFS-Wald, wobei r die Wurzel des Baumes ist, der u und v enthält.

Beispiel 6.3. Abbildung 6.8 zeigt den BFS- und DFS-Baum für unseren Beispielgraphen nebeneinander. In beiden Fällen wurde die Suche im Knoten a gestartet, d.h. a ist jeweils die Wurzel des Baumes. Die Höhe des BFS-Baumes für einen vorgegebenen Startknoten ist eindeutig bestimmt, da die Tiefe eines Blattes v genau der graphentheoretische Abstand im Graphen zur Wurzel des BFS-Baumes ist. Daraus folgt, dass der BFS-Baum minimale

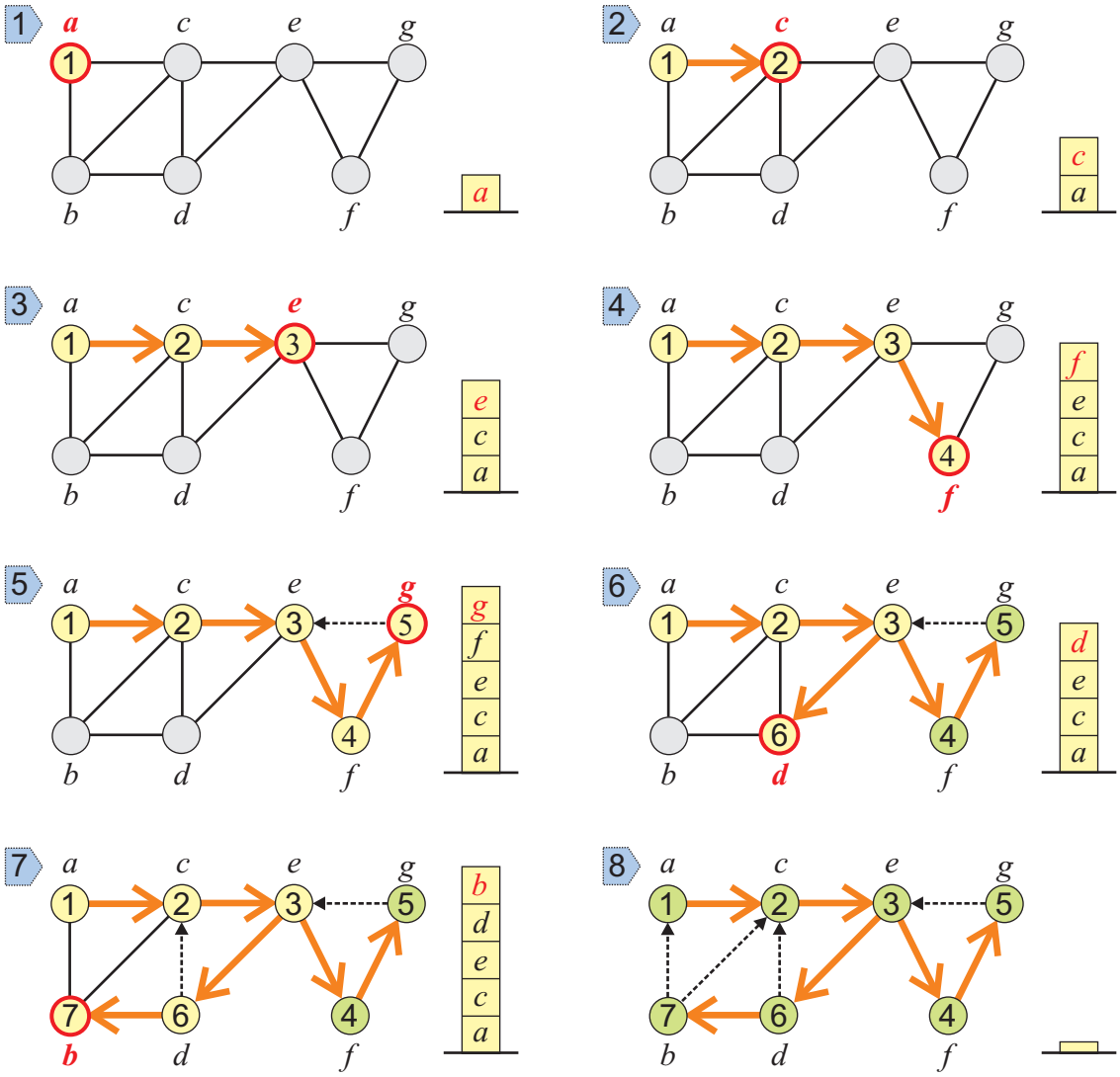


Abbildung 6.7: Ablauf des Algorithmus DFS. Die Beschriftung der Knoten ist die Reihenfolge, in der sie besucht werden, und der durch $\pi[v]$ definierte DFS-Baum wird durch die orangefarbenen Pfeile dargestellt; gefundene B-Kanten sind gestrichelt. Grüne Knoten sind bereits besucht, gelbe Knoten sind im Stack, graue Knoten wurden noch nicht gesehen; der in der aktuellen Iteration besuchte Knoten ist dick rot umrandet.

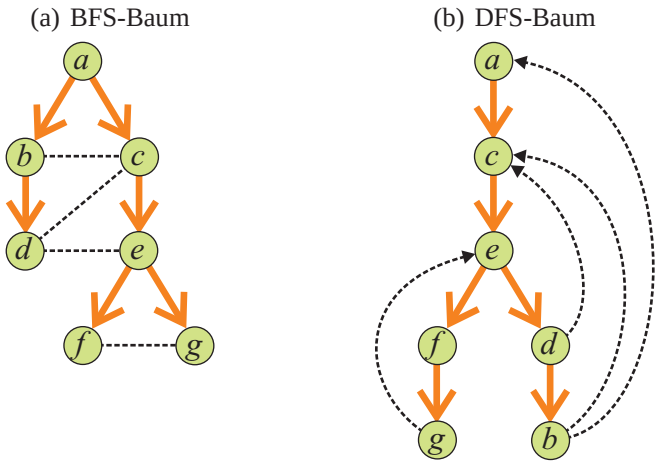


Abbildung 6.8: Der BFS- und der DFS-Baum in unseren Beispielen.

Höhe hat — wir können mit Hilfe der Kanten des Graphen keinen Baum mit gleicher Wurzel und kleinerer Höhe konstruieren.

Andererseits kann der DFS-Baum auch höher sein. In unserem Beispiel ist seine Höhe um eins größer als die des BFS-Baumes. Überlegen Sie sich, ob die Höhe des DFS-Baumes immer maximal ist. Ist sie überhaupt eindeutig? Oder ist es in unserem Beispiel möglich, einen DFS-Baum mit gleicher Wurzel zu erhalten, der aber eine größere Höhe hat?

6.4 Elementare Graphenalgorithmen

In diesem Abschnitt behandeln wir einige einfache, elementare Graphenalgorithmen. Die ersten beiden basieren auf der uns nun wohlbekannten Tiefensuche.

6.4.1 Die Komponenten eines Graphen

Wir haben bereits gesehen, dass wir mit Hilfe von DFS feststellen können, welche Knoten von einem Startknoten aus erreicht werden können: Alle Knoten in einem Baum des DFS-Waldes sind von der Wurzel aus erreichbar. Allgemein können wir sogar folgern, dass es zwischen zwei Knoten genau dann einen Weg im Graphen gibt, wenn sie im gleichen Baum des DFS-Waldes liegen. Das führt uns zu dem Problem, die *Komponenten* eines Graphens zu bestimmen:

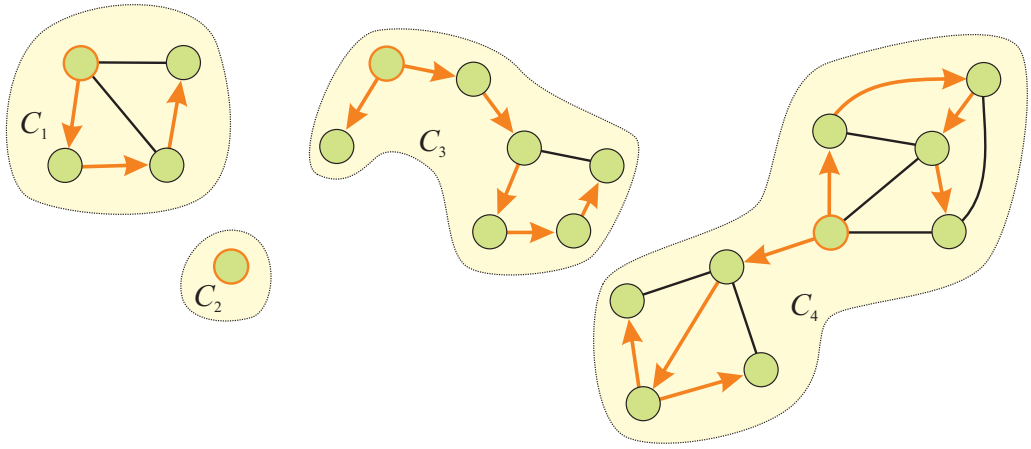


Abbildung 6.9: Ein Graph mit Komponenten $C_1, \ldots, C_4$ und zugehörigem DFS-Wald.

Definition 6.5. Sei $G = (V, E)$ ein ungerichteter Graph.

- G heißt *zusammenhängend* (engl. *connected*), wenn $|V| \geq 1$ und für jedes Paar u, v von Knoten ein Weg von u nach v in G existiert.
- Ein Graph $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E$ ist ein *Untergraph* (engl. *subgraph*) von G. Wir schreiben $G' \subseteq G$.
- Eine *Komponente* (engl. *component*) von G ist ein maximaler zusammenhängender Untergraph U von G, d.h. es gibt keinen anderen zusammenhängenden Untergraphen U' von G mit $U \subseteq U'$.

Die Komponenten eines Graphen werden häufig auch als *Zusammenhangskomponenten* (engl. *connected components*) bezeichnet. Abbildung 6.9 zeigt einen Graphen mit vier Komponenten. Ein möglicher DFS-Wald für diesen Graphen ist durch die orangefarbenen Pfeile eingezeichnet (die Wurzeln sind die Knoten mit orangefarbenem Rand).

Offensichtlich genügt es, für jede Komponente die Knoten zu bestimmen, die darin enthalten sind, da ja alle inzidenten Kanten eines Knotens zur selben Komponente gehören müssen. In unserer Implementierung bestimmen wir einfach für jeden Knoten v den Index *component*$[v]$ der ihn enthaltenden Komponente, wobei wir die Komponenten $1, 2, \ldots$ durchnummerieren. Der Basisalgorithmus zur Tiefensuche muss dabei nur geringfügig angepasst werden (siehe Listing 6.3). Der Wert von *numComps* ist die Nummer der aktuellen Komponente. Vor jedem neuen Starten von DFS (Zeile 11) wird *numComps* um eins erhöhen. Wir sparen uns das Markierungsfeld *marked* des Basisalgorithmus, indem wir "*marked*$[v] =$ **false**" mit "*component*$[v] = 0$" codieren.

Eingabe: ungerichteter Graph $G = (V, E)$
Ausgabe: *component*$[v]$ ist die ID der Komponente, die v enthält; *numComps* ist die Anzahl der Komponenten

```
1:  var int component[V]                          ▷ Komponente eines Knotens
2:  var int numComps                              ▷ Anzahl Komponenten

3:  procedure COMPONENTS(Graph G = (V, E))
4:      for all v ∈ V do                          ▷ Initialisierung
5:          component[v] := 0
6:      end for
7:      numComps := 0
8:      for all v ∈ V do
9:          if component[v] = 0 then
10:             numComps := numComps + 1          ▷ neue Komponente
11:             DFS-COMP(v)
12:         end if
13:     end for
14: end procedure

15: procedure DFS-COMP(Node v)
16:     component[v] := numComps
17:     for all w ∈ N(v) do
18:         if component[w] = 0 then DFS-COMP(w)
19:     end for
20: end procedure
```

Listing 6.3: Bestimmung der Komponenten eines Graphen.

Offensichtlich verändern unsere Modifikationen das asymptotische Laufzeitverhalten der Tiefensuche nicht. Daher erhalten wir das folgende Theorem:

Theorem 6.2. *Der Algorithmus in Listing 6.3 bestimmt die Komponenten eines Graphen $G = (V, E)$ in Zeit $\Theta(|V| + |E|)$.*

6.4.2 Kreise in Graphen

Im diesem Abschnitt betrachten wir das Problem festzustellen, ob ein Graph G einen Kreis enthält oder nicht. Sei F ein DFS-Wald für G. Enthält G keine Rückwärtskante bezüglich F, dann enthält F offenbar alle Kanten von G; da ein Wald offensichtlich keinen Kreis enthält, folgt daraus, dass auch G kreisfrei ist.

Eingabe: ungerichteter Graph $G = (V, E)$
Ausgabe: Kreis in G, falls einer existiert

```
 1: var Node π[V]                                          ▷ Vorgänger im DFS-Baum
 2: var bool marked[V]                                     ▷ Markierung für DFS-Suche
 3: var EdgeSet B                                          ▷ Menge der Rückwärtskanten
 4: function IsAcyclic(Graph G = (V, E)) : bool
 5:     B := ∅                                             ▷ Initialisierung
 6:     for all v ∈ V do
 7:         marked[v] := false; π[v] := nil
 8:     end for
 9:     for all v ∈ V do
10:         if not marked[v] then DFS-Acyclic(v)
11:     end for
12:     if B ≠ ∅ then
13:         return false                                   ▷ jede B-Kante induziert einen Kreis
14:     else
15:         return true                                    ▷ kein Kreis vorhanden
16:     end if
17: end function

18: procedure DFS-Acyclic(Node v)
19:     marked[v] := true
20:     for all w ∈ N(v) do
21:         if not marked[w] then
22:             π[w] := v
23:             DFS-Acyclic(w)
24:         else if π[v] ≠ w then
25:             B := B ∪ (v, w)        ▷ Rückwärtskante gefunden (siehe Anmerkung 6.5)
26:         end if
27:     end for
28: end procedure
```

Listing 6.4: Test auf Kreisfreiheit.

Gibt es andererseits eine Rückwärtskante (u, v) bezüglich F, dann gibt es auch einen Weg $\varphi = v, \ldots, u$ von Baumkanten in F. Somit bilden φ und die Kante (u, v) einen Kreis in G. Wir erhalten also folgendes Lemma.

Lemma 6.2. *Sei G ein ungerichteter Graph und F ein beliebiger DFS-Wald für G. Dann ist G genau dann kreisfrei, wenn G keine Rückwärtskante bezüglich F enthält.*

Die Funktion ISACYCLIC in Listing 6.4 realisiert den Test auf Kreisfreiheit mit Hilfe von DFS. Wir speichern dabei die Menge der Rückwärtskanten in B: Erforschen wir in DFS-ACYCLIC(v) eine Kante, die zu einem bereits markierten Knoten u führt, dann fügen wir diese Kante zu der Menge B hinzu, falls u nicht der Elter von v im DFS-Wald ist. Nachdem DFS abgeschlossen wurde, müssen wir nur noch testen, ob $B = \emptyset$ ist. Ist dies nicht der Fall, dann gibt es einen Kreis in G. Wir könnten einen Kreis ausgeben, indem wir eine beliebige Kante $(v, w) \in B$ wählen und den durch π kodierten Weg von v zur Wurzel verfolgen, bis wir auf w stoßen.

Da unsere Modifikationen am DFS-Basisalgorithmus auch in diesem Fall das asymptotische Laufzeitverhalten nicht verändern (wir können die Menge B z.B. als Liste speichern), erhalten wir folgendes Resultat.

Theorem 6.3. *Die Funktion* ISACYCLIC *in Listing 6.4 testet einen ungerichteten Graphen $G = (V, E)$ auf Kreisfreiheit in Zeit $\Theta(|V| + |E|)$.*

Anmerkung 6.5. Für jede B-Kante (v, w) wird die Anweisung in Zeile 25 der Prozedur DFS-ACYCLIC zweimal aufgerufen:

(i.) Zum ersten Mal, wenn vom Knoten v aus die Kante (v, w) erforscht wird und sich w noch auf dem Rekursionsstack befindet.

(ii.) Zum zweiten Mal, wenn die Kante vom Knoten w aus erforscht wird und der DFS-Aufruf für v bereits abgeschlossen ist (d.h. v ist markiert und befindet sich nicht mehr auf dem Rekursionsstack).

Da wir B als Menge deklariert haben, ist das zunächst kein Problem. In einer realen Implementierung würde man B z.B. aber als Liste implementieren, was dazu führt, dass jede B-Kante zweimal in der Liste ist. Eine mögliche Lösung besteht darin, in *marked*$[v]$ drei Markierungszustände zu speichern:

weiß: Der Knoten v wurde noch nicht besucht.
grau: Der Knoten v wurde bereits besucht, der Aufruf DFS-VISIT(v) ist aber noch nicht abgeschlossen.
schwarz: Der Aufruf DFS-VISIT(v) ist bereits abgeschlossen.

D.h. initial sind alle Knoten weiß, am Anfang der Prozedur DFS-VISIT(v) wird v als grau markiert und am Ende als schwarz. Wir modifizieren dann Zeile 25 so, dass wir die Kante (v, w) nur dann zu B hinzufügen, wenn w grau ist.

Anmerkung 6.6. Sind wir nicht an der ganzen Menge der B-Kanten interessiert, dann können wir natürlich die DFS-Suche abbrechen, sobald wir eine B-Kante gefunden haben. Dann wissen wir bereits, dass diese gefundene B-Kante einen Kreis im Graphen verursacht.

6.4.3 Topologisches Sortieren

Im Gegensatz zu den beiden vorangehenden Algorithmen betrachten wir jetzt gerichtete Graphen. Gerichtete Graphen werden häufig zur Darstellung von Abhängigkeiten verwendet.

Nehmen wir als Beispiel den Ablauf zum Bauen eines Hauses. Dazu sind verschiedene Aufgaben zu erledigen, z.B. das Ausheben der Baugrube, das Einsetzen der Fenster, die Fertigstellung des Mauerwerkes, die Fertigstellung des Dachstuhls oder das Eindecken des Daches. Offensichtlich können bestimmte Aufgaben erst begonnen werden, wenn andere abgeschlossen sind. So kann der Dachdecker sicher nicht mit dem Eindecken des Daches beginnen, bevor der Dachstuhl fertig ist. Andererseits sind Aufgaben wie das Einsetzen der Fenster und das Eindecken des Daches unabhängig voneinander. Diese Abhängigkeiten können wir durch gerichtete Kanten modellieren: Kann Aufgabe Y erst dann begonnen werden, wenn Aufgabe X abgeschlossen wurde, dann habe wir eine Kante (X, Y) im Graphen.

Offensichtlich müssen wir zum Bauen des Hauses eine Reihenfolge finden, die alle diese Abhängigkeiten berücksichtigt. Das ist natürlich nicht möglich, wenn der Abhängigkeitsgraph einen Kreis enthält. Daher interessieren wir uns in diesem Abschnitt für gerichtete, kreisfreie Graphen, die man auch als *DAG* bezeichnet.

Definition 6.6. Ein *DAG* (*directed acyclic graph*) ist ein gerichteter Graph, der keinen (gerichteten) Kreis enthält.

Eine mögliche Reihenfolge der zu erledigenden Aufgaben wird als *topologische Sortierung* bezeichnet. Eine topologische Sortierung der Knoten eines DAGs $G = (V, A)$ ist eine lineare Ordnung $<_V$ der Knoten, so dass für jede Kante $(u, v) \in A$ gilt: $u <_V v$. Abbildung 6.10 zeigt ein Beispiel für eine topologische Sortierung der Knoten eines Graphen (von links nach rechts).

Wir werden im Folgenden einen Algorithmus herleiten, der für jeden DAG eine topologische Sortierung bestimmt.

Topologisches Sortieren	
Gegeben:	DAG $G = (V, A)$
Gesucht:	eine Sortierung $v_1, \ldots, v_n$ der Knoten von G mit $i < j$ für alle $(v_i, v_j) \in A$

Wesentlich ist dabei, so genannte Quellen im Graphen zu finden. Eine *Quelle* in einem gerichteten Graphen ist ein Knoten ohne eingehende Kanten, also mit Eingangsgrad 0; analog ist eine *Senke* ein Knoten ohne ausgehende Kanten. Der Algorithmus zum topologischen Sortieren basiert auf der folgenden, einfachen Beobachtung:

Beobachtung 6.1. *Jeder nicht-leere DAG hat mindestens eine Quelle und eine Senke.*

Beweis. Nehmen wir an, wir hätten einen DAG $G = (V, A)$ ohne Senken. Dann können wir von einem Startknoten u_1 beginnend immer ausgehende Kanten weiterverfolgen, da ja jeder

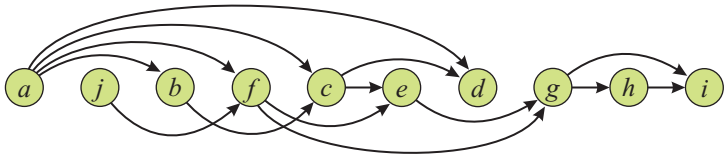


Abbildung 6.10: Eine topologische Sortierung (von links nach rechts) der Knoten eines DAGs.

Knoten ausgehende Kanten besitzt. Wir erhalten also einen Kantenzug $p_i = u_1, u_2, \ldots, u_i$, den wir beliebig lang fortsetzen können. Sobald aber $i > |V|$ ist, muss mindestens ein Knoten mehrmals in p_i vorkommen. Das bedeutet aber, dass es einen (gerichteten) Kreis in G gibt. Das ist ein Widerspruch, da G ein DAG ist. Also besitzt jeder DAG eine Senke. Analog können wir zeigen, dass jeder DAG auch eine Quelle besitzt. □

Da eine Quelle s keine eingehenden Kanten besitzt, ist diese offenbar von keinem Knoten abhängig; also können wir s gefahrlos als ersten Knoten in der topologischen Sortierung wählen. Alle von s ausgehenden Kanten brauchen uns dann nicht mehr zu interessieren, da ja alle anderen Knoten nach s in der Sortierung kommen werden. Das heißt aber gerade, dass wir nach s eine topologische Sortierung für den Graphen $G - s$ (das ist der Graph, der durch Löschen des Knotens s in G entsteht) anhängen können. Das Grundschema unseres Algorithmus ist also wie folgt:

while G ist nicht leer **do**
  Wähle eine Quelle s in G und gib sie aus
  $G := G - s$
end while

Unsere Aufgabe besteht nun darin, daraus einen effizienten Algorithmus zu machen. Folgende Fragestellungen müssen wir klären:

1. *Wie finden wir effizient eine Quelle, ohne jedesmal alle Knoten zu durchlaufen?*

   Neben den Quellen, die im Eingabegraphen schon vorhanden sind, können nur Knoten v zu Quellen werden, wenn ein Knoten gelöscht wird, der eine ausgehende Kante zu v besitzt. Es genügt also, beim Löschen eines Knotens dessen ausgehende Nachbarmenge zu betrachten.

2. *Können wir verhindern, dass wir tatsächlich Knoten im Graphen löschen müssen?*

   Eigentlich sind wir nur an der Veränderung der Eingangsgrade der Knoten interessiert. Anstatt Knoten tatsächlich zu löschen, verwalten wir ein Knotenfeld *indeg* mit dem aktuellen Eingangsgrad und modifizieren dies entsprechend.

Eingabe: azyklischer, gerichteter Graph $G = (V, A)$
Ausgabe: Ausgabe der Knoten von G in topologischer Sortierung

```
 1: var Queue Q
 2: var int indeg[V]
 3: for all v ∈ V do
 4:     indeg[v] := d⁺(v)
 5:     if indeg[v] = 0 then Q.PUT(v)                    ▷ Ist v Quelle?
 6: end for
 7: while not Q.ISEMPTY() do
 8:     v := Q.GET()
 9:     Gib v aus
10:     for all (v, u) ∈ A⁻(v) do
11:         indeg[u] := indeg[u] − 1          ▷ u hat jetzt eine eingehende Kante weniger
12:         if indeg[u] = 0 then Q.PUT(u)                ▷ Ist u jetzt Quelle?
13:     end for
14: end while
```

Listing 6.5: Topologisches Sortieren.

Wenn wir diese Ideen umsetzen, erhalten wir den in Listing 6.5 dargestellten Algorithmus zum topologischen Sortieren.

Beispiel 6.4. Abbildung 6.11 zeigt den Ablauf des Algorithmus zum topologischen Sortieren an einem Beispiel. Die „gelöschten“ Knoten und Kanten sind in grau dargestellt, die aktuellen Quellen in orange. Die Ausgabe des Algorithmus ist

$$a, j, b, f, c, e, d, g, h, i.$$

Die entsprechende topologische Sortierung ist auch in Abbildung 6.10 zu sehen.

Analyse der Laufzeit.

- Initialisierung (Zeilen 1–6): Das Initialisieren von *indeg* und der Queue mit den Quellen des Eingabegraphen hat Laufzeit $\Theta(|V|)$, da wir in konstanter Zeit den Eingangsgrad eines Knotens bestimmen können.
- Jeder Knoten kommt genau einmal in die Queue: Entweder ist er eine Quelle in G, oder sein Eingangsgrad wird durch das Dekrementieren in Zeile 11 auf 0 gesetzt. Daraus folgt, dass die **while**-Schleife genau $|V|$-mal durchlaufen wird.

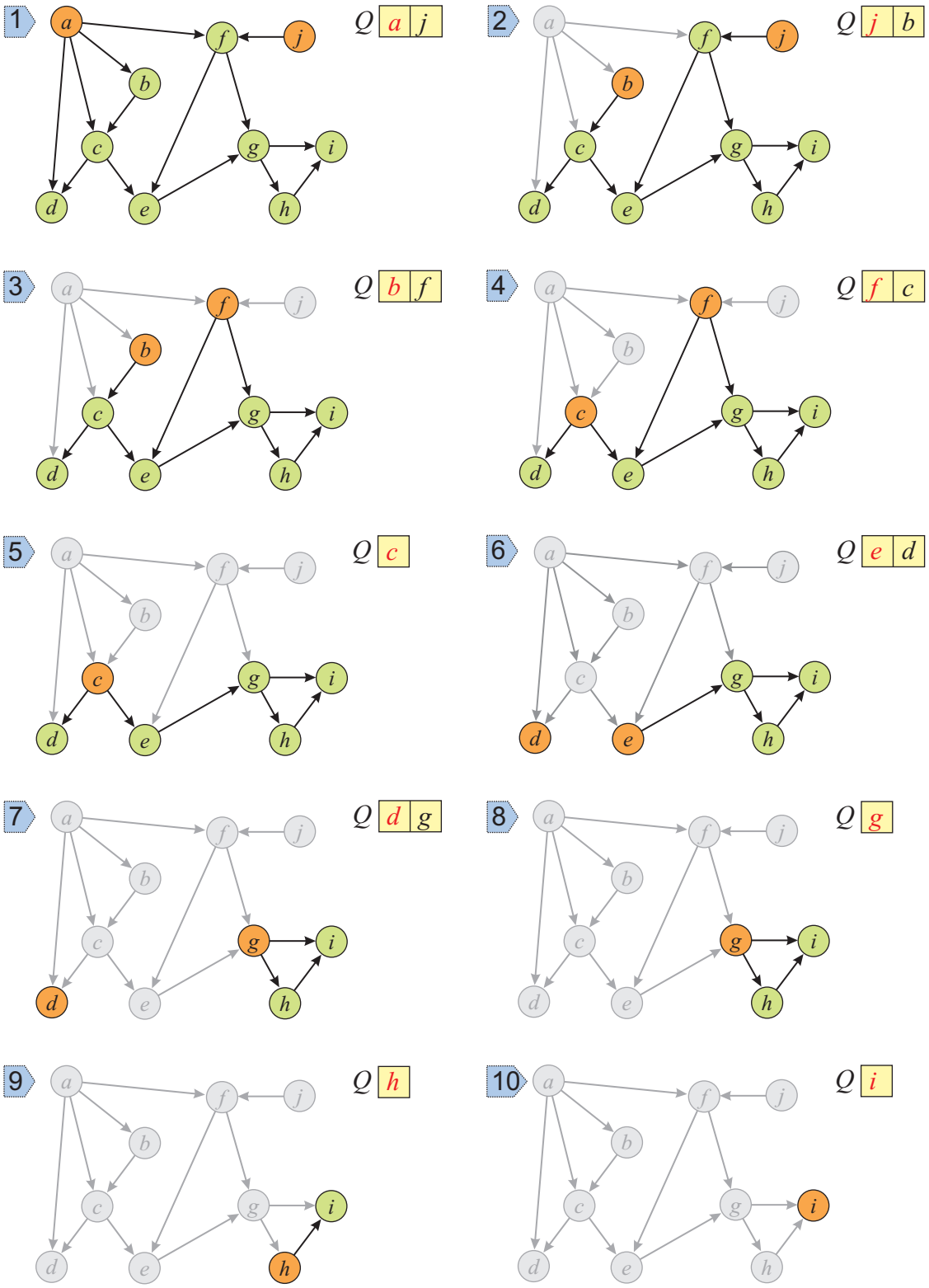


Abbildung 6.11: Ablauf des Algorithmus zum topologischen Sortieren.

- Die **for all**-Schleife in Zeile 10 wird insgesamt für jede Kante genau einmal aufgerufen, das ergibt also einen Aufwand von $\Theta(|A|)$.
- Der Gesamtaufwand is also $\Theta(|V| + |A|)$.

Theorem 6.4. *Der Algorithmus in Listing 6.5 berechnet eine topologische Sortierung der Knoten eines DAGs $G = (V, A)$ in Zeit $\Theta(|V| + |A|)$.*

6.5 Minimale Spannbäume

Im diesem Abschnitt befassen wir uns wieder mit ungerichteten Graphen. Wir nehmen außerdem an, dass die Graphen zusammenhängend sind. Wir interessieren uns für Untergraphen, die Bäume sind.

Definition 6.7. Ein ungerichteter Graph heißt *Baum* (engl. *tree*), wenn er zusammenhängend und kreisfrei (im ungerichteten Sinne!) ist.

Anmerkung 6.7. Bäume im Sinn unserer Definition werden manchmal auch als *freie* Bäume bezeichnet, um sie von *gewurzelten* Bäumen zu unterscheiden. Wählen wir in einem (freien) Baum einen beliebigen Knoten als *Wurzel*, so erhalten wir einen gewurzelten Baum.

Fordern wir in obiger Definition *nicht*, dass der Graph zusammenhängend sein muss, so spricht man von einem *Wald*. Ein Wald ist also ein Graph, dessen Komponenten alle Bäume sind.

Für einen Baum $T = (V, E)$ gilt immer $|V| = |E| + 1$. Ferner gibt es in einem Baum jeweils genau einen ungerichteten Weg zwischen je zwei Knoten. Entfernt man eine Kante aus einem Baum, so zerfällt dieser in genau zwei Komponenten. Man kann sich überlegen, dass die obigen Aussagen jeweils äquivalente Definitionen eines Baumes sind:

Theorem 6.5. *Sei $G = (V, E)$ ein ungerichteter Graph ohne Mehrfachkanten und Schleifen. Dann sind die folgenden Aussagen äquivalent.*

1) *G ist ein Baum.*
2) *Jedes Paar von verschiedenen Knoten in G ist durch einen eindeutigen Weg verbunden.*
3) *G ist zusammenhängend, zerfällt aber durch Entfernen einer beliebigen Kante aus E in zwei Komponenten.*
4) *G ist zusammenhängend und $|E| = |V| - 1$.*
5) *G ist kreisfrei und $|E| = |V| - 1$.*

6) G ist kreisfrei, aber durch Hinzufügen einer beliebigen Kante, die noch nicht in E ist, entsteht ein Kreis.

Beweis. Wir beweisen das Theorem durch Ringschluss.

1) $\Longrightarrow$ 2) Da ein Baum zusammenhängend ist, gibt es zwischen jedem Paar von Knoten mindestens einen Weg in G. Nehmen wir an, zwischen $u, v \in V$ gäbe es zwei verschiedene Wege p_1 und p_2. Sei x der erste Knoten, an dem sich die beiden Wege verzweigen, und sei y der erste Knoten, an dem die beiden Wege wieder zusammentreffen. Dann bilden die Kanten auf p_1 und p_2 auf den entsprechenden Teilstücken zwischen x und y einen Kreis. Das ist ein Widerspruch, da ein Baum kreisfrei ist!

2) $\Longrightarrow$ 3) Da jedes Knotenpaar durch einen Weg verbunden ist, ist G zusammenhängend. Nehmen wir an, es gäbe eine Kante (u, v), so dass G nach Löschen von (u, v) nicht in zwei Komponenten zerfällt. Dann muss es aber zwischen u und v noch einen Weg geben, der die Kante (u, v) nicht benutzt. Das ist aber ein Widerspruch zu Eindeutigkeit des Weges.

3) $\Longrightarrow$ 4) Wir haben bereits im Abschnitt 6.4.1 gesehen, dass jeder zusammenhängende Graph einen DFS-Baum T besitzt, welcher $|V| - 1$ T-Kanten besitzt. Wenn G mehr Kanten besitzt, dann muss es eine B-Kante e in T geben, und der Graph nach Löschen von e ist sicher noch zusammenhängend. Also hat G genau $|V| - 1$ Kanten.

4) $\Longrightarrow$ 5) Da G zusammenhängend und $|E| = |V| - 1$, sind die T-Kanten jedes DFS-Baums von G gerade die Kanten von G. Demzufolge ist G kreisfrei (vgl. Abschnitt 6.4.2).

5) $\Longrightarrow$ 6) Wenn G kreisfrei ist und aus k Komponenten besteht, dann hat G $|E| = |V| - k$ Kanten. Da $|E| = |V| - 1$ vorausgesetzt ist, folgt $k = 1$ und G ist somit zusammenhängend. Also ist jedes Paar von Knoten durch einen Weg verbunden, und das Hinzufügen einer beliebigen neuen Kante erzeugt somit einen Kreis.

6) $\Longrightarrow$ 1) Sei u, v ein beliebiges Paar unterschiedlicher Knoten. Entweder ist (u, v) bereits in G enthalten oder das Hinzufügen der Kante (u, v) erzeugt einen Kreis. Dann muss aber bereits ein Weg von u nach v in G enthalten sein. Daraus folgt, dass G zusammenhängend und somit ein Baum ist. □

Definition 6.8. Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. Ein Untergraph $T = (V, E_T)$ von G heißt *Spannbaum* (auch *aufspannender Baum*, engl. *spanning tree*) von G, falls T ein Baum ist.

Ein Spannbaum von G ist also ein kreisfreier, zusammenhängender Untergraph von G, der alle Knoten von V enthält. Haben wir zusätzlich für die Kanten von G eine Gewichtsfunktion $w : E \to \mathbb{R}$ gegeben, dann definieren wir das Gewicht eines Spannbaums $T = (V, E_T)$ als das Gewicht seiner Kantenmenge:

$$w(T) := w(E_T) := \sum_{e \in E_T} w(e)$$

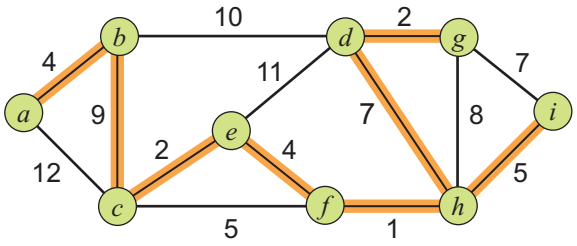


Abbildung 6.12: Ein minimaler Spannbaum eines gewichteten Graphen.

Wir werden im Folgenden zwei Algorithmen kennenlernen, die einen Spannbaum mit minimalem Gewicht berechnen: den *Algorithmus von Kruskal* und den *Algorithmus von Prim*. Solch einen Spannbaum bezeichnet man auch als *minimalen Spannbaum* (engl. *minimum spanning tree*).

Minimum Spanning Tree (MST)	
Gegeben:	ungerichteter, zusammenhängender Graph $G = (V, E)$ Gewichtsfunktion $w : E \to \mathbb{R}$
Gesucht:	ein Spannbaum T von G mit minimalem Gewicht $w(T)$

Beispiel 6.5. Abbildung 6.12 zeigt einen gewichteten Graphen G und einen minimalen Spannbaum von G (fett orange hinterlegte Kanten). Das Gesamtgewicht des MST ist 34. Beachten Sie, dass der MST nicht eindeutig sein muss. So kann in unserem Beispiel die Kante (d, h) durch (g, i) ersetzt werden und wir haben weiterhin einen aufspannenden Baum mit gleichem Gewicht.

Der MST eines gewichteten Graphen ist jedoch dann eindeutig, wenn die Gewichte aller Kanten paarweise verschieden sind.

Minimale Spannbäume haben Anwendung bei der Konstruktion von Netzwerken. So können beispielsweise die Kosten der Kanten die notwendige Länge des Kabels sein, um zwei Netzwerkknoten miteinander zu verbinden. Ein MST beschreibt dann die billigste Möglichkeit, alle Netzwerkknoten miteinander zu verbinden.

Wir zeigen im Folgenden eine wesentliche Eigenschaft von minimalen Spannbäumen, die wir in den Korrektheitsbeweisen der Algorithmen von Kruskal und Prim anwenden werden. Dazu benötigen wir noch zwei Begriffe:

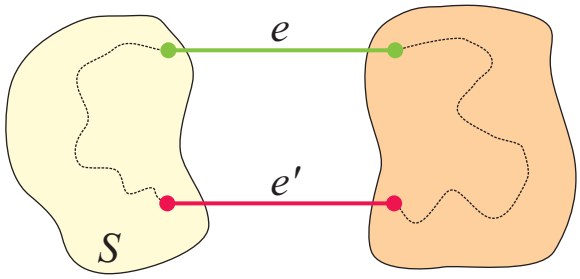


Abbildung 6.13: Beweis von Lemma 6.3.

Definition 6.9. Wir nennen eine Teilmenge T der Kanten von G *aussichtsreich*, wenn es einen MST von G gibt, der alle Kanten aus T enthält. Weiterhin sagen wir, eine Kante (u, v) *verlässt* eine Knotenmenge $S \subset V$, wenn $u \in S$ und $v \notin S$ gilt.

Gemäß dieser Definition ist die leere Menge immer aussichtsreich, und eine aussichtsreiche Menge T mit $|V| - 1$ Kanten ist die Kantenmenge eines MST von G. Das folgende Lemma ist die Grundlage der Algorithmen von Kruskal und Prim.

Lemma 6.3. *Sei $G = (V, E)$ zusammenhängend mit Gewichtsfunktion $w : E \to \mathbb{R}$. Sei*

- *$S \subset V$;*
- *$T \subset E$ aussichtsreich und keine Kante aus T verlässt S; und*
- *$e \in E$ eine Kante mit minimalem Gewicht, die S verlässt.*

Dann ist die Menge $T \cup \{e\}$ aussichtsreich.

Beweis. Da T aussichtsreich ist, gibt es einen MST mit Kantenmenge T' und $T \subset T'$. Falls $e \in T'$, dann gilt das Lemma offensichtlich.

Falls $e \notin T'$, dann erhalten wir durch Hinzufügen von e zu T' genau einen Kreis (vergl. Abbildung 6.13). Auf diesem Kreis muss es eine Kante $e' \neq e$ geben, die S verlässt. Wenn wir e' in T durch e ersetzen, erhalten wir einen neuen Spannbaum mit Kantenmenge $T'' := (T' \setminus \{e'\}) \cup \{e\}$ und Gewicht

$$w(T'') = w(T') - w(e') + w(e).$$

Da e minimales Gewicht einer Kante hat, die S verlässt, gilt $w(e) \leq w(e')$, und somit $w(T'') \leq w(T')$. Daraus folgt, dass T'' ein MST von G ist. Außerdem ist $T \cup \{e\} \subseteq T''$ und somit $T \cup \{e\}$ aussichtsreich. □

6.5.1 Der Algorithmus von Kruskal

Die Idee des Algorithmus von Kruskal besteht darin, die Kanten nach aufsteigendem Gewicht zu betrachten und jeweils zu testen, ob die betrachtete Kante mit den bereits gewählten Kanten einen Kreis verursacht; ist dies nicht der Fall, wird die Kante gewählt:

Sortiere die Kanten nach aufsteigendem Gewicht, so dass
$w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$ gilt.
$T := \emptyset$
for $i := 1$ **to** m **do**
  if $T \cup \{e_i\}$ ist kreisfrei **then** $T := T \cup \{e_i\}$
end for

Der Algorithmus arbeitet offensichtlich sehr kurzsichtig, da er in jedem Schritt einfach die leichteste, noch hinzufügbare Kante wählt. Die Lösung wird also stückweise konstruiert, wobei in jedem Schritt der zu diesem Zeitpunkt günstigste Teil hinzugefügt wird. Solche Algorithmen bezeichnet man als *greedy* (dt. *gierig*). Greedy Algorithmen berechnen nicht notwendigerweise optimale Lösungen. Wir werden jedoch für den Algorithmus von Kruskal zeigen, dass er immer eine optimale Lösung, also einen MST berechnet.

Beispiel 6.6. Abbildung 6.14 zeigt den Ablauf des Algorithmus von Kruskal an dem Beispielgraphen aus Abbildung 6.12. Die bisher gewählten Kanten T sind orange hinterlegt; die einzelnen Komponenten des Graphen, der aus den bisher gewählten Kanten besteht, sind gelb hinterlegt. Die nächste Kante, die zu T hinzugefügt wird, ist die billigste Kante die zwei Komponenten verbindet und ist blau hinterlegt.

Die Schritte, in denen eine Kante betrachtet aber nicht gewählt wird, da ein Kreis entstehen würde (d.h. die Kante verbindet *nicht* zwei Komponenten), sind nicht extra aufgeführt; die Kanten sind dann aber gestrichelt gezeichnet.

Bevor wir uns überlegen, wie wir den Algorithmus von Kruskal effizient realisieren können, zeigen wir seine Korrektheit:

Theorem 6.6. *Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$. Dann berechnet der Algorithmus von Kruskal einen MST von G.*

Beweis. Seien $e_{\sigma(1)}, \ldots, e_{\sigma(k)}$ die Kanten, die Kruskal in dieser Reihenfolge zu T hinzufügt. Wir zeigen durch Induktion, dass die Kantenmenge $T_i := \{e_{\sigma(1)}, \ldots, e_{\sigma(i)}\}$ mit $0 \leq i \leq k$ aussichtsreich ist.

$i = 0$**:** $T_0 = \emptyset$ und ist somit aussichtsreich.

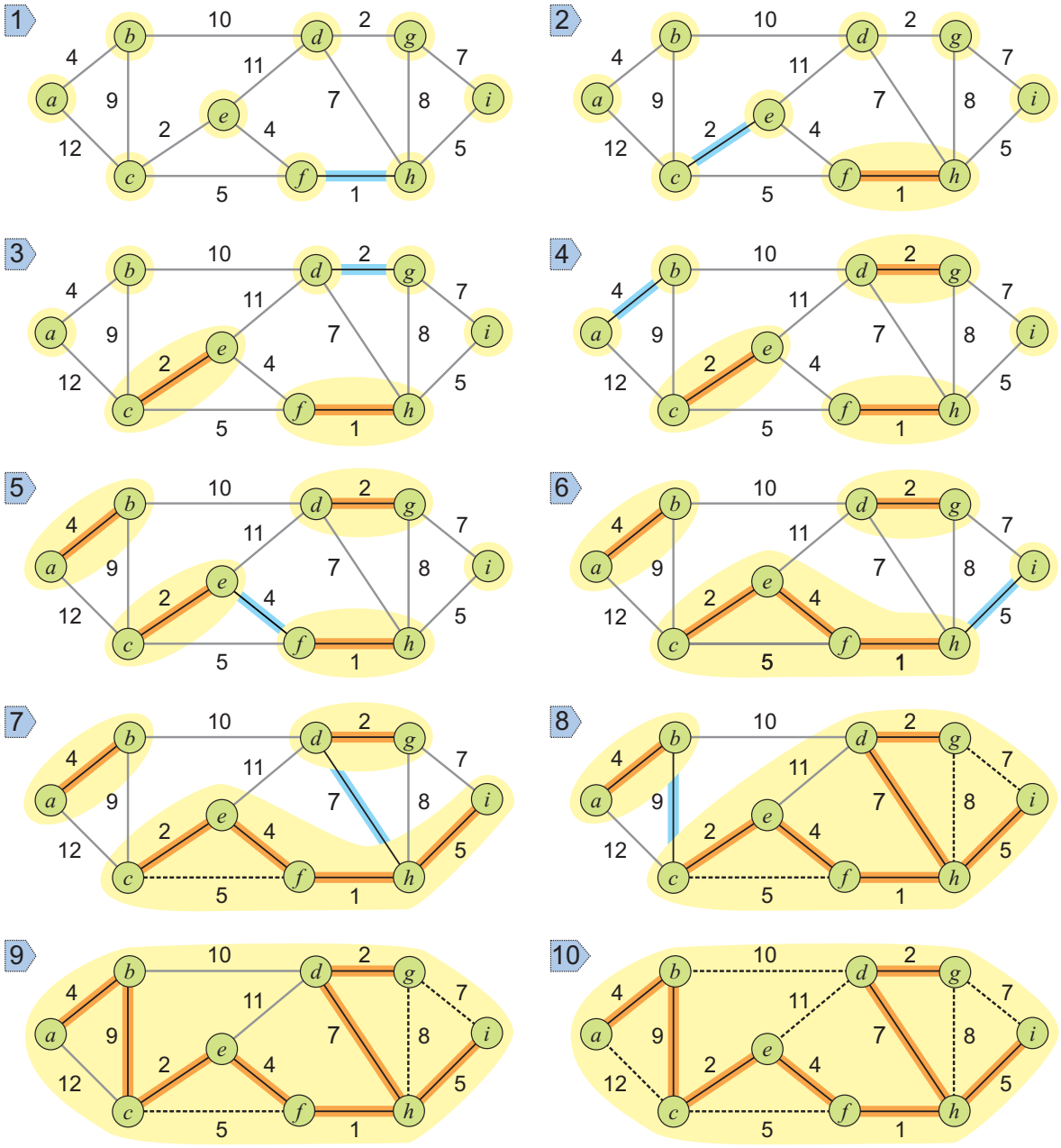


Abbildung 6.14: Ablauf des Algorithmus von Kruskal.

$1 \leq i \leq k$**:** Wir nehmen an, dass T_{i-1} aussichtsreich ist. Sei $e_{\sigma(i)} = (u, v)$ und $C_u = (U, E_u)$ die Komponente des Graphen $G_{i-1} = (V, T_{i-1})$, die u enthält. Da $T_{i-1} \cup \{e_{\sigma(i)}\}$ kreisfrei ist, ist $e_{\sigma(i)}$ eine Kante, die U verlässt. Weiterhin gilt, dass T_{i-1} kreisfrei bleibt, wenn wir irgendeine Kante hinzufügen, die U verlässt. Daraus folgt, dass keine der Kanten $e_1, \ldots, e_{\sigma(i)-1}$ die Menge U verlässt (sonst hätte der Algorithmus sie vorher gewählt!), und daher ist $e_{\sigma(i)}$ eine billigste Kante, die U verlässt. Nach Lemma 6.3 ist somit $T_i = T_{i-1} \cup \{e_{\sigma(i)}\}$ aussichtsreich.

Wir müssen nun noch zeigen, dass $G_T = (V, T)$ zusammenhängend ist. Das folgt aber daraus, dass G zusammenhängend ist und wir nur Kanten *nicht* wählen, wenn sie einen Kreis verursachen würden. □

Möchte man den Kruskal-Algorithmus implementieren, so stellt sich die Frage, wie man effizient das folgende Problem löst:

Teste, ob $T \cup \{e\}$ einen Kreis enthält.

Eine naive Lösung wäre es, jedesmal die Funktion IsAcyclic aus Abschnitt 6.4.2 für den Graphen $G_T := (V, T \cup \{e\})$ aufzurufen. Dies würde für $|E|$ Aufrufe zu einer Gesamtlaufzeit von $O(|V| \cdot |E|)$ führen, da sich in T bis zu $|V| - 1$ Kanten befinden können.

Im Folgenden werden wir sehen, dass wir für den zweiten Teil des Kruskal-Algorithmus, d.h. alles nach dem Sortieren der Kanten, eine fast lineare Laufzeit erzielen können und damit die Gesamtlaufzeit des Algorithmus durch das Sortieren der Kanten bestimmt wird, also $O(|E| \log |E|)$ ist. Dazu zeigen wir, dass das (wiederholte) Testen auf Kreisfreiheit in unserem speziellen Fall auf ein allgemeines Problem für Partitionen von Mengen zurückgeführt werden kann.

Betrachten wir den Graphen $G_T = (V, T)$ zu einem Zeitpunkt des Algorithmus. Wir wissen, dass G_T kreisfrei ist, daher besteht der Graph G_T aus mehreren Komponenten, die alle Bäume sind. Diese Komponenten partitionieren die Knotenmenge. Zu Beginn ist $T = \emptyset$ und jeder Knoten bildet eine eigene Komponente, d.h. jede Menge der Knotenpartition besteht aus einem einzelnen Knoten. Wenn wir testen, ob eine Kante (u, v) mit den Kanten in T einen Kreis schließt, dann ist das gleichbedeutend mit der Frage, ob u und v in der selben Komponente von G_T liegen, also in derselben Menge der Partition. Fügen wir schließlich eine Kante hinzu, dann werden die beiden Komponenten, in denen u bzw. v liegen, zu einer Komponente vereinigt, d.h. die beiden Mengen der Partition werden vereinigt. Wir benötigen also die folgenden Operationen für eine Partition:

- *Erzeuge* eine einelementige Menge der Partition.
- *Finde* die Partitionsmenge, in der sich ein gegebenes Element befindet.
- *Vereinige* zwei Partitionsmengen.

Wir werden im nächsten Abschnitt den Datentyp *UnionFind* einführen, der genau diese Operationen unterstützt. Danach werden wir eine sehr effiziente Realisierung dieses Datentyps kennenlernen und die Komplexität von Kruskal unter Verwendung dieser Datenstruktur untersuchen.

Beispiel 6.7. Wir betrachten nochmals das Beispiel aus Abbildung 6.14. Die Partition der Knoten sieht in den einzelnen Schritten des Algorithmus wie folgt aus:

Komponenten	*nächste Kante*	
$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$	(f, h)	ok
$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f, h\}, \{g\}, \{i\}$	(c, e)	ok
$\{a\}, \{b\}, \{c, e\}, \{d\}, \{f, h\}, \{g\}, \{i\}$	(d, g)	ok
$\{a\}, \{b\}, \{c, e\}, \{d, g\}, \{f, h\}, \{i\}$	(a, b)	ok
$\{a, b\}, \{c, e\}, \{d, g\}, \{f, h\}, \{i\}$	(e, f)	ok
$\{a, b\}, \{c, e, f, h\}, \{d, g\}, \{i\}$	(c, f)	Kreis!
$\{a, b\}, \{c, e, f, h\}, \{d, g\}, \{i\}$	(h, i)	ok
$\{a, b\}, \{c, e, f, h, i\}, \{d, g\}$	(d, h)	ok
$\{a, b\}, \{c, d, e, f, g, h, i\}$	(g, i)	Kreis!
$\{a, b\}, \{c, d, e, f, g, h, i\}$	(g, h)	Kreis!
$\{a, b\}, \{c, d, e, f, g, h, i\}$	(b, c)	ok
$\{a, b, c, d, e, f, g, h, i\}$	(b, d)	Kreis!
$\{a, b, c, d, e, f, g, h, i\}$	(e, d)	Kreis!
$\{a, b, c, d, e, f, g, h, i\}$	(a, c)	Kreis!

6.5.2 Der ADT UnionFind

Der Datentyp *UnionFind* verwaltet eine Partition einer endlichen Menge, deren Elemente aus einem endlichen *Universum* U sind.

Wertebereich: Ein Instanz P des Datentyps *UnionFind* ist eine Familie $P = \{S_1, S_2, \ldots, S_k\}$ paarweise disjunkter Teilmengen einer endlichen Menge U. Für jede Teilmenge S_i gibt es einen eindeutigen Repräsentanten $s_i \in S_i$.

Operationen: Sei $P = \{S_1, \ldots, S_k\}$ mit Repräsentanten s_i die Instanz vor Anwendung der Operation.

- MAKESET$(U\ x)$

  Voraussetzung: $x \notin S_1 \cup \ldots \cup S_k$

Erzeugt eine neue Menge $S_{k+1} := \{x\}$ mit Repräsentant $s_{k+1} := x$ und fügt sie zu P hinzu.

- UNION$(U\ x, U\ y)$

  Voraussetzung: x und y sind jeweils Repräsentant einer der Mengen $S_1, \ldots, S_k$.

  Sei x Repräsentant von S_x und y Repräsentant von S_y. Dann werden S_x und S_y in P durch $S' := S_x \cup S_y$ ersetzt, d.h. die Instanz P' nach der Operation ist

  $$P' := (P \setminus \{S_x, S_y\}) \cup \{S'\}.$$

  Als Repräsentant für S' wird ein beliebiges Element in S' gewählt.

- FIND$(U\ x) : U$

  Voraussetzung: x ist in einer der Mengen $S_1, \ldots, S_k$ enthalten.

  Gibt den Repräsentanten der Menge zurück, die x enthält.

Üblicherweise erzeugt man zunächst mittels MAKESET für jedes Element der Menge U', die man partitionieren möchte, eine einelementige Menge der Partition und benutzt danach die Operationen UNION und FIND. Wir gehen im Folgenden davon aus, dass $U = U'$ ist; insbesondere werden wir Felder benutzen, die mit Elementen aus U indiziert werden.

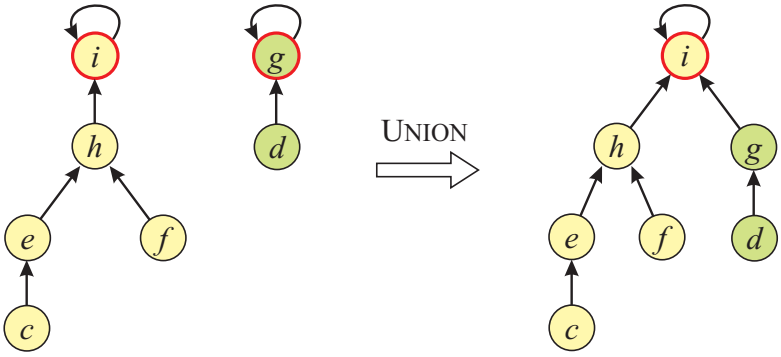


Abbildung 6.15: UNION angewendet auf zwei Mengen $\{c, e, f, h, i\}$ und $\{d, g\}$, die als gewurzelte Bäume dargestellt sind.

Realisierung durch gewurzelte Bäume

In unserer Darstellung repräsentieren wir jede Menge S_i durch einen gewurzelten Baum. Die Knoten des Baumes sind die Elemente der Menge und die Wurzel des Baumes ist der Repräsentant s_i der Menge. In unserer Implementierung hat jeder Knoten v einen Verweis $\pi[v]$ auf seinen Elter, wobei die Wurzel auf sich selbst zeigt. Die Operationen MAKESET, UNION und FIND sind dann sehr einfach zu implementieren; siehe Listing 6.6.

```
1:  ▷ Interne Repräsentation
2:  var U π[U]                                   ▷ Elter eines Knotens

3:  procedure MAKESET(U x)
4:      π[x] := x
5:  end procedure

6:  procedure UNION(U x, U y)
7:      π[x] := y
8:  end procedure

9:  function FIND(U x) : U
10:     while π[x] ≠ x do
11:         x := π[x]
12:     end while
13:     return x
14: end function
```

Listing 6.6: Implementierung von *UnionFind*

Beispiel 6.8. Abbildung 6.15 veranschaulicht die Darstellung der Mengen und die Funktionsweise von UNION an einem Beispiel. Auf der linken Seite sind die Bäume von zwei Mengen $\{c, e, f, h, i\}$ und $\{d, g\}$ der Partition dargestellt (die Wurzel ist fett rot umrandet). Rechts ist das Ergebnis von Union zu sehen: Wir hängen die Wurzel des zweiten Baumes einfach an die Wurzel des ersten Baumes.

Hinweis 6.1. Die Bäume in unserer Darstellung der Mengen der Partition kann man leicht mit den Bäumen verwechseln, die während des Kruskal-Algorithmus die Komponenten des Graphen $G_T = (V, T)$ sind. Verwenden wir diese Union-Find Datenstruktur für den Algorithmus von Kruskal, so sind zwar die Elemente in den Mengen (und damit in den gewurzelten Bäumen) genau die Knoten der Komponenten, jedoch müssen die Kanten $(v, \pi[v])$ *nicht* den Kanten in T entsprechen! Die Baumdarstellung der Mengen ist eine rein *interne* Darstellung der Datenstruktur.

Analyse der Laufzeit:

- MAKESET(x): $O(1)$
- UNION(x, y): $O(1)$
- FIND(x): $O(h(T_x))$, wobei T_x der Baum ist, der x enthält.

  Die Find Operation verfolgt die Elter-Verweise bis zur Wurzel, d.h. die Laufzeit von FIND(x) ist durch die Höhe des Baumes beschränkt, der x enthält.

Die Laufzeit von FIND ist also linear in der Höhe des Baumes. Allerdings verhindert unsere Konstruktion der Bäume (UNION Operation) nicht, dass die entstehenden Bäume degenerieren können. Im ungünstigsten Fall entartet ein Baum T zu einer linearen Liste und hat Höhe $s(T) - 1$, wobei wir mit $s(T)$ die Anzahl der Knoten im Baum T bezeichnen.

Wir gehen im Folgenden davon aus, dass wir zunächst n MAKESET Operationen ausführen, und interessieren uns für die Gesamtlaufzeit von $n - 1$ UNION und m FIND Operationen, die anschließend durchgeführt werden. Das entspricht der Vorgehensweise, zunächst eine Partition aus n einelementigen Mengen aufzubauen, und danach so viele UNION Operation auszuführen, bis die Partition nur noch eine einzige n-elementige Menge enthält. Bisher können wir die Höhe eines Baumes nur durch $O(n)$ nach oben beschränken, so dass wir die Worst-Case Laufzeit für $n - 1$ UNION und m FIND Operationen lediglich durch

$$O(n + mn)$$

beschränken können. Wir werden aber sehen, dass diese Laufzeit durch zwei sehr einfache Heuristiken deutlich verbessert werden kann.

Verbesserung 1: gewichtete Vereinigungsregel

Die gewichtete Vereinigungsregel verhindert, dass die entstehenden Bäume entarten. Die Idee dabei ist:

> *Wenn wir zwei Bäume vereinigen, dann hängen wir den Baum mit der kleineren Höhe an den Baum mit der größeren Höhe.*

Dazu verwalten wir für jede Wurzel x eines Baums T_x einen Rang, der die Höhe des Baumes T_x ist, d.h. *rank*$(x) := h(T_x)$. Vereinigen wir zwei Bäume T_x und T_y und gilt $h(T_x) > h(T_y)$, dann ist die Höhe des entstehenden Baumes $h(T_x)$, d.h. die Höhe nimmt nicht zu; nur wenn beide Bäume gleiche Höhe haben, also $h(T_x) = h(T_y)$, dann ist der entstehende Baum um eins höher, hat also Höhe $h(T_y) + 1$.

Allgemein können wir sogar sagen, dass die Höhe der entstehenden Bäume logarithmisch in der Anzahl der Elemente im Baum beschränkt ist:

Lemma 6.4. UNION *mit gewichteter Vereinigungsregel liefert immer Bäume, deren Höhe* $h(T)$ *durch* $\log s(T)$ *nach oben beschränkt ist.*

Beweis. Wir müssen zeigen, dass jeder erzeugte Baum mit Höhe d mindestens 2^d Knoten besitzt. Dann folgt sofort die Behauptung.

Wir zeigen dies durch Induktion nach d.

$d = 0$**:** Ein Baum mit Höhe 0 hat 1 Knoten, also $1 \geq 2^0$ gilt.

$d \geq 1$**:** Wir nehmen an, dass jeder erzeugt Baum mit Höhe $d' < d$ mindestens $2^{d'}$ Knoten hat.

Sei T ein erzeugter Baum mit Höhe d, der unter allen erzeugten Bäumen mit Höhe d die minimale Knotenanzahl hat. T entstand dadurch, dass bei UNION der Baum T_1 an die Wurzel des Baums T_2 gehängt wurde.

- Nach der gewichteten Vereinigungsregel folgt $h(T_1) \leq h(T_2)$. Außerdem muss $h(T_2) \leq d - 1$ sein, denn sonst hätte T_2 Höhe d aber weniger Knoten als T, was nach Wahl von T nicht möglich ist.
- Nach Konstruktion ist $h(T_1) \leq d - 1$, es muss aber sogar $h(T_1) = d - 1$ gelten, denn sonst hätte T_2 Höhe d.
- Es folgt $h(T_1) = h(T_2) = d-1$. Nach Induktionsvoraussetzung ist somit $s(T_1) \geq 2^{d-1}$ und $s(T_2) \geq 2^{d-1}$, und damit

$$s(T) \geq 2^{d-1} + 2^{d-1} = 2^d.$$

□

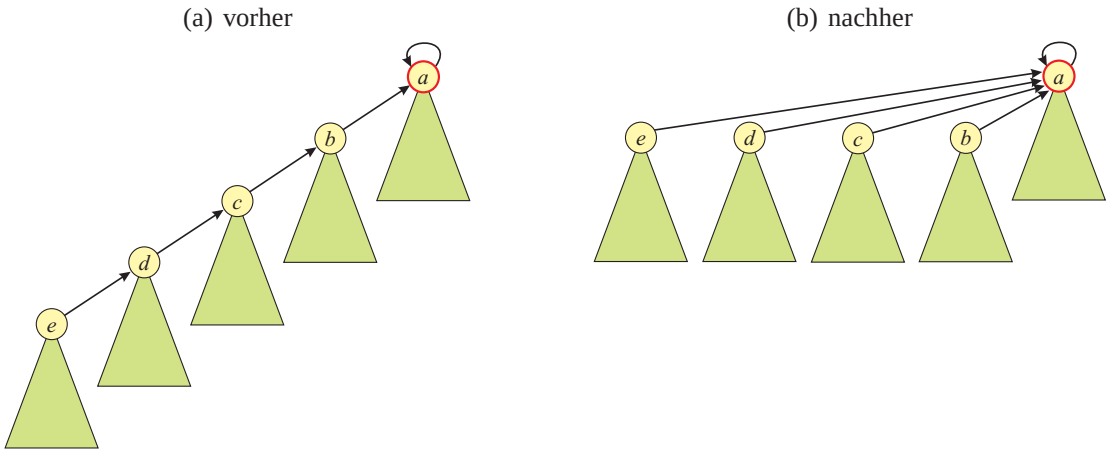


Abbildung 6.16: Effekt der Pfadkomprimierung nach einem FIND Aufruf; die Dreiecke unter den Knoten repräsentieren Unterbäume.

Aus diesem Lemma folgt sofort, dass die Worst-Case Laufzeit einer FIND Operation durch $O(\log n)$ beschränkt ist, und daher benötigen $n-1$ UNION und m FIND Operationen im Worst-Case eine Laufzeit von

$$O(n + m \log n).$$

Anmerkung 6.8. Statt als Rang einer Wurzel die Höhe des Baumes zu benutzen, kann man alternativ auch die Anzahl der Knoten im Baum verwenden, d.h. $\mathit{rank}(x) := s(T_x)$, wenn x Wurzel des Baumes T_x ist. Beide Varianten liefern Bäume T, deren Höhe durch $\log s(T)$ beschränkt ist.

Verbesserung 2: Pfadkomprimierung

Die zweite Heuristik merkt sich bei einer FIND Operation gewonnene Informationen, um spätere FIND Operationen schneller zu machen. Die Idee ist dabei wie folgt:

> *Bei einem Aufruf von* FIND(x) *werden alle Knoten auf dem Pfad* p_x *von* x *bis zur Wurzel zu direkten Kindern der Wurzel gemacht.*

Abbildung 6.16 zeigt den Effekt der Pfadkomprimierung nach einem FIND Aufruf. Da wir bei FIND(x) den Pfad p_x sowieso einmal durchlaufen müssen, ist die modifizierte FIND Operation nur um einen konstanten Faktor langsamer. Wird jedoch später FIND für einen Knoten auf p_x aufgerufen, dann hat dieser Aufruf nur konstante Laufzeit!

Wir verzichten an dieser Stelle auf eine genau Laufzeitanalyse; man kann jedoch mit Hilfe von amortisierter Analyse das folgende Theorem zeigen (siehe z.B. Cormen et al.):

Theorem 6.7. *Wird die gewichtete Vereinigungsregel und Pfadkomprimierung benutzt, so kosten* $n-1$ UNION *und* m FIND *Operationen im Worst-Case*

$$O((n+m)\cdot\alpha(n)).$$

Dabei ist $\alpha(n)$ eine *extrem langsam* wachsende Funktion, die wir formal weiter unten definieren.

Hinweis 6.2. Man beachte, dass das Theorem *nicht* aussagt, dass jede Operation für sich im Worst-Case nur $O(\alpha(n))$ Zeit benötigt. Es kann durchaus FIND Operationen geben, die Zeit $O(\log n)$ benötigen. Da die $n+m-1$ Operationen zusammen im Worst-Case jedoch immer nur $O((n+m)\cdot\alpha(n))$ Zeit kosten, sagt man, dass die *amortisierten* Kosten einer Operation

$$O\left(\frac{n+m}{n+m-1}\cdot\alpha(n)\right)=O(\alpha(n))$$

sind. Die Idee dabei ist, dass man die Mehrkosten einer teuren Operation auf viele billige Operationen verteilen kann, so dass die Kosten einer Operation im Schnitt wieder günstig sind.

Um die Funktion α zu definieren, benötigen wir zunächst die Funktion A_k. Für $k\geq 0$ und $j\geq 1$ definieren wir:

$$A_k(j):=\begin{cases} j+1 & \text{if } k=0\\ A_{k-1}^{(j+1)}(j) & \text{if } k\geq 1\end{cases}$$

Dabei bedeutet die Notation $f^{(j)}$, dass die Funktion f j-mal angewendet wird, d.h.

$$f^{(j)}(x):=\begin{cases} f(x) & \text{falls } j=1\\ f^{(j-1)}(x) & \text{falls } j\geq 2.\end{cases}$$

Man kann zeigen, dass $A_1(j)=2j+1$ und $A_2(j)=2^{j+1}(j+1)-1$ gilt. Wir beobachten, dass die Funktion A *extrem schnell* wächst:

$$\begin{aligned} A_0(1) &= 1+1=2\\ A_1(1) &= 2\cdot 1+1=3\\ A_2(1) &= 2^{1+1}(1+1)-1=7\\ A_3(1) &= A_2^{(2)}(1)=A_2(7)=2047\\ A_4(1) &= A_3^{(2)}(1)=A_3(2047)\\ &\gg A_2(2047)>16^{512}\gg 10^{80}\end{aligned}$$

Man schätzt, dass 10^{80} die Anzahl der Atome im Universum ist! Die Inverse der Funktion A_k ist definiert als

$$\alpha(n):=\min\{k\mid A_k(1)\geq n\}.$$

Daraus ergibt sich, dass die Funktion α *extrem langsam* wächst:

$$\alpha(n) = \begin{cases} 0 & \text{für } 0 \leq n \leq 2 \\ 1 & \text{für } n = 3 \\ 2 & \text{für } 4 \leq n \leq 7 \\ 3 & \text{für } 8 \leq n \leq 2047 \\ 4 & \text{für } 2048 \leq n \leq A_4(1) \end{cases}$$

Wir können also annehmen, dass $\alpha(n) \leq 4$ ist für alle praktisch relevanten Werte von n. Daher sind die amortisierten Kosten einer UNION bzw. FIND Operation praktisch konstant!

Anmerkung 6.9. Die Funktion A ist die in Cormen et al. beschriebene Variante der *Ackermann* Funktion. Unter dem Begriff Ackermann Funktion findet man verschiedene, leicht voneinander abweichende Definitionen. Jedoch haben alle Varianten gemeinsam, dass sie extrem schnell wachsen.

Listing 6.7 zeigt die Implementierung von *UnionFind* mit gewichteter Vereinigungsregel und Pfadkomprimierung. Auch bei Verwendung dieser Heuristiken, die die Laufzeit deutlich verbessern, bleibt die Implementierung kurz und elegant.

6.5.3 Realisierung von Kruskal mit UnionFind

Wir kommen nun zurück zum Algorithmus von Kruskal zur Berechnung eines MST. Listing 6.8 zeigt die Implementierung des Algorithmus mit Hilfe des Datentyps *UnionFind*. Man beachte, dass nicht unbedingt jede Kante von E betrachtet werden muss, sondern abgebrochen wird, wenn ein aufspannender Baum konstruiert ist. Allerdings ist es möglich, dass auch die teuerste Kante im MST benötigt wird.

Sei $n := |V|$ die Anzahl der Knoten und $m := |E|$ die Anzahl der Kanten von G. Dann ergibt sich die Worst-Case Laufzeit unter Verwendung der *UnionFind* Datenstruktur aus dem vorangehenden Abschnitt mit gewichteter Vereinigungsregel und Pfadkomprimierung wie folgt:

- Die Initialisierung inklusive der n MAKESET Aufrufe benötigt Zeit $O(n)$.
- Das Sortieren der Kanten benötigt Zeit $O(m \log m)$.
- Die Hauptschleife muss im schlechtesten Fall für alle Kanten durchlaufen werden. Dabei werden $2m$ FIND und $n - 1$ UNION Operationen ausgeführt. Nach Satz 6.7 benötigt das Zeit $O((n + m) \cdot \alpha(n)) = O(m \cdot \alpha(n))$, da $m \geq n - 1$.
- Die Gesamtlaufzeit wird also durch das Sortieren dominiert und ist $O(m \log m)$.

Theorem 6.8. *Der Algorithmus von Kruskal berechnet einen minimalen Spannbaum eines ungerichteten, zusammenhängenden Graphen $G = (V, E)$ in Zeit $O(|E| \log |E|)$.*

```
 1: ▷ Interne Repräsentation
 2: var U π[U]                                          ▷ Elter eines Knotens
 3: var int rank[U]                                     ▷ Rang eines Knotens

 4: procedure MakeSet(U x)
 5:     π[x] := x
 6:     rank[x] := 0
 7: end procedure

 8: procedure Union(U x, U y)
 9:     if rank[x] > rank[y] then                       ▷ gewichtet Vereinigungsregel
10:         π[y] := x
11:     else
12:         π[x] := y
13:         if rank[x] = rank[y] then rank[y] := rank[y] + 1
14:     end if
15: end procedure

16: function Find(U x) : U
17:     if x ≠ π[x] then
18:         π[x] := Find(π[x])                          ▷ Pfadkomprimierung
19:     end if
20:     return π[x]
21: end function
```

Listing 6.7: *UnionFind* mit gewichteter Vereinigungsregel und Pfadkomprimierung.

Eingabe: ungerichteter, zshgd. Graph $G = (V, E)$ mit Gewichtsfunktion $w : E \to \mathbb{R}$
Ausgabe: Kantenmenge T eines MST von G

```
1:  var UnionFind P
2:  for all v ∈ V do
3:      P.MakeSet(v)
4:  end for
5:  Sortiere die Kanten von G, so dass w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m) gilt.
6:  T := ∅;   i := 1
7:  while |T| < |V| − 1 do
8:      let e_i = (u, v)
9:      x := P.FIND(u)
10:     y := P.FIND(v)
11:     if x ≠ y then
12:         T := T ∪ {e_i}
13:         P.UNION(x, y)
14:     end if
15:     i := i + 1;
16: end while
```

Listing 6.8: Algorithmus von Kruskal

6.5.4 Der Algorithmus von Prim

Der Algorithmus von Prim arbeitet ebenfalls nach dem Greedy-Prinzip. Im Gegensatz zu Kruskal verwaltet der Algorithmus von Prim immer einen einzigen Baum und fügt in jedem Schritt einen weiteren Knoten hinzu, indem ein Knoten gewählt wird, der durch eine leichteste Kante mit dem bisher konstruierten Baum verbunden ist.

Wähle einen beliebigen Startknoten $s \in V$
$S := \{s\}; \quad T := \emptyset$
while $S \neq V$ **do**
    Sei $e = (u, v)$ eine Kante mit minimalem Gewicht, die S verlässt
    $T := T \cup \{e\}; \quad S := S \cup \{v\}$
end while

Beispiel 6.9. Abbildung 6.17 zeigt den Ablauf des Algorithmus von Prim an dem Beispielgraphen aus Abbildung 6.12. Die Knoten der Menge S sind rot dargestellt, der Startknoten s ist also der Knoten a. Der von S induzierte Untergraph (die Knoten in S und alle Kanten, die beide Endpunkte in S haben) ist gelb hinterlegt. Die Kanten in T sind orange hinterlegt.

In jedem Schritt wird eine Kante gewählt (und danach zu T hinzugefügt), die S verlässt und minimales Gewicht hat. Die Kanten, die S verlassen, sind jeweils hellblau hinterlegt und die Kante, die als nächstes gewählt wird, ist rot markiert.

Theorem 6.9. *Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$. Dann berechnet der Algorithmus von Prim einen MST von G.*

Beweis. Seien $e_1, \ldots, e_k$ die Kanten, die Prim in dieser Reihenfolge zu T hinzufügt. Wir zeigen durch Induktion, dass die Kantenmenge $T_i := \{e_1, \ldots, e_i\}$ mit $0 \leq i \leq k$ aussichtsreich ist.

$i = 0$**:** $T_0 = \emptyset$ und ist somit aussichtsreich.

$1 \leq i \leq k$**:** Wir nehmen an, dass T_{i-1} aussichtsreich ist. Sei S die Menge wie im Algorithmus vor Hinzunahme der Kante e_i. Dann ist e_i eine leichteste Kante, die S verlässt, und keine Kante aus T_{i-1} verlässt S, da S gerade die Knoten sind, die durch die Kanten in T_{i-1} verbunden werden. Nach Lemma 6.3 ist somit $T_i = T_{i-1} \cup \{e_i\}$ aussichtsreich.

Wir haben also am Ende des Algorithmus eine aussichtsreiche Kantenmenge T mit $|V| - 1$ Kanten, also einen MST. □

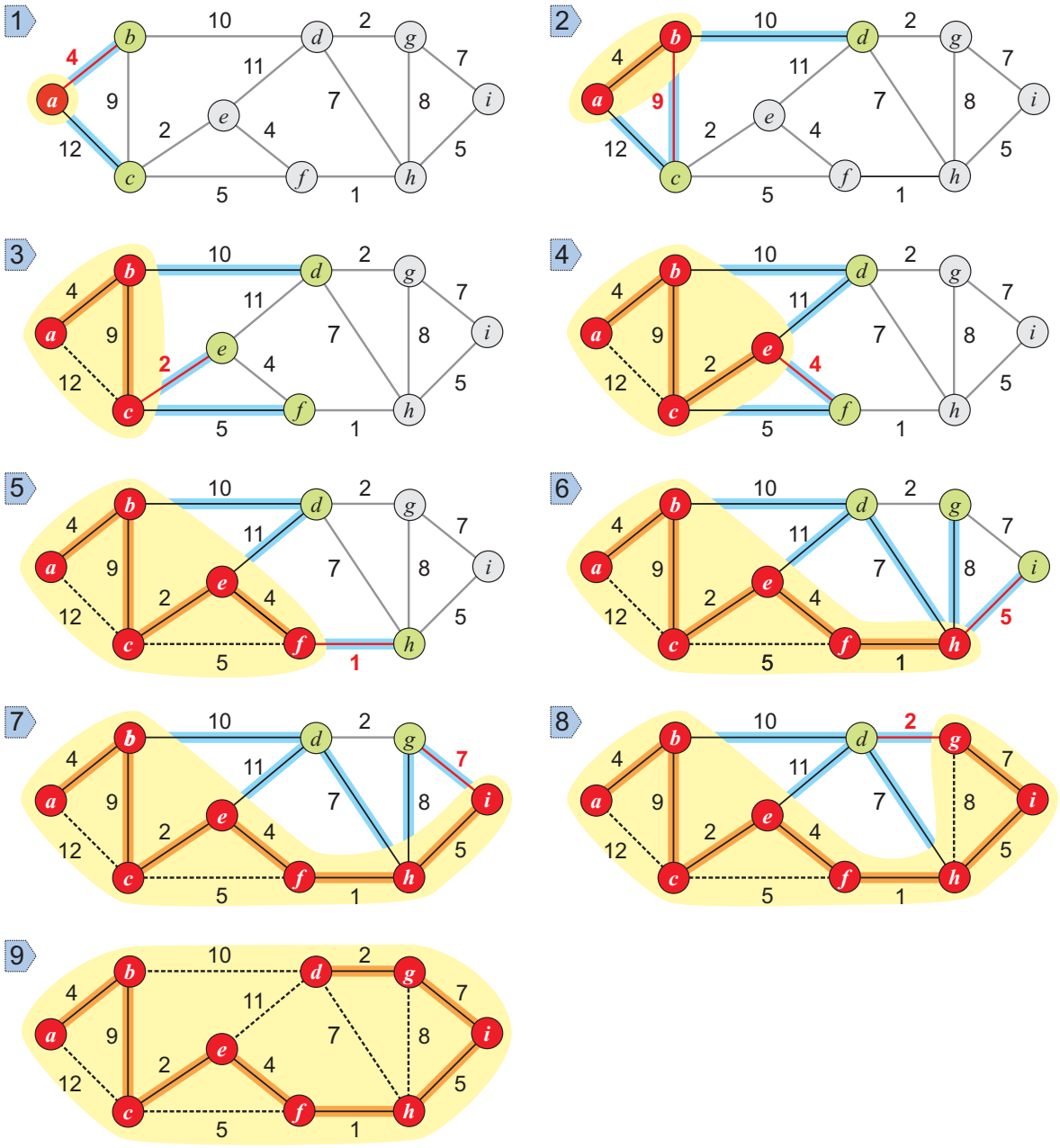


Abbildung 6.17: Ablauf des Algorithmus von Prim.

Das Ermitteln einer leichtesten Kante, die die Menge S verlässt, kann effizient gelöst werden, indem alle Knoten in $V \setminus S$ in einer Priority-Queue verwaltet werden. Die Sortierung erfolgt dabei nach dem Gewicht der jeweils leichtesten Kante, die einen Knoten in $V \setminus S$ mit einem Knoten in S verbindet.

Die Priority-Queue muss nach jedem Hinzufügen einer Kante aktualisiert werden. Dazu müssen lediglich die zum Knoten u, der gerade zu S hinzugefügt wurde, inzidenten Kanten überprüft werden. Führt eine solche Kante zu einem Knoten v in $V \setminus S$ und ist günstiger als die bisher günstigste Verbindung von v zu einem Knoten in S, dann wird die Priorität von v entsprechend reduziert. Die Implementierung des Algorithmus von Prim unter Verwendung einer Priority-Queue ist in Listing 6.9 dargestellt.

Eingabe: ungerichteter, zshgd. Graph $G = (V, E)$ mit Gewichtsfunktion $w : E \to \mathbb{R}$
Ausgabe: ein MST von G repräsentiert durch das Feld π

```
 1: var π[V]                                          ▷ Vorgänger eines Knotens im MST
 2: var PriorityQueue Q
 3: var pos[V]                                        ▷ Position eines Knotens in der PriorityQueue

 4: for each u ∈ V \ {s} do
 5:     pos[u] := Q.INSERT(∞, u)
 6: end for
 7: pos[s] := Q.INSERT(0, s)
 8: π[s] := nil
 9: while not Q.ISEMPTY() do
10:     (p, u) := Q.EXTRACTMIN()
11:     pos[u] := nil
12:     for all e = (u, v) ∈ E(u) do
13:         p_v := pos[v]
14:         if p_v ≠ nil and w(e) < Q.PRIORITY(p_v) then
15:             Q.DECREASEPRIORITY(p_v, w(e))
16:             π[v] := u
17:         end if
18:     end for
19: end while
```

Listing 6.9: Algorithmus von Prim

Analyse der Laufzeit. Wir analysieren jetzt die Laufzeit des Algorithmus von Prim unter Verwendung eines Min-Heaps als Priority-Queue.

- Die Initialisierung inklusive Aufbau des Min-Heaps (Zeilen 1–8) benötigt Zeit $O(|V|)$.

- Die **while**-Schleife wird $|V|$-mal durchlaufen, d.h. die EXTRACTMIN Aufrufe benötigen insgesamt Zeit $O(|V| \log |V|)$.

- Die **for all**-Schleife darin wird für jede Kante genau zweimal durchlaufen. DECREASEPRIORITY wird höchstens für jede Kante einmal aufgerufen, benötigt also insgesamt Zeit $O(|E| \log |V|)$.

- Die Gesamtlaufzeit ist somit $O(|E| \log |V|)$, da $|E| \geq |V| - 1$.

Theorem 6.10. *Der Algorithmus von Prim berechnet einen minimalen Spannbaum eines ungerichteten, zusammenhängenden Graphen $G = (V, E)$ in Zeit $O(|E| \log |V|)$.*

6.6 Kürzeste Wege

Betrachten wir die Knoten eines Graphen als Orte und die Kanten als mögliche Verbindungen zwischen den Orten, für die wir die notwendigen Fahrzeiten kennen, so können wir die Frage stellen, wie wir am schnellsten von Ort A zu Ort B kommen. In diesem Abschnitt betrachten wir genau solche *Kürzeste Wege Probleme*.

Wir gehen im folgenden davon aus, dass wir einen gerichteten Graphen $G = (V, A)$ mit einer nicht-negativen Gewichtsfunktion $w : A \to \mathbb{R}_0^+$ für die Kanten gegeben haben. Die Gewichte können wir zum Beispiel als Längen oder Fahrtzeiten interpretieren. Die Länge eines Weges $p = v_0, e_1, \ldots, e_k, v_k$ ist einfach die Summe der Kantengewichte, d.h.

$$w(p) := \sum_{i=1,\ldots,k} w(e_i).$$

Anmerkung 6.10. Wir werden in diesem Abschnitt davon ausgehen, dass der Graph *keine* Mehrfachkanten oder Schleifen besitzt. Gibt es zwischen zwei Knoten u und v mehrere Kanten (u, v), so ist sicher nur die Kanten mit dem geringsten Gewicht für einen kürzesten Weg relevant. Wir könnten also gefahrlos Mehrfachkanten zwischen zwei Knoten durch die jeweils leichteste Kante ersetzen.

6.6.1 Single Source Shortest Path

Beim *Single Source Shortest Path Problem* interessiert man sich für die kürzesten Wege von einem gegebenen Startknoten $s \in V$ zu allen anderen Knoten des Graphen. Das Problem ist analog zum USSSP, das wir bereits im Abschnitt 6.3.1 untersucht haben, nur das wir jetzt einen gerichteten Graphen betrachten und Kantengewichte beachten müssen.

Single-Source Shortest Paths (SSSP)	
Gegeben:	gerichteter Graph $G = (V, A)$ Gewichtsfunktion $w : A \to \mathbb{R}_0^+$ Startknoten $s \in V$
Gesucht:	ein kürzester Weg von s nach v für jeden Knoten $v \in V$

Die folgende Eigenschaft von kürzesten Wegen erlaubt es uns, für jeden Knoten einen kürzesten Weg vom Startknoten s aus wie im Falle von BFS in Form eines *Kürzeste-Wege Baums* (engl. *shortest-paths tree, SPT*) darzustellen.[2]

Beobachtung 6.2. *Ist $s = v_0, v_1, \ldots, v_k$ ein kürzester Weg von s nach v_k, dann ist auch jeder Weg $v_0, \ldots, v_i$ für $0 \leq i < k$ ein kürzester Weg von s nach v_i.*

Das ist unmittelbar klar, denn gäbe es einen kürzeren Weg von s nach v_i, dann könnten wir auch den Weg von s nach v_k verkürzen. Wir können also alle kürzesten Wege, die von s ausgehen, mit Hilfe eines Knotenfeldes π darstellen, so dass $\pi[v]$ der Vorgänger von v auf einem kürzesten Weg von s nach v ist.

Das SSSP kann effizient mit dem Algorithmus von Dijkstra gelöst werden. Die Idee dabei ist ganz ähnlich wie beim Algorithmus von Prim zur Bestimmung eines MST. Wir beginnen mit s als Wurzel des SPT und fügen nacheinander die Kanten des SPT hinzu, wobei wir jeweils eine Kante auswählen, die einen kürzesten Weg von s zu einem Knoten, der noch nicht im SPT ist, ergibt.

Für eine Teilmenge S der Knoten bezeichnen wir mit $A(S)$ die Menge der Kanten, die von einem Knoten aus S zu einem Knoten nicht aus S führen, also

$$A(S) := \{(x, y) \in A \mid x \in S, y \notin S\}.$$

Dann können wir die Idee des Algorithmus von Dijkstra wie folgt skizzieren:

$S := \{s\}$ ▷ S ist Menge der Knoten im SPT
$d[s] := 0$; $\pi[s] :=$ *nil* ▷ s ist Wurzel des SPT
while $A(S) \neq \emptyset$ **do**
  Sei $e = (u, v) \in A(S)$ eine Kante für die $d[u] + w(e)$ minimal ist
  $S := S \cup \{v\}$
  $d[v] := d[u] + w(e)$; $\pi[v] := u$
end while

Neben dem SPT, der durch das Feld π repräsentiert wird, berechnet der Algorithmus in $d[v]$ den *Abstand* jedes Knotens v zu s, d.h. die Länge des kürzesten Weges von s nach v.

[2] Genaugenommen sind die Knoten des Kürzeste-Wege Baums die Knoten, die von s aus erreichbar sind.

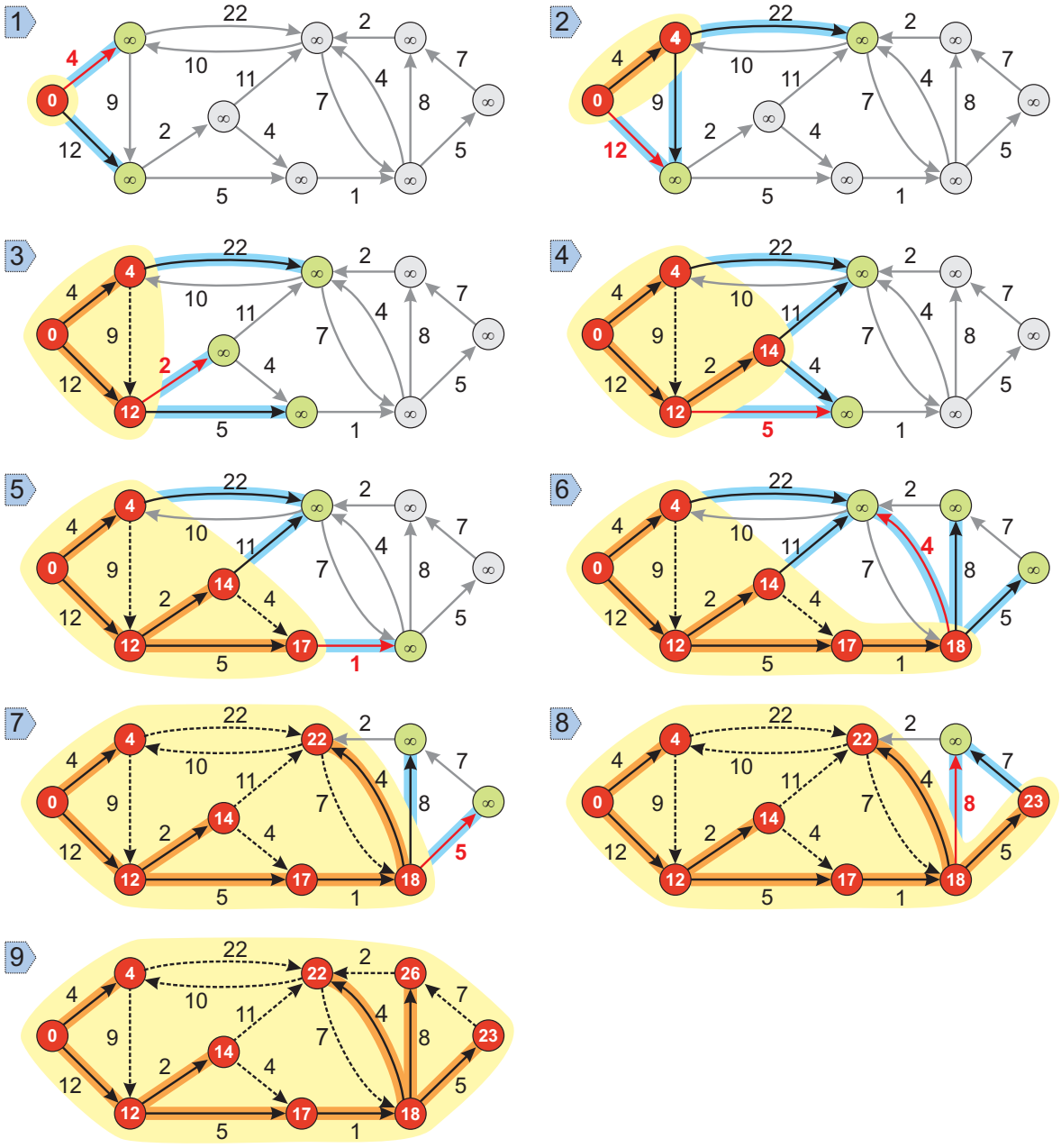


Abbildung 6.18: Ablauf des Algorithmus von Dijkstra.

Beispiel 6.10. Abbildung 6.18 zeigt den Ablauf des Algorithmus von Dijkstra an einem Beispiel. Die Knoten der Menge S sind rot dargestellt und jeder Knoten v ist mit seinem Abstand $d[v]$ vom Startknoten beschriftet; noch nicht erreichte Knoten sind mit ∞ beschriftet. Der Startknoten s ist also mit 0 beschriftet. Der von S induzierte Untergraph ist gelb hinterlegt und die Kanten des Kürzeste-Wege Baumes sind orange hinterlegt.

Die Kanten der Menge $A(S)$ sind blau hinterlegt und die Kante $e = (u, v)$, die der Algorithmus als nächstes wählt, ist rot markiert. Man beachte, dass der Algorithmus *nicht* immer die Kante aus $A(S)$ mit dem geringsten Gewicht wählt! Analog zu BFS werden die Knoten nach aufsteigendem Abstand zu s besucht.

Wir werden später sehen, wir wir die Auswahl der Kante (u, v) effizient realisieren können. Zuvor zeigen wir die Korrektheit des Algorithmus.

Lemma 6.5. *Der Algorithmus von Dijkstra berechnet für jeden Knoten $v \in V$ den kürzesten Weg von s nach v.*

Beweis. Seien $e_1, \ldots, e_n$ die Kanten, die der Algorithmus wählt, und sei $e_i = (u_i, v_i)$ für $i = 1, \ldots, n$ und $v_0 := s$.

Wir zeigen durch Induktion nach i, dass $v_0, \ldots, v_i$ ein kürzester Weg von v_0 nach v_i in G ist.

$i = 0$**:** Der Weg von v_0 nach v_0 ohne Benutzung einer Kante ist offensichtlich ein kürzester Weg.

$1 \leq i \leq k$**:** Wir nehmen an, dass $v_0, \ldots, v_j$ ein kürzester Weg von v_0 nach v_j ist für $j < i$. Wir müssen zeigen, dass der vom Algorithmus gewählte Weg ein kürzester Weg ist. Dieser Weg hat Kosten $d[u_i] + w(e_i)$

Jeder andere Weg von v_0 nach v_i beginnt mit einem Wegstück, das nur Knoten aus $V_{i-1} := \{v_0, \ldots, v_{i-1}\}$ enthält, gefolgt von einer Kante $(x, y) \in A(V_{i-1})$. Alleine das Wegstück von v_0 nach y hat aber bereits mindestens Kosten $d[u_i] + w(e_i)$, da die Kante e_i gerade so gewählt wurde. Folglich ist $v_0, \ldots, v_i$ ein kürzester Weg.

Wir müssen noch begründen, warum alle Knoten, die nach dem Ablauf des Algorithmus nicht im SPT sind, auch von s aus nicht erreichbar sind. Das ist aber klar, da der Algorithmus solange eine Kante zu einem Knoten in $V \setminus S$ wählt, wie eine solche existiert. □

Wir überlegen uns nun, wir wir die Auswahl der nächsten Kante $e = (u, v)$, die zum SPT hinzugefügt wird, effizient realisieren können. Wir bemerken zunächst, dass es uns ausreicht, den Knoten v zu kennen. Wenn wir für jeden Knoten $v \notin S$ den Wert

$$\delta(v) := \min\{d[u] + w(e) \mid e = (u, v) \in A(S)\} \cup \{\infty\}$$

speichern, dann ist der nächste Knoten, den wir betrachten müssen, der Knoten mit minimalem δ-Wert. Für einen Knoten $v \notin S$ ist $\delta(v)$ die Länge des kürzesten Weges von s nach v, der nur über Knoten aus S führt. Ähnlich wie beim Algorithmus von Prim speichern wir die δ-Werte in einer Priority-Queue Q (siehe Listing 6.10). Die Initialisierung von Q ist einfach, denn zu Beginn ist $\delta(s) = 0$ und $\delta(v) = \infty$ für alle $v \neq s$.

Wie können wir nun die δ-Werte effizient updaten, wenn wir den nächsten Knoten u zum SPT hinzufügen? Dazu müssen wir lediglich alle von u ausgehenden Kanten betrachten, denn nur für die Knoten v, für die es eine Kante (u, v) gibt, kann sich eventuell der Wert $\delta(v)$ verringern. Wenn wir feststellen, dass wir einen Wert $\delta(v)$ (also eine Priorität) verringern müssen ($\rightarrow$ DECREASEPRIORITY), dann können wir uns in $\pi[v]$ den Knoten u merken, der auf dem derzeit kürzesten Weg von s nach v der Vorgänger von v ist. Beim Hinzufügen des nächsten Knotens u zum SPT ($\rightarrow$ EXTRACTMIN) ist dann ist der Vorgänger $\pi[u]$ bereits korrekt gesetzt.

Eingabe: gerichteter Graph $G = (V, A)$ mit Gewichtsfunktion $w : A \rightarrow \mathbb{R}_0^+$ und ein Startknoten $s \in V$
Ausgabe: Kürzeste-Wege Baum im Feld π

```
1:  var π[V]                          ▷ Vorgänger eines Knotens im Kürzeste-Wege Baum
2:  var PriorityQueue Q
3:  var pos[V]                        ▷ Position eines Knotens in der PriorityQueue

4:  for each u ∈ V \ {s} do
5:      pos[u] := Q.INSERT(∞, u)
6:      π[u] := nil
7:  end for
8:  pos[s] := Q.INSERT(0, s);
9:  π[s] := nil
10: while not Q.ISEMPTY() do
11:     (d_u, u) := Q.EXTRACTMIN()            ▷ d_u ist Abstand von s zu u
12:     pos[u] := nil
13:     for all e = (u, v) ∈ A⁻(u) do
14:         if d_u + w(e) < Q.PRIORITY(pos[v]) then
15:             Q.DECREASEPRIORITY(pos[v], d_u + w(e))
16:             π[v] := u
17:         end if
18:     end for
19: end while
```

Listing 6.10: Algorithmus von Dijkstra

Analyse der Laufzeit. Wir analysieren die Laufzeit des Algorithmus von Dijkstra unter Verwendung eines Binary-Heaps als Priority-Queue.

- Die Initialisierung inklusive Aufbau des Min-Heaps (Zeilen 1–9) benötigt Zeit $O(|V|)$. Man beachte, dass sich zu jedem Zeitpunkt maximal $|V|$ Elemente in der Priority-Queue befinden.
- Die **while**-Schleife wird höchstens $|V|$-mal durchlaufen, d.h. die EXTRACTMIN Aufrufe benötigen insgesamt Zeit $O(|V| \log |V|)$.
- DECREASEPRIORITY wird höchstens für jede Kante einmal aufgerufen, benötigt also insgesamt Zeit $O(|A| \log |V|)$.
- Die Gesamtlaufzeit ist somit $O((|V| + |A|) \log |V|)$.

Theorem 6.11. *Der Algorithmus von Dijkstra löst das SSSP für einen gerichteten Graphen $G = (V, A)$ mit Gewichtsfunktion $w : A \to \mathbb{R}_0^+$ in Zeit $O((|V| + |A|) \log |V|)$.*

Hinweis 6.3. Man beachte, dass die Korrektheit des Algorithmus von Dijkstra darauf beruht, dass es keine Kanten mit *negativem* Gewicht gibt, d.h. das „Verlängern“ eines Weges um eine Kante kann den Weg nicht verkürzen. Hat der Graph auch Kanten mit negativem Gewicht, so ist die Korrektheit des Algorithmus nicht mehr garantiert!

6.6.2 All-Pairs Shortest Paths

Im vorigen Abschnitt haben wir für einen gegebenen Startknoten s zu jedem anderen Knoten jeweils einen kürzesten Weg bestimmt. In diesem Abschnitt wollen wir nun für jedes Knotenpaar (v, w) einen kürzesten Weg von v nach w bestimmen. Dieses Problem ist als das *All-Pairs Shortest Paths Problem* bekannt:

All-Pairs Shortest Paths (APSP)	
Gegeben:	gerichteter Graph $G = (V, A)$ Gewichtsfunktion $w : A \to \mathbb{R}_0^+$
Gesucht:	ein kürzester Weg von u nach v für jedes Paar $u, v \in V$

Natürlich könnten wir einfach den Algorithmus von Dijkstra für jeden Knoten $v \in V$ aufrufen. Sein n die Anzahl der Knoten und m die Anzahl der Kanten des Graphen. Dann haben n Aufrufe von Dijkstra Zeitkomplexität $O(n(n+m) \log n)$. Für *dichte Graphen* (also $m = \Omega(n^2)$) ist die Laufzeit $O(n^3 \log n)$. Wir werden sehen, dass wir einen sehr einfachen Algorithmus für das APSP entwickeln können, der für dichte Graphen sogar eine bessere Laufzeit besitzt.

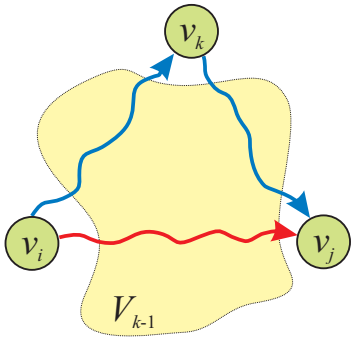


Abbildung 6.19: Zwei Möglichkeiten für einen kürzesten Weg von v_i nach v_j, der nur Zwischenknoten aus V_k benutzt: der Weg läuft über v_k (blau) oder benutzt v_k nicht (rot).

Wir nehmen im folgenden an, dass $v_1, \ldots, v_n$ die Knoten des Graphen sind und die Kantengewichte als $n \times n$-Matrix gegeben sind, so dass $w(v_i, v_j)$ das Gewicht der Kante (v_i, v_j) ist; falls die Kante (v_i, v_j) gar nicht in G enthalten ist, so ist $w(v_i, v_j) := \infty$.

Die Idee des Algorithmus besteht darin, zunächst *eingeschränkte Teilprobleme* zu betrachten und daraus dann die optimale Lösung zu konstruieren. Sei $k \in \mathbb{N}$ eine feste Zahl mit $0 \leq k \leq n$. Wir bezeichnen mit $V_k := \{v_1, \ldots, v_k\}$ die Menge der ersten k Knoten. Unsere Teilprobleme sind kürzeste Wege, die nur Knoten aus V_k als Zwischenknoten benutzen dürfen (*Zwischenknoten* sind alle Knoten auf einem Weg außer dem Anfangs- und dem Endknoten). Wir bezeichnen mit $d_{ij}^{(k)}$ die Länge eines kürzesten Weges von v_i nach v_j, der nur Knoten aus V_k als Zwischenknoten benutzt. Dann ist offensichtlich

$$d_{ij}^{(0)} = w(v_i, v_j).$$

Wie können wir nun $d_{ij}^{(k)}$ berechnen, wenn wir bereits $d_{i'j'}^{(k-1)}$ für alle $1 \leq i', j' \leq n$ bestimmt haben? Für einen kürzesten Weg p von v_i nach v_j gibt es zwei Möglichkeiten (vergleiche Abbildung 6.19):

1. Der Weg p benutzt v_k *nicht* als Zwischenknoten. Dann ist $d_{ij}^{(k)} = d_{ij}^{(k-1)}$.
2. Der Weg p benutzt v_k als Zwischenknoten. Dann setzt sich p aus einem kürzesten Weg von v_i nach v_k und einem von v_k nach v_j zusammen, die jeweils nur Knoten aus V_{k-1} als Zwischenknoten benutzen! In dem Fall ist also $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Offensichtlich erhalten wir einen entsprechenden kürzesten Weg, wenn wir das Minimum der beiden Alternativen wählen, d.h.

$$d_{ij}^{(k)} := \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}). \tag{6.1}$$

Für $k = n$ haben wir dann die Länge des kürzesten Weges für jedes Knotenpaar bestimmt. Der Algorithmus in Listing 6.11 löst auf diese Art und Weise das APSP.

Eingabe: gerichteter Graph $G = (V, A)$ mit Knoten $v_1, \ldots, v_n$ und Gewichtsfunktion $w : V \times V \to \mathbb{R}_0^+$
Ausgabe: Länge des kürzesten Weges von v_i nach v_j in $d_{ij}^{(n)}$

```
1: for i := 1 to n do
2:     for j := 1 to n do
3:         d_ij^(0) := w(v_i, v_j)
4:     end for
5: end for
6: for k := 1 to n do
7:     for i := 1 to n do
8:         for j := 1 to n do
9:             d_ij^(k) := min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
10:        end for
11:    end for
12: end for
```

Listing 6.11: Algorithmus von Floyd-Warshall

Sind wir neben der Längen der kürzesten Wege auch an den Wegen selbst interessiert, so können wir diese in einer $n \times n$ Matrix $\Pi = (\pi_{ij})$ darstellen. Dabei ist π_{ij} der Vorgänger von j auf einem kürzesten Weg von i nach j, bzw. *nil*, falls $i = j$ oder gar kein Weg von i nach j existiert. Wir können parallel zu den $d_{ij}^{(k)}$-Werten jeweils die Vorgänger auf unseren eingeschränkten kürzesten Wegen in $\pi_{ij}^{(k)}$ berechnen. Die Initialisierung ist dann

$$\pi_{ij}^{(0)} := \begin{cases} \textit{nil} & \text{falls } i = j \text{ oder } w(v_i, v_j) = \infty \\ i & \text{sonst} \end{cases}$$

und wir bestimmen $\pi_{ij}^{(k)}$ entsprechend der Wahl des Minimums in (6.1):

$$\pi_{ij}^{(k)} := \begin{cases} \pi_{ij}^{(k-1)} & \text{falls } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{sonst} \end{cases}$$

Dann ist $\pi_{ij} := \pi_{ij}^{(n)}$ die gesuchte Lösung.

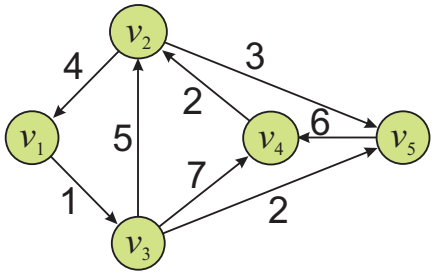


Abbildung 6.20: Beispielgraph für Anwendung von Floyd-Warshall.

Beispiel 6.11. Die Distanzmatrizen $D^{(k)} = \left(d_{ij}^{(k)}\right)_{1\leq i,j\leq n}$ für den Beispielgraphen aus Abbildung 6.20 sehen wie folgt aus:

$$D^{(0)} = \begin{pmatrix} 0 & \infty & 1 & \infty & \infty \\ 4 & 0 & \infty & \infty & 3 \\ \infty & 5 & 0 & 7 & 2 \\ \infty & 2 & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(1)} = \begin{pmatrix} 0 & \infty & 1 & \infty & \infty \\ 4 & 0 & 5 & \infty & 3 \\ \infty & 5 & 0 & 7 & 2 \\ \infty & 2 & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & \infty & 1 & \infty & \infty \\ 4 & 0 & 5 & \infty & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(3)} = \begin{pmatrix} 0 & 6 & 1 & 8 & 3 \\ 4 & 0 & 5 & 12 & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 6 & 1 & 8 & 3 \\ 4 & 0 & 5 & 12 & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ 12 & 8 & 13 & 6 & 0 \end{pmatrix} \quad D^{(5)} = \begin{pmatrix} 0 & 6 & 1 & 8 & 3 \\ 4 & 0 & 5 & 9 & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ 12 & 8 & 13 & 6 & 0 \end{pmatrix}$$

Analyse der Laufzeit und des Speicherverbrauchs. So, wie wir den Algorithmus angegeben haben, benötigen er zur Speicherung aller $d_{ij}^{(k)}$-Werte n Matrizen der Größe n^2, was einem Speicherverbrauch von $\Theta(n^3)$ entspricht. Es ist aber leicht zu sehen, dass das verbessert werden kann. Für festes k benötigen wir nämlich zum Berechnen der Werte $d_{ij}^{(k)}$ nur die Werte $d_{ij}^{(k-1)}$ (für $1 \leq i, j \leq n$), aber kein $d_{ij}^{(k')}$ mit $k' < k-1$. Wir können also mit zwei Matrizen auskommen (eine für die bereits berechneten Werte, eine für die neu zu berechnenden Werte). Dann ist der Speicherverbrauch $\Theta(n^2)$. Analoges gilt, wenn wir auch die Vorgänger

π_{ij} berechnen wollen.

Die Laufzeit ist auf Grund der drei geschachtelten **for**-Schleifen offensichtlich $\Theta(n^3)$. Wir sparen also gegenüber der Variante, die n-mal den Algorithmus von Dijkstra aufruft, einen Faktor von $\log n$.

Theorem 6.12. *Der Algorithmus von Floyd-Warshall (mit obigen Modifikationen) kann das APSP in einer Laufzeit von $\Theta(n^3)$ mit einem Speicherverbrauch von $\Theta(n^2)$ lösen.*

Anmerkung 6.11. Es ist auch möglich, den Algorithmus von Dijkstra durch eine andere Implementierung der Priority-Queue mit einer Laufzeit von $O(n^2 + m)$ zu realisieren, wobei m die Anzahl der Kanten ist. Dazu speichert man einfach die Priorität jedes Knotens in einem Knotenfeld P, d.h. $P[v]$ is die Priorität von Knoten v. Dann benötigen INSERT und DECREASEPRIORITY jeweils konstante und EXTRACTMIN lineare Zeit. Die dadurch erzielte Laufzeit für Dijkstra ist zwar für dünne Graphen schlechter als die Implementierung mit Binary-Heaps, jedoch benötigen n Aufrufe von Dijkstra (für beliebige Graphen) auch nur Zeit $O(n^3)$.

Die theoretisch beste Laufzeit für Dijkstra erzielt man, wenn man die Priority-Queue mittels Fibonacci-Heaps (siehe z.B. Cormen et al.) implementiert. Damit erreicht man eine Laufzeit von $O(m + n \log n)$ für Dijkstra, und somit $O(n \cdot m + n^2 \log n)$ für APSP. Das ist für dichte Graphen genauso gut wie Floyd-Warshall und für dünne Graphen sogar besser.

Der entscheidende Vorteil des Algorithmus von Floyd-Warshall ist jedoch, dass er konzeptionell einfacher und damit auch einfacher zu implementieren ist.

Kapitel 7

Optimierung

Optimierungsprobleme sind Probleme, die im Allgemeinen viele zulässige Lösungen besitzen. Jeder Lösung ist ein bestimmter Wert (Zielfunktionswert, Kosten) zugeordnet. Optimierungsalgorithmen suchen in der Menge aller zulässigen Lösungen diejenigen mit dem besten, dem *optimalen*, Wert. Beispiele für Optimierungsprobleme sind:

- Minimal aufspannender Baum (MST)
- Kürzeste Wege in Graphen
- Längste Wege in Graphen
- Handelsreisendenproblem (Traveling Salesman Problem, TSP)
- Rucksackproblem
- Scheduling (z.B. Maschinen, Crew)
- Zuschneideprobleme (z.B. Bilderrahmen, Anzüge)
- Packungsprobleme

Wir unterscheiden zwischen Optimierungsproblemen, für die wir Algorithmen mit polynomiellem Aufwand kennen („in P“), und solchen, für die noch kein polynomieller Algorithmus bekannt ist. Darunter fällt auch die Klasse der NP-schwierigen Optimierungsprobleme. Die Optimierungsprobleme dieser Klasse besitzen die Eigenschaft, dass das Finden eines polynomiellen Algorithmus für eines dieser Optimierungsprobleme auch polynomielle Algorithmen für alle anderen Optimierungsprobleme dieser Klasse nach sich ziehen würde. Die meisten Informatiker glauben, dass für NP-schwierige Optimierungsprobleme keine polynomiellen Algorithmen existieren. Die oben erwähnten Beispiele für Optimierungsprobleme

sind fast alle NP-schwierig, lediglich für das MST-Problem sowie das kürzeste Wege Probleme sind Algorithmen mit polynomieller Laufzeit bekannt.

Ein Algorithmus hat polynomielle Laufzeit, wenn die Laufzeitfunktion $T(n)$ durch ein Polynom in n beschränkt ist. Dabei steht n für die Eingabegröße der Instanz. Z.B. ist das Kürzeste-Wegeproblem polynomiell lösbar, während das Längste-Wegeproblem im Allgemeinen NP-schwierig ist. (Transformation vom Handlungsreisendenproblem: Wäre ein polynomieller Algorithmus für das Längste-Wegeproblem bekannt, dann würde dies direkt zu einem polynomiellen Algorithmus für das Handlungsreisendenproblem führen.) Dies scheint auf den ersten Blick ein Widerspruch zu sein: wenn man minimieren kann, dann sollte man nach einer Kostentransformation („mal -1“) auch maximieren können. Allerdings ist für das Kürzeste-Wegeproblem nur dann ein polynomieller Algorithmus bekannt, wenn die Instanz keine Kreise mit negativen Gesamtkosten enthält. Und genau das ist nach der Kostentransformation nicht mehr gewährleistet.

Wir konzentrieren uns hier meist auf *kombinatorische Optimierungsprobleme*, das sind solche, die auf eine endliche Grundmenge beschränkt sind.

Definition 7.1. Ein *kombinatorisches Optimierungsproblem* ist folgendermaßen definiert. Gegeben sind eine endliche Menge E (*Grundmenge*), eine Teilmenge I der Potenzmenge 2^E von E (*zulässige Mengen*), sowie eine Kostenfunktion $c : E \to K$. Gesucht ist eine Menge $I^* \in I$, so dass

$$c(I^*) = \sum_{e \in I^*} c(e)$$

so groß (klein) wie möglich ist.

Hat man ein Optimierungsproblem gegeben, so bezeichnet man eine Instantiierung dieses Problems mit speziellen Eingabedaten als *Probleminstanz*. Im Prinzip lassen sich Instanzen von kombinatorischen Optimierungsproblemen durch Enumeration aller möglichen zulässigen Lösungen in endlicher Zeit lösen. Allerdings dauert eine Enumeration i.A. viel zu lange (s. Kapitel 7.3).

Für polynomielle Optimierungsaufgaben existieren verschiedene Strategien zur Lösung. Oft sind das speziell für das jeweilige Problem entwickelte Algorithmen. Manchmal jedoch greifen auch allgemeine Strategien wie das *Greedy-Prinzip* oder die *dynamische Programmierung*, die wir in diesem Kapitel kennen lernen werden.

NP-schwierige Optimierungsprobleme treten in der Praxis häufiger auf. Für sie betrachten wir *exakte Verfahren*, d.h. solche, die immer optimale Lösungen liefern, aber wegen des exponentiell steigenden Zeitaufwands im Allgemeinen nur für kleinere Probleminstanzen verwendet werden können. Als exakte Verfahren studieren wir *Enumeration* (s. Kapitel 7.3), *Branch-and-Bound* (s. Kapitel 7.4) und *Dynamische Programmierung* (s. Kapitel 7.5) an

ausgewählten Beispielen. In der Praxis werden für NP-schwierige Optimierungsprobleme häufig *Heuristiken* verwendet, die meist die optimale Lösung nur annähern.

Wir beginnen dieses Kapitel mit der Einführung in verschiedene Optimierungsprobleme und einfachen konstruktiven Heuristiken (s. Kapitel 7.1). Wir werden sehen, dass es auch Heuristiken gibt, die eine Gütegarantie für die Lösung abgeben. Diese Algorithmen nennt man *approximative Algorithmen* (s. Kapitel 7.2). Am Ende des Kapitels werden wir allgemeine *Verbesserungsheuristiken* kennenlernen, deren Ziel es ist, eine gegebene Lösung durch lokale Änderungen zu verbessern (s. Kapitel 7.6).

7.1 Heuristiken

In diesem Abschnitt stellen wir typische NP-schwierige Optimierungsprobleme vor, an denen wir die verschiedenen Algorithmen zur Lösung von Optimierungsproblemen demonstrieren werden. Es handelt sich hierbei um kombinatorische Optimierungsprobleme.

7.1.1 Travelling Salesman Problem

Ein klassisches kombinatorisches Optimierungsproblem ist das *Travelling Salesman Problem* (TSP) (auch symmetrisches TSP, Rundreiseproblem bzw. Handlungsreisendenproblem genannt).

Travelling Saleman Problem (TSP)	
Gegeben:	vollständiger ungerichteter Graph $G = (V, E)$ mit Gewichtsfunktion $c : E \to \mathbb{R}$
Gesucht:	Tour T (Kreis, der jeden Knoten genau einmal enthält) von G mit minimalem Gewicht $c(T) = \sum_{e \in T} c(e)$

Das TSP ist ein NP-schwieriges Optimierungsproblem. Man kann sich leicht überlegen, dass es in einer TSP-Instanz mit $n = |V|$ Städten genau $(n-1)!/2$ viele verschiedene Touren gibt. Die Enumeration aller möglichen Touren ist also nur für sehr wenige Städte (kleines n) möglich. Bereits für $n = 12$ gibt es 19.958.400 verschiedene Touren. Für $n = 25$ sind es bereits ca. 10^{23} Touren und für $n = 60$ ca. 10^{80} Touren (zum Vergleich: die Anzahl der Atome im Weltall wird auf ca. 10^{80} geschätzt. Selbst mit einem „Super-Rechner“, der 40 Millionen Touren pro Sekunde enummerieren könnte, würde für 10^{80} Touren ca. 10^{64} Jahre benötigen.

Und trotzdem ist es möglich mit Hilfe von ausgeklügelten Branch-and-Bound Verfahren (s. Kapitel 7.4) TSP-Instanzen mit bis zu 85.900 Städten praktisch (beweisbar!) optimal zu

Eingabe: vollständiger ungerichteter Graph $G = (V, E)$ mit Kantenkosten c(e)
Ausgabe: zulässige Lösung für die gegebene TSP-Instanz

1: Beginne mit einer leeren Tour $T := \emptyset$
2: Beginne an einem Knoten $v := v_0$ ▷ Ende der Tour
3: **while** noch freie Knoten existieren **do**
4: Suche den nächsten freien (noch nicht besuchten) Knoten w zu v ▷ billigste Kante (v, w)
5: Addiere die Kante (v, w) zu T.
6: **end while**
7: Addiere die Kante (v, v_0)

Listing 7.1: Nearest-Neighbor (NN) Heuristik für das TSP.

lösen. Dies jedoch nur nach relativ langer Rechenzeit (siehe `www.tsp.gatech.edu` bzw. Folien zur Vorlesung). Mit diesen Methoden kann man auch TSP-Instanzen bis zu einer Größe von ca. 300 Städten relativ schnell lösen.

Wir betrachten in diesem Abschnitt eine einfache Greedy-Heuristik für das TSP. *Greedy-Algorithmen* sind *„gierige“* Verfahren zur exakten oder approximativen Lösung von Optimierungsaufgaben, welche die Lösung iterativ aufbauen (z.B. durch schrittweise Hinzunahme von Objekten) und sich in jedem Schritt für die jeweils lokal beste Lösung entscheiden. Eine getroffene Entscheidung wird hierbei niemals wieder geändert. Daher sind Greedy-Verfahren bezogen auf die Laufzeit meist effizient, jedoch führen sie nur in seltenen Fällen (wie etwa für das kürzeste Wegeproblem oder das MST-Problem) zur optimalen Lösung.

Die Nearest-Neighbor Heuristik ist in Algorithmus 7.1 gegeben. Sie beginnt an einem Startknoten und addiert iterativ jeweils eine lokal beste Kantezum Ende der Tour hinzu. Das Problem bei diesem Vorgehen ist, dass das Verfahren sich selbst ab und zu in eine Sackgasse führt, so dass besonders teure (lange) Kanten benötigt werden, um wieder weiter zu kommen. Auch die letzte hinzugefügte Kante ist hier i.A. sehr teuer (s. Beispiel auf den Folien).

Wie gut ist nun die Nearest-Neighbor Heuristik (NN-Heuristik) im allgemeinen? Man kann zeigen, dass es Probleminstanzen gibt, für die sie beliebig schlechte Ergebnisse liefert. Eine solche Instanz ist z.B. in Beispiel 7.1 gegeben.

Beispiel 7.1. *Eine Worst-Case Instanz für die NN-Heuristik:* Die Knoten der TSP-Instanz seien nummeriert von 1 bis n und die Kantengewichte sind wie folgt: $c(i, i + 1) = 1$ für $i = 1, ..., n - 1$, $c(1, n) = M$, wobei M eine sehr große Zahl ist) und $c(i, j) = 2$ sonst.

Eine von der NN-Heuristik produzierte Lösung mit Startknoten 1 enthält die Kanten $(1, 2), (2, 3), ..., (n - 1, n)$ und $(n, 1)$. Der Wert dieser Lösung ist $(n - 1) + M$.

Eine optimale Tour ist z.B. gegeben durch die Kanten $(1, 2), (2, 3), ..., (n-3, n-2), (n-2, n), (n, n - 1)$ und $(n - 1, 1)$. Der optimale Wert ist also $(n - 2) + 4 = n + 2$.

Da M beliebig groß sein kann, kann die NN-Heuristik einen Lösungswert liefern, der beliebig weit vom optimalen Lösungswert entfernt ist (also beliebig schlecht ist).

7.1.2 Verschnitt- und Packungsprobleme

Ein anderes klassiches NP-schwieriges Optimierungsproblem ist die *Eindimensionale Verschnittoptimierung*.

Eindimensionale Verschnittoptimierung	
Gegeben:	Gegenstände $1, ..., N$ der Größe w_i und beliebig viele Rohlinge der GroßeK
Gesucht:	Finde die kleinste Anzahl von Rohlingen, aus denen alle Gegenstände geschnitten werden können.

Zu unserem Verschnittproblem gibt es ein äquivalentes Packungsproblem, das *Bin-Packing Problem*, bei dem es um möglichst gutes Packen von Kisten geht:

Bin-Packing Problem	
Gegeben:	Gegenstände $1, \ldots, N$ der Größe w_i und beliebig viele Kisten der Größe K.
Gesucht:	Finde die kleinste Anzahl von Kisten, die alle Gegenstände aufnehmen.

Eine bekannte Greedy-Heuristik ist die *First-Fit Heuristik* (FF-Heuristik), die folgendermaßen vorgeht: Die Gegenstände werden in einer festen Reihenfolge betrachtet. Jeder Gegenstand wird in die erstmögliche Kiste gelegt, in die er paßt. Beispiel 7.2 führt die FF-Heuristik an einer Probleminstanz durch.

Beispiel 7.2. Wir führen die First-Fit Heuristik an der folgenden Probleminstanz durch. Gegeben sind beliebig viele Kisten der Größe $K = 101$ und 37 Gegenstände der folgenden Größe: 7× Größe 6, 7× Größe 10, 3× Größe 16, 10× Größe 34, und 10× Größe 51.

Die First-Fit Heuristik berechnet die folgende Lösung:

1	×	(7×) 6 | (5×)10	$\sum = 92$
1	×	(2×)10 | (3×)16	$\sum = 68$
5	×	(2×)34	$\sum = 68$
10	×	(1×)51	$\sum = 51$

Das ergibt insgesamt 17 Kisten.

Wieder fragen wir uns, wie gut die von der First-Fit Heuristik berechneten Lösungen im allgemeinen sind. Für unser Beispiel sieht die optimale Lösung folgendermaßen aus:

$3\times$	51	34	16		$\sum = 101$
$7\times$	51	34	10	6	$\sum = 101$

Die dargestellte Lösung benötigt nur 10 Kisten. Offensichtlich ist diese auch die optimale Lösung, da jede Kiste auch tatsächlich ganz voll gepackt ist.

Für die Probleminstanz P_1 in Beispiel 7.2 weicht also der Lösungswert der First-Fit Heuristik $c_{FF}(P_1)$ nicht beliebig weit von der optimalen Lösung $c_{opt}(P_1)$ ab. Es gilt: $c_{FF}(P_1) \leq \frac{17}{10} c_{opt}(P_1)$. Aber vielleicht existiert ja eine andere Probleminstanz, bei der die FF-Heuristik nicht so gut abschneidet? Wir werden dieser Frage im nächsten Kapitel nachgehen.

7.1.3 0/1-Rucksackproblem

Wir betrachten ein weiteres NP-schwieriges Optimierungsproblem, nämlich das *0/1-Rucksackproblem*. Hier geht es darum aus einer gegebenen Menge von Gegenständen möglichst wertvolle auszuwählen, die in einem Rucksack mit gegebener Maximalgröße zu verpacken sind.

0/1-Rucksackproblem	
Gegeben:	N Gegenstände i mit Gewicht (Größe) w_i, und Wert (Kosten) c_i, und ein Rucksack der Größe K.
Gesucht:	Menge der in den Rucksack gepackten Gegenstände mit maximalem Gesamtwert; dabei darf das Gesamtgewicht den Wert K nicht überschreiten.

Wir führen 0/1-Entscheidungsvariablen für die Wahl der Gegenstände ein:

$$x_1, \ldots, x_N, \text{ wobei } x_i = \begin{cases} 0 & \text{falls Element } i \text{ nicht gewählt wird} \\ 1 & \text{falls Element } i \text{ gewählt wird} \end{cases}$$

Das 0/1-Rucksackproblem lässt sich nun formal folgendermaßen definieren:

Maximiere

$$\sum_{i=1}^{N} c_i x_i,$$

wobei

$$x_i \in \{0,1\} \wedge \sum_{i=1}^{N} w_i x_i \leq K.$$

Anmerkung 7.1. Das Rucksackproblem taucht in vielen Varianten auf, u.a. auch in einer Variante, bei der man mehrere Gegenstände des gleichen Typs hat. Um unser Problem von dieser Variante abzugrenzen, nennen wir unser Problem 0/1-Rucksackproblem. Wenn es jedoch aus dem Zusammenhang klar ist, welche Variante hier gemeint ist, dann verzichten wir auch auf den Zusatz 0/1.

Die *Greedy-Heuristik* für das 0/1-Rucksackproblem geht folgendermaßen vor. Zunächst werden die Gegenstände nach ihrem Nutzen sortiert. Der Nutzen von Gegenstand i ist gegeben durch c_i/w_i. Für alle Gegenstände i wird dann in der sortierten Reihenfolge ausprobiert, ob der Gegenstand noch in den Rucksack paßt. Wenn dies so ist, dann wird er hinzugefügt.

Beispiel 7.3. Das Rucksack-Problem mit $K = 17$ und folgenden Daten:

Gegenstand	a	b	c	d	e	f	g	h
Gewicht	3	4	4	6	6	8	8	9
Wert	3	5	5	10	10	11	11	13
Nutzen	1	1,25	1,25	1,66	1,66	1,375	1,375	1,44

Der Nutzen eines Gegenstands i errechnet sich aus c_i/w_i.
Das Packen der Gegenstände a, b, c und d führt zu einem Gesamtwert von 23 bei dem maximal zulässigen Gesamtgewicht von 17. Entscheidet man sich für die Gegenstände c, d und e, dann ist das Gewicht sogar nur 16 bei einem Gesamtwert von 25.

Die Greedy-Heuristik sortiert die Gegenstände zuerst nach ihrem Nutzen, damit erhalten wir die Reihenfolge d, e, h, f, g, b, c, a. Es werden also die Gegenstände d, e und b in den Rucksack gepackt mit Gesamtwert 25.

Wir stellen uns wieder die Frage, wie gut nun die Greedy-Heuristik im allgemeinen ist. Hierzu zeigt uns die in Beispiel 7.4 gegebene Instanz, dass die erzielten Lösungswerte der Greedy-Heuristik beliebig weit von den optimalen Werten entfernt sein können.

Beispiel 7.4. *Eine Worst-Case Rucksack-Probleminstanz für die Greedy-Heuristik.* Die gegebenen Gegenstände haben jeweils Gewichte und Werte $w_i = c_i = 1$ für $i = 1, ..., N-1$ und der n-te Gegenstand $c_n = K - 1$ und $w_N = K = MN$, wobei M eine sehr große Zahl ist. Der Rucksack hat Größe K (ist also sehr groß).

Die Greedy-Heuristik packt die ersten $N - 1$ Gegenstände in den Rucksack. Danach ist kein Platz mehr für das N-te Element. Die erzielte Lösung hat einen Lösungswert gleich $N - 1$.

Die optimale Lösung hingegen packt nur das N-te Element ein und erreicht dadurch einen Lösungswert von $K - 1 = MN - 1$.

Da M beliebig groß sein kann, ist auch dies eine Instanz bei der die Greedy-Heuristik beliebig *schlecht* sein kann, da der berechnete Lösungswert beliebig weit vom optimalen Wert entfernt sein kann.

Wir haben also in diesem Abschnitt drei verschiedene Greedy-Heuristiken für drei verschiedene NP-schwierige Optimieruungsprobleme kennengelernt. Dabei haben wir gesehen, dass manchmal die erzielte Lösung sehr weit von der Optimallösung entfernt sein kann. Im Fall der First-Fit Heuristik für Bin-Packing haben wir jedoch eine solche *schlechte* Instanz nicht gefunden. Im folgenden Abschnitt gehen wir diesen Effekten genauer nach.

7.2 Approximative Algorithmen und Gütegarantien

Ist ein Problem NP-schwierig, so können größere Instanzen im Allgemeinen nicht exakt gelöst werden, d.h. es besteht kaum Aussicht, mit Sicherheit optimale Lösungen in akzeptabler Zeit zu finden. In einer solchen Situation ist man dann auf *Heuristiken* angewiesen, also Verfahren, die zulässige Lösungen für das Optimierungsproblem finden, die aber nicht notwendigerweise optimal sind.

In diesem Abschnitt beschäftigen wir uns mit Heuristiken, die Lösungen ermitteln, über deren Qualität (Güte) man bestimmte Aussagen treffen kann. Man nennt solche Heuristiken mit Gütegarantien *approximative Algorithmen*.

Die „Güte“ eines Algorithmus sagt etwas über seine Fähigkeiten aus, optimale Lösungen gut oder schlecht anzunähern. Die formale Definition lautet wie folgt:

Definition 7.2. Sei A ein Algorithmus, der für jede Probleminstanz P eines Optimierungsproblems Π eine zulässige Lösung mit positivem Wert liefert. Dann definieren wir $c_A(P)$ als den Wert der Lösung des Algorithmus A für Probleminstanz $P \in \Pi$; $c_{\text{opt}}(P)$ sei der optimale Wert für P.

Für Minimierungsprobleme gilt: Falls

$$\frac{c_A(P)}{c_{\text{opt}}(P)} \leq \varepsilon$$

für alle Probleminstanzen P und $\varepsilon > 0$, dann heißt A ein ε-*approximativer* Algorithmus und die Zahl ε heißt *Gütegarantie* von Algorithmus A.

Für Maximierungsprobleme gilt: Falls

$$\frac{c_A(P)}{c_{\text{opt}}(P)} \geq \varepsilon$$

für alle Probleminstanzen P und $\varepsilon > 0$, dann heißt A ein ε-*approximativer* Algorithmus und die Zahl ε heißt *Gütegarantie* von Algorithmus A.

Beispiel 7.5. Wir betrachten als Beispiel ein Minimierungsproblem und jeweils die Lösungen einer Heuristik A und die optimale Lösung zu verschiedenen Probleminstanzen P_i:

$$c_A(P_1) = 100 \quad \text{und} \quad c_{\text{opt}}(P_1) = 50$$
$$c_A(P_2) = 70 \quad \text{und} \quad c_{\text{opt}}(P_2) = 40$$

Die Information, die wir daraus erhalten, ist lediglich, dass die Güte der Heuristik A nicht besser als 2 sein kann. Nur, falls

$$\frac{c_A(P_i)}{c_{\text{opt}}(P_i)} \leq 2$$

für alle möglichen Probleminstanzen P_i gilt, ist A ein 2-approximativer Algorithmus.

Anmerkung 7.2. Es gilt

- für Minimierungsprobleme: $\varepsilon \geq 1$,
- für Maximierungsprobleme: $\varepsilon \leq 1$,
- $\varepsilon = 1 \Leftrightarrow A$ ist exakter Algorithmus

Im Folgenden betrachten wir verschiedene approximative Algorithmen für das Bin-Packing-Problem und für das Travelling Salesman Problem.

7.2.1 Bin-Packing – Packen von Kisten

In Beispiel 7.2 hat die First-Fit Heuristik eine Lösung mit 17 Kisten generiert, während die optimale Lösung nur 10 Kisten benötigt. Daraus folgt

$$\frac{c_{FF}(P_1)}{c_{\text{opt}}(P_1)} = \frac{17}{10}$$

für diese Probleminstanz P_1. Dies lässt bisher nur den Rückschluss zu, dass die Gütegarantie der FF-Heuristik auf keinen Fall besser als $\frac{17}{10}$ sein kann.

Wir beweisen zunächst eine Güte von asymptotisch 2 für die First-Fit Heuristik. Das heißt also, dass die Lösungen der FF-Heuristik für alle möglichen Probleminstanzen nie mehr als doppelt so weit von der optimalen Lösung entfernt sein können. Die FF-Heuristik ist also ein Approximationsalgorithmus mit Faktor 2 für das Bin-Packing Problem.

Theorem 7.1. *Es gilt:* $c_{FF}(P) \leq 2c_{\text{opt}}(P) + 1$ *für alle* $P \in \Pi$.

Beweis. Offensichtlich gilt: Jede FF-Lösung füllt alle bis auf eine der belegten Kisten mindestens bis zur Hälfte. Daraus folgt für alle Probleminstanzen $P \in \Pi$:

$$c_{FF}(P) \leq \frac{\sum_{j=1}^{N} w_j}{K/2} + 1$$

Da

$$\sum_{j=1}^{N} w_j \leq K c_{\text{opt}}(P)$$

folgt

$$c_{FF}(P) \leq 2c_{\text{opt}}(P) + 1 \quad \Rightarrow \quad \frac{c_{FF}(P)}{c_{\text{opt}}(P)} \leq 2 + \frac{1}{c_{\text{opt}}(P)}$$

□

Man kann sogar noch eine schärfere Güte zeigen. Es gilt (ohne Beweis):

$$\frac{c_{FF}(P)}{c_{\text{opt}}(P)} < \frac{17}{10} + \frac{2}{c_{\text{opt}}(P)} \qquad \forall P \in \pi$$

Weiterhin kann man zeigen, dass $\frac{17}{10}$ der asymptotisch beste Approximationsfaktor für die FF-Heuristik ist. Für interessierte Studierende ist der Beweis in Johnson, Demers, Ullman, Garey, Graham in *SIAM Journal on Computing*, vol. 3 no. 4, S. 299-325, 1974, zu finden.

7.2.2 Das symmetrische TSP

Wir haben im vorigen Abschnitt gesehen, dass die Nearest-Neighbor Heuristik Lösungen für das TSP generieren kann, die beliebig weit von dem optimalen Lösungswert entfernt sein können. Im folgenden betrachten wir eine andere beliebte Heuristik für das TSP, die *Spanning Tree Heuristik*.

Eingabe: vollständiger ungerichteter Graph $G = (V, E)$ mit Kantenkosten c(e)
Ausgabe: zulässige Tour T für die gegebene TSP-Instanz

1: Bestimme einen minimalen aufspannenden Baum B von K_n.
2: Verdopple alle Kanten aus $B \rightarrow$ Graph (V, B_2).
3: Bestimme eine Eulertour F im Graphen (V, B_2).
4: Gib dieser Eulertour F eine Orientierung $\rightarrow F$
5: Wähle einen Knoten $i \in V$, markiere i, setze $p = i, T = \emptyset$.
6: **while** noch unmarkierte Knoten existieren **do**
7: Laufe von p entlang der Orientierung F bis ein unmarkierter Knoten q erreicht ist.
8: Setze $T = T \cup \{(p, q)\}$, markiere q, und setze $p = q$.
9: **end while**
10: Setze $T = T \cup \{(p, i)\} \rightarrow$ STOP; T ist die Ergebnis-Tour.

Listing 7.2: Spanning Tree (ST) Heuristik für das TSP.

Spanning-Tree-Heuristik (ST)

Die Spanning Tree (ST) Heuristik basiert auf der Idee einen minimal aufspannenden Baum zu generieren und daraus eine Tour abzuleiten. Dafür bedient sie sich einer sogenannten Eulertour.

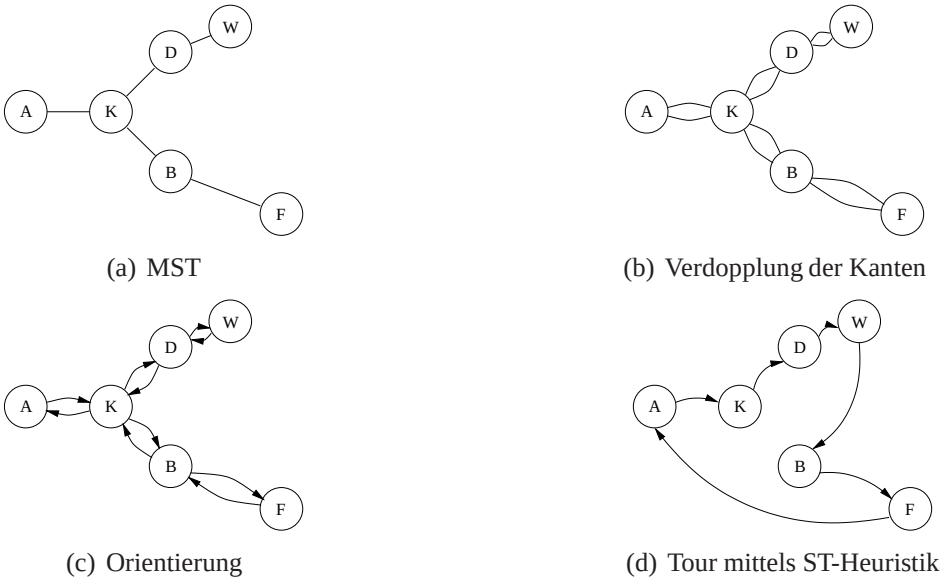
Definition 7.3. Eine *Eulertour* ist ein geschlossener Kantenzug, der jede Kante des Graphen genau einmal enthält.

In einem Graphen G existiert genau dann eine Eulertour wenn alle Knotengrade in G gerade sind. Dies kann man sich leicht überlegen: Jede Eulertour, die einen Knoten besucht (hineingeht) muss auch wieder aus dem Knoten heraus. Eine Eulertour kann in linearer Zeit berechnet werden (ohne Beweis).

Die ST-Heuristik berechnet also zuerst einen Minimum Spanning Tree T und verdoppelt dann die Kanten in T um alle Knotengrade gerade zu bekommen. Die Eulertour ist jedoch keine zulässige TSP-Tour, da diese im allgemeinen Knoten mehrfach besuchen kann. Deswegen muss aus der Eulertour eine zulässige Tour konstruiert werden. Dies wird durch "Abkürzungen" entlang der Eulertour erreicht. Listing 7.2 zeigt die einzelnen Schritte der ST-Heuristik.

Abbildung 7.1 zeigt eine Visualisierung der Spanning-Tree Heuristik anhand eines Beispiels (Städte in Nord-Rhein-Westfalen und Hessen: Aachen, Köln, Bonn, Düsseldorf, Frankfurt, Wuppertal).

Analyse der Gütegarantie. Leider ist das allgemeine TSP sehr wahrscheinlich nicht mit einer Konstanten approximierbar. Denn man kann zeigen: Gibt es ein $\varepsilon > 1$ und einen polyno-



(a) MST (b) Verdopplung der Kanten

(c) Orientierung (d) Tour mittels ST-Heuristik

Abbildung 7.1: ST-Heuristik

miellen Algorithmus A, der für jedes symmetrische TSP eine Tour T_A liefert mit $\frac{c_A(P)}{c_{\text{opt}}(P)} \leq \varepsilon$, dann ist P = NP.

Theorem 7.2. *Das Problem, das symmetrische TSP für beliebiges $\varepsilon > 1$ zu approximieren ist NP-schwierig (ohne Beweis).*

Doch es gibt auch positive Nachrichten. Wenn die gegebene TSP-Instanz gewisse Eigenschaften aufweist, dann ist diese doch sehr gut approximierbar:

Definition 7.4. Ein TSP heißt *euklidisch* wenn für die Distanzmatrix C die Dreiecks-Ungleichung gilt, d.h. für alle Knoten i, j, k gilt: $c_{ik} \leq c_{ij} + c_{jk}$ und alle $c_{ii} = 0$ sind.

Das heißt: es kann nie kürzer sein, von i nach k einen Umweg über eine andere Stadt j zu machen. Denkt man beispielsweise an Wegstrecken, so ist die geforderte Dreiecksungleichung meist erfüllt. Gute Nachrichten bringt das folgende Theorem:

Theorem 7.3. *Für die Probleminstanz P seien $c_{ST}(P)$ der von der ST-Heuristik erzielte Lösungswert und $c_{\text{opt}}(P)$ der Optimalwert. Für das euklidische TSP und die ST-Heuristik gilt:*

$$\frac{c_{ST}(P)}{c_{\text{opt}}(P)} \leq 2 \quad \text{für alle } P \in \Pi.$$

Ausgabe: Christofides-Tour: Ersetze Schritt 2: in Listing 7.2 (ST-Heuristik) durch:
1: Sei W die Menge der Knoten in (V, B) mit ungeradem Grad.
2: Bestimme im von W induzierten Untergraphen von K_n ein Minimum Perfektes Matching M.
3: Setze $B_2 = B \cup M$

Listing 7.3: Neue Teile der Christophides (CH) Heuristik für das TSP.

Beweis. Es gilt

$$c_{ST}(P) \leq c_{B_2}(P) = 2c_B(P) \leq 2c_{\text{opt}}(P).$$

Die erste Abschätzung gilt wegen der Dreiecksungleichung, die letzte, da ein Minimum Spanning Tree die billigste Möglichkeit darstellt, in einem Graphen alle Knoten zu verbinden. □

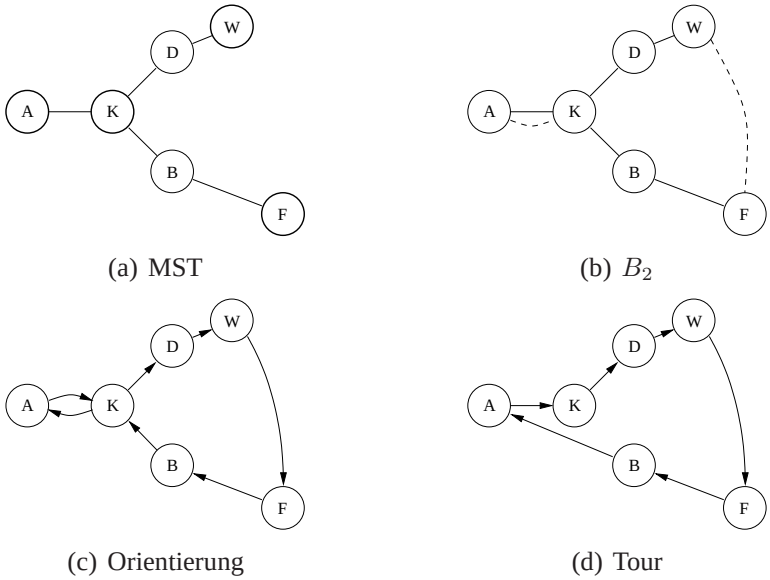
Im folgenden Abschnitt lernen wir eine Heuristik kennen, die das TSP noch besser approximiert.

Christophides-Heuristik (CH)

Diese 1976 publizierte Heuristik funktioniert ähnlich wie die Spanning-Tree-Heuristik, jedoch erspart man sich die Verdopplung der Kanten. Diese wird in der ST-Heuristik nur deswegen gemacht, weil dann sichergestellt ist, dass in dem Graphen eine Eulertour existiert. Es genügt jedoch statt einer Verdopplung aller Kanten, nur jeweils Kanten an den ungeraden Knoten zu dem Spanning Tree hinzuzunehmen. Diese zusätzliche Kantenmenge entspricht einem sogenannten Perfekten Matching auf den ungeraden Knoten im Spanning Tree.

Definition 7.5. Ein *Perfektes Matching* M in einem Graphen ist eine Kantenmenge, die jeden Knoten genau einmal enthält. Ein *Minimum Perfektes Matching* M ist ein Perfektes Matching mit dem kleinsten Gesamtgewicht, d.h. die Summe aller Kantengewichte c_e über die Kanten $e \in M$ ist minimal unter allen Perfekten Matchings.

Ein Perfektes Matching zwischen den ungeraden Knoten ordnet also jedem solchen Knoten einen eindeutigen Partnerknoten zu (Alle ungeraden Knoten werden zu Paaren zusammengefasst). Das Matching „geht auf“, denn wir wissen: die Anzahl der ungeraden Knoten ist immer gerade (s. Abschnitt 6.1). Um eine gute Approximationsgüte zu erreichen, berechnet die Christophides-Heuristik nicht irgendein Perfektes Matching, sondern dasjenige mit kleinstem Gesamtgewicht. Ein Minimum Perfektes Matching kann in polynomieller Zeit berechnet werden (ohne Beweis). Listing 7.3 zeigt die Schritte der Christofides-Heuristik, die Schritt (2) aus dem Listing 7.2 der ST-Heuristik ersetzen.



(a) MST (b) B_2

(c) Orientierung (d) Tour

Abbildung 7.2: Die CH-Heuristik an einem Beispiel

Abbildung 7.2 zeigt die Konstruktion an unserem Beispiel. Dabei betrachten wir für die Berechnung eines Minimalen Perfekten Matchings den vollständigen Untergraphen, der von den Knoten A, K, W, F induziert wird.

Analyse der Gütegarantie. Die Christofides-Heuristik erreicht eine bessere Approximationsgüte als die Spanning-Tree Heuristik für das euklidische TSP.

Theorem 7.4. *Für die Probleminstanz P seien $c_{CH}(P)$ der von der CH-Heuristik erzielte Lösungswert und $c_{\text{opt}}(P)$ der Optimalwert. Für das euklidische TSP und die Christophides-Heuristik gilt:*

$$\frac{c_{CH}(P)}{c_{\text{opt}}(P)} \leq \frac{3}{2} \text{ für alle } P \in \Pi$$

Beweis. Seien $i_1, i_2, \ldots, i_{2M}$ die Knoten von B mit ungeradem Grad und zwar so nummeriert, wie sie in einer optimalen Tour T_{opt} vorkommen, d.h. $T_{\text{opt}} = \{i_1, \ldots, i_2, \ldots, i_{2M}, \ldots\}$. Sei $M_1 = \{(i_1, i_2), (i_3, i_4), \ldots\}$ und $M_2 = \{(i_2, i_3), \ldots, (i_{2M}, i_1)\}$. Es gilt:

$$c_{\text{opt}}(P) \geq c_{M_1}(P) + c_{M_2}(P) \geq c_M(P) + c_M(P)$$

Die erste Abschätzung gilt wegen der Dreiecks-Ungleichung (Euklidisches TSP). Die zweite Abschätzung gilt, weil M das Matching kleinsten Gewichts ist. Weiterhin gilt:

$$c_{CH}(P) \leq c_{B_2}(P) = c_B(P) + c_M(P) \leq c_{\text{opt}}(P) + \frac{1}{2}c_{\text{opt}}(P) = \frac{3}{2}c_{\text{opt}}(P).$$

□

Anmerkung 7.3. Die Christophides-Heuristik (1976) war lange Zeit die Heuristik mit der besten Gütegarantie. Erst im Jahr 1996 zeigte Arora, dass das euklidische TSP beliebig nah approximiert werden kann: die Gütegarantie $\varepsilon > 1$ kann mit Laufzeit $O(N^{\frac{1}{\varepsilon-1}})$ approximiert werden. Eine solche Familie von Approximationsalgorithmen wird als *Polynomial Time Approximation Scheme* (PTAS) bezeichnet.

Anmerkung 7.4. Konstruktionsheuristiken für das symmetrische TSP erreichen in der Praxis meistens eine Güte von ca. 10-15% Abweichung von der optimalen Lösung. Die Christophides-Heuristik liegt bei ca. 14%.

7.3 Enumerationsverfahren

Exakte Verfahren, die auf einer vollständigen Enumeration beruhen, eignen sich für Optimierungsprobleme diskreter Natur. Für solche kombinatorische Optimierungsprobleme ist es immer möglich, die Menge aller zulässigen Lösungen aufzuzählen und mit der Kostenfunktion zu bewerten. Die am besten bewertete Lösung ist die optimale Lösung. Bei NP-schwierigen Problemen ist die Laufzeit dieses Verfahrens dann nicht durch ein Polynom in der Eingabegröße beschränkt.

Im Folgenden betrachten wir ein Enumerationsverfahren für das 0/1-Rucksackproblem.

7.3.1 Ein Enumerationsalgorithmus für das 0/1-Rucksack-Problem

Eine Enumeration aller zulässigen Lösungen für das 0/1-Rucksackproblem entspricht der Aufzählung aller Teilmengen einer N-elementigen Menge (bis auf diejenigen Teilmengen, die nicht in den Rucksack passen).

Der Algorithmus *Enum()* (s. Listing 7.4) basiert auf genau dieser Idee. Zu jedem Lösungsvektor x gehört ein Zielfunktionswert (Gesamtkosten von x) $xcost$ und ein Gesamtgewichtswert $xweight$. Die bisher beste gefundene Lösung wird in dem globalen Vektor $bestx$ und der zugehörige Lösungswert in der globalen Variablen $maxcost$ gespeichert. Der Algorithmus wird mit dem Aufruf $Enum(0, 0, 0, x)$ gestartet.

In jedem rekursiven Aufruf wird die aktuelle Lösung x bewertet. Danach werden die Variablen $x[1]$ bis $x[z]$ als fixiert betrachtet. Der dadurch beschriebene Teil des gesamten Suchraums wird weiter unterteilt: Wir betrachten alle möglichen Fälle, welche Variable $x[i]$ mit $i = z+1$ bis $i = N$ als nächstes auf 1 gesetzt werden kann; die Variablen $x[z+1]$ bis $x[i-1]$ werden gleichzeitig auf 0 fixiert. Alle so erzeugten kleineren Unterprobleme werden durch rekursive Aufrufe gelöst.

Wir folgen hier wieder dem Prinzip des *Divide & Conquer* („Teile und herrsche“), wie wir es beispielsweise schon von Sortieralgorithmen kennen: Das Problem wird rekursiv in kleinere Unterprobleme zerteilt, bis die Unterprobleme trivial gelöst werden können.

Eingabe: Anzahl z der fixierten Variablen in x; Gesamtkosten $xcost$; Gesamtgewicht $xweight$; aktueller Lösungsvektor x
Ausgabe: aktualisiert die globale bisher beste Lösung $bestx$ und ihre Kosten $maxcost$, wenn eine bessere Lösung gefunden wird

```
1:  if xweight ≤ K then
2:      if xcost > maxcost then
3:          maxcost = xcost;
4:          bestx = x;
5:      end if
6:      for i = z + 1, ..., N do
7:          x[i] = 1;
8:          Enum (i, xcost + c[i], xweight + w[i], x);
9:          x[i] = 0;
10:     end for
11: end if
```

Listing 7.4: Enum $(z, xcost, xweight, x)$;

Der Algorithmus ist korrekt, denn die Zeilen (6)-(9) enumerieren über alle möglichen Teilmengen einer N-elementigen Menge, die in den Rucksack passen. Die Zeilen (1)-(5) sorgen dafür, dass nur zulässige Lösungen betrachtet werden und die optimale Lösung gefunden wird.

Analyse der Laufzeit. Offensichtlich ist die Laufzeit in $O(2^N)$.

Wegen der exponentiellen Laufzeit ist das Enumerationsverfahren i.A. nur für kleine Instanzen des 0/1-Rucksackproblems geeignet. Bereits für $N \geq 50$ ist das Verfahren nicht mehr praktikabel. Deutlich besser geeignet für die Praxis ist das Verfahren der Dynamischen Programmierung, das wir in Abschnitt 7.5.1 betrachten werden.

7.4 Branch-and-Bound

Eine spezielle Form der beschränkten Enumeration, die auch auf dem Divide-&-Conquer-Prinzip basiert, ist das Branch-and-Bound Verfahren. Dabei werden durch Benutzung von Primal- und Dualheuristiken möglichst große Gruppen von Lösungen als nicht optimal bzw. gültig erkannt und von der vollständigen Enumeration ausgeschlossen. Hierzu bedient man sich sogenannter unterer bzw. oberer Schranken.

Definition 7.6. Für eine Instanz P eines Optimierungsproblems heißt $L > 0$ *untere Schranke* (*lower bound*), wenn für den optimalen Lösungswert gilt $c_{\text{opt}}(P) \geq L$. Der Wert $U > 0$ heißt *obere Schranke* (*upper bound*), wenn gilt $c_{\text{opt}}(P) \leq U$.

Die Idee von Branch-and-Bound ist einfach. Wir gehen hier von einem Problem aus, bei dem eine Zielfunktion minimiert werden soll. (Ein Maximierungsproblem kann durch Vorzeichenumkehr in ein Minimierungsproblem überführt werden.)

- Zunächst berechnen wir z.B. mit einer Heuristik eine zulässige (im Allgemeinen nicht optimale) Startlösung mit Wert U und eine untere Schranke L für alle möglichen Lösungswerte (Verfahren hierzu nennt man auch *Dualheuristiken*).
- Falls $U = L$ sein sollte, ist man fertig, denn die gefundene Lösung muss optimal sein.
- Ansonsten wird die Lösungsmenge partitioniert (*Branching*) und die Heuristiken werden auf die Teilmengen angewandt (*Bounding*).
- Ist für eine (oder mehrere) dieser Teilmengen die für sie berechnete untere Schranke L (*lower bound*) nicht kleiner als die beste überhaupt bisher gefundene obere Schranke (*upper bound* = eine Lösung des Problems), braucht man die Lösungen in dieser Teilmenge nicht mehr beachten. Man erspart sich also die weitere Enumeration.
- Ist die untere Schranke kleiner als die beste gegenwärtige obere Schranke, muss man die Teilmengen weiter zerkleinern. Man fährt solange mit der Zerteilung fort, bis für alle Lösungsteilmengen die untere Schranke mindestens so groß ist wie die (global) beste obere Schranke.

Ein Branch-and-Bound Algorithmus besteht also aus den Schritten

- **Branching:** Partitioniere die Lösungsmenge
- **Search:** Wähle eines der bisher erzuegten Teilprobleme
- **Bounding:** Berechne für die ausgewählte Teilmenge je eine untere und obere Schranke und prüfe, ob dieses Teilproblem weiter betrachtet werden muß.

Listing 7.5 beschreibt dieses Grundprinzip im Pseudocode. Die Hauptschwierigkeit besteht darin, *gute* Zerlegungstechniken und einfache Datenstrukturen zu finden, die eine effiziente Abarbeitung und Durchsuchung der einzelnen Teilmengen ermöglichen. Außerdem sollten die Heuristiken möglichst gute Schranken liefern, damit möglichst große Teilprobleme ausgeschlossen werden können.

7.4.1 Branch-and-Bound für das 0/1-Rucksackproblem

In diesem Abschnitt stellen wir einen konkreten Branch-and-Bound Algorithmus für das 0/1-Rucksackproblem vor. Hierbei handelt es sich um ein Maximierungsproblem, d.h. jede zulässige Lösung bildet eine untere Schranke für die gegebene Probleminstanz P.

Eingabe: Minimierungsproblem P
Ausgabe: Optimale Lösung U (und globale obere Schranke)

```
1:  var Liste offener (Sub-)probleme Π; lokale untere Schranke L'; lokale heuristische
    Lösung U'
2:  Setze U = ∞; Π = {P};
3:  while ∃P' ∈ Π do
4:      entferne P' von Π
5:      berechne für P' lokale untere Schranke L' mit Dualheuristik;        ▷ Bounding:
6:      if L' < U then                  ▷ Fall L' ≥ U braucht nicht weiter verfolgt zu werden
7:          berechne zulässige Lösung für P' mit Heuristik → obere Schranke U';
8:          if U' < U then
9:              U = U';                                  ▷ neue bisher beste Lösung gefunden
10:             entferne aus Π alle Subprobleme mit lokaler unterer Schranke ≥ U
11:         end if
12:         if L' < U then              ▷ Fall L' ≥ U braucht nicht weiter verfolgt zu werden
13:             partitioniere P' in Teilprobleme P_1, ..., P_k              ▷ Branching:
14:             Π = Π ∪ {P_1, ..., P_k};
15:         end if
16:     end if
17: end while
```

Listing 7.5: Branch-and-Bound für Minimierung

- **Init.** Eine zulässige Startlösung kann z.B. mit der Greedy-Heuristik berechnet werden. Diese bildet eine untere Schranke für P. Eine mögliche Berechnung einer oberen Schranke U wird weiter unten vorgestellt.

- **Branching.** Wir zerlegen das Problem in zwei Teilprobleme: Wir wählen Gegenstand i und wählen es für Teilproblem T_{1i} aus (d.h. $x_i = 1$) und für Teilproblem T_{i2} explizit nicht aus (d.h. $x_i = 0$). Für T_1 muss das Gewicht von Gegenstand i von der Rucksackgröße abgezogen werden und der Wert zum Gesamtwert hinzuaddiert werden. Danach streichen wir Gegenstand i aus unserer Liste.

- **Search.** Wähle eines der bisher erzeugten Teilprobleme, z.B. dasjenige mit der besten (größten) oberen Schranke, denn wir hoffen, dass wir hier die beste Lösung finden werden.

- **Bounding.** Berechne für eine dieser Teilmengen T_i je eine untere und obere Schranke L_i und U_i. Sei L der Wert der besten bisher gefundenen Lösung. Falls $U_i \leq L_i$, braucht man die Teillösungen in der Teilmenge T_i nicht weiter betrachten.

Eine obere Schranke kann z.B. mit dem folgenden Verfahren berechnet werden. Zunächst werden die Gegenstände nach ihrem Nutzen $f_i = c_i/w_i$ sortiert. Seien $g_1, g_2, \ldots, g_n$ die in dieser Reihenfolge sortierten Gegenstände. Berechne das maximale r mit $g_1 + g_2 + \cdots g_r \leq$

K. Genau diese r Gegenstände werden dann eingepackt (also $x_i = 1$ für $i = 1, \ldots, r$). Danach ist noch Platz für $K - (g_1 + \ldots + g_r)$ Einheiten an Gegenständen. Diesen freien Platz füllen wir mit $(K - (g_1 + \ldots + g_r))/g_{r+1}$ Einheiten von Gegenstand $r + 1$ auf.

Wir zeigen, dass der so berechnete Wert eine obere Schranke für die Probleminstanz darstellt. Denn die ersten r Gegenstände mit dem höchsten Nutzen pro Einheit sind im Rucksack enthalten. Und zwar jeweils so viel davon, wie möglich ist. Allerdings wird eventuell vom letzten Gegenstand $r+1$ ein nicht-gazzahliger Anteil eingepackt. Deswegen ist die generierte Lösung i.A. nicht zulässig (sie erfüllt die Ganzzahligkeitsbedingung i.A. nicht). Aber es gilt: der berechnete Lösungswert der Gegenstände im Rucksack ist mindestens so groß wie die beste Lösung.

Beispiel 7.6 zeigt die Berechnung einer oberen Schranke für die 0/1-Rucksack-Probleminstanz aus Beispiel 7.4.

Beispiel 7.6. Wir berechnen eine obere Schranke für Beispiel 7.4. Die Sortierung nach Nutzen ergibt die Reihenfolge d, e, h, f, g, b, c, a. Wir packen also die Gegenstände d und e in den Rucksack, d.h. $x_d = x_e = 1$. Danach ist noch Platz frei für 5 Einheiten, aber Gegenstand h hat leider Gewicht 9 und paßt nicht mehr ganz in den Rucksack hinein. Wir nehmen also noch $x_h = 5/9$ Einheiten dazu. Der Wert der aktuellen (nicht-zulässingen) Lösung ist $10 + 10 + 5/9(13) < 27,3$. Eine obere Schranke für die beste erreichbare Lösung der Probleminstanz ist also 27.

Die Berechnung einer oberen Schranke inmitten eines Branch-and-Bound Baumes (BB-Baumes), bei dem eine Teillösung bereits fixiert wurde, ist sehr ähnlich. Die bisher erhaltene Teillösung wird sukzessive um die noch nicht fixierten Gegenstände mit dem größten Nutzen erweitert. Dabei muss beachtet werden, dass bei jeder Fixierung eines Gegenstandes im BB-Baum die verbleibende Rucksackgröße neu berechnet wird bzw. die explizit nicht gewählten Gegenstände aus der iste gestrichen werden.

7.4.2 Branch-and-Bound für das asymmetrische TSP

Der „Großvater“ aller Branch-and-Bound Algorithmen in der kombinatorischen Optimierung ist das Verfahren von Little, Murty, Sweeny und Karel zur Lösung asymmetrischer Travelling-Salesman-Probleme, das 1963 veröffentlicht wurde.

Asymmetrisches Travelling Salesman Problem (ATSP)

Gegeben: Gerichteter vollständiger Graph $G(V, E)$ mit $N = |V|$ Knoten, die Städten entsprechen, und eine Distanzmatrix C mit $c_{ii} = +\infty$ und $c_{ij} \geq 0$.

Gesucht: Rundtour $T \subset E$ (Hamiltonscher Kreis) durch alle N Städte, die jede Stadt genau einmal besucht und minimalen Distanzwert aufweist:

$$c(T) = \sum_{(i,j)\in T} c_{ij}$$

Lösungsidee: Zeilen- und Spaltenreduktion der Matrix C und sukzessive Erzeugung von Teilproblemen beschrieben durch $P(EINS, NULL, I, J, C, L, U)$.

Dabei bedeutet:

$EINS$: Menge der bisher auf 1 fixierten Kanten
$NULL$: Menge der bisher auf 0 fixierten Kanten
I: Noch relevante Zeilenindizes von C
J: Noch relevante Spaltenindizes von C
C: Distanzmatrix des Teilproblems
L: Untere Schranke für das Teilproblem
U: Obere Schranke für das globale Problem

Vorgangsweise:

1. Das Anfangsproblem ist

   $$P(\emptyset, \emptyset, \{1, \ldots, n\}, \{1, \ldots, n\}, C, 0, \infty),$$

   wobei C die Ausgangsmatrix ist.
   Setze dieses Problem auf eine globale Liste der zu lösenden Probleme.

2. Sind alle Teilprobleme gelöst $\longrightarrow$ STOP.

   Andernfalls wähle ein ungelöstes Teilproblem

   $$P(EINS, NULL, I, J, C, L, U)$$

   aus der Problemliste.

3. **Bounding:**

   a) **Zeilenreduktion:**
   Für alle $i \in I$:

   Berechne das Minimum der i-ten Zeile von C

   $$c_{ij_0} = \min_{j\in J} c_{ij}$$

   Setze $c_{ij} = c_{ij} - c_{ij_0}\ \forall j \in J$ und $L = L + c_{ij_0}$

b) **Spaltenreduktion:**
Für alle $j \in J$:

Berechne das Minimum der j-ten Spalte von C

$$c_{i_0 j} = \min_{i \in I} c_{ij}$$

Setze $c_{ij} = c_{ij} - c_{i_0 j}\ \forall i \in I$ und $L = L + c_{i_0 j}$

c) Ist $L \geq U$, so entferne das gegenwärtige Teilproblem aus der Problemliste und gehe zu 2.

Die Schritte (d) und (e) dienen dazu, eine neue (globale) Lösung des Gesamtproblems zu finden. Dabei verwenden wir die Information der in diesem Teilproblem bereits festgesetzten Entscheidungen.

d) Definiere den „Nulldigraphen“ $G_0 = (V, A)$ mit
$A = EINS \cup \{(i,j) \in I \times J \mid c_{ij} == 0\}$

e) Versuche, mittels einer Heuristik, eine Rundtour in G_0 zu finden. Wurde keine Tour gefunden, so gehe zu 4: Branching.
Sonst hat die gefundene Tour die Länge L. In diesem Fall:

e_1) Enferne alle Teilprobleme aus der Problemliste, deren lokale, untere Schranke größer gleich L ist.

e_2) Setze in allen noch nicht gelösten Teilproblemen $U = L$.

e_3) Gehe zu 2.

4. **Branching:**

a) Wähle nach einem Plausibilitätskriterium eine Kante $(i,j) \in I \times J$, die 0 bzw. 1 gesetzt wird.

b) Definiere die neuen Teilprobleme

b_1) $P(EINS \cup \{(i,j)\}, NULL, I \backslash \{i\}, J \backslash \{j\}, C', L, U)$
wobei C' aus C durch Streichen der Zeile i und der Spalte j entsteht – von i darf nicht noch einmal hinaus- und nach j nicht noch einmal hineingelaufen werden.

b_2) $P(EINS, NULL \cup \{(i,j)\}, I, J, C'', L, U)$,
wobei C'' aus C entsteht durch $c_{ij} = \infty$.

Füge diese Teilprobleme zur Problemliste hinzu und gehe zu 2.

Bemerkungen zum Algorithmus

zu 3. Bounding:

- Hinter der Berechnung des Zeilenminimums steckt die folgende Überlegung: Jede Stadt muss in einer Tour genau einmal verlassen werden. Zeile i der Matrix C enthält jeweils die Kosten für das Verlassen von Stadt i. Und das geht auf keinen Fall billiger als durch das Zeilenminimum c_{ij_0}. Das gleiche Argument gilt für das Spaltenminimum, denn jede Stadt muss genau einmal betreten werden, und die Kosten dafür sind in Spalte j gegeben.

- Schritte (a) und (b) dienen nur dazu, eine untere Schranke zu berechnen (Bounding des Teilproblems). Offensichtlich erhält man eine untere Schranke dieses Teilproblems durch das Aufsummieren des Zeilen- und Spaltenminimums.

- Für die Berechnung einer globalen oberen Schranke können Sie statt (d) und (e) auch jede beliebige TSP-Heuristik verwenden.

zu 4. Branching:

- Ein Plausibilitätskriterium ist beispielsweise das folgende:

  Seien

  $$u(i,j) = \min\{c_{ik} \mid k \in J \setminus \{j\}\} + \min\{c_{kj} \mid k \in I \setminus \{i\}\} - c_{ij}$$

  die minimalen Zusatzkosten, die entstehen, wenn wir von i nach j gehen, ohne die Kante (i,j) zu benutzen. Wir wählen eine Kante $(i,j) \in I \times J$, so dass

  $$c_{ij} = 0 \qquad \text{und} \qquad u(i,j) = \max\{u(p,q) \mid c_{pq} = 0\}.$$

- Dahinter steckt die Überlegung: Wenn die Zusatzkosten sehr hoch sind, sollten wir lieber direkt von i nach j gehen. Damit werden „gute“ Kanten beim Enumerieren erst einmal bevorzugt. Wichtig dabei ist, dass $c_{ij} = 0$ sein muss, sonst stimmen die Berechnungen von L und U nicht mehr.

- Bei der Definition eines neuen Teilproblems kann es vorkommen, dass die dort (in $EINS$ oder $NULL$) fixierten Kanten zu keiner zulässigen Lösung mehr führen können (weil z.B. die zu $EINS$ fixierten Kanten bereits einen Kurzzyklus enthalten). Es ist sicher von Vorteil, den Algorithmus so zu erweitern, dass solche Teilprobleme erst gar nicht zu der Problemliste hinzugefügt werden (denn in diesem Zweig des Enumerationsbaumes kann sich keine zulässige Lösung mehr befinden).

Durchführung des Branch & Bound Algorithmus an einem ATSP Beispiel

Die Instanz des Problems ist gegeben durch die Distanzmatrix C:

$$C = \begin{pmatrix} \infty & 5 & 1 & 2 & 1 & 6 \\ 6 & \infty & 6 & 3 & 7 & 2 \\ 1 & 4 & \infty & 1 & 2 & 5 \\ 4 & 3 & 3 & \infty & 5 & 4 \\ 1 & 5 & 1 & 2 & \infty & 5 \\ 6 & 2 & 6 & 4 & 5 & \infty \end{pmatrix}$$

Das Ausgangsproblem:

$$P = (\emptyset, \emptyset, \{1, \ldots, 6\}, \{1, \ldots, 6\}, C, 0, +\infty)$$

Bounding:

Die Zeilenreduktion ergibt:

$$C = \begin{pmatrix} \infty & 4 & 0 & 1 & 0 & 5 \\ 4 & \infty & 4 & 1 & 5 & 0 \\ 0 & 3 & \infty & 0 & 1 & 4 \\ 1 & 0 & 0 & \infty & 2 & 1 \\ 0 & 4 & 0 & 1 & \infty & 4 \\ 4 & 0 & 4 & 2 & 3 & \infty \end{pmatrix}$$

$$L = 1 + 2 + 1 + 3 + 1 + 2 = 10$$

Die Spaltenreduktion ändert hier nichts, da bereits in jeder Spalte das Minimum 0 beträgt.

Nun wird der Nulldigraph aufgestellt (siehe Abbildung 7.3). In diesem Nulldigraphen existiert keine Tour. 6 kann nur über 2 erreicht werden und von 6 müßte wieder auf 2 gegangen werden.

Branching: Wähle Kante (2,6) und setze diese einmal gleich 1 und einmal gleich 0. Dies ergibt zwei neue Teilprobleme.

Das erste neue Teilproblem:

$$P = (\{(2,6)\}, \emptyset, \{1, 3, \ldots, 6\}, \{1, \ldots, 5\}, C', 10, +\infty)$$

$$C' = \begin{pmatrix} \infty & 4 & 0 & 1 & 0 \\ 0 & 3 & \infty & 0 & 1 \\ 1 & 0 & 0 & \infty & 2 \\ 0 & 4 & 0 & 1 & \infty \\ 4 & \infty & 4 & 2 & 3 \end{pmatrix}$$

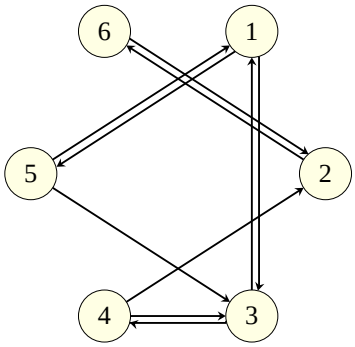


Abbildung 7.3: Der Nulldigraph in Branch & Bound

Die untere Schranke für dieses Teilproblem nach Zeilen- und Spaltenreduktion ist $L = 12$, da nur die Zeile mit Index (6) um 2 reduziert werden muss.

Das zweite neue Teilproblem:

$$P = (\emptyset, \{(2,6)\}, \{1,2,3,4,5,6\}, \{1,2,3,4,5,6\}, C'', 10, +\infty)$$

C'' entsteht aus C nach Setzen von C_{26} auf $+\infty$. Die untere Schranke für dieses Problem ist ebenfalls 12.

Abbildung 7.4 zeigt einen möglichen Enumerationsbaum für dieses Beispiel. (Dabei wird nicht das vorgestellte Plausibilitätskriterium genommen.) Hier wird als nächstes die Kante $(4,2)$ im Branching betrachtet, was Teilprobleme (4) und (5) erzeugt. Danach wird die Kante $(3,4)$ fixiert, was die Teilprobleme (6) und (7) begründet. Nach dem Fixieren der Kante $(5,1)$ wird zum ersten Mal eine globale obere Schranke von 13 gefunden (d.h. eine Tour im Nulldigraphen). Nun können alle Teilprobleme, deren untere Schranke gleich 13 ist, aus der Problemliste entfernt werden. In unserem Beispiel wird als nächstes das Teilproblem (3) gewählt, wobei dann über die Kante $(2,4)$ gebrancht wird, was Teilprobleme (9) und (10) erzeugt. Weil wir mit Teilproblem (10) niemals eine bessere Lösung als 15 erhalten können (untere Schranke), können wir diesen Teilbaum beenden. Wir machen mit Teilproblem (9) weiter und erzeugen die Probleme (11) und (12), die auch sofort beendet werden können. Es bleibt Teilproblem (7), das jedoch auch mit dem Branchen auf der Kante $(3,1)$ zu „toten“ Teilproblemen führt.

Analyse der Laufzeit. Die Laufzeit ist im Worst-Case exponentiell, denn jeder Branching-Schritt verdoppelt die Anzahl der neuen Teilprobleme. Dies ergibt im schlimmsten Fall bei N Städten eine Laufzeit von 2^N. Allerdings greift das Bounding im Allgemeinen recht gut, so dass man deutlich weniger Teilprobleme erhält. Doch bereits das Berechnen einer Instanz mit $N = 80$ Städten mit Hilfe dieses Branch-and-Bound Verfahrens dauert schon zu lange. Das vorgestellte Verfahren ist für TSPs mit bis zu 50 Städten anwendbar.

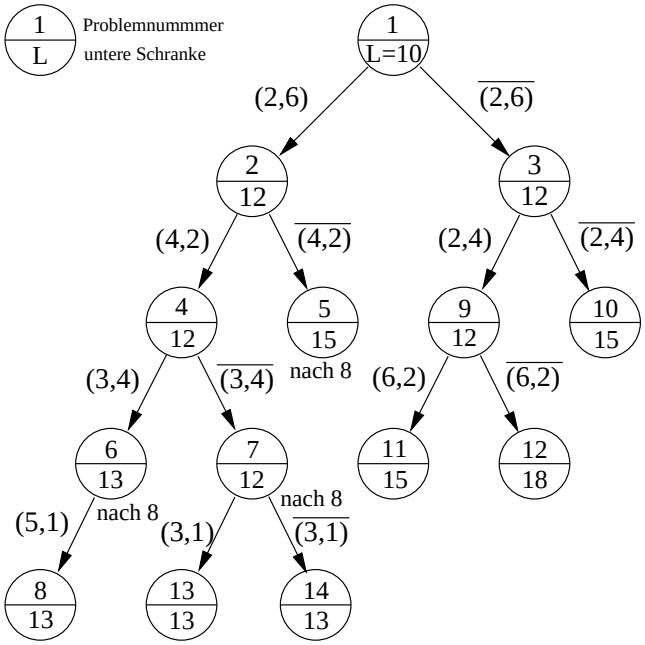


Abbildung 7.4: Möglicher Enumerationsbaum für das Beispiel

In der Praxis hat sich für die exakte Lösung von TSPs die Kombination von Branch-and-Bound Methoden mit linearer Programmierung bewährt. Solche Verfahren, die als Branch-and-Cut Verfahren bezeichnet werden, sind heute in der Lage in relativ kurzer Zeit TSPs mit ca. 500 Städten exakt zu lösen. Die größten nicht-trivialen bisher exakt gelösten TSPs wurden mit Hilfe von Branch-and-Cut Verfahren gelöst und umfassen ca. 85.900 Städte (siehe `http://www.tsp.gatech.edu`).

Typische Anwendungen von Branch-and-Bound Verfahren liegen in Brettspielen, wie z.B. Schach oder Springer-Probleme. Die dort erfolgreichsten intelligenten Branch-and-Bound Verfahren funktionieren in Kombination mit parallelen Berechnungen.

7.5 Dynamische Programmierung

Idee: Zerlege das Problem in kleinere Teilprobleme P_i ähnlich wie bei Divide & Conquer. Allerdings: während die P_i bei Divide & Conquer unabhängig sind, sind sie hier voneinander abhängig.

Dazu: Löse jedes Teilproblem und speichere das Ergebnis E_i so ab, dass E_i zur Lösung größerer Probleme verwendet werden kann.

Allgemeines Vorgehen:

1. Wie ist das Problem zerlegbar? Definiere den Wert einer optimalen Lösung rekursiv.
2. Bestimme den Wert der optimalen Lösung „bottom-up“.

Wir haben bereits einen Dynamischen Programmierungsalgorithmus kennengelernt: den Algorithmus von Floyd-Warshall für das All-Pairs-Shortest-Paths Problem. Dort wurde der Wert einer optimalen Lösung für Problem P als einfache (Minimum-) Funktion der Werte optimaler Lösungen von kleineren (bzw. eingeschränkten) Problemen ausgedrückt. Dieses Prinzip nennt man auch *Bellmansche Optimalitätsgleichung*.

Wir betrachten im Folgenden eine effiziente Dynamische Programmierung für das 0/1-Rucksackproblem.

7.5.1 Dynamische Programmierung für das 0/1-Rucksackproblem

In diesem Abschnitt behandeln wir das 0/1-Rucksackproblem bei dem alle Daten ganzzahlig sind, d.h. die Gewichte w_i und die Werte c_i der $n = N$ Gegenstände sowie die Rucksackgröße K sind ganzzahlig.

Eine Möglichkeit, das 0/1-Rucksackproblem aus Abschnitt 7.3.1 in Teilprobleme zu zerlegen, ist die folgende: Wir betrachten zunächst die Situation nach der Entscheidung über die ersten i Gegenstände. Wir werden sehen, wie man aus den guten Lösungen des eingeschränkten Problems mit i Objekten gute Lösungen des Problems mit $i + 1$ Objekten erhält.

Definition 7.7. Für ein festes $i \in \{1, \ldots, n\}$ und ein festes $W \in \{0, \ldots, K\}$ sei $R(i, W)$ das eingeschränkte Rucksackproblem mit den ersten i Objekten, deren Gewichte $w_1, \ldots, w_i$ und deren Werte $c_1, \ldots, c_i$ betragen und bei dem das Gewichtslimit W beträgt.

Wir legen eine Tabelle $T(i, W)$ an mit $i + 1$ Zeilen für $j = 0, \ldots, i$ und $W + 1$ Spalten für $j = 0, \ldots, W$, wobei an Position $T(i, W)$ folgende Werte gespeichert werden:

- $F(i, W)$: der optimale Lösungswert für Problem $R(i, W)$
- $D(i, W)$: die dazugehörige optimale Entscheidung über das i-te Objekt

Dabei setzen wir $D(i, W) = 1$, wenn es in $R(i, W)$ optimal ist, das i-te Element einzupacken, und $D(i, W) = 0$ sonst. Der Wert einer optimalen Rucksackbepackung für das Originalproblem ist dann gegeben durch $F(n, K)$.

Wir studieren nun, wie für ein gegebenes i und W der Wert $F(i, W)$ aus bereits bekannten Lösungswerten $F(i - 1, W')$ ($W' \leq W$) berechnet werden kann. Hierbei etrachten wir zwei Fälle:

Eingabe: n Gegenstände mit Gewichten $w_1, \ldots, w_n$ und Werten $c_1, \ldots, c_n$; Rucksack der Größe K
Ausgabe: Optimaler Lösungswert mit ihrem Gesamtwert und Gesamtgewicht

```
1:  var 2-dimensionales Array F[i, W] für i = 0, ..., n und W = 0, ..., K
2:  for W = 0, ..., K do
3:      F[0, W] := 0
4:  end for                                                    ▷ Initialisierung
5:  for i = 1, ..., n do
6:      for W = 0, ..., w_i − 1 do
7:          F(i, W) := F(i − 1, W)
8:          D(i, W) := 0
9:      end for
10:     for W = w_i, ..., K do
11:         if F(i − 1, W − w_i) + c_i > F(i − 1, W) then
12:             F(i, W) := F(i − 1, W − w_i) + c_i; D(i, W) := 1
13:         else F(i, W) := F(i − 1, W); D(i, W) := 0
14:         end if
15:     end for
16: end for
```

Listing 7.6: Rucksack Dynamische Programmierung

- 1. Fall: Das i-te Element wird eingepackt: In diesem Fall setzt sich die neue Lösung $F(i, W)$ aus der optimalen Lösung von $R(i-1, W-w_i)$ und dem Wert des i-ten Elements zusammen.

- 2. Fall: Das i-te Element wird nicht eingepackt: Dann erhält man den gleichen Lösungswert wie für $R(i-1, W)$, also $F(i-1, W)$.

Damit haben wir wieder eine Bellmansche Optimalitätsgleichung gegeben und können die neuen Lösungswerte effizient berechnen:

$$F(i, W) := \max\{F(i-1, W-w_i) + c_i), F(i-1, W)\}$$

Für die Berechnung setzen wir die Anfangswerte $F(0, W) := 0$ für alle $W = 0, \ldots, K$. Für kleine W passiert es oft, dass $w_i > W$ gilt; in diesem Fall paßt Gegenstand i nicht in den auf W eingeschränkten Rucksack. Listing 7.6 enthält den Pseudocode zur Dynamischen Programmierung für das Rucksackproblem.

Beispiel 7.7. Beispiel: Es sei ein Rucksack der Größe $K = 9$ gegeben und $n = 7$ Gegenstände mit folgenden Gewichten und Werten.

Gegenstand i	1	2	3	4	5	6	7
Wert c_i	6	5	8	9	6	7	3
Gewicht w_i	2	3	6	7	5	9	4

Die Dynamische Programmierungstabelle der Lösungswerte $F(i, W)$ ergibt sich wie folgt.

W/i	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	11	11	11	11	11	12	14	15
6	0	0	11	11	11	11	11	12	14	15
7	0	0	11	11	11	12	12	12	14	15

Die Korrektheit ergibt sich direkt aus der Bellmannschen Optimalitätsgleichung.

Wie können wir nun aus dem optimalen Lösungswert die Lösung selbst, d.h. die einzelnen dazugehörigen Entscheidungen berechnen? In $T[i, W]$ sind die optimalen Lösungswerte Werte $F(i, W)$ gespeichert. An der Stelle $F(n, K)$ steht also der optimale Lösungswert für unser Originalproblem. Wir starten bei $F(n, K)$. Falls $D(n, K) = 0$, dann packen wir Gegenstand n nicht ein. Wir gehen weiter zu Problem $R(n-1, K)$. Falls $D(n, K) = 1$, dann packen wir Gegenstand n ein. Damit ist das für die ersten $n-1$ Gegenstände erlaubte Gewichtslimit $K - w_n$. Gehe weiter zu Problem $R(n-1, K-w_n)$. Wiederhole dies bis $R(0, 0)$ erreicht ist.

Beispiel 7.8. Für unser Beispiel bedeutet dies, dass man von $R(7, 9)$ nach $R(4, 9)$ wandert, wo Gegenstand 4 eingepackt wurde, und von dort über $R(3, 2)$ nach $R(1, 2)$ wandert, wo Gegenstand 1 eingepackt wurde. Danach gelangt man zu $R(0, 0)$ ohne einen weiteren Gegenstand einzupacken. Damit hat man die optimale Lösung (Gegenstände 1 und 4) berechnet.

Analyse der Laufzeit. Die Laufzeit der Berechnung der Werte $F(i, W)$ und $D(i, W)$ ist $O(nK)$ für n Gegenstände und Rucksackgröße K. Die Laufzeit der Berechnung der Lösung ist $O(n)$.

Oberflächlich betrachtet sieht die Laufzeit $O(nK)$ polynomiell aus. Aber VORSICHT, dies ist nicht der Fall, denn die Rechenzeit muss auf die Länge (genauer Bitlänge) der Eingabe

bezogen werden. Wenn alle Zahlen der Eingabe nicht größer sind als 2^n und $K = 2^n$, dann ist die Länge der Eingabe $\theta(n^2)$, denn wir haben $2n+1$ Eingabezahlen mit Kodierungslänge je $\theta(n)$. Die Laufzeit ist jedoch von der Größenordnung $N2^n$ und somit exponentiell in der Inputlänge (also nicht durch ein Polynom beschränkt). Falls aber alle Zahlen der Eingabe in $O(n^2)$ sind. dann liegt die Eingabelänge im Bereich zwischen $\Omega(n)$ und $O(n \log n)$. Dann ist die Laufzeit in $O(n^3)$ und damit polynomiell.

Definition 7.8. Rechenzeiten, die exponentiell sein können, aber bei polynomiell kleinen Zahlen in der Eingabe polynomiell sind, heißen *pseudopolynomiell*.

InsbesondereAlgorithmen mit einer Laufzeit, die direkt von den Eingabewerten und nicht nur von der Anzahl der eingegebenen Objekte abhängen, sind pseudopolynomiell.

Theorem 7.5. *Das 0/1-Rucksackproblem kann mit Hilfe von Dynamischer Programmierung in pseudopolynomieller Rechenzeit gelöst werden.*

In der Praxis kann man Rucksackprobleme meist mit Hilfe von Dynamischer Programmierung effizient lösen, obwohl das Problem NP-schwierig ist, da die auftauchenden Zahlen relativ klein sind.

Das besprochene DP-Verfahren heißt *Dynamische Programmierung durch Gewichte*. Denn wir betrachten die Lösungswerte als Funktion der Restkapazitäten im Rucksack. Durch Vertauschung der Rollen von Gewicht und Wert kann auch *Dynamische Programmierung durch Werte* durchgeführt werden. Dort betrachtet man alle möglichen Lösungswerte und überlegt, wie man diese mit möglichst wenig Gewicht erreichen kann.

7.6 Verbesserungsheuristiken

Wir schließen das Kapitel Optimierung mit allgemeinen, beliebten und weit verbreiteten Heuristiken ab, die versuchen eine gegebene Lösung zu verbessern.

Die in Abschnitt 7.1 gezeigten Heuristiken werden auch als *konstruktive Heuristiken* bezeichnet, da sie neue Lösungen von Grund auf generieren. Im Gegensatz dazu gibt es auch Verfahren, die eine zulässige Lösung als Input verlangen und diese durch lokale Änderungen verbessern. Die Ausgangslösung kann hierbei eine zufällig erzeugte, gültige Lösung sein oder aber durch eine vorangestellte konstruktive Heuristik erzeugt werden.

Wir unterscheiden zwischen einfacher lokaler Suche (s. Abschnitt 7.6.1) und sogenannten Meta-Heuristiken wie Simulated Annealing (s. Abschnitt 7.6.2) und evolutionäre Algorithmen (s. Abschnitt 7.6.3). Aus Zeitgründen betrachten wir alternative Methoden wie z.B. Tabu Suche oder Ameisenverfahren nur sehr kurz (s. Abschnitt 7.6.4).

7.6.1 Einfache lokale Suche

Für die einfache lokale Suche ist die Definition der *Nachbarschaft* $N(x)$ einer Lösung x wesentlich. Das ist die Menge von Lösungen, die von einer aktuellen Lösung x aus durch eine kleine Änderung – einen sogenannten *Zug* – erreichbar sind.

Werden Lösungen beispielsweise durch Bitvektoren repräsentiert, wie das im Rucksackproblem der Fall ist, so könnte $N(x)$ die Menge aller Bitvektoren sein, die sich von x in genau einem Bit unterscheiden. Größere Nachbarschaften können definiert werden, indem man alle Lösungen, die sich in maximal r Bits von x unterscheiden, als Nachbarn betrachtet. Dabei ist r eine vorgegebene Konstante.

Etwas allgemeiner kann eine sinnvolle Nachbarschaft für ein gegebenes Problem oft wie folgt durch *Austausch* definiert werden: Entferne aus der aktuellen Lösung x bis zu r Lösungselemente. Sei S die Menge aller verbleibenden Elemente der Lösung. Die Nachbarschaft von x besteht nun aus allen zulässigen Lösungen, die S beinhalten.

Eingabe: eine Optimierungsaufgabe
Ausgabe: heuristische Lösung x
1: **var** Nachbarlösung x' zu aktueller Lösung x
2: $x =$ Ausgangslösung;
3: **repeat**
4: Wähle $x' \in N(x)$; ▷ leite eine Nachbarlösung ab
5: **if** x' besser als x **then**
6: $x = x'$;
7: **end if**
8: **until** Abbruchkriterium erfüllt

Listing 7.7: Einfache lokale Suche

Algorithmus 7.7 zeigt das Prinzip der einfachen lokalen Suche. In Zeile 3 wird aus der Nachbarschaft eine neue Lösung x' ausgewählt. Diese Auswahl kann auf folgende Art erfolgen.

Random neighbor: Es wird eine Nachbarlösung zufällig abgeleitet. Diese Variante ist die schnellste, jedoch ist die neue Lösung x' häufig schlechter als x.

Best improvement: Die Nachbarschaft $N(x)$ wird vollständig durchsucht und die beste Lösung x' wird ausgewählt. Dieser Ansatz ist am zeitaufwändigsten.

First improvement: Die Nachbarschaft $N(x)$ wird systematisch durchsucht, bis die erste Lösung, welche besser als x ist, gefunden wird bzw. alle Nachbarn betrachtet wurden.

Bei lokaler Suche mit der „random neighbor“ Auswahlstrategie sind normalerweise wesentlich mehr Iterationen notwendig, um eine gute Endlösung zu finden, als bei den beiden

anderen Strategien. Dieser Umstand wird aber oft durch den geringen Aufwand einer einzelnen Iteration ausgeglichen. Welche Strategie in der Praxis am Besten ist, ist vor allem auch vom konkreten Problem abhängig.

In den Zeilen 4 bis 6 von Algorithmus 7.7 wird x' als neue aktuelle Lösung akzeptiert, wenn sie besser ist als die bisherige.

Das Abbruchkriterium ist meist, dass in $N(x)$ keine Lösung mehr existiert, die besser als die aktuelle Lösung x ist. Eine solche Endlösung kann durch Fortsetzung der lokalen Suche nicht mehr weiter verbessert werden und wird daher auch als *lokales Optimum* in Bezug auf $N(x)$ bezeichnet. Formal gilt für eine zu minimierende Zielfunktion $f(x)$:

$$x \text{ ist ein lokales Optimum} \quad \leftrightarrow \quad \forall x' \in N(x) \mid f(x') \geq f(x)$$

Im Allgemeinen ist ein lokales Optimum aber nicht unbedingt ein globales Optimum, an dem wir eigentlich interessiert sind.

Beispiel: Zweier-Austausch für das Symmetrische TSP (2-OPT)

Es wird versucht, eine Ausgangstour iterativ zu verbessern, indem Paare von Kanten temporär entfernt werden und die so resultierenden zwei Pfade durch neue Kanten verbunden werden. Abbildung 7.5 zeigt ein Beispiel. In folgendem Algorithmus wird die Nachbarschaft $N(T)$ einer Tour T systematisch nach einer Verbesserung durchforstet. Die erste Verbesserung wird akzeptiert, was der Auswahlstrategie „first improvement“ entspricht. Im Schritt (2) werden speziell nur die Paare von in der Tour *nicht nebeneinander liegenden* Kanten ausgewählt, da das Entfernen von nebeneinander liegenden Kanten nie zu einer neuen Tour führen kann.

(1) Wähle eine beliebige Anfangstour $T = \{(i_1, i_2), (i_2, i_3), \ldots, (i_n, i_1)\}$; sei $i_{n+1} = i_1$

(2) Setze $Z = \{\{(i_p, i_{p+1}), (i_q, i_{q+1})\} \subset T \mid 1 \leq p, q \leq n \wedge p+1 < q \wedge q+1 \bmod n \neq p\}$

(3) Für alle Kantenpaare $\{(i_p, i_{p+1}), (i_q, i_{q+1})\}$ aus Z:
Falls $c_{i_p i_{p+1}} + c_{i_q i_{q+1}} > c_{i_p i_q} + c_{i_{p+1} i_{q+1}}$:
setze $T = (T \backslash \{(i_p, i_{p+1}), (i_q, i_{q+1})\}) \cup \{(i_p, i_q), (i_{p+1}, i_{q+1})\}$
gehe zu (2)

(4) T ist das Ergebnis.

Beispiel: r-Austausch für das Symmetrische TSP (r-OPT)

Bei dieser Verallgemeinerung werden nicht nur Paare von Kanten durch neue Kanten ersetzt, sondern systematisch alle r-Tupel.

(1) Wähle eine beliebige Anfangstour $T = \{(i_1, i_2), (i_2, i_3), \ldots, (i_n, i_1)\}$

(2) Sei Z die Menge aller r-elementigen Teilmengen von T

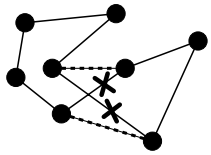


Abbildung 7.5: Prinzip eines Zweieraustausches beim symmetrischen TSP.

(3) Für alle $R \in Z$:
Setze $S = T \setminus R$ und konstruiere alle Touren, die S enthalten. Gibt es eine unter diesen, die besser als T ist, wird diese das aktuelle T und gehe zu (2).

(4) T ist das Ergebnis.

Eine Tour, die durch einen r-Austausch nicht mehr verbessert werden kann, heißt in diesem Zusammenhang r-*optimal*.

Einfache lokale Suche ist eine in der Praxis sehr beliebte Methode, um bereits vorhandene Lösungen oft deutlich und in relativ kurzer Zeit zu verbessern.

Anmerkung 7.5. Für das TSP kommt 3-Opt meist sehr nahe an die optimale Lösung (ca. 3-4%). Jedoch erhält man bereits für $N \geq 200$ Städte sehr hohe Rechenzeiten, da der Aufwand pro Iteration bei $O(N^3)$ liegt. In der Praxis hat sich für das TSP als beste Heuristik die Heuristik von Lin und Kernighan (1973) herausgestellt, die oft bis zu 1-2% nahe an die Optimallösung herankommt. Bei dieser Heuristik werden generierte Kandidatenlösungen analysiert und r wird aus der aktuellen Situation heraus dynamisch gewählt. Ein solcher Ansatz wird allgemein auch *variable Tiefensuche* genannt.

7.6.2 Simulated Annealing

Einfachen lokale Sucheverfahren wie 2-Opt ist es oft nicht möglich, mitunter schlechten lokalen Optima zu „entkommen“. Eine Vergrößerung der Nachbarschaft kann dieses Problem manchmal lösen, es steigt aber der Aufwand meist allzu stark mit der Problemgröße an. *Simulated Annealing* ist eine allgemeine Erweiterung der einfachen lokalen Suche (eine *Meta-Heuristik*), die es bei skalierbarem Zeitaufwand erlaubt, lokalen Optima grundsätzlich auch zu entkommen, ohne dass die Nachbarschaft vergrößert werden muss.

Abgeleitet wurde dieses Verfahren von dem physikalischen Prozess, ein Metall in einen Zustand möglichst geringer innerer Energie zu bringen, in dem die Teilchen möglichst strukturiert angeordnet sind. Dabei wird das Metall erhitzt und sehr langsam abgekühlt. Die Teilchen verändern anfangs ihre Positionen und damit ihre Energieniveaus sehr stark, mit sinkender Temperatur werden die Bewegungen von lokal niedrigen Energieniveaus weg aber immer geringer.

Eingabe: eine Optimierungsaufgabe
Ausgabe: heuristische Lösung x

```
1:  var Zeit t; aktuelle Temperatur T; Ausgangstemperatur T_init; Nachbarlösung x'
2:  t = 0;
3:  T = T_init;
4:  x = Ausgangslösung;
5:  repeat
6:      Wähle x' ∈ N(x) zufällig;                    ▷ leite eine Nachbarlösung ab
7:      if x' besser als x then
8:          x = x';
9:      else
10:         if Z < e^{-|f(x')-f(x)|/T} then
11:             x = x';
12:         end if
13:         T = g(T, t);
14:         t = t + 1;
15:     end if
16: until Abbruchkriterium erfüllt
```

Listing 7.8: Simulated Annealing

Algorithmus 7.8 zeigt das von Kirkpatrick et al. (1983) vorgeschlagene Simulated Annealing. Z ist hierbei eine Zufallszahl $\in [0, 1)$, $f(x) > 0$ die Bewertung der Lösung x.

Die Ausgangslösung kann wiederum entweder zufällig oder mit einer Konstruktionsheuristik generiert werden. In jeder Iteration wird dann ausgehend von der aktuellen Lösung x eine Nachbarlösung x' durch eine kleine zufällige Änderung abgeleitet, was der Auswahlstrategie „random neighbor“ entspricht. Ist x' besser als x, so wird x' in jedem Fall als neue aktuelle Lösung akzeptiert. Ansonsten, wenn also x' eine schlechtere Lösung darstellt, wird diese mit einer Wahrscheinlichkeit $p_{accept} = e^{-|f(x')-f(x)|/T}$ akzeptiert. T ist dabei die aktuelle „Temperatur“. Dadurch ist die Möglichkeit gegeben, lokalen Optima zu entkommen. Diese Art, Nachbarlösungen zu akzeptieren wird in der Literatur auch *Metropolis-Kriterium* genannt.

Die Wahrscheinlichkeit p_{accept} hängt von der Differenz der Bewertungen von x und x' ab – nur geringfügig schlechtere Lösungen werden mit höherer Wahrscheinlichkeit akzeptiert als viel schlechtere. Außerdem spielt die Temperatur T eine Rolle, die über die Zeit hinweg kleiner wird: Anfangs werden Änderungen mit größerer Wahrscheinlichkeit erlaubt als später.

Wie T initialisiert und in Abhängigkeit der Zeit t vermindert wird, beschreibt das *Cooling-Schema*.

Geometrisches Cooling: Für T_{init} wird z.B. $f_{\max} - f_{\min}$ gewählt. Sind $f_{\max}$ bzw. $f_{\min}$ nicht bekannt, so werden Schranken bzw. Schätzungen hierfür verwendet. Als Cooling-Funktion wird z.B. gewählt $g(T, t) = T \cdot \alpha, \quad \alpha < 1$ (z.B. 0,999).

Adaptives Cooling: Es wird der Anteil der Verbesserungen an den letzten erzeugten Lösungen gemessen und auf Grund dessen T stärker oder schwächer reduziert.

Als Abbruchkriterium sind unterschiedliche Bedingungen vorstellbar:
Ablauf einer bestimmten Zeit; eine gefundene Lösung ist „gut genug“; oder keine Verbesserung in den letzten k Iterationen.

Damit Simulated Annealing grundsätzlich funktioniert und Chancen bestehen, das globale Optimum zu finden, müssen folgende zwei Bedingungen erfüllt sein.

- **Erreichbarkeit eines Optimums**: Ausgehend von jeder möglichen Lösung muss eine optimale Lösung durch wiederholte Anwendung der Nachbarschaftsfunktion grundsätzlich mit einer Wahrscheinlichkeit größer Null erreichbar sein.
- **Lokalitätsbedingung:** Eine Nachbarlösung muss aus ihrer Ausgangslösung durch eine „kleine“ Änderung abgeleitet werden können. Dieser kleine Unterschied muss im Allgemeinen (d.h. mit Ausnahmen) auch eine kleine Änderung der Bewertung bewirken. Sind diese Voraussetzungen erfüllt, spricht man von *hoher Lokalität* – eine effiziente Optimierung ist möglich. Haben eine Ausgangslösung und ihre abgeleitete Nachbarlösung im Allgemeinen große Unterschiede in der Bewertung, ist die Lokalität schwach. Die Optimierung ähnelt dann der Suche „einer Nadel im Heuhaufen“ bzw. einer reinen Zufallssuche und ist nicht effizient.

Wir betrachten ein Beispiel für sinnvolle Operatoren, um eine Nachbarlösung abzuleiten.

Beispiel: Symmetrisches TSP. *Operator Inversion:* Ähnlich wie im 2-Opt werden zwei nicht benachbarte Kanten einer aktuellen Tour zufällig ausgewählt und entfernt. Die so verbleibenden zwei Pfade werden mit zwei neuen Kanten zu einer neuen Tour zusammengefügt.

Anmerkung 7.6. Dass dieser Operator keinerlei Problemwissen wie etwa Kantenkosten ausnutzt, ist einerseits eine Schwäche; andererseits ist das Verfahren aber so auch für schwierigere Varianten des TSPs, wie dem *blinden TSP*, einsetzbar. Bei dieser Variante sind die Kosten einer Tour nicht einfach die Summe fixer Kantenkosten, sondern im Allgemeinen eine nicht näher bekannte oder komplizierte, oft nicht-lineare Funktion. Beispielsweise können die Kosten einer Verbindung davon abhängen, *wann* diese verwendet wird. (Wenn Sie mit einem PKW einerseits in der Stoßzeit, andererseits bei Nacht durch die Stadt fahren, wissen Sie, was gemeint ist.)

7.6.3 Evolutionäre Algorithmen

Unter dem Begriff *evolutionäre Algorithmen* werden eine Reihe von Meta-Heuristiken (genetische Algorithmen, Evolutionsstrategien, Evolutionary Programming, Genetic Programming, etc.) zusammengefasst, die Grundprinzipien der natürlichen Evolution in einfacher

Weise nachahmen. Konkret sind diese Mechanismen vor allem die *Selektion* (natürliche Auslese, „survival of the fittest“), die *Rekombination* (Kreuzung) und die *Mutation* (kleine, zufällige Änderungen).

Ein wesentlicher Unterschied zu Simulated Annealing ist, dass nun nicht mehr mit nur einer aktuellen Lösung, sondern einer ganze Menge (= *Population*) gearbeitet wird. Durch diese größere *Vielfalt* ist die Suche nach einer optimalen Lösung „breiter“ und robuster, d.h. die Chance lokalen Optima zu entkommen ist größer.

Algorithmus 7.9 zeigt das Schema eines evolutionären Algorithmus.

Eingabe: eine Optimierungsaufgabe
Ausgabe: beste gefundene Lösung
1: **var** selektierte Eltern Q_s; Zwischenlösungen Q_r
2: P = Menge von Ausgangslösungen;
3: bewerte(P);
4: **repeat**
5: Q_s = Selektion(P);
6: Q_r = Rekombination(Q_s);
7: P = Mutation(Q_r);
8: bewerte(P);
9: **until** Abbruchkriterium erfüllt

Listing 7.9: Grundprinzip eines evolutionären Algorithmus

Ausgangslösungen können wiederum entweder zufällig oder mit einfachen Erzeugungsheuristiken generiert werden. Wichtig ist jedoch, dass sich diese Lösungen unterscheiden und so Vielfalt gegeben ist.

Selektion. Die Selektion kopiert aus der aktuellen Population P Lösungen, die in den weiteren Schritten neue Nachkommen produzieren werden. Dabei werden grundsätzlich bessere Lösungen mit höherer Wahrscheinlichkeit gewählt. Schlechtere Lösungen haben aber im Allgemeinen auch Chancen, selektiert zu werden, damit lokalen Optima entkommen werden kann.

Eine häufig eingesetzte Variante ist die **Tournament Selektion**, die wie folgt funktioniert:

(1) Wähle aus der Population k Lösungen gleichverteilt zufällig aus (Mehrfachauswahl ist üblicherweise erlaubt).

(2) Die beste der k Lösungen ist die selektierte.

Rekombination. Die Aufgabe der Rekombination ist, aus zwei selektierten Elternlösungen eine neue Lösung zu generieren. Vergleichbar mit der bereits beim Simulated Annealing

beschriebenen Lokalitätsbedingung ist hier wichtig, dass die neue Lösung möglichst ausschließlich aus Merkmalen der Eltern aufgebaut wird.

Mutation. Die Mutation entspricht der Nachbarschaftsfunktion beim Simulated-Annealing. Sie dient meist dazu, neue oder verlorengegangene Merkmale in die Population hineinzubringen.

Häufig werden die Rekombination und Mutation nicht immer, sondern nur mit einer bestimmten Wahrscheinlichkeit ausgeführt, sodass vor allem gute Lösungen manchmal auch unverändert in die Nachfolgegeneration übernommen werden.

Wir sehen uns ein konkretes Beispiel an.

Beispiel: Symmetrisches TSP. *Edge-Recombination* (ERX): Aus den Kanten zweier Elterntouren T^1 und T^2 soll eine neue Tour T erstellt werden, die möglichst nur aus Kanten der Eltern aufgebaut ist. Algorithmus 7.10 zeigt ein mögliches Vorgehen.

Eingabe: Zwei gültige Touren T^1 und T^2
Ausgabe: Neue abgeleitete Tour T

```
1:  var aktueller Knoten v; Nachfolgeknoten w; Kandidatenmenge für Nachfolgeknoten W
2:  beginne bei einem beliebigen Startknoten v = v_0; T = {};
3:  while es noch unbesuchte Knoten gibt do
4:      Sei W die Menge der noch unbesuchten Knoten, die in T^1 ∪ T^2 adjazent zu v sind;
5:      if W ≠ {} then
6:          wähle einen Nachfolgeknoten w ∈ W zufällig aus;
7:      else
8:          wähle einen zufälligen noch nicht besuchten Nachfolgeknoten w;
9:      end if
10:     T = T ∪ {(v, w)}; v = w;
11: end while
12: schließe die Tour: T = T ∪ {(v, v_0)};
```

Listing 7.10: Edge-Recombination(T^1, T^2)

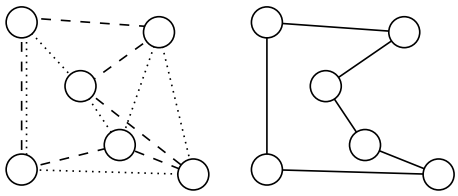


Abbildung 7.6: Beispiel zur Edge-Recombination.

Abb. 7.6 zeigt links zwei übereinandergelegte Elterntouren und rechts eine mögliche, durch Edge-Recombination neu erzeugte Tour, die nur aus Kanten der Eltern besteht.

Im Algorithmus werden neue Kanten, die nicht von den Eltern stammen, nur dann zu T hinzugefügt, wenn W leer ist („edge-fault“). Um die Wahrscheinlichkeit, dass es zu diesen Situationen kommt, möglichst gering zu halten, kann die Auswahl in Schritt (5) wie folgt verbessert werden: Ermittle für jeden Knoten $w_i \in W$ die Anzahl der gültigen Knoten, die von diesem dann weiter unter Verwendung von Elternkanten angelaufen werden können, d.h. $a_i = |\{u \mid (\{w_i, u\} \in T^1 \cup T^2 \wedge u \text{ noch nicht besucht in } T\}|$.
Wähle ein $w = w_i$ für das a_i minimal ist.

In einer anderen Variante der Edge-Recombination werden die Kantenlängen als lokale, problem-spezifische Heuristik mitberücksichtigt: In Schritt 5 wird entweder immer oder mit höherer Wahrscheinlichkeit die Kante kürzester Länge ausgewählt. Dies führt einerseits zu einer rascheren Konvergenz des evolutionären Algorithmus zu guten Lösungen, kann aber auch ein vorzeitiges „Hängenbleiben“ bei weniger guten, lokal-optimalen Touren bewirken.

Anmerkung 7.7. Abschließend sei zu evolutionären Algorithmen noch angemerkt, dass sie mit anderen Heuristiken und lokalen Verbesserungstechniken sehr effektiv kombiniert werden können. So ist es möglich

- Ausgangslösungen mit anderen Heuristiken zu erzeugen,
- in der Rekombination und Mutation Heuristiken einzusetzen (z.B. können beim TSP kostengünstige Kanten mit höherer Wahrscheinlichkeit verwendet werden), und
- alle (oder einige) erzeugte Lösungen mit einem anderen Verfahren (z.B. 2-Opt) noch lokal weiter zu verbessern.

Auch können die Vorteile paralleler Hardware gut genutzt werden.

7.6.4 Alternative Heuristiken

Es gibt noch einige andere Klassen von sogenannten Meta-Heuristiken:

- Eine beliebte und in vielen Fällen auch sehr erfolgreiche Alternative zu Simulated Annealing stellt die 1986 von Glover vorgestellte *Tabu-Suche* dar. Auch sie ist eine Erweiterung der einfachen lokalen Suche, die darauf abzielt, lokalen Optima grundsätzlich entkommen zu können, um ein globales Optimum zu finden. Die Tabu-Suche verwendet ein Gedächtnis über den bisherigen Optimierungsverlauf und nutzt dieses, um bereits erforschte Gebiete des Suchraums nicht gleich nochmals zu betrachten.
- Ameisen-Verfahren (*Ant Colony, ACO*) wurden ursprünglich für Wegeprobleme (z.B. TSP) entwickelt und basieren auf der Idee der Pheromon-Spuren, die Ameisen auf ihren Wegen hinterlassen. Die Idee dabei ist, dass Ameisen, die einen kurzen Weg zur Futterquelle nehmen, diesen öfter hin-und-her laufen, und dort also stärkere Pheromonspuren hinterlassen, wovon andere Ameisen angezogen werden. Allerdings sind ACO Verfahren meist sehr langsam.

- Sintflut-Verfahren entsprichtt der Idee von Simulated Annealing, allerdings sorgt hier der steigende Wasserpegel (bei SA entspricht dies der Temperatur) dafür, dass schlechtere Nachbarlösungen mit zunehmender Laufzeit des Verfahrens seltener akzeptiert werden. D.h. i.W. ist der Zufall Z von Simulated Annealing hier ausgeschaltet.

- Für Schlagzeilen sorgte vor einiger Zeit der Artikel über DNA-Computing, bei dem ein kleines TSP ”im Reagenzglas” mit linear vielen Schritten gelöst wurde. Inzwischen ist die Euphorie abgeklungen, denn statt exponentiell viel Zeit benötigt dieser Ansatz exponentiell viel Platz. Um z.B ein 100-Städte TSP damit zu lösen, müßte ein nichtrealisierbares riesengroßes Reagenzglas gebaut werden...

Weiterführende Literatur

- C.H. Papadimitriou und K. Steiglitz: „Combinatorial Optimization: Algorithms and Complexity“, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982

- W.J. Cook, W.H. Cunningham, W.R. Pulleyblank und A. Schrijver: „Combinatorial Optimization“, John Wiley & Sons, Chichester, 1998

- E. Aarts und J.K. Lenstra: „Local Search in Combinatorial Optimization“, John Wiley & Sons, Chichester, 1997

- Z. Michalewicz: „Genetic Algorithms + Data Structures = Evolution Programs“, Springer, 1996