Algorithmen und Datenstrukturen 1

Skriptum zur gleichnamigen
Vorlesung mit Übungen von
Univ.-Prof. Günther Raidl und Univ.-Ass. Bin Hu

im Sommersemester 2012

VERSION 4.00
INSTITUT FÜR COMPUTERGRAPHIK UND ALGORITHMEN
TECHNISCHE UNIVERSITÄT WIEN

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 werden, ist sehr hoch.

Egal in welchem Gebiet Sie sich vertiefen werden, ob im Banken- oder Finanzwesen, Kryptographie, Computergraphik, Software-Entwicklung oder in Datenbanken: Sie werden immer wieder auf die Grundbegriffe stoßen, die Sie in dieser Lehrveranstaltung lernen.

Die Vorlesung mit Übung **Algorithmen und Datenstrukturen 1** ist im Studienplan der Bakkalaureatsstudien der Informatik, dem Bakkalaureat Wirtschaftsinformatik, sowie Geodäsie und Geoinformatik angesetzt und behandelt den Entwurf und die Analyse von Algorithmen. Fortgesetzt wird diese Lehrveranstaltung mit der VU Algorithmen und Datenstrukturen 2, die weiterführende Themen – unter anderem aus dem Bereich der Optimierung – behandelt.

Bisher haben Sie gelernt, wie man grundsätzlich programmiert. Sie sollten in der Lage sein, eine an Sie herangetragene Aufgabenstellung in ein lauffähiges Programm umzusetzen.

Mit dem Basiswissen aus Algorithmen und Datenstrukturen werden Sie auch in der Lage sein, **gut** und **effizient** zu programmieren. Was dies genau bedeutet, werden Sie im Laufe der Vorlesung erfahren.

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

> Angenommen Sie haben nach Studienende eine Stellung als ProjektleiterIn in der Software-Branche inne. Sie erhalten die Aufgabe, die Datenbank einer Telekom-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 Algorithmen und Datenstrukturen verstanden hat, schreibt hingegen ein Programm, das für jede Abfrage nur 1 Sekunde benötigt, also 120 Mal so schnell ist. Das ist klarerweise schlecht für Sie und Ihre Zukunft in der Firma.

akzeptiert werden. Da wir Sie im Zuge der LVA-Abwicklung via E-Mail erreichen können müssen, vergewissern Sie sich bitte, dass Ihre *zur Zeit der Anmeldung* in TISS eingetragene E-Mailadresse gültig ist, oder aber – wenn Sie keine Adresse in TISS hinterlegen möchten – stellen Sie sicher, dass Sie E-Mails an Ihre generische Studentenadresse der TU Wien (`e#######@student.tuwien.ac.at`) fristgerecht empfangen und lesen.

Der Übungsteil setzt sich wie folgt zusammen:

- Eingangstest
- 4 Übungseinheiten mit Anwesenheitspflicht
- 2 schriftlich auszuarbeitende Beispielblätter
- 1 Programmieraufgabe
- 2 Übungstests und 1 Nachtragstest

Genauere Informationen dazu finden Sie in den nächsten Abschnitten und auf der Webseite zur LVA.

Eingangstest

Nach der Anmeldung zur Lehrveranstaltung ist für rund eine Woche auf der TU Wien E-Learning Plattform TUWEL (`http://tuwel.tuwien.ac.at`) ein elektronischer Eingangstest freigeschaltet. Die erfolgreiche Absolvierung dieses Eingangstests ist zwingend erforderlich, andernfalls wird Ihre Anmeldung für eine der Übungsgruppen widerrufen.

Dieser Eingangstest dient dazu, Ihnen unter Umständen vorhandene Schwächen in einzelnen Teilbereichen aufzuzeigen, die für diese Lehrveranstaltung absolut notwendige Grundlagen darstellen. Beispielsweise gibt es dabei Fragen zur O-Notation (erste VO-Einheiten) aber auch zu Rechenregeln für Potenzen und Logarithmen.

Der elektronische Eingangstest besteht aus rund 10 Multiple Choice Fragen, die Sie großteils (mindestens 80%) korrekt beantworten müssen, um den Test erfolgreich abzuschließen. Solange der Test freigeschaltet ist, können Sie diesen beliebig oft wiederholen, wobei das beste von Ihnen erreichte Resultat gewertet wird.

Beachten Sie bitte, dass Sie mit dem positiven Abschluss des Eingangstests definitiv für diese LVA angemeldet sind und eine prüfungsrelevante Leistung erbracht haben. Sie erhalten daher ab diesem Zeitpunkt auf jeden Fall ein (ohne weitere Leistungen negatives) Zeugnis für diese Lehrveranstaltung.

Übungsstunden

Vier Mal im Semester treffen sich Kleingruppen zu jeweils 55 Minuten langen **Übungsstunden**. Sie müssen sich bei der **Anmeldung** zu dieser Lehrveranstaltung auf eine bestimmte Übungsgruppe festlegen und können während des gesamten Semesters nur an den Treffen genau dieser Gruppe teilnehmen; eine Teilnahme an anderen Übungsstunden ist ausnahmslos unzulässig.

Die **Übungsaufgaben** für eine Übungsstunde werden rund zwei Wochen vor dem jeweiligen Übungstermin auf der Webseite der Lehrveranstaltung zum Herunterladen angeboten. Die Übungsaufgaben sind aber auf Wunsch auch in unserem Sekretariat in ausgedruckter Form erhältlich.

Für jedes Übungsblatt geben Sie über TUWEL bis zu einem für alle Übungsgruppen gleichen Termin an, welche der aktuellen Übungsbeispiele Sie gelöst haben. Der Leiter bzw. die Leiterin Ihrer Übungsgruppe wählt aufgrund dieser Liste TeilnehmerInnen aus, die ihre Lösungen dann an der Tafel präsentieren. Die Anzahl der von Ihnen angekreuzten Beispiele und Ihre Leistungen bei diesen Präsentationen fließen in Ihre Beurteilung ein.

Bitte beachten Sie, dass aufgrund der zeitlichen Einschränkungen möglicherweise nicht immer die Gelegenheit besteht, alle Beispiele in den Kleingruppen ausführlich zu bearbeiten.

Beispielblätter

Neben den Übungsblättern, die für die jeweiligen Übungsstunden zu lösen sind, gibt es zwei Beispielblätter, die Sie schriftlich ausarbeiten und elektronisch via TUWEL abgeben sollen. Ihre Lösungen werden begutachtet und Sie bekommen eine Rückmeldung, die Ihnen primär als Hilfe für die Vorbereitung auf die Übungstests dienen soll. Die Anzahl der von Ihnen richtig gelösten Beispiele fließt ebenfalls in die Benotung ein.

Programmieraufgabe

Weiters ist für einen positiven Abschluss dieser Lehrveranstaltung eine **Programmieraufgabe** in Java zu bewältigen.

Die Angabe für diese Programmieraufgabe steht rechtzeitig auf der Webseite der LVA zum Download bereit. Dort finden Sie auch ein Codegerüst des Programms, das Ihnen Routinearbeiten wie die Ein- und Ausgabe abnimmt. Sie müssen konkrete Implementationen von Algorithmen in das zur Verfügung gestellte Gerüst einfügen. Auf der Webseite haben Sie weiters auch Zugriff auf Testdaten und Dokumentationen.

Für die Programmieraufgabe gibt es einen Abgabetermin, bis zu dem das jeweilige Programm funktionstüchtig über ein automatisiertes **Abgabesystem in TUWEL** eingereicht

werden muss. Ihr Programm wird automatisch überprüft, wobei das Ergebnis der Tests danach im Abgabesystem abrufbar ist. Sollte Ihr Programm die internen Tests des Abgabesystems nicht vollständig bestehen, so steht Ihnen eine festgelegte Anzahl an Versuchen zur Verfügung, um ein voll funktionstüchtiges Programm abzugeben. Die letzte abgegebene Programmversion wird zur Bewertung herangezogen. Testen Sie daher Ihre Programme vor der Abgabe ausführlich!

In den Tagen nach der Programmabgabe finden persönliche **Abgabegespräche** im Informatiklabor (Favoritenstraße 11, Erdgeschoß, `http://www.inflab.tuwien.ac.at`) statt, wo Sie Ihren Lösungsweg im Detail präsentieren. Für einen konkreten Termin zu Ihrem Abgabegespräch melden Sie sich über TUWEL an. Diese Gespräche dauern rund 10 Minuten.

Für die Programmieraufgabe können Sie eine bestimmte Punkteanzahl erreichen, die in die Gesamtbeurteilung einfließt. Die korrekte Funktionsweise des abgegebenen Programms ist dabei maßgeblich, in die Bewertung fließen aber auch Eigenständigkeit, Effizienz und Eleganz der Lösung ein.

Übungstests

Zusätzlich zu Übungsgruppen, Übungsbeispielen und Programmieraufgabe gibt es **zwei Übungstests** und **einen Nachtragstest**. Wo und zu welcher genauen Uhrzeit diese stattfinden, wird jeweils in der Woche vor dem jeweiligen Test separat auf der Webseite zur Lehrveranstaltung angekündigt.

Zur Absolvierung dieser LVA benötigen Sie lediglich zwei Testresultate, Sie können also durchaus einen Termin versäumen, aber auch an allen drei Tests teilnehmen. In letzterem Fall verfällt das schlechteste der drei Testresultate – es gehen dann also nur Ihre beiden besten Ergebnisse in die Beurteilung ein. Um die Lehrveranstaltung positiv zu beenden, müssen Sie auf die beiden gewerteten Tests zusammen mehr als die Hälfte der erreichbaren Punkte erhalten.

Für alle drei Übungstests ist zwingend eine Anmeldung über TISS notwendig!

Stoff jedes Übungstests ist der gesamte bis dahin in der Übung inklusive den dazugehörigen Vorlesungseinheiten durchgenommene Lehrinhalt. Schriftliche Unterlagen und elektronische Hilfsmittel sind nicht erlaubt. Vom Schwierigkeitsgrad und der Art der Fragestellung entsprechen die Testbeispiele den Übungsbeispielen; es lohnt sich also spätestens bei den Tests, wenn Sie die Übungsblätter wirklich eigenständig ausgearbeitet haben.

Positiver Abschluss

Zusammenfassend hier nochmals die Kriterien, um diese LVA positiv zu absolvieren:

1. Den Eingangstest mit mindestens 8 Punkten bestehen,
2. insgesamt mehr als 50 Punkte auf die zwei besten Übungstests erhalten,
3. mindestens 28 Punkte auf die Übungsbeispiele in den Übungsgruppen und die Beispielblätter erhalten,
4. mindestens einen Punkt auf die Programmieraufgabe erhalten,
5. mindestens die Hälfte der bei dieser LVA möglichen Punkteanzahl erreichen.

Details entnehmen Sie bitte der LVA-Homepage (siehe unten)!

Weitere Informationen, Fragen, Probleme

Weitere, auch aktuelle Informationen und die genauen Termine finden Sie auf unseren Webseiten:

- Unsere Homepage:
  `http://www.ads.tuwien.ac.at/`
- VU Algorithmen und Datenstrukturen 1:
  `http://www.ads.tuwien.ac.at/teaching/lva/186813.html`

Bei Fragen oder Problemen wenden Sie sich bitte

- via E-Mail-Hotline an uns
  `algodat1-ss12@ads.tuwien.ac.at`
- kommen Sie in unsere Sprechstunden
  (Termine siehe `http://www.ads.tuwien.ac.at/w/Staff`)
- oder vereinbaren Sie einen individuellen Gesprächstermin.

Literaturliste

Die Vorlesung hält sich teilweise eng an die Ausführungen in (1). Auch (2) ist ein sehr schönes, gut lesbares Buch zur Algorithmik; (3) erklärt den Stoff meist besonders nachvollziehbar, einfach und ausführlich. Weiters sind die Bücher (4), (5) und (6) empfehlenswert. Für den Teil zur Optimierung empfehlen wir insbesondere (7). Speziellere weitere Literaturhinweise in diesem Bereich finden Sie am Ende der entsprechenden Kapitel. Darüber hinaus steht es Ihnen natürlich frei, jedes andere geeignete Buch zum behandelten Stoff auszuwählen.

(1) T. Ottmann und P. Widmayer: „Algorithmen und Datenstrukturen", 5. Auflage, Spektrum Akademischer Verlag, 2012

(2) R. Sedgewick: „Algorithmen in Java, Teil 1–4", 3. Auflage, Pearson Education, 2003

(3) R. Sedgewick: „Algorithms in Java, Part 5", Pearson Education, 2004

(4) T.H. Cormen, C.E. Leiserson und R.L. Rivest: „Introduction to Algorithms", MIT Press, 1990

(5) A.V. Aho, J.E. Hopcroft und J.D. Ullman: „Data Structures and Algorithms", Addison-Wesley, 1987

(6) R. Johnsonbaugh und M. Schaefer: „Algorithms", Pearson Education, 2004

(7) M. T. Goodrich und R. Tamassia: „Data Structures & Algorithms in Java", 4. Auflage, John Wiley & Sons, 2006

(8) G. Saake und K.-U. Sattler: „Algorithmen und Datenstrukturen – Eine Einführung mit Java", 3. Auflage, dPunkt Verlag, 2006

Dankesworte

An dieser Stelle sei allen gedankt, die einen Beitrag zu diesem Skriptum geleistet haben: Teile dieses Skriptums wurden aus einer Vorlage entnommen, die von Constantin Hellweg, Universität zu Köln, erstellt wurde. Verantwortlich für den Inhalt sind Univ.-Prof. Dr. Petra Mutzel und Univ.-Prof. Dr. Günther Raidl sowie die Univ.-AssistentInnen Bin Hu, Christian Schauer, Marian Rainer-Harbach, Emir Causevic, Martin Gruber, Matthias Prandtstetter, Andreas Chwatal, Gunnar Klau, Gabriele Koller, Ivana Ljubic, Martin Schönhacker und René Weiskircher, die StudienassistentInnen Thorsten Krenek, Anna Potocka, Raul Fechete und Georg Kraml. Des Weiteren haben Anmerkungen von Univ.-Prof. Dr. Thomas Eiter zur Verbesserung des Skriptums entscheidend beigetragen. Auch danken wir unseren StudentInnen für wertvolles Feedback.

Kapitel 1

Einführung in Algorithmen und Datenstrukturen

In diesem Kapitel klären wir zunächst wichtige Begriffe wie *Algorithmus, Datenstruktur und Programm*. Um gute Algorithmen entwerfen zu können, ist es auch sehr wichtig, bestehende Algorithmen zu analysieren (siehe Abschnitt 1.2). Um die Algorithmen zu spezifizieren, verwenden wir in dieser Vorlesung einen sogenannten *Pseudocode*, der in Abschnitt 1.3 eingeführt wird.

1.1 Begriffe

Nach *Brockhaus* ist ein *Algorithmus* ein Rechenverfahren, das in genau festgelegten Schritten vorgeht.

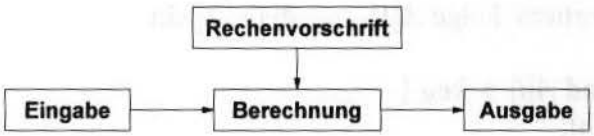


Abbildung 1.1: Definition des Begriffs Algorithmus.

Ein Algorithmus besitzt eine Eingabe, eine Ausgabe sowie eine Berechnung, die durch eine Rechenvorschrift vorgegeben ist (siehe Abb. 1.1). Weiterhin fordern wir für einen Algorithmus, dass seine Beschreibung endliche Größe hat und er in endlich vielen Schritten terminiert.

Nach *Wirth* sind *Programme* letztlich konkrete Formulierungen abstrakter Algorithmen, die sich auf bestimmte Darstellungen wie Datenstrukturen stützen. *Programmerstellung und Datenstrukturierung* sind untrennbare ineinandergreifende Themen.

Beispiel Sortierproblem

Eingabe: Eine Folge von n Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$.
Ausgabe: Eine Permutation (Umordnung) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ der Eingabefolge, so dass

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

gilt.

Jede konkrete Zahlenfolge ist eine *Instanz* des Sortierproblems, z.B. soll $\langle 32, 25, 13, 48, 39 \rangle$ in $\langle 13, 25, 32, 39, 48 \rangle$ überführt werden. Generell gilt: Eine Instanz eines Problems besteht aus allen Eingabewerten, die zur Lösung benötigt werden.

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

Ein möglicher, einfacher Algorithmus zur Lösung des Sortierproblems geht von dem Gedanken aus, dass man eine sortierte Teilfolge hat, in die weitere Elemente sukzessive eingeordnet werden. Dabei müssen Elemente am oberen Ende der sortierten Teilfolge „Platz machen" und jeweils die richtige Stelle in der Folge zur Aufnahme des neuen Elementes vorbereiten. Im Folgenden sehen wir diesen Algorithmus in Pseudocode[1]:

Algorithmus 1 Insertion-Sort (**var** A)

Eingabe: zu sortierende Folge in Feld A (d.h. in $A[1], \ldots, A[n]$)
Ausgabe: sortierte Folge in Feld A
Variable(n): Zahl key; Indizes i, j

```
1: für j = 2, ..., n {
2:     key = A[j];
3:     // füge A[j] in sortierte Folge A[1], ..., A[j − 1] ein
4:     i = j − 1;
5:     solange i > 0 und A[i] > key {
6:         A[i + 1] = A[i];
7:         i = i − 1;
8:     }
9:     A[i + 1] = key;
10: }
```

Der Ablauf des Algorithmus bei Eingabe der Folge $\langle 5, 2, 4, 6, 1, 3 \rangle$ ist in Abbildung 1.2 beschrieben.

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

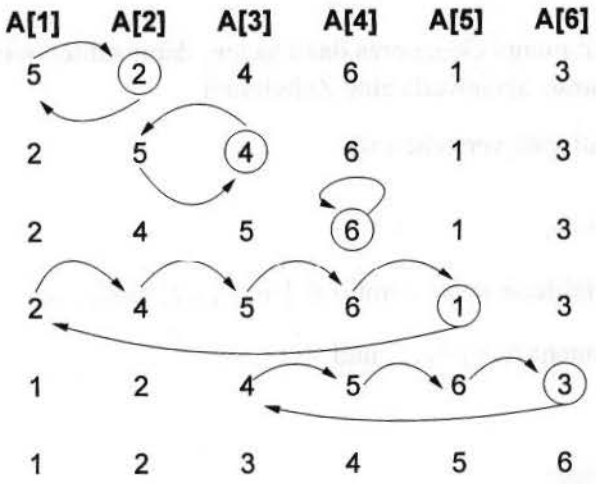


Abbildung 1.2: Illustration von Insertion-Sort.

1.2 Analyse von Algorithmen

Die Analyse von Algorithmen ist sehr eng mit dem Algorithmenentwurf verbunden. Hier geht es darum festzustellen, wie gut bzw. effizient ein vorliegender Algorithmus ist.

Es gibt verschiedene Maße für die Effizienz von Algorithmen:

- Speicherplatz
- Laufzeit
- Besondere charakterisierende Parameter (problemabhängig), das sind z.B. beim Sortieren:
  - die Anzahl der Vergleichsoperationen sowie
  - die Anzahl der Bewegungen der Datensätze.

Wir sind im Folgenden fast immer an der Laufzeit eines Algorithmus interessiert. Dazu müssen wir die Maschine festlegen, auf der wir rechnen. Wir betrachten hier eine *RAM: „random access machine“* mit folgenden Eigenschaften:

- Es gibt genau einen Prozessor.
- Alle Daten liegen im Hauptspeicher.
- Die Speicherzugriffe dauern alle gleich lang.

Wenn wir im Folgenden nichts Genaueres dazu sagen, dann zählen wir die *primitiven Operationen* eines Algorithmus als jeweils eine Zeiteinheit.

Unter primitiven Operationen verstehen wir

- Zuweisungen: $a = b$,
- arithmetische Befehle: $a = b \circ c$ mit $\circ \in \{+, -, \cdot, /, mod, \ldots\}$,
- logische Operationen: $\wedge, \vee, \neg, \ldots$ und
- Sprungbefehle:

  „**gehe zu** *label*"

  „**falls** $a \diamond b$ ...; **sonst** ...;" mit $\diamond \in \{<, \leq, ==, \neq, \geq, >\}$.

Wir vernachlässigen also

- die Indexrechnungen,
- den Typ der Operanden und
- die Länge der Operanden.

Die *Laufzeit eines Algorithmus* legen wir als die Anzahl der von ihm ausgeführten primitiven Operationen fest. Es ist klar, dass das Sortieren von 1000 Zahlen länger als das Sortieren von 10 Zahlen dauert. Deshalb beschreiben wir die Laufzeit als *Funktion der Eingabegröße*. Beim Sortieren einer n-elementigen Folge ist diese Eingabegröße n. Bei einigen Sortieralgorithmen geht das Sortieren bei gleich langen Folgen für „fast schon sortierte" Folgen schneller als für nicht sortierte Folgen. Man muss also auch bei Eingaben der gleichen Länge verschiedene Fälle unterscheiden.

Wir analysieren nun die Laufzeit unseres Algorithmus Insertion-Sort. Dazu weisen wir jeder Zeile des Algorithmus eine Laufzeit c_i zu und bestimmen, wie oft der Befehl in der Zeile ausgeführt wird.

Zeile	Kosten	Wie oft?
1	c_1	n
2	c_2	$n-1$
4	c_4	$n-1$
5	c_5	$\sum_{j=2}^{n} t_j$
6	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	c_7	$\sum_{j=2}^{n} (t_j - 1)$
9	c_9	$n-1$

t_j ist dabei die Anzahl der Durchführungen der Zeile (5) („**solange**"-Schleifenabfrage) für j.

Daraus ergibt sich eine Gesamtlaufzeit von:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_9(n-1).$$

$T(n)$ wird auch als *Laufzeitfunktion in* n bezeichnet.

„Best-Case"-Analyse

Die Best-Case-Analyse misst die kürzeste mögliche Laufzeit über alle möglichen Eingabe-Instanzen. Die kürzeste Laufzeit hat der Algorithmus Insertion-Sort, falls die Eingabefolge bereits sortiert ist. In Zeile (5) ist dann immer $A[i] \leq key$. Also gilt hier

$$t_j = 1 \text{ für } j = 2, 3, \ldots n.$$

Daraus folgt eine Laufzeit von

$$\begin{aligned} T_{\text{best}}(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_9)n - (c_2 + c_4 + c_5 + c_9) \\ &= an + b \text{ für Konstanten } a \text{ und } b, \end{aligned}$$

also eine lineare Funktion in n.

„Worst-Case"-Analyse

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

Der Worst-Case tritt bei Insertion-Sort ein, wenn die Eingabefolge in umgekehrter Reihenfolge sortiert ist. Hier gilt

$$t_j = j \text{ für } j = 2, 3, \ldots, n.$$

Weil

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

und

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

folgt

$$\begin{aligned} T_{\text{worst}}(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) \\ &\quad + c_7\left(\frac{n(n-1)}{2}\right) + c_9(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_9\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_9) \\ &= an^2 + bn + c \text{ für Konstanten } a, b \text{ und } c, \end{aligned}$$

also eine quadratische Funktion in n.

„Average-Case"-Analyse

Die *Average-Case-Analyse* misst die durchschnittliche Laufzeit über alle Instanzen der Größe n (einer gegebenen Instanzklasse). Der Average-Case kann manchmal erheblich besser, manchmal aber auch genauso schlecht wie der Worst-Case sein. In unserem Beispiel führt z.B. $t_j = \frac{j}{2}$ auch zu einer quadratischen Laufzeitfunktion $T_{\text{avg}}(n)$. Das Problem beim Average-Case ist oft die Berechnung der durchschnittlichen Laufzeit.

Notationen: Θ, O und Ω

Da die ausführliche Laufzeitfunktion oft zu umfangreich ist, vereinfacht man hier, indem man nur die *Ordnung der Laufzeit* betrachtet. Die Ordnung der Funktion $T(n) = an^2+bn+c$ ist z.B. n^2. Die Ordnung der Laufzeitfunktion $T(n) = an + b$ ist n. Man schreibt:

$$\begin{aligned} an^2 + bn + c &= \Theta(n^2) \\ an + b &= \Theta(n) \end{aligned}$$

Eine exakte Definition dieser Schreibweise werden wir noch einführen.

Motivation für die Einführung der Θ-Notation

Wir nehmen an, dass F, G, und H jeweils einen Algorithmus entwickelt haben, die die Laufzeitfunktion $f(n), g(n)$ bzw. $h(n)$ besitzen, wobei $f(n) = n + 3$, $g(n) = 2n + 2$ und $h(n) = n^2$ gilt.

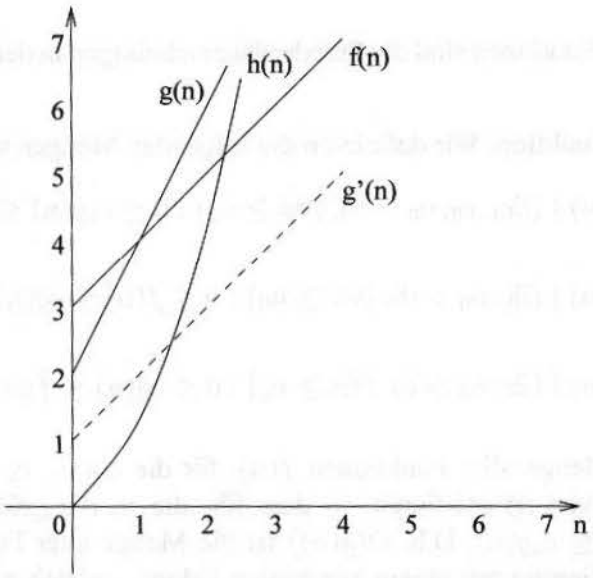


Abbildung 1.3: Laufzeitfunktion $f(n), g(n), h(n)$ und $g'(n)$.

1. Welcher Algorithmus ist der schnellste?
   Intuitiv halten wir $f(n)$ für schneller als $h(n)$, obwohl für kleine Eingabegrößen n die Situation umgekehrt ist (siehe Abb. 1.3). Unsere Intuition täuscht uns aber nicht, denn für uns ist die Laufzeitfunktion für große n von Interesse – kleine n sind vernachlässigbar. Wir sagen also (und daran hält sich auch die formale Definition):
   „$f(n)$ *wächst langsamer als* $h(n)$" wenn $f(n) \leq h(n)$ für alle n ab einem gewissen n_0 gilt. Wir interessieren uns also für das *asymptotische Wachstum* (für $n \to \infty$).

2. G kauft sich einen schnelleren Computer; die Laufzeit ist jetzt nur noch halb so lang: $g'(n) = \frac{1}{2}(g(n)) = n + 1$.
   Vorher dachten wir intuitiv: „$f(n)$ *wächst langsamer als* $g(n)$";
   Jetzt hingegen meinen wir: „$g'(n)$ *wächst langsamer als* $f(n)$".
   Eine Skalierung mit einem konstanten Faktor ändert nichts an dem asymptotischem Wachstum. Darum sagen wir:
   „$f(n)$ *wächst ungefähr gleich wie* $g(n)$" wenn $c_1 g(n) \leq f(n) \leq c_2 g(n)$ für zwei Konstanten c_1 und c_2 gilt.

Zusammenfassung:

$f(n)$ und $g(n)$ besitzen die gleiche Wachstumsrate, wenn sie für große Eingabegrößen nur durch Skalierung voneinander abweichen.

Die folgenden wichtigen Definitionen der Θ-, *O- und* Ω*-Notationen* formalisieren unsere

Überlegungen. Diese Notationen sind die Standardbezeichnungen in der Informatik für Laufzeitanalysen.

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

$$\Theta(g(n)) = \{f(n) \mid (\exists c_1, c_2, n_0 > 0), (\forall n \geq n_0) : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$$O(g(n)) = \{f(n) \mid (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq f(n) \leq cg(n)\}$$

$$\Omega(g(n)) = \{f(n) \mid (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq cg(n) \leq f(n)\}$$

$\Theta(g(n))$ 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, die größer als n_0 sind, gilt: $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$. D.h. $\Theta(g(n))$ ist die Menge aller Funktionen, die asymptotisch – bis auf Skalierung mit einem konstanten Faktor – gleich großes Wachstum wie $g(n)$ besitzen.

$O(g(n))$ ist die Menge aller Funktionen $f(n)$, für die ein c und ein $n_0 > 0$ existieren, so dass für alle $n \geq n_0$ gilt, dass $0 \leq f(n) \leq cg(n)$ ist, d.h. $f(n)$ ist durch $cg(n)$ von oben beschränkt (für große n und eine Konstante c, die unabhängig von n ist). Man stellt sich dabei vor, dass n immer weiter (bis in das Unendliche) wächst.

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

Schreibweise:

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

Wir benutzen die

O-Notation für die *oberen Schranken*,

Ω-Notation für die *unteren Schranken* und die

Θ-Notation für die „genaue" Wachstumsrate einer Laufzeitfunktion.

Bisher haben wir die Größenordnung einer Laufzeitfunktion nur intuitiv abgeleitet, wie z.B.

$$\frac{1}{2}n^2 - 3n = \Theta(n^2),$$

jetzt ist dies formal beweisbar. Einen solchen formalen Beweis führen wir jetzt für dieses Beispiel durch.

Beispiel 1:

Wir müssen also ein c_1, c_2 und n_0 finden, so dass

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \text{ für alle } n \geq n_0.$$

Mit Division durch $n^2 > 0$ erhalten wir

$$c_1 \overset{(1)}{\leq} \frac{1}{2} - \frac{3}{n} \overset{(2)}{\leq} c_2.$$

(1) gilt für alle $n \geq 7$, wenn $c_1 \leq \frac{1}{14}$.

(2) gilt für alle $n \geq 1$, wenn $c_2 \geq \frac{1}{2}$.

Also wählen wir z.B.

$$c_1 = \frac{1}{14}, c_2 = \frac{1}{2} \text{ und } n_0 = 7.$$

Damit ist gezeigt, dass $\frac{1}{2}n^2 - 3n$ in $\Theta(n^2)$ ist. Für einen Beweis ist es also sehr wichtig und hilfreich, diese Konstanten c_1, c_2 und n_0 auch anzugeben. Natürlich müssen Sie diese Konstanten nicht durch das exakte Lösen von Gleichungssystemen ausrechnen. Grobe Abschätzungen genügen bereits. Beispielsweise können Sie für dieses Beispiel auch $c_1 = \frac{1}{4}$, $c_2 = 1$ und $n_0 = 100$ angeben.

Beispiel 2:

Eine lineare Funktion ist in $\Theta(n)$, d.h. wir beweisen nun, dass

$$f(n) = an + b = \Theta(n)$$

für $a, b \in \mathbb{N}_0$.

Beweis: Zu zeigen ist $\exists c_1, c_2, n_0 > 0$, so dass $\forall n \geq n_0$ gilt: $0 \leq c_1 n \leq an + b \leq c_2 n$

Division durch n ergibt:

$$0 \leq c_1 \overset{(1)}{\leq} a + \frac{b}{n} \overset{(2)}{\leq} c_2$$

1. erfüllt für $c_1 \leq a$, da $\frac{b}{n} \geq 0$
2. erfüllt für $a + 1 \leq c_2$, für alle $n \geq b$; denn dann gilt $\frac{b}{n} \leq 1$

$\Longrightarrow$ Wähle also z.B.:

$$c_1 = a, c_2 = a + 1, n_0 \geq b$$

Ganz ähnlich kann man zeigen, dass gilt

$$an^2 + bn + c = \Theta(n^2).$$

Beispiel 3:

In einem dritten Beispiel zeigen wir, dass z.B. $6n^3$ nicht in $\Theta(n^2)$ ist.

Annahme: $\exists c_2, n_0$, so dass $\forall n \geq n_0$ gilt: $6n^3 \leq c_2 n^2$

Daraus folgt: $\exists c_2, n_0$, so dass $\forall n \geq n_0$ gilt: $n \leq \frac{c_2}{6}$, was ein Widerspruch ist, da n immer weiter wächst und c_2 nur eine Konstante unabhängig von n ist.

Notationen spielen eine *wichtige Rolle* für die Analyse von Algorithmen; wir werden sie immer wieder benötigen.

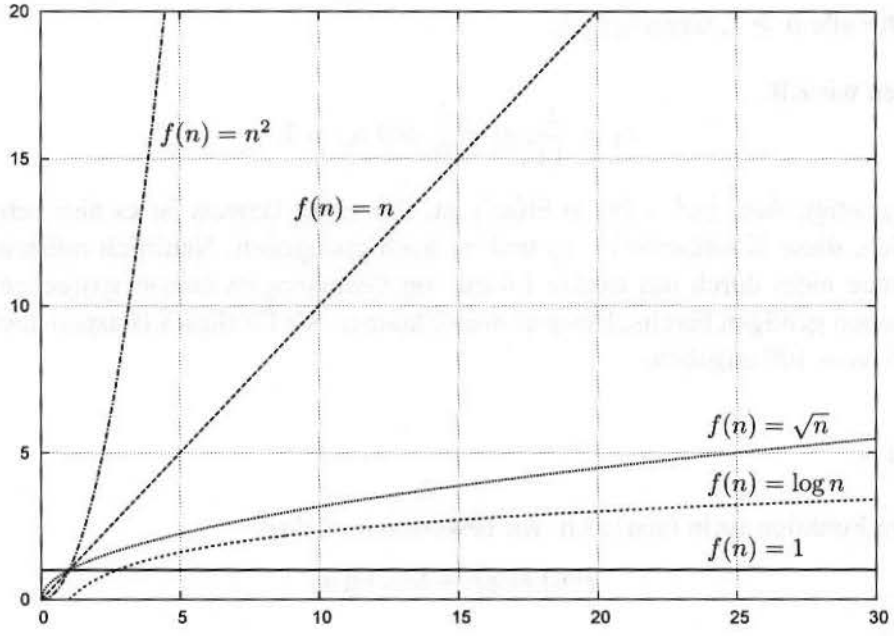


Abbildung 1.4: Vergleich verschiedener Laufzeitfunktionen.

Abbildung 1.4 veranschaulicht das Verhältnis verschiedener Laufzeitfunktionen zueinander. Es gilt: $\Theta(1) < \Theta(\log n) < \Theta(\sqrt{n}) < \Theta(n) < \Theta(n \cdot \log n) < \Theta(n^2) < \Theta(2^n) < \Theta(n!) < \Theta(n^n)$.

Beziehungen am Beispiel „Insertion-Sort"

- Die *Worst-Case* Laufzeit $\Theta(n^2)$ folgt sofort aus der Laufzeitfunktion (s. oben). Daraus folgt eine Laufzeit von $O(n^2)$ für beliebige Eingaben.
- Von der *Worst-Case* Laufzeit $\Theta(n^2)$ kann *nicht* auf eine Laufzeit $\Theta(n^2)$ für beliebige Eingabe gefolgert werden. Im Beispiel gilt eine Laufzeit von $\Theta(n)$, falls die Eingabe bereits sortiert ist.

- Von der *Best-Case* Laufzeit $\Theta(n)$ kann auf eine Laufzeit $\Omega(n)$ für beliebige Eingaben gefolgert werden.

Interpretation von Gleichungen

Manchmal sieht man Gleichungen, in denen 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: $f(n) = 3n + 1$).

Ein anderes Beispiel ist $2n^2 + \Theta(n) = \Theta(n^2)$, das wie folgt zu interpretieren ist.

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

In der Literatur wird außerdem die Schreibweise $\Omega(f(n))$ auch für „unendlich oft $\geq c \cdot f(n)$" verwendet.

1.3 Spezifikation eines Algorithmus

Ein Algorithmus kann auf verschiedene Art und Weise spezifiziert werden,

- als Text (Deutsch, Englisch, Pseudocode,...)
- als Computerprogramm
- als Hardwaredesign

In der Vorlesung werden wir die Algorithmen in Pseudocode formulieren.

Eine Alternative wäre eine existierende Programmiersprache. Diese enthält jedoch oft notwendige Zeilen und Befehle, die vom eigentlichen Algorithmus „ablenken". Pseudocode erhält man aus einem in Programmiersprache geschriebenen Programm, indem man alles weglässt, was „unmittelbar klar" ist und sich nur auf das Wesentliche beschränkt. Auch die Variablendeklarationen können unter Umständen weggelassen werden. Die Merkregel lautet, man kann bis zu 3-5 Programmierzeilen zu einer Zeile in Pseudocode zusammenfassen, wenn die genaue Umsetzung dieser Zeile in ein Programm eindeutig ist. Ein Beispiel hierfür haben wir schon gesehen: Algorithmus 1 auf Seite 16.

Kapitel 2

Sortieren

Untersuchungen zeigen, dass mehr als ein Viertel der kommerziell benutzten Rechenzeit für das Sortieren verschiedenster Daten verwendet wird. Daher wurden in den vergangenen Jahren große Anstrengungen unternommen, um möglichst effiziente Sortieralgorithmen zu entwickeln.

Viele dieser Algorithmen folgen allgemeinen Design-Prinzipien für Algorithmen und sind so auch für viele andere Probleme einsetzbar. Das ist der Grund, warum sich ein großer Teil dieser Vorlesung den Sortierverfahren widmet.

Zunächst werden wir das Sortierproblem formal 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 „$\leq$" erklärt ist.

Gesucht: Permutation

$$\pi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$$

der Zahlen von 1 bis n, so dass die Umordnung der Sätze gemäß π die Schlüssel in aufsteigende Reihenfolge bringt:

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \ldots \leq k_{\pi(n)}.$$

Internes Sortieren: Alle Daten sind im Hauptspeicher.

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

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

Laufzeitmessung: Statt alle primitiven Operationen zu zählen, interessieren wir uns hier jeweils für die besonderen Charakteristika:

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

Wir analysieren die Algorithmen jeweils im besten Fall, im schlechtesten Fall und im Durchschnittsfall über alle $n!$ möglichen Anfangsanordnungen.

Die Laufzeitfunktionen für die Schlüsselvergleiche in den jeweiligen Fällen sind $C_{\text{best}}(n)$, $C_{\text{worst}}(n)$ und $C_{\text{avg}}(n)$ (C steht für *Comparisons*).

Für die Bewegungen sind es dementsprechend $M_{\text{best}}(n)$, $M_{\text{worst}}(n)$ und $M_{\text{avg}}(n)$ (M steht für *Movements*).

Sind die Informationsfelder der Datensätze sehr groß, dann bedeutet dies, dass jedes Kopieren der Daten (d.h. jede Bewegung) sehr aufwändig und zeitintensiv ist. Deshalb erscheint es manchmal günstiger, einen Algorithmus mit mehr Schlüsselvergleichen aber weniger Datenbewegungen einem anderen Algorithmus mit weniger Schlüsselvergleichen aber mehr Datenbewegungen vorzuziehen.

Im Folgenden werden die wichtigsten Sortieralgorithmen vorgestellt.

2.1 Sortieren durch Einfügen (*Insertion-Sort*)

Dieser Algorithmus wurde bereits in den Abschnitten 1.1 bis 1.3 behandelt und auch der Aufwand berechnet. Wir holen daher an dieser Stelle nur noch die Aufteilung dieses Aufwands in Schlüsselvergleiche und Datenbewegungen nach.

Der einzige Schlüsselvergleich findet sich in Zeile 5 von Algorithmus 1. Aus der Analyse in Abschnitt 1.2 entnehmen wir, dass diese Zeile (dort durch die Konstante c_5 repräsentiert) im Best-Case $\Theta(n)$ mal durchlaufen wird, während sie im Worst-Case $\Theta(n^2)$ mal bearbeitet werden muss. Daher gilt:

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

Datenbewegungen gibt es in den Zeilen 2, 6 und 9 von Algorithmus 1, wobei aber Zeile 6 für die Analyse am interessantesten ist, weil sie sich in der inneren Schleife befindet. Diese Zeilen werden in Abschnitt 1.2 durch die Konstanten c_2, c_6 und c_8 repräsentiert. Damit ergibt sich wie zuvor aus den dort berechneten Laufzeitfunktionen:

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

Die Analyse des durchschnittlichen Falles (mit $t_j = \frac{j}{2}$) ergibt weiters $C_{\text{avg}}(n) = \Theta(n^2)$ und $M_{\text{avg}}(n) = \Theta(n^2)$.

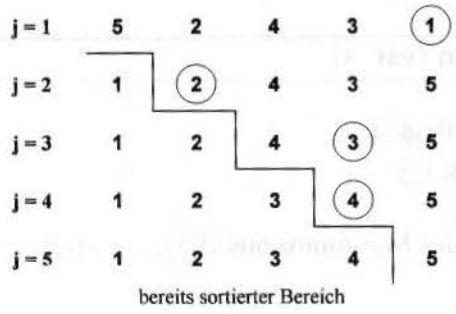


Abbildung 2.1: Beispiel für Selection Sort.

2.2 Sortieren durch Auswahl (*Selection-Sort*)

Selection-Sort ist ein einfacher, elementarer Algorithmus zum Sortieren. Genauso wie Insertion-Sort geht er davon aus, dass links der aktuellen Position j bereits alles 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.

Methode: Bestimme die Position $i_1 \in \{1, 2, \ldots, n\}$, an der das Element mit minimalem Schlüssel auftritt, vertausche $A[1]$ mit $A[i_1]$; bestimme $i_2 \in \{2, 3, \ldots, n\}$, vertausche $A[2]$ mit $A[i_2]$; usw.

Abbildung 2.1 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 wurde.

Pseudocode des Algorithmus: In Algorithmus 2 findet sich der Pseudocode des Verfahrens Selection-Sort. In vielen Implementierungen wird die Abfrage in Zeile 9 weggelassen und einfach immer die Vertauschung durchgeführt. Was in der Praxis effizienter ist, hängt vom relativen Aufwand des Vergleichs bzw. der Vertauschung ab.

Analyse von Selection-Sort

- Wir zählen die Schlüsselvergleiche, das sind die Vergleiche in Zeile 5:

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

Algorithmus 2 Selection-Sort (**var** A)

Eingabe: Folge in Feld A
Ausgabe: sortierte Folge in Feld A
Variable(n): Indizes minpos, i, j

```
1: für j = 1, 2, ..., n − 1 {
2:    // Bestimme Position des Minimums aus A[j], ..., A[n]
3:    minpos = j;
4:    für i = j + 1, ..., n {
5:       falls A[i].key < A[minpos].key dann {
6:          minpos = i;
7:       }
8:    }
9:    falls minpos > j dann {
10:      Vertausche A[minpos] mit A[j];
11:   }
12: }
```

- Die Datenbewegungen finden in Zeile 10 statt. Wird Zeile 9 weggelassen, so gilt:

$$M_{\text{best}}(n) = M_{\text{worst}}(n) = M_{\text{avg}}(n) = 3 \cdot (n-1) = \Theta(n)$$

  Bei vorhandener Zeile 9 gilt:

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

- Die Anzahl der Zuweisungen in Zeile 6 hängt von der Eingabeinstanz ab:
  - *Best-Case*: Die Zuweisung in Zeile 6 wird im besten Fall nie ausgeführt: 0
  - *Worst-Case*: Die Zuweisung wird immer ausgeführt: $\Theta(n^2)$
  - *Average-Case*: Dieser Fall ist schwer zu analysieren; das ersparen wir uns.

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

Einsatz von *Selection-Sort*, falls

- die Bewegungen von Datensätzen teuer sind, und die
- Vergleiche zwischen den Schlüsseln billig.

2.3 Sortieren durch Verschmelzen (*Merge-Sort*)

Merge-Sort ist einer der ältesten für den Computer entwickelten Sortieralgorithmen und wurde erstmals 1945 durch John von Neumann vorgestellt.

Idee:

- **Teile:** Teile die n-elementige Folge in der Mitte in zwei Teilfolgen.
- **Herrsche:** Sortiere beide Teilfolgen rekursiv mit *Merge-Sort*.
- **Kombiniere:** Verschmelze die beiden Teilfolgen zu einer sortierten Gesamtfolge.

$\rightarrow$ Für einelementige Teilfolgen ist nichts zu tun.

Merge-Sort folgt einem wichtigen Entwurfsprinzip für Algorithmen, das als *„Teile und Herrsche"* (*„divide and conquer"*) bezeichnet wird:

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

Abbildung 2.2 illustriert die Idee von Merge-Sort.

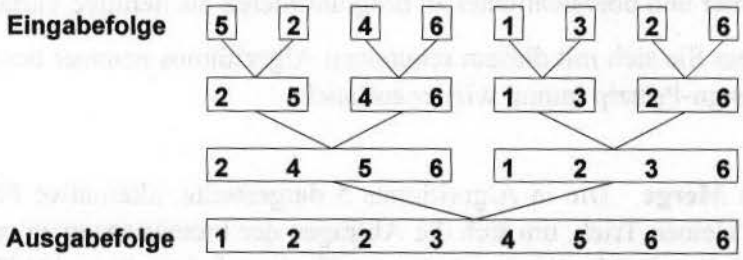


Abbildung 2.2: Illustration von Merge-Sort.

Im Folgenden gilt:

- A: Feld, in dem die Datensätze gespeichert sind.
- l, r, m: Indizes mit $l \leq m \leq r$.

- *Merge* (A, l, m, r) verschmilzt die beiden Teilarrays $A[l, \ldots, m]$ und $A[m+1, \ldots, r]$ die bereits sortiert sind. Das Resultat ist ein sortiertes Array $A[l, \ldots, r]$. Die Laufzeit von *Merge* beträgt $\Theta(r - l + 1)$.
- *Merge-Sort* (A, l, r) sortiert $A[l, \ldots, r]$ (falls $l \geq r$ gilt, so ist nichts zu tun).
- Das Sortieren eines n-elementigen Arrays geschieht durch *Merge-Sort* $(A, 1, n)$.

Pseudocode des Algorithmus: In Algorithmus 3 findet sich die rekursive Implementierung von *Merge-Sort*. Wird der Algorithmus für eine leere oder einelementige Teilfolge aufgerufen, so terminiert die Rekursion. Algorithmus 4 zeigt den zugehörigen Algorithmus *Merge*.

Algorithmus 3 Merge-Sort (**var** A, l, r)

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

```
1: falls l < r dann {
2:     m = ⌊(l + r)/2⌋;
3:     Merge-Sort (A, l, m);
4:     Merge-Sort (A, m + 1, r);
5:     Merge (A, l, m, r);
6: }
```

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, dass Sie sich mit diesem rekursiven Algorithmus genauer beschäftigen, weil gerade dieses Design-Prinzip immer wieder auftaucht.

Pseudocode von Merge Die in Algorithmus 5 dargestellte, alternative Prozedur Merge verwendet einen kleinen Trick, um sich die Abfragen der Indexgrenzen zu ersparen. Sie ist damit schneller als jene in Algorithmus 4. Achten Sie darauf, dass die rechte Hälfte der Folge von $m + 1$ bis r in umgekehrter Reihenfolge nach B kopiert wird.

Analyse von Merge-Sort

Worst-Case

Teile: Bestimmung der Mitte des Teilvorgangs in konstanter Zeit: $\Theta(c)$, wobei c konstant.
Herrsche: Lösen zweier Teilprobleme der Größe $\lceil \frac{n}{2} \rceil$ bzw. $\lfloor \frac{n}{2} \rfloor$: $T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right)$.

Algorithmus 4 Merge (**var** A, l, m, r)

Eingabe: sortierte Teilfolgen $A[l], \ldots, A[m]$ und $A[m+1], \ldots, A[r]$
Ausgabe: verschmolzene, sortierte Teilfolge in $A[l], \ldots, A[r]$
Variable(n): Hilfsfolge B; Indizes i, j, k, h

```
1:  i = l; // läuft durch A[l] bis A[m] der ersten Teilfolge
2:  j = m + 1; // läuft durch A[m + 1] bis A[r] der zweiten Teilfolge
3:  k = l; // das nächste Element der Resultatfolge ist B[k]
4:  solange i ≤ m und j ≤ r {
5:     falls A[i].key ≤ A[j].key dann {
6:        B[k] = A[i];   i = i + 1;    // übernimm A[i] nach B[k]
7:     } sonst {
8:        B[k] = A[j];   j = j + 1;    // übernimm A[j] nach B[k]
9:     }
10:    k = k + 1
11: }
12: falls i > m dann {
13:    // erste Teilfolge erschöpft; übernimm zweite
14:    für h = j, ..., r {
15:       B[k + h − j] = A[h];
16:    }
17: } sonst {
18:    // zweite Teilfolge erschöpft; übernimm erste
19:    für h = i, ..., m {
20:       B[k + h − i] = A[h];
21:    }
22: }
23: // speichere sortierte Folge von B zurück nach A
24: für h = l, ..., r {
25:    A[h] = B[h];
26: }
```

Algorithmus 5 Merge (**var** A, l, m, r)

Eingabe: sortierte Teilfolgen $A[l], \ldots, A[m]$ und $A[m+1], \ldots, A[r]$
Ausgabe: verschmolzene, sortierte Teilfolge in $A[l], \ldots, A[r]$
Variable(n): Hilfsfolge B; Index i

```
1:  B[l,...,m] = A[l,...,m];
2:  B[m+1,...,r] = A[r,...,m+1];
3:  p = l;   q = r;
4:  für i = l,...,r {
5:     falls B[p].key ≤ B[q].key dann {
6:        A[i] = B[p];   p = p + 1;
7:     } sonst {
8:        A[i] = B[q];   q = q − 1;
9:     }
10: }
```

Kombiniere: $\Theta(n)$.
Insgesamt: Rekursionsgleichung der Laufzeitfunktion:

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

d.h. die gesamte Laufzeit zum Sortieren einer Instanz der Größe n setzt sich zusammen als die Summe der Zeiten zum Sortieren der linken und rechten Hälfte und der Zeit für „Merge".

Die Lösung dieser Rekursionsgleichung lautet $T(n) = \Theta(n \log_2 n)$. Das werden wir nun beweisen. Fast alle Logarithmen, die in der Vorlesung auftauchen, haben Basis zwei.

Beweis: Wir zeigen $T(n) = O(n \log_2 n)$. Es folgt aus der Definition der O-Notation: 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: Für $n \geq 3$ und $c = 3a$ gilt $\boxed{T(n) \leq c\,n \log_2(n-1)}$

Induktionsanfang:

$$\begin{aligned} n = 3: \quad T(3) &\leq T(1) + T(2) + 3a \\ &\leq 3T(1) + 5a \\ &\leq 8a = \tfrac{8}{3}c \\ &\leq 3c = c\,\underbrace{3 \log_2(3-1)}_{=1} \end{aligned} \qquad \begin{aligned} n = 4: \quad T(4) &\leq T(2) + T(2) + 4a \\ &\leq 4T(1) + 8a \\ &\leq 12a = 4c \\ &\leq c\,\underbrace{4 \log_2(4-1)}_{>1} \end{aligned}$$

$$
\begin{aligned}
n=5: \quad T(5) &\leq T(2)+T(3)+5a \\
&\leq 2T(1)+2a+T(2)+T(1)+3a+5a \\
&\leq 2T(1)+2a+2T(1)+2a+T(1)+3a+5a \\
&= 5T(1)+12a \\
&\leq 17a = \tfrac{17}{3}c \\
&\leq 10c = c\,5\log_2(5-1)
\end{aligned}
$$

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

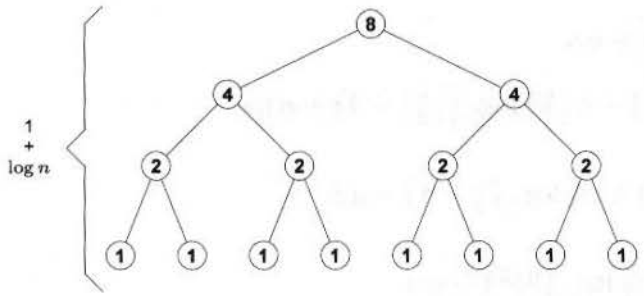
$$
\begin{aligned}
T(n) &\leq T\left(\lceil \tfrac{n}{2} \rceil\right) + T\left(\lfloor \tfrac{n}{2} \rfloor\right) + a\,n \\
&\overset{(I.V.)}{\leq} c\lceil \tfrac{n}{2} \rceil \log_2\left(\lceil \tfrac{n}{2} \rceil - 1\right) + c\lfloor \tfrac{n}{2} \rfloor \log_2\left(\lfloor \tfrac{n}{2} \rfloor - 1\right) + a\,n \\
&\leq c\tfrac{n+1}{2}\log_2\left(\tfrac{n+1}{2}-1\right) + c\tfrac{n}{2}\log_2\left(\tfrac{n}{2}-1\right) + a\,n \\
&\leq c\tfrac{n+1}{2}\log_2\left(\tfrac{n-1}{2}\right) + c\tfrac{n}{2}\log_2\left(\tfrac{n-2}{2}\right) + a\,n \\
&\overset{(*)}{=} \tfrac{1}{2}c(n+1)\log_2(n-1) + \tfrac{1}{2}c\,n\log_2(n-2) - c\tfrac{n+1}{2} - c\tfrac{n}{2} + a\,n \\
&\leq \tfrac{1}{2}c\,n\log_2(n-1) + \tfrac{1}{2}c\log_2(n-1) + \tfrac{1}{2}c\,n\log_2(n-1) - c\,n \\
&\quad -\tfrac{c}{2} + a\,n \\
&= c\,n\log_2(n-1) + \tfrac{c}{2}\log_2(n-1) - c\,n - \tfrac{c}{2} + \tfrac{c}{3}n \\
&\leq c\,n\log_2(n-1) - \tfrac{c}{2}\underbrace{(2n+1-\log_2(n-1))}_{(\text{für } n \geq 2)} + \tfrac{c}{2}n \\
&\leq c\,n\log_2(n-1) - \tfrac{c}{2}n + \tfrac{c}{2}n \\
&= c\,n\log_2(n-1)
\end{aligned}
$$

$(*): (\log_2 \tfrac{Z}{N} = \log_2 Z - \log_2 N)$

Ähnlich zeigt man, dass $T(n) = \Omega(n \log_2 n)$ (freiwillige Übung!). Damit ist die Behauptung bewiesen. Wir haben gezeigt, dass $T(n) = \Theta(n \log_2 n)$.

Intuitive Herleitung der Laufzeitfunktion

Man kann sich die Laufzeit $\Theta(n \log_2 n)$ auch folgendermaßen (informell) herleiten: Wir nehmen der Einfachheit halber an, dass $n = 2^k$. In jedem Aufteilungsschritt wird die Anzahl der zu lösenden Teilinstanzen verdoppelt, die Zeit für das Lösen dieser Teilinstanzen jedoch halbiert. Damit bleibt der Gesamtaufwand in allen Teile- und Kombiniere-Schritten der gleichen Stufe gleich n. Abbildung 2.3 zeigt eine Visualisierung für $n = 2^3 = 8$.



Anzahl der Instanzen	Zeit pro Instanz	Gesamtzeit
$2^0 = 1$	$2^3 = 8$	$2^3 = 8$
$2^1 = 2$	$2^2 = 4$	$2^3 = 8$
$2^2 = 4$	$2^1 = 2$	$2^3 = 8$
$2^3 = 8$	$2^0 = 1$	$2^3 = 8$

Abbildung 2.3: Visualisierung für $n = 2^3 = 8$ und $a = 1$.

Nun stellt sich die Frage: Wieviele Stufen gibt es? Bei $x + 1$ Stufen, wird n so lange (x-Mal) halbiert, bis $\frac{n}{2^x}$ gleich 1 ist, also gilt $x = \log_2 n$. Es gibt also genau $\log_2 n + 1$ solcher Stufen.

Daraus ergibt sich für $n = 2^k$:

$$\begin{aligned} T(n) &= n(1 + \log_2 n) \\ &= n + n \log_2 n \\ &= \Theta(n \log_2 n) \end{aligned}$$

Fortsetzung der Analyse

$$\begin{aligned} C_{\text{worst}}(n) &= \Theta(n \log_2 n) = C_{\text{best}}(n) = C_{\text{avg}}(n) \\ M_{\text{worst}}(n) &= \Theta(n \log_2 n) = M_{\text{best}}(n) = M_{\text{avg}}(n) \end{aligned}$$

Diskussion

- Merge-Sort benötigt $\Theta(n)$ zusätzlichen Speicherplatz.
- Als alternative Datenstruktur eignen sich verkettete Listen, da alle Teilfolgen nur sequentiell verarbeitet werden. Dann werden keine Daten bewegt, nur Verweise

verändert.
⇒ Merge-Sort ist deshalb auch gut als externes Sortierverfahren geeignet (klassisch: Sortieren auf Bändern).

Wir sind hier also insgesamt im Worst-Case und Average-Case besser als beim Sortieren durch Einfügen und Sortieren durch Auswahl.

Bedeutung in der Praxis

Nehmen wir an, es ginge in einem Wettbewerb um das Sortieren von 1 000 000 Zahlen. Die Teilnehmer sind ein Supercomputer mit einem *schlechten* Programm und ein PC mit einem *guten* Programm, wie die folgende Tabelle zeigt.

	Algorithmus	Implementierung	Geschwindigkeit
Supercomputer	Insertion-Sort	$2n^2$	1 000 Mio. Op./sec
PC	Merge-Sort	$50n \log_2 n$	10 Mio. Op./sec

Zeitbedarf:

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

$$\begin{aligned} \text{PC:} \quad & \frac{50 \cdot 10^6 \cdot \log_2 10^6 \text{ Operationen}}{10^7 \text{ Operationen/sec.}} \\ \approx \; & 100 \text{ sec} \end{aligned}$$

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

Wie Computer-Hardware sind auch Algorithmen Technologie!

2.4 Quicksort

Dieser in der Praxis sehr schnelle und viel verwendete Algorithmus wurde 1960 von C.A.R. Hoare entwickelt und basiert ebenfalls auf der Strategie „Teile und Herrsche". Während jedoch bei Merge-Sort die Folgen zuerst aufgeteilt wurden, bevor sortiert wurde, ist dies bei Quicksort umgekehrt: hier wird zunächst Vorarbeit geleistet, bevor rekursiv aufgeteilt wird.

Idee: Wähle ein Pivotelement (z.B. das letzte in der Folge) und bestimme 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. Nach diesem Schleifendurchlauf sitzt das Pivotelement 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 die gleiche Idee rekursiv an.

Der Algorithmus ist wie folgt in das Prinzip „Teile und Herrsche" einzuordnen:

Methode: Falls A die leere Folge ist oder nur ein Element hat, so bleibt A unverändert, sonst

Teile: Wähle „Pivotelement" x aus A. Dann teile A ohne x in zwei Teilfolgen A_1 und A_2, so dass gilt:

- A_1 enthält nur Elemente $\leq x$
- A_2 enthält nur Elemente $\geq x$

Herrsche:

- Rekursiver Aufruf: *Quicksort*(A_1);
- Rekursiver Aufruf: *Quicksort*(A_2);

Danach sind A_1 und A_2 sortiert.

Kombiniere: Bilde A durch Hintereinanderfügen in der Reihenfolge A_1, x, A_2.

Algorithmus 6 Quicksort (**var** A, l, r)

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

```
1: falls l < r dann {
2:     x = A[r].key;
3:     p = Partition(A, l, r, x);
4:     Quicksort (A, l, p − 1);
5:     Quicksort (A, p + 1, r);
6: }
```

Der Aufruf lautet: *Quicksort*($A, 1, n$);

Der Ablauf des Algorithmus *Partition*($A, 4, 9, 4$) beim Aufruf von *Quicksort*($A, 4, 9$) ist in Abbildung 2.4 illustriert.

Algorithmus 7 Partition (**var** A, l, r, x)

Eingabe: Folge A; Indexgrenzen l und r (falls $l \geq r$ ist nichts zu tun); Pivotelement x
Ausgabe: Pivotstelle i; partitionierte Teilfolge $A[l],\ldots,A[r]$
Variable(n): Indizes i, j

```
1:  i = l − 1;   j = r;
2:  wiederhole
3:     wiederhole
4:        i = i + 1;
5:     bis A[i].key ≥ x;
6:     wiederhole
7:        j = j − 1;
8:     bis j ≤ i oder A[j].key ≤ x;
9:     falls i < j dann {
10:       vertausche A[i] und A[j];
11:    }
12: bis i ≥ j;
13: // i ist Pivotstelle: dann sind alle links ≤ x, alle rechts ≥ x
14: vertausche A[i] und A[r];
15: retourniere i;
```

Korrektheit

Die Korrektheit des Algorithmus im Falle seiner Terminierung ist offensichtlich. Das Programm ist auch korrekt, falls Schlüssel mehrmals vorkommen. Allerdings gibt es dann unnötige Vertauschungen.

Wir diskutieren nun die Terminierung. Warum stoppt der Algorithmus *Partition*?

Terminierung der inneren *Wiederhole*-Schleifen:

- Erste Schleife: Durch Wahl des Pivotelements „ganz rechts" kann i durch die Abfrage $A[i].key \geq x$ in Zeile 5 nicht über Position r hinaus wachsen.
- Zweite Schleife: Die Bedingung $j \leq i$ in Zeile 8 bricht diese Schleife ab, sobald j den Index i (der im Intervall $[l, r]$ liegen muss) erreicht oder unterschreitet.

Analyse von Quicksort

Worst-Case: Der schlechteste Fall tritt hier auf, wenn die Eingabeinstanz z.B. eine bereits aufsteigend sortierte Folge ist(!). Eine Illustration des Aufrufbaumes in diesem Fall zeigt

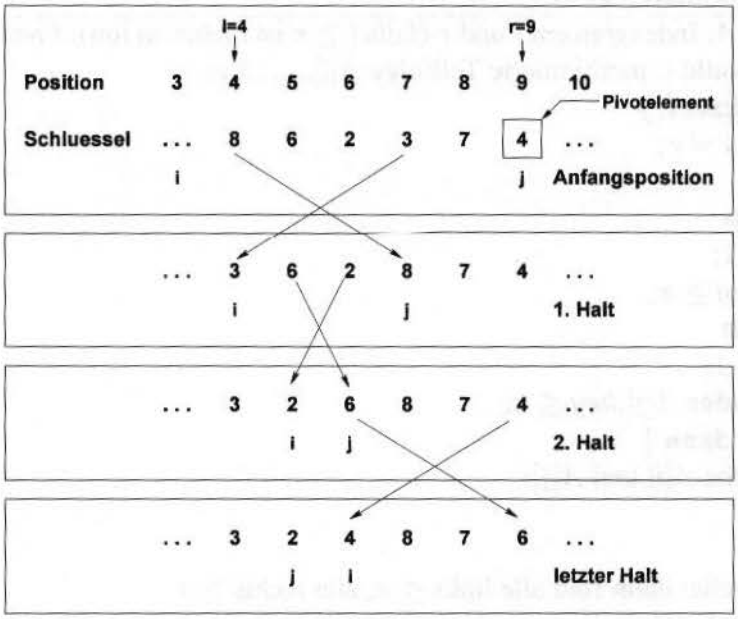


Abbildung 2.4: Illustration eines Aufrufs von Quicksort.

Abbildung 2.5. Es gilt:

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

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). (**Achtung:** Dies steht falsch im Buch von Ottmann und Widmayer. Dort steht $\Theta(n^2)$.)

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 $\Theta(\log n)$ und auf jedem Niveau werden $\Theta(n)$ Vergleiche/Bewegungen durchgeführt, also

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

Dabei ist 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: Übliche Grundannahme: Alle Schlüssel sind verschieden und entsprechen o.B.d.A. genau der Menge $\{1, 2, \ldots, n\}$; alle Permutationen werden als gleich wahrscheinlich angenommen.

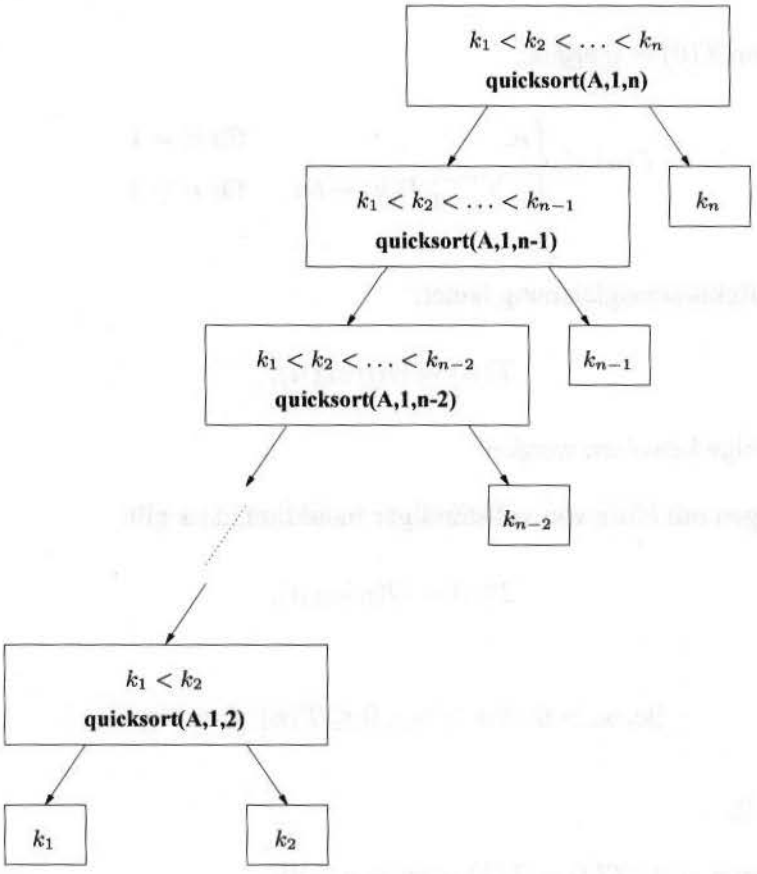


Abbildung 2.5: Aufrufbaum für Quicksort im Worst-Case.

Jede Zahl $k \in \{1, 2, \ldots, n\}$ tritt mit gleicher Wahrscheinlichkeit $\frac{1}{n}$ an Position n auf.

Das Pivotelement k erzeugt zwei Folgen der Längen $(k-1)$ und $(n-k)$.

Beide sind wieder zufällig: Teilt man sämtliche Folgen mit n Elementen mit dem Pivotelement k, so erhält man sämtliche Folgen der Länge $(k-1)$ und $(n-k)$.

Rekursionsformel für die Laufzeitfunktionen für Konstanten a und b

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

wobei bn der Aufteilungsaufwand für eine Folge der Länge n ist.

Das Einsetzen von $T(0) = 0$ ergibt:

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

Die Lösung der Rekursionsgleichung lautet:

$$T(n) = \Theta(n \log n),$$

was wir in der Folge beweisen werden.

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

$$T(n) = O(n \log n),$$

d.h.

$$\exists c, n_0 > 0 : \forall n \geq n_0 : 0 \leq T(n) \leq cn \log n$$

Induktion: $n_0 = 2$:

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

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

$$\begin{aligned} T(n) &\leq \frac{2}{n} \sum_{k=1}^{n-1} T(k) + bn \\ &\leq \frac{2}{n} \sum_{k=1}^{n-1} (ck \log k) + bn \ldots \leq \\ &\leq cn \log n - \frac{c}{4} n - \frac{c}{2} + bn \\ &\leq cn \log n, \text{ falls } c \text{ groß genug} \end{aligned}$$

Wähle z.B. $c = 4b$; dann gilt $T(n) \leq cn \log n$. Mit diesem c und dem gewählten n_0 ist die Behauptung $T(n) = O(n \log n)$ bewiesen. Der Beweis von $T(n) = \Omega(n \log n)$ ist ähnlich. Damit gilt $T(n) = \Theta(n \log n)$. Insgesamt: wähle

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

Abschließende Bemerkungen zu *Quicksort*

- Quicksort ist das beste Verfahren in der Praxis.
- Es gibt viele Varianten, auch solche, die den Worst-Case einer sortierten Folge in der Praxis effektiv verhindern:
  - zufällige Wahl des Pivotelements
  - mittleres (Median) von zufällig gewählten Elementen
- Bei jedem rekursiven Aufruf werden die aktuellen Werte der Parameter und der lokalen Variablen auf den Stack gelegt (Operation „push"), und nach Beendigung des rekursiven Aufrufs wieder vom Stack geholt (Operation „pop"). Wir haben gesehen, dass der Stack für *Quicksort* im Worst-Case $\Theta(n)$ Platz benötigt. Im Best- bzw. Average-Case ist die Höhe des Aufrufbaumes und damit der hierfür zusätzlich benötigte Speicherplatz $\Theta(\log n)$.
- Man kann *Quicksort* mit Hilfe von Stacks ohne rekursive Aufrufe implementieren. Dadurch kann man den zusätzlich benötigten Speicherplatz auf $\Theta(\log n)$ drücken.

2.5 Heapsort

Der Algorithmus *Heapsort* folgt dem Prinzip des Sortierens durch Auswahl, wobei diese Auswahl jedoch geschickt organisiert ist. Dazu wird eine spezielle Datenstruktur, der *Heap*, verwendet.

Definition: Wir nennen eine Folge $F = \langle k_1, k_2, \ldots, k_n \rangle$ von Schlüsseln einen *Maximum-Heap* (*Haufen*), wenn für alle $i \in \{2, 3, \ldots, n\}$ gilt $k_i \leq k_{\lfloor \frac{i}{2} \rfloor}$. Anders ausgedrückt: Falls $2i \leq n$ bzw. $2i + 1 \leq n$, dann muss gelten: $k_i \geq k_{2i}$ bzw. $k_i \geq k_{2i+1}$.
Die Definition eines *Minimum-Heaps* erfolgt analog.

Ein Heap kann sehr gut als binärer Baum veranschaulicht werden. 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 2.6 veranschaulicht den durch die Folge F gegebenen Heap für

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

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

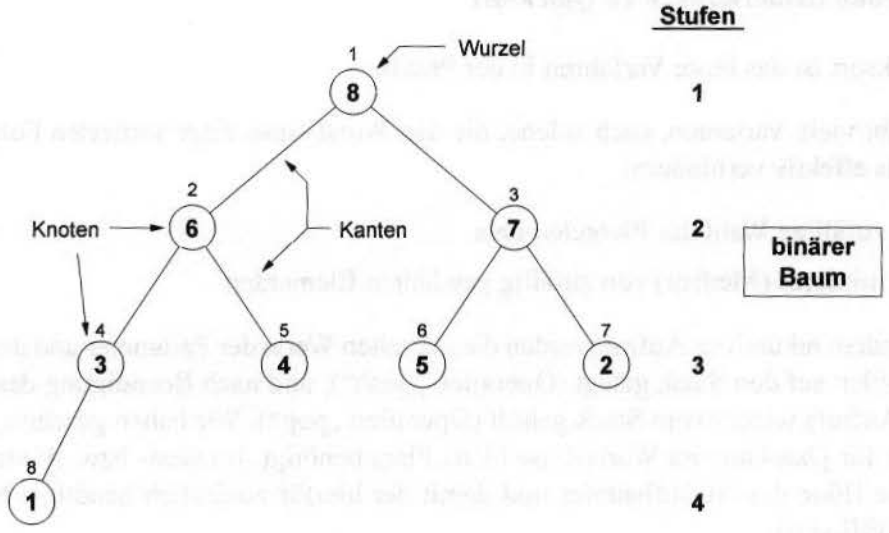


Abbildung 2.6: Der Heap F veranschaulicht als binärer Baum.

- 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.
- k_{2i} in T ist linkes Kind zu k_i.
- k_{2i+1} in T ist rechtes Kind zu k_i.
- k_i in T ist Mutter/Vater von k_{2i} und k_{2i+1}.

Andere Formulierung: Ein binärer Baum ist ein Heap, falls 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 Position eins der Folge bzw. ist die Wurzel des dazugehörigen Binärbaumes. Denn: das erste Element muss größer sein als seine beiden Kinder, diese müssen größer sein als deren Kinder, etc.

Sortieren

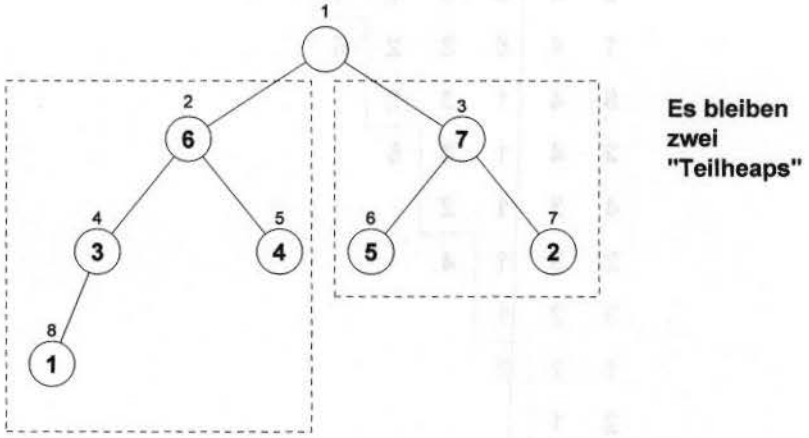
Ein Heap kann sehr gut zum Sortieren benutzt werden. Dazu sind folgende Schritte notwendig:

- Erstelle aus der Eingabefolge einen Heap
- Solange der Heap nicht leer ist:

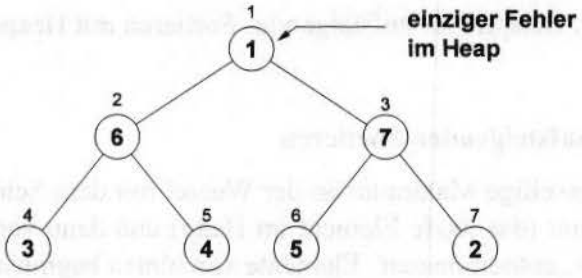
- entferne k_1, das aktuelle Maximum, aus dem Heap (ausgeben, zwischenspeichern, etc.)
- erstelle aus den restlichen Schlüsseln einen Heap

Wir werden diese Schritte im Folgenden genauer untersuchen. Das folgende Beispiel illustriert den zweiten Schritt.

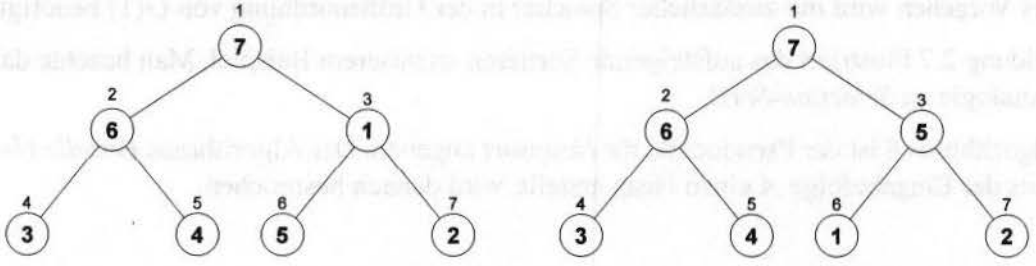
Entfernen des Wertes **8**:



Der Schlüssel mit höchstem Index wird an die Wurzel kopiert:



„Versickern" von 1 nach unten durch Vertauschen mit dem jeweils größten Nachfolger:



Damit ist wieder ein gültiger Heap hergestellt.
Nun folgt die Entfernung von (7), usw.

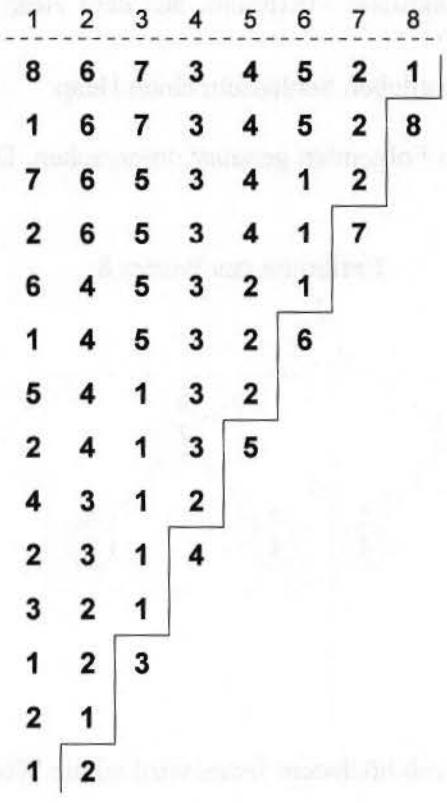


Abbildung 2.7: Beispiel für aufsteigendes Sortieren mit Heapsort.

Heapsort: Methode zum aufsteigenden Sortieren

In Heapsort wird nun das jeweilige Maximum an der Wurzel mit dem Schlüssel vertauscht, der neu an die Wurzel kommt (das letzte Element im Heap) und dann versickert wird. Dadurch können die dem Heap „entnommenen" Elemente von hinten beginnend in den Bereich des Feldes geschrieben werden, der vom Heap selbst nicht mehr verwendet wird. Am Ende ist der Heap leer und die Elemente stehen aufsteigend sortiert im ursprünglichen Feld. Durch dieses Vorgehen wird nur zusätzlicher Speicher in der Größenordnung von $O(1)$ benötigt.

Abbildung 2.7 illustriert das aufsteigende Sortieren an unserem Beispiel. Man beachte dabei die Analogie zu *Selection-Sort*!

In Algorithmus 8 ist der Pseudocode für *Heapsort* angeben. Der Algorithmus *Erstelle Heap*, der aus der Eingabefolge A einen Heap erstellt, wird danach besprochen.

Konstruktion eines Heaps

Idee: Wir interpretieren A als Binärbaum. Wir gehen „von rechts nach links" vor und lassen jeweils Schlüssel versickern, deren beide „Unterbäume" bereits die Heapeigenschaft haben.

Algorithmus 8 Heapsort (**var** A, n)

Eingabe: Folge A, Länge von A ist n
Ausgabe: sortierte Folge A
Variable(n): Index i

```
1: Erstelle Heap (A, n);
2: für i = n, ..., 2 {
3:    Tausche A[1] mit A[i];
4:    Versickere (A, 1, i − 1);
5: }
```

Algorithmus 9 Versickere (**var** A, i, m)

Eingabe: Folge A; Indexgrenzen i und m
Ausgabe: Eingangswert $A[i]$ ist bis maximal $A[m]$ versickert
Variable(n): Index j

```
1: solange 2i ≤ m {
2:    // A[i] hat linkes Kind
3:    j = 2i; // A[j] ist linkes Kind
4:    falls j < m dann {
5:       // A[i] hat rechtes Kind
6:       falls A[j].key < A[j + 1].key dann {
7:          j = j + 1; // A[j] ist größtes Kind
8:       }
9:    }
10:   falls A[i].key < A[j].key dann {
11:      Vertausche A[i] mit A[j];
12:      i = j; // versickere weiter
13:   } sonst {
14:      i = m; // Stopp: Heap-Bedingung erfüllt
15:   }
16: }
```

Algorithmus 10 Erstelle Heap (**var** A, n)

Eingabe: Folge A; maximaler Index n
Ausgabe: Heap in A
Variable(n): Index i

```
1: für i = ⌊n/2⌋, ..., 1 {
2:    Versickere (A, i, n);
3: }
```

Abbildung 2.8 zeigt die Konstruktion eines Heaps an einem Beispiel.

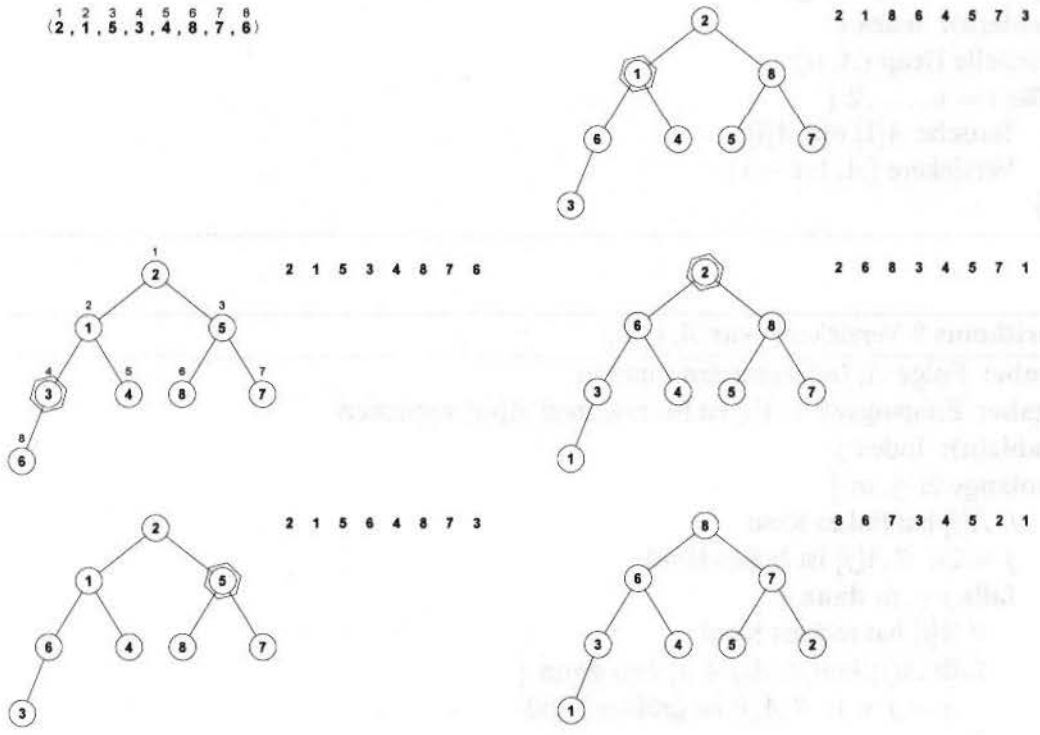


Abbildung 2.8: Konstruktion eines Heaps.

Der Pseudocode zur Konstruktion eines Heaps ist in Algorithmus 10 zu sehen.

Analyse von Heapsort

Analyse für den Aufbau des ersten Heaps

Da ein vollständiger Binärbaum mit j Stufen genau $2^j - 1$ Knoten besitzt, hat ein Heap mit n Schlüsseln genau $j = \lceil \log(n+1) \rceil$ Stufen.
Sei $2^{j-1} \leq n \leq 2^j - 1$, d.h. j ist die Anzahl der Stufen des Binärbaums. 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 $j - k$, d.h. 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 Cormen et al., 1990: Introduction to Algorithms).

Analyse von Versickere $(A, 1, m)$

Die Schleife in Zeile 1 von Algorithmus 9 wird höchstens so oft durchlaufen, wie es der Anzahl der Stufen des Heaps entspricht; das ist $\lceil \log(n+1) \rceil$ mal.

Analyse des gesamten Heapsort

Der Algorithmus *Versickere* wird $n-1$ Mal aufgerufen. Das ergibt insgesamt $O(n \log n)$ Ausführungen der Zeilen 4, 10 und 11 in Algorithmus 9, d.h. $C_{\text{worst}} = \Theta(n \log n)$ und $M_{\text{worst}}(n) = \Theta(n \log n)$.

Die Schritte 4 und 10 werden sowohl im Worst-Case als auch im Best-Case benötigt. Selbst wenn man als Eingabefolge eine invers sortierte Folge hat, benötigt man Aufwand $\Theta(n \log n)$. Damit ergibt sich:

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

Diskussion: Quicksort ist im Durchschnitt schneller als Heapsort, aber Heapsort benötigt nur konstant viel zusätzlichen Speicherplatz.

2.6 Eine untere Schranke

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

Frage: Wieviele Schritte werden mindestens zum Sortieren von n Datensätzen benötigt?

Die Antwort hängt davon ab, was wir über die zu sortierenden Datensätze wissen, und welche Operationen wir zulassen. Wir betrachten das folgende Szenario für allgemeine Sortierverfahren.

Szenario: Wir haben kein Wissen über die Datensätze, nur soviel, dass alle Schlüssel verschieden sind. Um festzustellen, dass ein Schlüssel kleiner als ein anderer ist, sind lediglich elementare Vergleiche zwischen den Schlüsseln erlaubt.

Elementarer Vergleich: 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. Die Blätter entsprechen den Permutationen (siehe Abbildung 2.9).

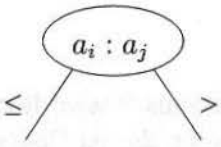


Abbildung 2.9: Ein elementarer Vergleich a_i mit a_j im Entscheidungsbaum.

Abbildung 2.10 zeigt als Beispiel den Entscheidungsbaum des Insertion-Sort Algorithmus für $\langle a_1, a_2, a_3 \rangle$.

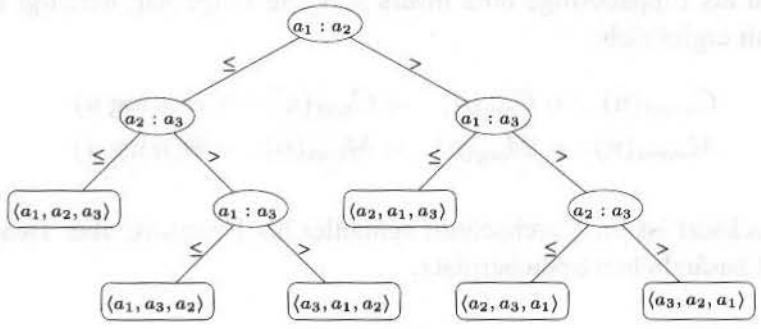


Abbildung 2.10: Entscheidungsbaum für Insertion-Sort von drei Elementen.

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. Um unsere gestellte Frage zu beantworten, suchen wir also nach einer unteren Schranke für die Höhe eines Entscheidungsbaums.

Satz: Jeder Entscheidungsbaum für die Sortierung von n paarweise verschiedenen Schlüsseln hat die Höhe $\Omega(n \log_2 n)$.

Beweis: Wir betrachten einen Entscheidungsbaum der Höhe h, der n disjunkte Schlüssel sortiert. Der Baum hat wenigstens $n!$ Blätter. Ein Binärbaum der Höhe h hat höchstens 2^h Blätter, also

$$n! \leq 2^h.$$

Das impliziert

$$h \geq \log_2(n!).$$

Es gilt

$$\log_2(n!) = \sum_{i=1}^{n} \log_2 i \geq \frac{n}{2} \log_2 \frac{n}{2} = \frac{n}{2}(\log_2 n - 1) = \Omega(n \log_2 n).$$

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

Folgerung: Heapsort und Mergesort sind asymptotisch zeitoptimale Sortieralgorithmen. Es geht nicht besser!

2.7 Lineare Sortierverfahren

Die bisherigen Sortieralgorithmen haben keine algorithmischen Eigenschaften der Schlüssel benutzt. Tatsächlich sind aber meist Wörter über einem bestimmten Alphabet (d.h. einer bestimmten Definitionsmenge) gegeben:

Beispiel

- Wörter über $\{a, b, \ldots, z, A, B, \ldots, Z\}$
- Dezimalzahlen
- ganzzahlige Werte aus einem kleinen Bereich

Diese Information über die Art der Werte und ihren Wertebereich kann man zusätzlich ausnützen und dadurch im Idealfall Verfahren erhalten, die in linearer Worst-Case-Zeit sortieren.

Beispiel 1: Einfaches Bucket-Sort

Geg.: Schlüssel sind ganzzahlige Werte aus einem kleinen Bereich, z.B. $M = \{1, \ldots, m\}$

Idee: Man führt m „Buckets" (Eimer) ein und verteilt zunächst die Schlüssel nacheinander in ihre jeweiligen Buckets. In einem zweiten Schritt sammelt man die Schlüssel der Reihe nach aus den Buckets wieder auf.

Beispiel 2: Sortieren durch Fachverteilung (Radix-Sort)

Dieses Verfahren eignet sich besonders für Schlüssel, welche Wörter über einem aus m Elementen bestehenden Alphabet M mit einer Länge l sind: $w = (w_1, \ldots, w_l) \in M^l$

Idee: t durchläuft alle Zeichen-Positionen absteigend von l bis 1, d.h. von der niederwertigsten zur höchstwertigsten Position.

Das Verfahren wechselt zwischen *Verteilungsphase* und *Sammelphase*. In der Verteilungsphase werden die Datensätze auf m Fächer verteilt, so dass das i-te Fach F_i alle Datensätze enthält, deren Schlüssel an Position t das i-te Zeichen im Alphabet besitzen. Der jeweils nächste Satz wird stets nach den in seinem Fach bereits vorhandenen Sätze gereiht.

In der Sammelphase werden die Sätze in den Fächern $F_1, \ldots, F_m$ so eingesammelt, dass die Sätze im Fach F_{i+1} als Ganzes hinter die Sätze im Fach F_i kommen. In der auf eine Sammelphase folgenden Verteilungsphase müssen die Datensätze wieder in der entsprechenden Reihenfolge verteilt werden.

Beispiel: Die Wortfolge $(ABC, CCC, ACB, BBB, ACA)$, d.h. $l = 3$, mit dem Alphabet $M = \{A, B, C\}$, $m = 3$, soll sortiert werden.

1. Verteilungsphase nach der hintersten Stelle ($t = 3$):

Fach	Datensätze
$F_1\ (A)$	$AC\mathbf{A}$
$F_2\ (B)$	$AC\mathbf{B}, BB\mathbf{B}$
$F_3\ (C)$	$AB\mathbf{C}, CC\mathbf{C}$

1. Sammelphase: $(ACA, ACB, BBB, ABC, CCC)$

2. Verteilungsphase nach der mittleren Stelle ($t = 2$):

Fach	Datensätze
$F_1\ (A)$	–
$F_2\ (B)$	$B\mathbf{B}B, A\mathbf{B}C$
$F_3\ (C)$	$A\mathbf{C}A, A\mathbf{C}B, C\mathbf{C}C$

2. Sammelphase: $(BBB, ABC, ACA, ACB, CCC)$

3. Verteilungsphase nach der vordersten Stelle ($t = 1$):

Fach	Datensätze
$F_1\ (A)$	$\mathbf{A}BC, \mathbf{A}CA, \mathbf{A}CB$
$F_2\ (B)$	$\mathbf{B}BB$
$F_3\ (C)$	$\mathbf{C}CC$

3. Sammelphase: $(ABC, ACA, ACB, BBB, CCC) \hat{=}$ sortierte Folge.

2.8 Zusammenfassung Sortierverfahren

Wir haben nun eine Reihe von Sortierverfahren kennengelernt. An diesem Punkt ist es Zeit, unsere Erkenntnisse noch einmal zusammenzufassen und zu überdenken.

Nehmen Sie sich in Ruhe Zeit und beantworten Sie folgende Fragen (das sind auch beliebte Prüfungsfragen!).

1. Welche Sortierverfahren kennen Sie?
2. Können Sie alle diese Verfahren anhand der Folge $\langle 7, 1012, 13, 128, 5, 4711, 34 \rangle$ durchführen und visualisieren?
3. Wie lautet jeweils die Anzahl der Schlüsselvergleiche und der Datenvertauschungen in Worst-Case, Best-Case und Average-Case?
4. Welches Sortierverfahren würden Sie vorziehen, wenn die Daten bereits „fast" sortiert sind? Warum?
5. Welche Verfahren würden Sie für externes Sortieren empfehlen? Warum?
6. Wie ist die Laufzeit der einzelnen Sortierverfahren, wenn alle Schlüssel den gleichen Wert besitzen?
7. Wie müssen die Sortierverfahren abgeändert werden, damit Sie auch Datensätze korrekt sortieren, die gleiche Schlüssel enthalten?
8. Ein Sortierverfahren heißt stabil, wenn die Reihenfolge der Elemente mit gleichem Schlüssel vor und nach dem Sortiervorgang gleich ist. Welche der Sortierverfahren sind stabil? Begründung?
9. Welche Sortierverfahren funktionieren nach dem Prinzip „Teile und Herrsche"?

Wenn Sie alle diese Fragen korrekt beantworten können, dann sind sie in Sachen Sortieralgorithmen schon relativ „fit".

Kapitel 3

Abstrakte Datentypen und Datenstrukturen

Für den Entwurf eines guten Algorithmus ist die Verwendung einer geeigneten Datenstruktur von großer Bedeutung. Hierzu klären wir zunächst die Begriffe *Abstrakter Datentyp* und *Datenstruktur*.

Ein *Abstrakter Datentyp (ADT)* besteht aus einer oder mehreren Mengen von *Objekten* und darauf definierten *Operationen*. Man nimmt keine Rücksicht auf die programmtechnische Realisierung.

Eine *Datenstruktur* ist eine *Realisierung/Implementierung* eines ADT mit den Mitteln einer Programmiersprache (oft nicht ganz auf der programmiersprachlichen Ebene sondern nur soweit, dass die endgültige Festlegung mit Mitteln einer Programmiersprache nicht mehr schwierig ist).

Wir zeigen am Beispiel des ADT *Lineare Liste*, dass es verschiedene Realisierungen desselben ADT geben kann.

3.1 Definition des ADT Lineare Liste

Objekte: Menge aller endlichen Folgen eines gegebenen Grundtyps. Die Schreibweise für eine lineare Liste ist

$$L = \langle a_1, a_2, \ldots, a_n \rangle,$$

wobei n die Anzahl der enthaltenen Elemente und $L = \langle\rangle$ die leere Liste ist.

Operationen: Im Folgenden gilt: x ist ein Wert vom Basistyp der Datenstruktur (z.B. eine Zahl), p eine Position und L eine lineare Liste.

- *Füge_ein* (x, p, L)

$L \neq \langle\rangle, 1 \leq p \leq n+1$:	Vorher:	$\langle a_1, a_2, \ldots, a_{p-1}, a_p, a_{p+1}, \ldots, a_n\rangle$
	Nachher:	$\langle a_1, a_2, \ldots, a_{p-1}, x, a_p, a_{p+1}, \ldots, a_n\rangle$
$L = \langle\rangle, p = 1$:	Vorher:	$\langle\rangle$
	Nachher:	$\langle x\rangle$

  Sonst undefiniert.

- *Entferne* (p, L)
  Für $L \neq \langle\rangle$ und $1 \leq p \leq n$ gilt: Aus

  $$\langle a_1, a_2, \ldots, a_{p-1}, a_p, a_{p+1}, \ldots a_n\rangle$$

  wird

  $$\langle a_1, a_2, \ldots, a_{p-1}, a_{p+1}, \ldots a_n\rangle.$$

  Sonst undefiniert.

- *Suche*(x, L)
  Liefert die erste Position p in $L = \langle a_1, a_2, \ldots, a_n\rangle$ mit $a_p = x$, falls diese existiert, sonst „nicht gefunden".

- *Zugriff*(p, L)
  Liefert a_p in $L = \langle a_1, a_2, \ldots, a_n\rangle$, falls $1 \leq p \leq n$, sonst undefiniert.

- *Verkette* (L_1, L_2)
  (kann auch zum Anhängen von einzelnen Elementen verwendet werden.)
  Liefert für

  $$L_1 = \langle a_1, a_2, \ldots, a_{n_1}\rangle \text{ und } L_2 = \langle b_1, b_2, \ldots, b_{n_2}\rangle$$

  die Liste

  $$L\langle a_1, a_2, \ldots, a_{n_1}, b_1, b_2, \ldots, b_{n_2}\rangle.$$

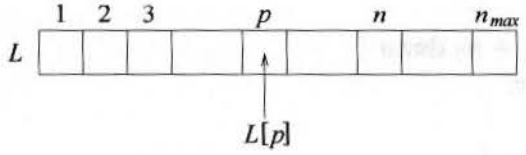
Anmerkung: Die spezielle Variante der linearen Liste, bei der immer nur an einem Ende eingefügt und entfernt wird, wird als *Stack (Stapel)* bezeichnet. Wird immer nur an einem Ende eingefügt und am anderen entfernt, so wird der ADT *Queue (Warteschlange)* genannt.

Um den ADT der linearen Liste zu realisieren, sind verschiedene Datenstrukturen möglich. Wir werden uns zwei verschiedene Implementierungen genauer ansehen.

3.2 Implementierungen des ADT Lineare Liste

3.2.1 Sequentiell gespeicherte lineare Listen

Speicherung in einem Array $L[1], \ldots, L[n_{max}]$:



Leere Liste: $n = 0$;

- Füge_ein (x, p, L):

```
1: falls p < 1 ∨ p > n + 1 ∨ n == n_max dann {
2:     STOP: Fehler;
3: }
4: falls n > 0 dann {
5:     für i = n, n − 1, ..., p {
6:         L[i + 1] = L[i];
7:     }
8: }
9: L[p] = x;
10: n = n + 1;
```

- Entferne (p, L):

```
1: falls p < 1 ∨ p > n dann {
2:     STOP: Fehler;
3: }
4: für i = p, ..., n − 1 {
5:     L[i] = L[i + 1];
6: }
7: n = n − 1;
```

- Suche (x, L):

```
1: für i = 1, ..., n {
2:     falls L[i] == x dann {
3:         retourniere i;
4:     }
5: }
6: retourniere „nicht gefunden“;
```

- Zugriff (p, L):
  1: **falls** $p < 1 \vee p > n$ **dann** {
  2: STOP: Fehler;
  3: }
  4: retourniere $L[p]$;

- Verkette (L_1, L_2):
  1: **falls** $n_{max} < n_1 + n_2$ **dann** {
  2: STOP: Fehler;
  3: }
  4: **für** $j = 1, \ldots, n_2$ {
  5: $L_1[n_1 + j] = L_2[j]$;
  6: }
  7: $n_1 = n_1 + n_2$;
  8: retourniere L_1;

Worst-Case-Analyse der Laufzeit

Füge_ein (x, p, L):	$\Theta(n)$	(bei $p = 1$)
Entferne (p, L):	$\Theta(n)$	(bei $p = 1$)
Suche (x, L):	$\Theta(n)$	(bei $L[i] \neq x \, \forall i$)
Zugriff (p, L):	$\Theta(1)$	
Verkette (L_1, L_2):	$\Theta(n_2)$	

Average-Case-Analyse

Die Zeit für Einfügen, Entfernen und Suchen ist in $\Theta(n)$, falls alle Positionen gleich wahrscheinlich sind.

Z.B. Suchen wenn $x \in L$: Erwartungswert $E[Z_x]$

$$
\begin{aligned}
&= \textstyle\sum_{i=1}^{n} i \cdot P(Z_x = i) \\
&= \textstyle\sum_{i=1}^{n} i \cdot 1/n \\
&= \tfrac{1}{n} \textstyle\sum_{i=1}^{n} i = \tfrac{1}{n} \tfrac{n(n+1)}{2} = \tfrac{n+1}{2} \\
&= \Theta(n)
\end{aligned}
$$

Z_x... Zugriffspfadlänge für Wert x (Zufallsvariable)

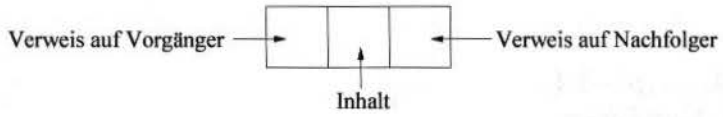
Sind die Elemente in der Liste sortiert, so kann man schneller suchen (siehe Kapitel 2 bzw. Kapitel 4).

3.2.2 Implementierung mit verketteten Listen

Alternativ zu sequentiell gespeicherten linearen Listen kann man den ADT *Lineare Liste* auch durch einfach oder doppelt verkettete Listen realisieren. Es gibt verschiedene Möglich-

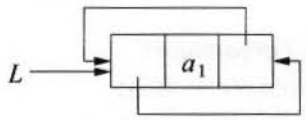
keiten, den ADT damit zu realisieren. Wir zeigen hier als Beispiel eine ringförmig verkettete Liste.

Listenelement:



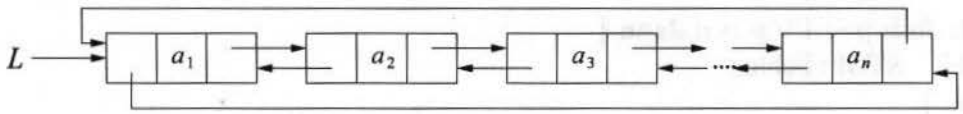
Leere Liste: Es gibt kein Listenelement; dargestellt durch $L = \text{NULL}$ und $n = 0$.

Eine Liste mit einem Element a_1 sieht wie folgt aus:



D.h., die Verweise auf den Vorgänger und Nachfolger zeigen auf ein und das selbe existierende Element.

Allgemein sieht die Darstellung von $L = \langle a_1, a_2, \ldots, a_n \rangle$ wie folgt aus:



Für die nachstehende Implementierung im Pseudocode gelten die folgenden Definitionen bzw. Deklarationen:

a_i :	Listenelement an der Stelle i
$a_i.Vorgaenger$:	Vorgänger von a_i
$a_i.Inhalt$:	Inhalt von a_i
$a_i.Nachfolger$:	Nachfolger von a_i
h :	temporäres Listenelement
neu :	neues Listenelement
L :	Verweis auf das erste Listenelement (d.h. „die Liste“)

- Füge_ein (x, p, L):

```
1: falls p < 1 ∨ p > n + 1 dann {
2:    STOP: Fehler;
3: }
4: h = L;
5: für i = 1, ..., p − 1 {
6:    h = h.Nachfolger;
7: }
8: erzeuge neues Listenelement neu;
9: neu.Inhalt = x;
10: falls n == 0 dann {
11:    neu.Nachfolger = neu.Vorgaenger = neu;
12: } sonst {
13:    neu.Nachfolger = h;
14:    neu.Vorgaenger = h.Vorgaenger;
15:    neu.Nachfolger.Vorgaenger = neu;
16:    neu.Vorgaenger.Nachfolger = neu;
17: }
18: falls p == 1 dann {
19:    L = neu;
20: }
21: n = n + 1;
```

- Entferne (p, L):

```
1: falls p < 1 ∨ p > n dann {
2:    STOP: Fehler;
3: }
4: falls n == 1 dann {
5:    L = NULL;
6: } sonst {
7:    h = L;
8:    für i = 1, ..., p − 1 {
9:       h = h.Nachfolger;
10:   }
11:   h.Vorgaenger.Nachfolger = h.Nachfolger;
12:   h.Nachfolger.Vorgaenger = h.Vorgaenger;
13:   falls p == 1 dann {
14:      L = L.Nachfolger;
15:   }
16: }
17: n = n − 1;
```

- Suche (x, L):

```
1: h = L;
2: i = 1;
3: solange i ≤ n ∧ h.Inhalt ≠ x {
4:     h = h.Nachfolger;
5:     i = i + 1;
6: }
7: falls i ≤ n dann {
8:     retourniere i;
9: }
10: retourniere „nicht gefunden“;
```

- Zugriff (p, L):

```
1: falls p < 1 ∨ p > n dann {
2:     STOP: Fehler
3: }
4: h = L;
5: für i = 1, ..., p − 1 {
6:     h = h.Nachfolger;
7: }
8: retourniere h.Inhalt;
```

- Verkette (L_1, L_2):

```
1: falls L1 == NULL dann {
2:     L1 = L2;
3: } sonst falls L2 ≠ NULL dann {
4:     last = L2.Vorgaenger;
5:     L1.Vorgaenger.Nachfolger = L2;
6:     last.Nachfolger = L1;
7:     L2.Vorgaenger = L1.Vorgaenger;
8:     L1.Vorgaenger = last;
9: }
10: n1 = n1 + n2;
11: retourniere L1;
```

Worst-Case-Analyse der Laufzeit

Füge_ein (x, p, L):	$\Theta(n)$	$(p = n + 1)$
Entferne (p, L):	$\Theta(n)$	$(p = n)$
Suche (x, L):	$\Theta(n)$	$(a_i \neq x \quad \forall i)$
Zugriff (p, L):	$\Theta(n)$	$(p = n)$
Verkette (L_1, L_2):	$\Theta(1)$	

Average-Case-Analyse

Einfügen, Entfernen, Suchen und Zugriff dauern $\Theta(n)$, falls alle Positionen gleich wahrscheinlich sind.

Diskussion

Erwartet man viele Operationen vom Typ Zugriff, ist die Datenstruktur der sequentiell in einem Array gespeicherten Liste schneller. Dafür entsteht ein Nachteil beim Einfügen oder Löschen, weil immer alle Elemente bis zum Ende des Arrays verschoben werden müssen.

Erwartet man viele Operationen vom Typ Verketten bzw. Einfügen oder Entfernen an Anfang oder Ende der Folge, dann sind doppelt verkettete Listen schneller, weil auf Anfang und Ende unmittelbar zugegriffen werden kann und das Umspeichern entfällt. Dafür ergibt sich hier ein Nachteil beim Zugriff auf beliebige Elemente.

Abschließend sei auch daran erinnert, dass bei der sequentiell in einem Array gespeicherten Variante die maximale Anzahl der Elemente $n_{\max}$ bekannt sein muss und die Datenstruktur immer entsprechend viel Speicherplatz benötigt, während im Fall der doppelt verketteten Liste der Speicherbedarf $\Theta(n)$ ist.

3.2.3 Wörterbücher (engl. *Dictionaries*)

Ein weiteres Beispiel für einen Abstrakten Datentyp ist das *Wörterbuch* (engl. *Dictionary*).

Als Wörterbuch wird eine Menge von Elementen eines gegebenen Grundtyps bezeichnet, auf der man die Operationen *Suchen*, *Einfügen* und *Entfernen* von Elementen ausführen kann. Man nimmt dabei meistens an, dass alle Elemente über einen in der Regel ganzzahligen Schlüssel identifizierbar sind. Sei S das gegebene Wörterbuch, dann sind die Operationen definiert durch:

- *Suchen* (S, x): Wenn x in S vorkommt und x Schlüssel eines Elementes ist, so soll als Ergebnis der Suchoperation der Zugriff auf die jeweilige Komponente möglich sein; andernfalls ist der Rückgabewert „NULL“, „false“ oder „0“ für „nicht gefunden“.
- *Einfügen* (S, x): Ersetze S durch $S \cup \{x\}$
- *Entfernen* (S, x): Ersetze S durch $S \setminus \{x\}$

Das Problem, eine Datenstruktur zusammen mit möglichst effizienten Algorithmen zum Suchen, Einfügen und Entfernen von Schlüsseln zu finden, nennt man das *Wörterbuchproblem*. Unsere Aufgabe ist es nun, effiziente Implementierung dafür zu finden.

Eine triviale mögliche Realisierung des Datentyps Wörterbuch ist diejenige durch sequentiell oder verkettet gespeicherte lineare Listen. Wir haben bereits in Abschnitt 3.2 gesehen, dass der Aufwand für die Operationen Suchen, Einfügen und Entfernen in einer solchen Liste mit n Elementen jeweils $\Theta(n)$ beträgt.

Im Kapitel 4 werden wir sehen, dass man den Aufwand zu $\Theta(\log_2 n)$ verringern kann (also deutlich kleiner, denn z.B. $\log_2 n \leq 20$ für $n = 1\,000\,000$), indem man balancierte binäre Suchbäume verwendet (siehe Abschnitt 4.2.2). Ändert sich die Datenmenge nicht laufend, dann empfielt es sich, in sequentiell gespeicherten linearen Listen mit Hilfe von *Binärer Suche* mit Aufwand $\Theta(\log_2 n)$ zu suchen (siehe Kapitel 4.1). Unter gewissen Bedingungen kann man das Wörterbuchproblem sogar in annähernd konstanter Zeit lösen (siehe Hashing-Verfahren in Kapitel 5).

Kapitel 4

Suchverfahren

Ein wichtiges Thema der Algorithmentheorie ist neben dem Sortieren von Datensätzen **das Suchen in Datenmengen**.

Das Suchen ist eine der grundlegendsten Operationen, die man immer wieder mit Computern ausführt, z.B. das Suchen von Datensätzen in Datenbanken oder das Suchen nach Wörtern in Wörterbüchern. Sehr aktuell ist momentan das Suchen von Stichwörtern im WWW. Wir behandeln zunächst nur elementare Suchverfahren, d.h. solche Verfahren, die nur Vergleichsoperationen zwischen den Schlüsseln zulassen (wie bei allgemeinen Sortierverfahren). In Kapitel 5 werden wir andere Suchverfahren kennenlernen, bei denen zusätzlich arithmetische Operationen erlaubt sind.

4.1 Suchen in sequentiell gespeicherten Folgen

Voraussetzung: In diesem Abschnitt nehmen wir an, dass die Datenelemente als Feld $(A[1], \ldots, A[n])$ implementiert sind. Wir suchen in A ein Element mit Schlüssel s.

Wir werden zwei Suchverfahren kennenlernen: ein einfaches, naives Verfahren sowie die binäre Suche, die in vielen Fällen empfehlenswert ist.

4.1.1 Lineare Suche (Naives Verfahren)

Die lineare Suche macht das, was man als Programmier-Anfänger intuitiv tun würde.

Idee: Durchlaufe alle Elemente eines Feldes von vorne nach hinten und vergleiche jeden Schlüssel mit dem Suchschlüssel, solange bis der Schlüssel gefunden wird.

Analyse

- Best-Case (d.h. sofortiger Treffer): $C_{\text{best}}(n) = 1$
- Worst-Case (erfolglose Suche): $C_{\text{worst}}(n) = n$
- Average-Case (für erfolgreiche Suche unter der Annahme: Jede Anordnung ist gleich wahrscheinlich):

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

Diskussion

- Diese einfache Methode eignet sich auch für einfach verkettet implementierte Listen.
- Lineare Suche ist nur für kleine n empfehlenswert.

4.1.2 Binäre Suche

Idee: Die Idee der binären Suche kann man sich sehr leicht anhand der Suche in einem Telefonbuch vergegenwärtigen. Sucht man dort nach einem Namen, schlägt man es zunächst in der Mitte auf. Ist der gesuchte Namen „kleiner" als die Namen auf der aufgeschlagenen Seite, so vergisst man die rechte Hälfte des Buches und wendet dasselbe Verfahren auf die linke Hälfte an, und so weiter.

Methode: 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 mit dem Schlüssel des Elementes in der Mitte. Falls

- s gefunden (Treffer): STOPP
- s ist kleiner: Durchsuche Elemente „links der Mitte"
- s ist größer: Durchsuche Elemente „rechts der Mitte"

Algorithmus 11 zeigt eine nicht-rekursive Implementierung für die binäre Suche. Die Suche wird durch den Aufruf *Binaere-Suche*$(A, s, 1, n)$ gestartet.

Algorithmus 11 Binäre-Suche (A, s, l, r)

Eingabe: Feld $A[l..r]$; Schlüssel s; Indizes l und r
Ausgabe: Position von s oder 0, falls s nicht vorhanden
Variable(n): Index m

```
1: wiederhole
2:    m = ⌊(l + r)/2⌋;
3:    falls s < A[m].key dann {
4:       r = m − 1;
5:    } sonst {
6:       l = m + 1;
7:    }
8: bis s == A[m].key ∨ l > r;
9: falls s == A[m].key dann {
10:   retourniere m;
11: } sonst {
12:   retourniere 0;
13: }
```

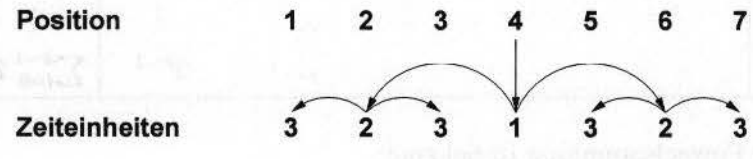
Analyse

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 11.

- Best-Case (d.h. sofortiger Treffer): $C_{\text{best}}(n) = \Theta(1)$.
- Worst-Case: Wie lange dauert es, das Element an Position $i \in \{1, 2, \ldots, n\}$ zu finden?

1	für mittleres Element
2	für mittleres in linker Hälfte
2	für mittleres in rechter Hälfte
	usw.

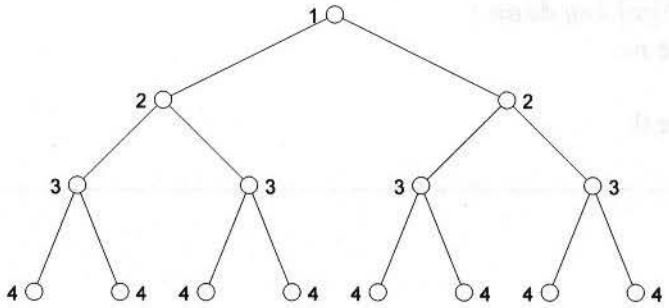
Beispiel für $n = 7 = 2^3 - 1$ ($k = 3$):



Sowohl für erfolgreiches als auch für erfolgloses Suchen gilt:
$C_{\text{worst}}(n) = \log_2(n + 1) = \Theta(\log n)$.

- Average-Case:

Zeit-einheiten	Anzahl Positionen	Produkt	aufsummiert
1	1	1	1
2	2	4	5
3	4	12	17
4	8	32	49
⋮	⋮	⋮	⋮
k	2^{k-1}	$k \cdot 2^{k-1}$	$\sum_{i=1}^{k} i \cdot 2^{i-1}$



Durchschnittliche Zeit für erfolgreiche Suche:

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

Die Summe $\sum_{i=1}^{k} i \cdot 2^{i-1}$ kann wie folgt berechnet werden:

	$1 \cdot 2^0$	$+ \; 2 \cdot 2^1$	$+ \; 3 \cdot 2^2$	$+ \; \ldots$	$+ \; k \cdot 2^{k-1}$	
0	2^0	2^1	2^2	$\ldots$	2^{k-1}	$\sum_{i=0}^{k-1} 2^i$
1		2^1	2^2	$\ldots$	2^{k-1}	$\sum_{i=0}^{k-1} 2^i - \sum_{i=0}^{0} 2^i$
2			2^2	$\ldots$	2^{k-1}	$\sum_{i=0}^{k-1} 2^i - \sum_{i=0}^{1} 2^i$
⋮				$\ldots$	⋮	⋮ ⋮
$k-1$				$\ldots$	2^{k-1}	$\sum_{i=0}^{k-1} 2^i - \sum_{i=0}^{k-2} 2^i$

Aus der Formelsammlung ist bekannt:

$$\sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1} \Rightarrow \sum_{i=0}^{k-1} 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1.$$

Also gilt:

$$\begin{aligned}\sum_{i=1}^{k} i \cdot 2^{i-1} &= k(2^k - 1) - \sum_{i=0}^{k-1}(2^i - 1) \\ &= k(2^k - 1) - (2^k - 1) + k \\ &= (k-1)(2^k - 1) + k \\ &= (k-1)2^k - k + 1 + k \\ &= (k-1)2^k + 1\end{aligned}$$

Durch Einsetzen von $k = \log_2(n+1)$ ergibt sich schließlich:

$$\begin{aligned}&= (\log_2(n+1) - 1)(n+1) + 1 \\ &= (n+1)\log_2(n+1) - n\end{aligned}$$

Daraus berechnet sich die durchschnittliche Zeit für die Suche:

$$\begin{aligned}\frac{1}{n}\left[(n+1)\log_2(n+1) - n\right] &= \frac{n+1}{n}\log_2(n+1) - 1 \\ &= \log_2(n+1) + \frac{\log_2(n+1)}{n} - 1 \\ &\underset{n\to\infty}{\to} \log_2(n+1) - 1\end{aligned}$$

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. In unserem Telefonbuch-Beispiel würde lineare Suche bei ca. 2 Millionen Telefonteilnehmern würde schlimmstenfalls 2 Millionen Vergleiche benötigen, binäre Suche hingegen nur $\log(2 \cdot 10^6) \approx 20$ Vergleiche. Beachten Sie die **große Zeitersparnis!**
- In der Praxis macht man meist Interpolationssuche, 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.: „*AAAAAAAABZ*".
- Binäre Suche ist nur sinnvoll, wenn sich die Daten nicht allzu oft ändern; sonst sind binäre Suchbäume besser geeignet (siehe Abschnitt 4.2).

4.2 Binäre Suchbäume

Binäre Suchbäume eignen sich zum Suchen in Datenmengen, die sich immer wieder ändern, sogenannte dynamische Datenmengen. Binäre Suchbäume sind eine Datenstruktur zur Unterstützung der folgenden Operationen auf dynamischen Mengen:

- Suchen nach einem Element mit dem gegebenen Schlüssel
- Minimum / Maximum: das Element mit dem kleinsten / größten Schlüssel im Baum finden
- Predecessor / Successor: ausgehend von einem gegebenen Element das Element mit dem nächstkleineren / nächstgrößeren Schlüssel im Baum finden
- Einfügen / Entfernen eines bestimmten Elementes

Damit lösen sie das Wörterbuchproblem (s. Abschnitt 3.2.3). Zunächst führen wir einige Begriffe im Zusammenhang mit binären Bäumen ein. Wir haben binäre Bäume bereits benutzt (Heapsort, untere Schranke für Sortieren).

Bezeichnungen

- Ein *Baum* ist eine kreisfreie Struktur, die aus einer Menge von Knoten sowie einer Menge von (gerichteten) Kanten besteht, die jeweils Paare von Knoten miteinander verbinden, so dass zwischen je zwei Knoten genau ein Pfad existiert. Ein Baum enhält also keine Kreise. Die Länge eines Pfades berechnet sich aus der Anzahl der Kanten, die auf dem Weg zwischen den beiden Endknoten des Pfades liegen. Z.B. ist der Pfad zwischen den Knoten 3 und 8 im Baum T_2 in Abb. 4.1 gegeben durch $3 - 7 - 8$ und hat Länge 2.
- In der Regel wird einer der Knoten des Baumes als *Wurzel* (*root*) ausgezeichnet. Dieser ist der einzige Knoten ohne Vorgänger. Alle anderen können genau über einen Weg (Pfad) von der Wurzel erreicht werden. Jeder Knoten ist gleichzeitig die Wurzel eines *Unterbaumes*, der aus ihm selbst und seinen direkten und indirekten Nachfolgern besteht. Abbildung 4.1 zeigt als Beispiel den linken Unterbaum der Wurzel von T_1.
- Ein Baum heißt *binärer Baum*, wenn jeder Knoten höchstens zwei Kinder (Nachfolger) besitzt: ein linkes und ein rechtes Kind.
- Die *Tiefe* $d(v)$ eines Knotens v ist die Anzahl der Kanten auf dem Pfad von diesem Knoten zur Wurzel. Sie ist eindeutig, da nur ein solcher Pfad existiert. Die Tiefe der Wurzel ist 0. Die Menge aller Knoten mit gleicher Tiefe im Baum wird auch *Ebene* oder *Niveau* genannt.

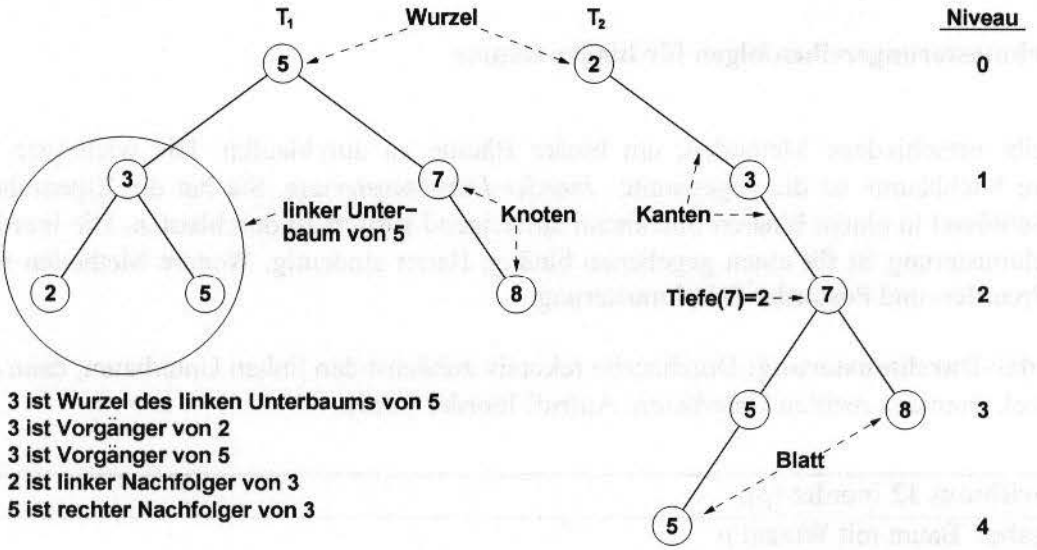


Abbildung 4.1: Zwei Bäume mit den entsprechenden Bezeichnungen.

- Die *Höhe* $h(T_r)$ eines (Teil-)Baumes T_r mit dem Wurzelknoten r ist definiert als die Länge eines längsten Pfades in dem Teilbaum von r zu einem beliebigen Knoten v in T_r. In Abb. 4.1 gilt z.B., $h(T_1) = 2$ und $h(T_2) = 4$.

- Ein Knoten heißt *Blatt*, wenn er keine Kinder besitzt. Alle anderen Knoten nennt man *innere Knoten.*

Implementierung von binären Bäumen

Wir implementieren binäre Bäume als verallgemeinerte Listen mit bis zu zwei Nachfolgern:

$x.key$:	Schlüssel von x
$x.info$:	zum Schlüssel zu speichernde Daten
$x.parent$:	Vorgänger von x
$x.leftchild$:	linkes Kind von x
$x.rightchild$:	rechtes Kind von x

Der Zugriff auf den Baum erfolgt über seinen Wurzelknoten *root*.

Durchmusterungsreihenfolgen für binäre Bäume

Es gibt verschiedene Methoden, um binäre Bäume zu durchlaufen. Die wichtigste für binäre Suchbäume ist die sogenannte *Inorder-Durchmusterung*. Sie hat die Eigenschaft, die Schlüssel in einem binären Suchbaum aufsteigend sortiert zu durchlaufen. Die Inorder-Durchmusterung ist für einen gegebenen binären Baum eindeutig. Weitere Methoden sind die Preorder- und Postorder-Durchmusterung.

Inorder-Durchmusterung: Durchsuche rekursiv zunächst den linken Unterbaum, dann die Wurzel, dann den rechten Unterbaum. Aufruf: Inorder (*root*).

Algorithmus 12 Inorder (p)

Eingabe: Baum mit Wurzel p
Ausgabe: Inorder-Durchmusterung des Baumes

```
1: falls p ≠ NULL dann {
2:    Inorder(p.leftchild);
3:    Ausgabe von p;
4:    Inorder(p.rightchild);
5: }
```

Beispiel: „Inorder"-Durchmusterung von T_1:

```
Inorder (5)
  Inorder (3)
    Inorder (2)
      Ausgeben (2)
    Ausgeben (3)
    Inorder (5)
      Ausgeben (5)
  Ausgeben (5)
  Inorder (7)
    Ausgeben (7)
    Inorder (8)
      Ausgeben (8)
```

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

Preorder-Durchmusterung: Durchsuche rekursiv zunächst die Wurzel, dann den linken Unterbaum, danach den rechten Unterbaum. Dies ergibt die folgende Durchmusterungsreihenfolge für T_1 und T_2:

$$T_1 : 5, 3, 2, 5, 7, 8$$

$$T_2 : 2, 3, 7, 5, 5, 8$$

Postorder-Durchmusterung: Durchsuche rekursiv zunächst den linken Unterbaum, dann den rechten Unterbaum, danach die Wurzel. Dies ergibt die folgende Durchmusterungsreihenfolge für T_1 und T_2:

$$T_1 : 2, 5, 3, 8, 7, 5$$

$$T_2 : 5, 5, 8, 7, 3, 2$$

Bei gegebener Inorder- und Preorder-Durchmusterungsreihenfolge eines beliebigen binären Baums kann dieser eindeutig rekonstruiert werden kann, sofern er lauter verschiedene Schlüssel enthält. Dies ist nicht möglich für gegebene Preorder- und Postorder-Folgen, wie das Beispiel in Abbildung 4.2 zeigt. Überlegen Sie sich, wie das für Inorder und Postorder ist.

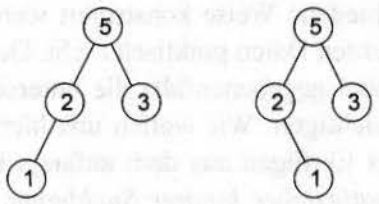


Abbildung 4.2: Zwei Bäume mit den gleichen Preorder (5, 2, 1, 3) und Postorder (1, 2, 3, 5) Durchmusterungsreihenfolgen.

Binäre Suchbaumeigenschaft

Für Suchbäume gilt die *Binäre Suchbaumeigenschaft*, die folgendermaßen definiert ist:
Sei x ein Knoten eines binären Suchbaumes. Ist y ein Knoten des linken Unterbaumes von x so gilt $y.key \leq x.key$. Ist z ein Knoten des rechten Unterbaumes von x so gilt $z.key \geq x.key$.

Prinzipiell können gleiche Schlüssel also im linken oder im rechten Unterbaum gespeichert werden, in einer konkreten Implementierung muss man sich aber für eine Variante entscheiden und diese dann auch konsequent verfolgen.

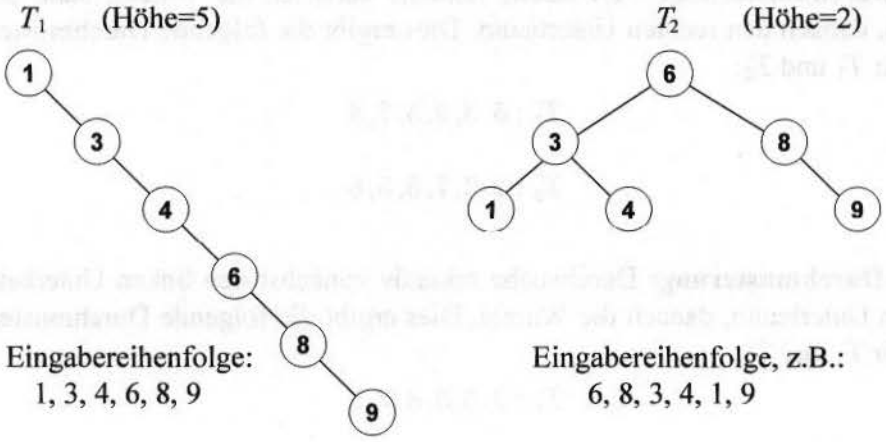


Abbildung 4.3: Zwei Bäume mit denselben Datenelementen: Baum T_1 ist zur linearen Liste entartet, Baum T_2 dagegen ist vollständig balanciert.

Aufbau binärer Suchbäume

Ein Binärbaum kann auf verschiedene Weise konstruiert werden. In manchen Anwendungen verändern sich die gespeicherten Daten praktisch nicht. Dann kann man einen statischen Suchbaum konstruieren und dabei gegebenenfalls die unterschiedliche Suchhäufigkeit für verschiedene Schlüssel berücksichtigen. Wir wollen uns hier jedoch solchen Bäumen zuwenden, die durch wiederholtes Einfügen aus dem anfangs leeren Baum erzeugt werden. Auf diese Weise entsteht ein *natürlicher binärer Suchbaum*, dessen Gestalt entscheidend von der Reihenfolge abhängt, in der die Schlüssel eingefügt werden.

Im Extremfall kann ein Baum T sogar zu einer linearen Liste ausarten, z.B. wenn er entsteht, indem die Einträge mit aufsteigenden Schlüsseln in einen anfangs leeren Baum eingefügt werden. Der Baum verhält sich dann bei der Suche wie eine lineare Liste. Andererseits können auch vollständig ausgeglichene Bäume entstehen, bei denen alle Ebenen bis auf eventuell die unterste voll besetzt sind. Abbildung 4.3 zeigt zwei Beispiele für diese beiden Extremfälle.

Aufgrund dieser Abhängigkeit von der Einfüge-Reihenfolge 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 in logarithmischer Schrittzahl ausführbar sind. Diese sogenannten *balancierten Suchbäume* werden in Abschnitt 4.2.2 diskutiert.

4.2.1 Natürliche binäre Suchbäume

Wir geben nun eine Implementierung aller oben genannten Operationen mit Laufzeit $O(h(T))$ für natürliche binäre Suchbäume an, wobei $h(T)$ die Höhe des gegebenen Suchbaumes ist. Abbildung 4.4 zeigt Beispiele für die Operationen *Minimum, Maximum, Successor, Predecessor* und *Einfügen* in einem Baum. Die Entfernung eines Knotens ist ein wenig aufwändiger und wird im Folgenden besprochen.

Bemerkung: Zur Auflösung der Mehrdeutigkeit im Falle gleicher Schlüssel wird die binäre Suchbaumeigenschaft leicht modifiziert: es wird im Folgenden davon ausgegangen, dass gleiche Schlüssel stets im rechten Unterbaum gespeichert werden.

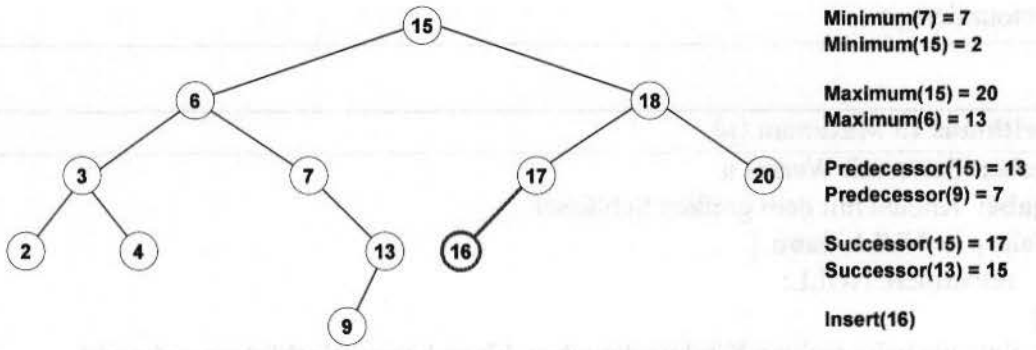


Abbildung 4.4: Illustration der Operationen *Minimum, Maximum, Successor, Predecessor* und *Einfügen*.

Algorithmus 13 Suchen (p, s)

Eingabe: Baum mit Wurzel p; Schlüssel s
Ausgabe: Knoten mit Schlüssel s oder NULL, falls s nicht vorhanden

```
1: solange p ≠ NULL ∧ p.key ≠ s {
2:   falls s < p.key dann {
3:     p = p.leftchild;
4:   } sonst {
5:     p = p.rightchild;
6:   }
7: }
8: retourniere p;
```

Algorithmus 14 Minimum (p)

Eingabe: Baum mit Wurzel p
Ausgabe: Knoten mit dem kleinsten Schlüssel

```
1: falls p = NULL dann {
2:     retourniere NULL;
3: }
4: // Solange beim linken Kind weitergehen, bis es keinen Nachfolger mehr gibt
5: solange p.leftchild ≠ NULL {
6:     p = p.leftchild;
7: }
8: retourniere p;
```

Algorithmus 15 Maximum (p)

Eingabe: Baum mit Wurzel p
Ausgabe: Knoten mit dem größten Schlüssel

```
1: falls p = NULL dann {
2:     retourniere NULL;
3: }
4: // Solange beim rechten Kind weitergehen, bis es keinen Nachfolger mehr gibt
5: solange p.rightchild ≠ NULL {
6:     p = p.rightchild;
7: }
8: retourniere p;
```

Algorithmus 16 Successor (p)

Eingabe: Baum mit Wurzel p
Ausgabe: Nachfolger von Knoten $p \neq$ NULL in der Inorder-Durchmusterungsreihenfolge
Variable(n): Knoten q

```
1: falls p.rightchild ≠ NULL dann {
2:     retourniere Minimum (p.rightchild);
3: } sonst {
4:     q = p.parent;
5:     // Wenn p linker Nachf. von q → fertig: q ist Successor
6:     solange q ≠ NULL ∧ p == q.rightchild {
7:         p = q;
8:         q = q.parent;
9:     }
10:    retourniere q;
11: }
```

Algorithmus 17 Predecessor (p)

Eingabe: Baum mit Wurzel p
Ausgabe: Vorgänger von Knoten $p \neq$ NULL in der Inorder-Durchmusterungsreihenfolge
Variable(n): Knoten q

```
1: falls p.leftchild ≠ NULL dann {
2:    retourniere Maximum (p.leftchild);
3: } sonst {
4:    q = p.parent;
5:    // Wenn p rechter Nachf. von q → fertig: q ist Predecessor
6:    solange q ≠ NULL ∧ p == q.leftchild {
7:       p = q;
8:       q = q.parent;
9:    }
10:   retourniere q;
11: }
```

Algorithmus 18 Einfügen (**var** $root, q$)

Eingabe: Baum mit Wurzel $root$; Knoten q
Variable(n): Knoten r

```
1: // Fügt Knoten q in Baum mit Wurzel root ein
2: r = NULL; // r wird Vorgänger von q
3: p = root;
4: solange p ≠ NULL {
5:    r = p;
6:    falls q.key < p.key dann {
7:       p = p.leftchild;
8:    } sonst {
9:       p = p.rightchild; // gleiche Schlüssel kommen nach rechts
10:   }
11: }
12: q.parent = r;
13: q.leftchild = NULL;
14: q.rightchild = NULL;
15: falls r == NULL dann {
16:    root = q;
17: } sonst {
18:    falls q.key < r.key dann {
19:       r.leftchild = q;
20:    } sonst {
21:       r.rightchild = q;
22:    }
23: }
```

Entfernung eines Knotens

Wir nehmen an, wir wollen Knoten z aus einem binären Suchbaum entfernen. Abhängig von der Anzahl seiner Kinder gibt es drei verschiedene Fälle zu beachten.

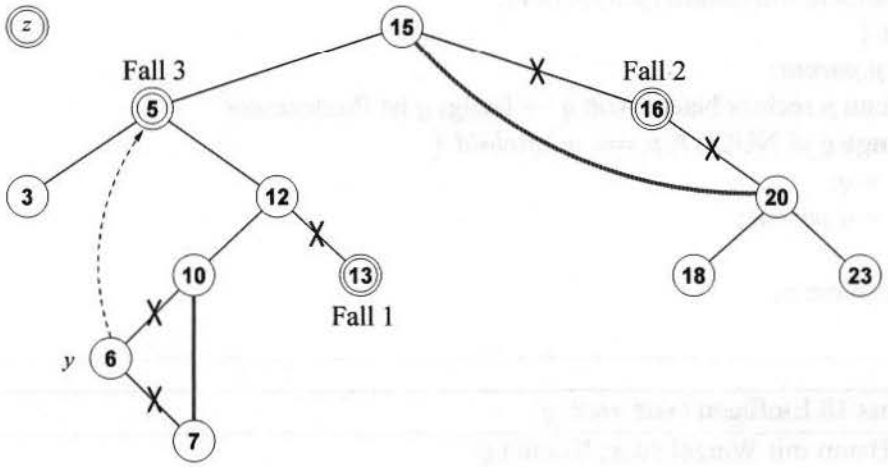


Abbildung 4.5: Drei mögliche Fälle beim Löschen eines Knotens (die zu entfernenden Knoten sind mit zwei Kreisen gekennzeichnet).

Fall 1: z hat keine Kinder, z.B. Knoten 13 in Abb. 4.5.
$\rightarrow$ In diesem Fall kann z einfach entfernt werden.

Fall 2: z hat ein Kind, z.B. Knoten 16 in Abb. 4.5.
$\rightarrow$ Hier müssen einfach nur zwei Verweise abgeändert werden. z wird sozusagen aus dem Baum „herausgeschnitten".

Fall 3: z hat zwei Kinder, z.B. Knoten 5 in Abb. 4.5.
Um die binäre Suchbaumeigenschaft weiterhin aufrecht zu erhalten, muss ein geeigneter Ersatzknoten für z gefunden werden. Geeignet sind der Knoten mit dem größten Schlüssel des linken Unterbaumes (Predecessor) bzw. der Knoten mit dem kleinsten Schlüssel des rechten Unterbaumes (Successor), da sie z am ähnlichsten sind. Im Prinzip ist es gleich, für welche Variante man sich entscheidet, weil beides wieder zu einem gültigen binären Suchbaum führt. In diesem Skriptum wollen wir uns auf den Successor beschränken.
$\rightarrow$ Sei y der Successor von z. Dann hat y kein linkes Kind. z wird durch den Knoten y ersetzt und y wird an seiner ursprünglichen Stelle entfernt (s. Fall 2). Dadurch gehen keine Daten verloren und wir erhalten das gewünschte Ergebnis.

Algorithmus 19 zeigt den Pseudocode für das Entfernen eines Knotens.

Algorithmus 19 Entfernen (**var** $root, q$)

Eingabe: Baum mit Wurzel $root$; Knoten q
Variable(n): Knoten r

```
1: // Entfernt Knoten q im Baum mit Wurzel root
2: // Bestimme einen Knoten r zum Herausschneiden
3: falls q.leftchild == NULL ∨ q.rightchild == NULL dann {
4:    // q hat max. 1 Nachfolger → wird selbst entfernt
5:    r = q;
6: } sonst {
7:    // q hat 2 Nachfolger → wird durch Successor ersetzt, dieser wird entfernt
8:    r = Successor(q);
9:    // Umhängen der Daten von r nach q
10:   q.key = r.key;
11:   q.info = r.info;
12: }
13: // Lasse p auf Kind von r verweisen (p = NULL, falls r keine Kinder hat)
14: falls r.leftchild ≠ NULL dann {
15:   p = r.leftchild;
16: } sonst {
17:   p = r.rightchild;
18: }
19: falls p ≠ NULL dann {
20:   p.parent = r.parent;
21:   // Erzeugt einen Verweis von p auf seinen neuen Vorgänger (den Vorgänger von r)
22: }
23: // Hänge p anstelle von r in den Baum ein
24: falls r.parent == NULL dann {
25:   // r war Wurzel: neue Wurzel ist p
26:   root = p;
27: } sonst {
28:   // Hänge p an der richtigen Seite des Vorgängerknotens von r ein
29:   falls r == r.parent.leftchild dann {
30:     r.parent.leftchild = p; // p sei linker Nachfolger
31:   } sonst {
32:     r.parent.rightchild = p; // p sei rechter Nachfolger
33:   }
34: }
35: Gib Speicherplatz von r frei
```

Analyse: Die Laufzeit aller Operationen ist offensichtlich $O(h(T))$.

Im Extremfall, wenn der Baum zur linearen Liste degeneriert ist, dauert die Suche 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. Ein Beispiel für einen vollständig balancierten Baum stellt T_2 in Abb. 4.3 dar.

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.

4.2.2 Balancierte Suchbäume: AVL-Bäume

Balancierte Suchbäume lösen das Wörterbuchproblem sehr effizient, indem sie Extremfälle, wie sie für natürliche binäre Suchbäume auftauchen, vermeiden. Die Idee balancierter Suchbäume ist einfach: Man versucht, den Suchbaum von Zeit zu Zeit „effizient und elegant" auszubalancieren, um zu garantieren, dass seine Tiefe logarithmisch bleibt. Dazu gibt es verschiedene Möglichkeiten:

- höhenbalancierte Bäume: Die Höhen der Unterbäume unterscheiden sich um höchstens eine Konstante 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 (z.B. $a = 2$, $b = 4$).

Wir betrachten im Folgenden ein Beispiel zu höhenbalancierten Bäumen (AVL-Bäume) und eines zu (a,b)-Bäumen (B-Bäume, vgl. Abschnitt 4.3).

Zunächst untersuchen wir die sogenannten AVL-Bäume näher. AVL-Bäume wurden von G. M. Adelson-Velskii und Y. M. Landis im Jahr 1962 eingeführt.

Definition

- 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. 4.6).

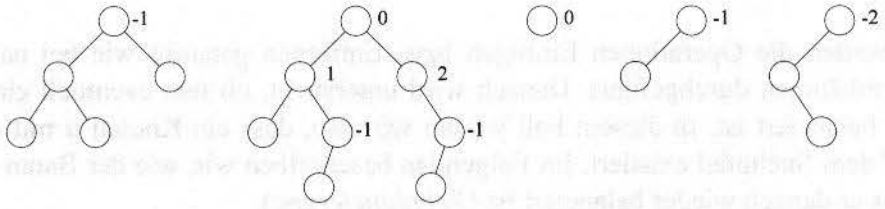


Abbildung 4.6: Beispiele für die Balance von Knoten in einem binären Baum.

- Zur Erinnerung: Die Höhe von v ist die Länge eines längsten Pfades in T von v zu einem Nachkommen von v. Die Höhe eines Baumes ist die Höhe seiner Wurzel.
- Zur Erinnerung: Die Tiefe von v ist definiert als die Länge des Pfades von v zur Wurzel (s. Abb. 4.7).
- Ein Knoten v heißt balanciert, falls $bal(v) \in \{-1, 0, +1\}$, sonst unbalanciert.
- Die Höhe eines leeren Baumes ist als -1 definiert.
- Ein *AVL-Baum* ist ein Baum, in dem alle Knoten balanciert sind.

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

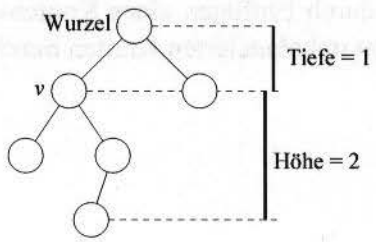


Abbildung 4.7: Höhe und Tiefe des Knotens v.

Das folgende wichtige Theorem beschränkt die Höhe von AVL-Bäumen und damit auch deren Suchtiefe. Da der Beweis aufwändig ist, wird es hier ohne Beweis zitiert.

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

Die Operationen *Suchen, 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 Einfügen und Entfernen, da hier eventuell die AVL-Baum-Eigenschaft verloren geht.

Zunächst werden die Operationen Einfügen bzw. Entfernen genauso wie bei natürlichen binären Suchbäumen durchgeführt. Danach wird untersucht, ob nun eventuell ein Knoten nicht mehr balanciert ist. In diesem Fall wissen wir also, dass ein Knoten u mit $bal(u) \in (-2, 2)$ auf dem Suchpfad existiert. Im Folgenden beschreiben wir, wie der Baum geändert wird, sodass er danach wieder balanciert ist (*Rebalancierung*).

Definition: Eine *AVL-Ersetzung* ist eine Operation (z.B. Einfügen, Löschen), die einen Unterbaum T eines Knotens durch einen modifizierten (gültigen) AVL-Baum T^* ersetzt, dessen Höhe um höchstens 1 von der Höhe von T abweicht.

In Abb. 4.8 wird beispielsweise der nur aus Knoten v bestehende Unterbaum von u durch den aus den Knoten v und z bestehenden, neuen Unterbaum ersetzt.

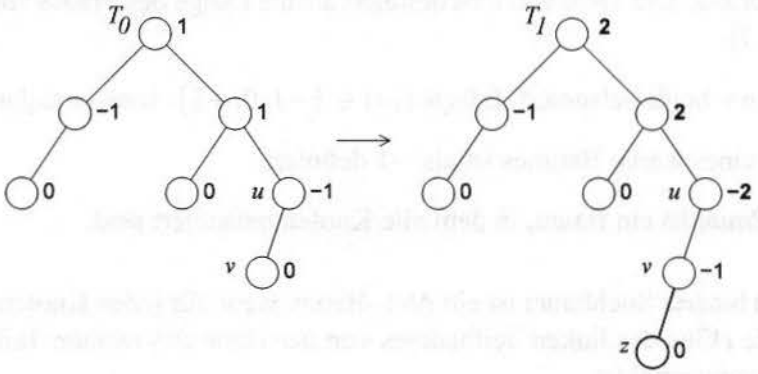


Abbildung 4.8: Ein Baum, der durch Einfügen eines Knotens z seine Balance-Eigenschaft verliert. Knoten u bezeichnet den unbalancierten Knoten maximaler Tiefe.

Rebalancierung

Sei T_0 ein gültiger AVL-Baum vor der AVL-Ersetzung (z.B. dem Einfügen oder Entfernen) und T_1 der unbalancierte Baum hinterher. Sei u der unbalancierte Knoten ($bal(u) \in \{-2, +2\}$) *maximaler Tiefe*. Seien nun 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: $bal(v) \subset \{-1, 0\}$, d.h. der linke Unterbaum des linken Kindes von u ist höher als oder gleich hoch wie der rechte (wobei eine Balance in v von 0 nur beim Löschen eines Knotens aus dem rechten Unterbaum von u auftreten kann, aber niemals beim Einfügen eines Knotens). Wir erreichen die Rebalancierung von u durch *einfache Rotation nach rechts* an u, d.h. u wird rechtes Kind von v. Das rechte Kind von v wird als linkes Kind an u abgegeben (vgl. Abb. 4.9). Der Name Rotation ist passend, weil man sich gut vorstellen kann, den Baum an u zu drehen. (Man stelle sich eine Uhr unter u vor und drehe diese im Uhrzeigersinn, d.h.

nach rechts. Dadurch wandert v nach oben und u nach unten. Der rechte Teilbaum von v fällt dabei von v nach u).
$\Rightarrow$ 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 im Fall $bal(v) = -1$ die Höhen der Teilbäume B und C gleich der Höhe von A minus eins sind, bei $bal(v) = 0$ besitzt Teilbaum B die gleiche Höhe wie A. Der Teilbaum mit Wurzel v ist also nach der Rotation balanciert (siehe Abb. 4.9). Sei T^* dieser neue Teilbaum.

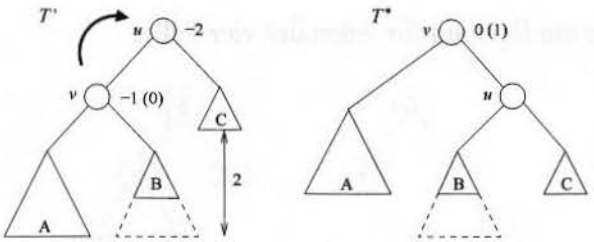


Abbildung 4.9: Rebalancierung durch Rotation nach rechts an Knoten u: der Pfeil im linken, unbalancierten Baum deutet die Richtung der Rotation an.

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 Rotation nach links an v und eine anschließende Rotation nach rechts an u. Dies nennt man auch eine *Doppelrotation links-rechts* (s. Abb. 4.10). 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 resultierende Teilbaum T^* mit Wurzel w ist demnach balanciert.

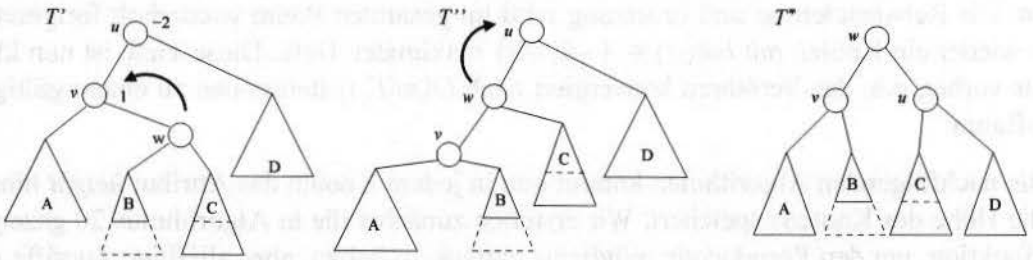


Abbildung 4.10: Rebalancierung durch Doppelrotation links-rechts an Knoten u: die beiden Pfeile in den unbalancierten Bäumen kennzeichnen die Richtung der Rotationen.

Fall 2: $bal(u) = 2$. Dieser Fall ist genau symmetrisch zu Fall 1. Man muss nur den Baum seitenverkehrt gespiegelt betrachten. Hier ist der rechte Unterbaum um 2 Ebenen höher als der linke. Sei also v das rechte Kind von u.

Fall 2.1: $bal(v) \in \{0, 1\}$, d.h. der rechte Unterbaum des rechten Kindes von u ist höher als oder gleich hoch wie der linke. Wir erreichen die Rebalancierung von u durch *einfache*

Rotation nach links an u, d.h. u wird linkes Kind von v. Das linke Kind von v wird als rechtes Kind an u abgegeben.

Fall 2.2: $bal(v) = -1$, d.h. der linke Unterbaum des rechten Kindes von u ist höher als der rechte. Dann existiert das linke Kind w von v. Wir rebalancieren durch eine Rotation nach rechts an v und eine anschließende Rotation nach links an u. Dies nennt man auch eine *Doppelrotation rechts-links*.

Abbildung 4.11 zeigt je ein Beispiel für jeden der vier Fälle.

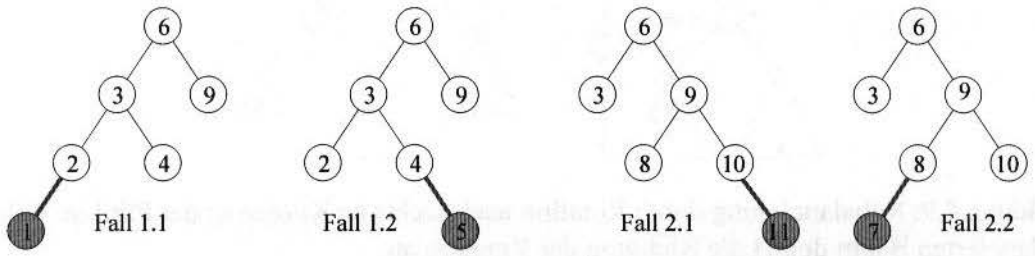


Abbildung 4.11: Vier Beispiele für AVL-Bäume, die durch das Einfügen eines Knotens aus der Balance geraten sind (die eingefügten Knoten sind hervorgehoben).

Für alle Rotationsformen gilt $|h(T^*) - h(T)| \leq 1$, d.h. der resultierende Teilbaum T^* hat entweder die gleiche Höhe oder ist um maximal eine Ebene kleiner oder größer als der Ausgangs-Teilbaum T von T_0 (vor dem ursprünglichen Einfügen oder Entfernen).

Daraus folgt: Alle Transformationen von T nach T^* sind als eine AVL-Ersetzung aufzufassen. Die Rebalancierung und Ersetzung wird im gesamten Baum wiederholt fortgesetzt: Sei u wieder ein Knoten mit $bal(u) \in \{-2, +2\}$ maximaler Tiefe. Diese Tiefe ist nun kleiner als vorher, d.h. das Verfahren konvergiert 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 erstellen zunächst die in Algorithmus 20 gezeigte Hilfsfunktion, um den Pseudocode möglichst einfach zu halten, aber allfällige Zugriffe auf nicht vorhandene Unterbäume zu verhindern.

Algorithmus 20 Height (u)

Eingabe: Teilbaum mit Wurzel u
Ausgabe: Höhe des Teilbaums

```
1: falls u == NULL dann {
2:     retourniere −1; // Teilbaum ist leer
3: } sonst {
4:     retourniere u.height; // gespeicherte Höhe zurückliefern
5: }
```

Algorithmus 21 zeigt den Pseudocode für das Einfügen eines Knotens in einen AVL-Baum. In Algorithmus 22 bzw. 23 werden die *einfache Rotation nach rechts* bzw. die *Doppelrotation links-rechts* in Pseudocode dargestellt.

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

Der Pseudocode zum Entfernen eines Knotens ist ähnlich, jedoch noch aufwändiger. Ist ein AVL-Baum nach dem Einfügen eines Knotens kurzzeitig nicht mehr balanciert, so genügt ein einziger Rebalancierungsschritt, um den Baum wieder zu balancieren (ohne Beweis). Beim Entfernen kann es hingegen sein, dass mehrere Rebalancierungsschritte notwendig sind. Dieser Effekt ist gut an dem folgenden Beispiel zu erkennen.

Entfernen eines Elementes aus einem AVL-Baum an einem Beispiel

Wir betrachten den AVL-Baum in Abb. 4.12. 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.

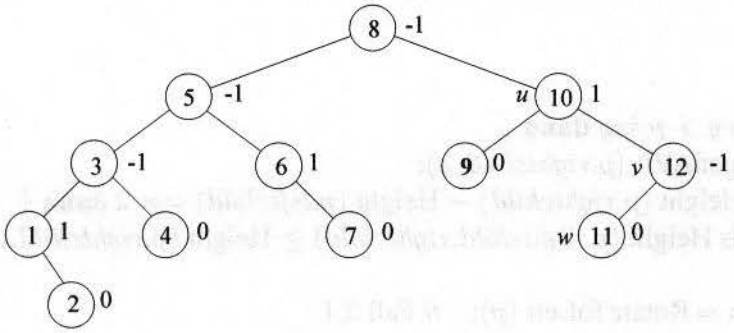


Abbildung 4.12: Ein AVL-Baum, aus dem wir Knoten 9 entfernen wollen.

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 $bal(v) = -1$, tritt Fall 2.2 ein. Sei w linkes Kind von v. Wir führen eine Doppelrotation rechts-links aus, d.h. zunächst eine einfache Rotation nach rechts an v, dann eine Rotation nach links an u (s. Abb. 4.13).

Es ergibt sich der in Abb. 4.14 dargestellte Baum.

Obwohl der Knoten 8 (u' in Abb. 4.14) 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 einfache Rotation nach rechts an u' (siehe Abb. 4.15).

Algorithmus 21 EinfügenAVL (**var** p, q)

Eingabe: Baum mit Wurzel p; Knoten q

```
1:  // Fügt Knoten q in AVL-Baum mit Wurzel p ein und führt Rebalancierung durch
2:  falls p == NULL dann {
3:     p = q; q.leftchild = q.rightchild = NULL; q.height = 0;
4:  } sonst {
5:     falls q.key < p.key dann {
6:        EinfügenAVL (p.leftchild, q);
7:        falls Height (p.rightchild) − Height (p.leftchild) == −2 dann {
8:           falls Height (p.leftchild.leftchild) ≥ Height (p.leftchild.rightchild) dann {
9:              // linker Unterbaum von p.leftchild ist höher (Gleichheit nicht möglich)
10:             p = RotateToRight (p);   // Fall 1.1
11:          } sonst {
12:             p = DoubleRotateLeftRight (p);   // Fall 1.2
13:          }
14:       }
15:    } sonst {
16:       falls q.key > p.key dann {
17:          EinfügenAVL (p.rightchild, q);
18:          falls Height (p.rightchild) − Height (p.leftchild) == 2 dann {
19:             falls Height (p.rightchild.rightchild) ≥ Height (p.rightchild.leftchild) dann
                {
20:                p = RotateToLeft (p);   // Fall 2.1
21:             } sonst {
22:                p = DoubleRotateRightLeft (p);   // Fall 2.2
23:             }
24:          }
25:       } sonst {
26:          // FERTIG: Knoten q schon vorhanden
27:       }
28:    }
29:    p.height = max (Height (p.leftchild), Height (p.rightchild)) + 1;
30: }
```

Algorithmus 22 RotateToRight (u)

Eingabe: Teilbaum mit Wurzel u
Ausgabe: Neue Wurzel v des rebalancierten Teilbaums
1: $v = u.leftchild$;
2: $u.leftchild = v.rightchild$; // u bekommt das rechte Kind von v
3: $v.rightchild = u$; // u wird zum neuen rechten Kind von v
4: // Berechne die Höhen der betroffenen Knoten neu
5: $u.height =$ max (Height $(u.leftchild)$, Height $(u.rightchild)) + 1$
6: $v.height =$ max (Height $(v.leftchild)$, Height $(u)) + 1$;
7: retourniere v; // Liefere die neue Wurzel des Teilbaums, v, zurück

Algorithmus 23 DoubleRotateLeftRight (u)

Eingabe: Teilbaum mit Wurzel u
Ausgabe: Neue Wurzel des rebalancierten Teilbaums
1: $u.leftchild =$ RotateToLeft $(u.leftchild)$;
2: retourniere RotateToRight (u);

Analyse

Rotationen und Doppelrotationen können in konstanter Zeit ausgeführt werden, da nur konstant viele Verweise 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:

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 und 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 der worst-case der sortierten Reihenfolge nicht erwartet wird, dann sind natürliche binäre Suchbäume vorzuziehen.

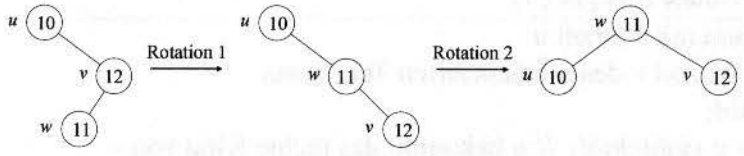


Abbildung 4.13: Doppelrotation rechts-links an u.

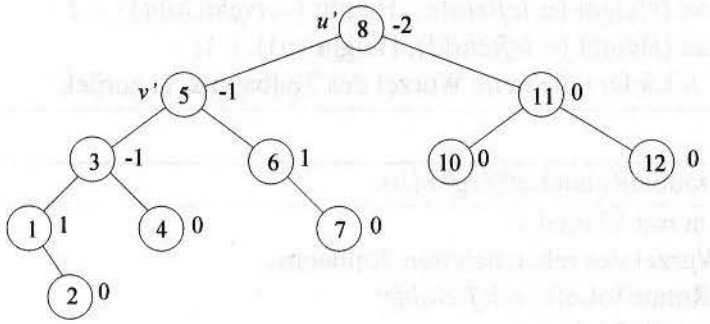


Abbildung 4.14: Nach der ersten Rotation ist der Baum immer noch nicht balanciert.

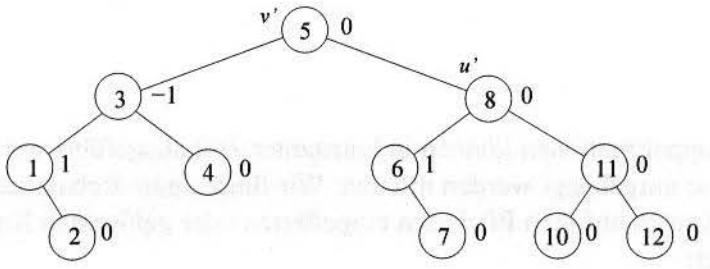


Abbildung 4.15: Nach der Rotation nach rechts an u' ist der Baum wieder balanciert.

4.3 B-Bäume und B*-Bäume

Die bisher behandelten Suchmethoden sind für „internes" Suchen gut geeignet, d.h. die Daten, in denen gesucht wird, finden vollständig im Hauptspeicher Platz. Oft ist dies jedoch nicht der Fall. Ist die Datenmenge sehr groß (wie z.B. oft bei Datenbanken), so werden die Daten auf Festplatten oder anderen externen Speichern abgelegt. Bei Bedarf werden dann Teile davon in den Hauptspeicher geladen. Ein solcher Zugriff benötigt deutlich mehr Zeit als ein Zugriff auf den Hauptspeicher.

Betrachten wir z.B. einen großen balancierten binären Suchbaum der Größe N und nehmen an, dass dieser extern gespeichert wurde. Wir gehen davon aus, dass die Verweise auf die linken und rechten Unterbäume jeweils Adressen auf dem externen Speichermedium darstellen. Wenn wir nun nach einem Schlüssel in diesem Baum suchen, würde dies ungefähr $\log_2 N$ Zugriffe auf das externe Speichermedium nach sich ziehen. Für $N = 10^6$ wären das

20 externe Speicherzugriffe. Wir können jedoch jeweils 7 Knoten des Suchbaumes zu einer Speicherseite (engl.: *page*) zusammenfassen, wie es in Abb. 4.16 gezeigt wird. Dabei nehmen wir an, dass jeweils mit einem externen Speicherzugriff eine gesamte Speicherseite in den Hauptspeicher geladen wird. In diesem Fall würde sich die Anzahl der externen Zugriffe dritteln, d.h. die Suche wäre ungefähr drei Mal so schnell.

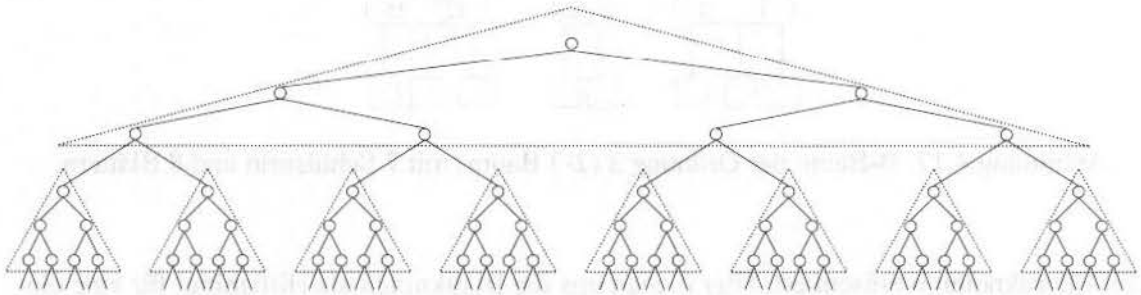


Abbildung 4.16: Binärer Suchbaum, bei dem jeweils 7 Knoten zu einer Speicherseite zusammengefasst wurden.

Durch das Gruppieren der Knoten in *pages* wird aus unserem binären Suchbaum mit jeweils zwei Kindern ein Suchbaum mit acht Kindern. In der Praxis wählt man die Größe einer *page* deutlich größer; oft erhält man dadurch Bäume mit 128 Kindern. Dies hat zur Folge, dass man z.B. in einem Baum mit $N = 10^6$ gespeicherten Schlüsseln mit nur drei externen Speicherzugriffen auskommt.

B-Bäume setzen diese Idee in die Praxis um, sodass effizientes Suchen, Einfügen und Entfernen von Datensätzen möglich ist; dabei bezieht sich die Effizienz insbesondere auf externe Speicherzugriffe.

4.3.1 B-Bäume

B-Bäume wurden 1972 von Bayer und McCreight entwickelt. In der Praxis wählt man die Speicherseitengröße so groß, dass mit einem externen Speicherzugriff genau eine Seite in den Hauptspeicher geladen werden kann. Der B-Baum wird so aufgebaut, dass jeder Knoten des B-Baums genau einer Speicherseite entspricht. Jeder Knoten enthält jeweils mehrere Schlüssel und Verweise auf weitere Knoten.

Man kann sich B-Bäume als natürliche Verallgemeinerung von binären Suchbäumen vorstellen. Der wesentliche Unterschied ist: Die inneren Knoten in B-Bäumen enthalten i.A. mehr als zwei Kinder; damit hier effiziente Suche möglich ist, enthalten sie auch mehrere Schlüssel, nämlich genau einen weniger als Kinder. Es folgt die formale Definition von B-Bäumen. Dabei ist ein *Blatt* eines B-Baumes ein Knoten ohne Inhalt. In einer realen Implementation werden Blätter nicht wirklich gespeichert, sondern durch NULL-Verweise in

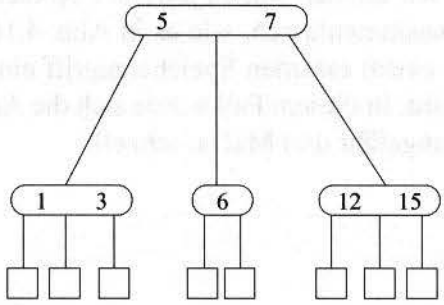


Abbildung 4.17: B-Baum der Ordnung 3 (2-3 Baum) mit 7 Schlüsseln und 8 Blättern.

den Elternknoten repräsentiert. Hier dienen uns die Blattknoten als Hilfsmittel für eine einfachere Definition und Analyse.

Definition: Ein B-Baum der Ordnung m ist ein Baum mit folgenden Eigenschaften:

- Alle Blätter haben die gleiche Tiefe.
- Jeder Knoten mit Ausnahme der Wurzel und der Blätter hat mindestens $\lceil \frac{m}{2} \rceil$ Kinder. Die Wurzel hat mindestens 2 Kinder (wenn der B-Baum nicht leer ist).
- Jeder Knoten hat höchstens m Kinder.
- Jeder Knoten mit $l+1$ Kindern hat l Schlüssel.
- Für jeden Knoten r mit l Schlüsseln $s_1, \ldots, s_l$ und $l+1$ Kindern $v_0, \ldots, v_l$ ($\lceil \frac{m}{2} \rceil \leq l+1 \leq m$) gilt: Für jedes $i, 1 \leq i \leq l$, sind alle Schlüssel in $T_{v_{i-1}}$ kleiner gleich s_i, und s_i ist kleiner gleich alle Schlüssel in T_{v_i}. Dabei bezeichnet T_{v_i} den Teilbaum mit Wurzel v_i.

Abbildung 4.17 zeigt einen B-Baum der Ordnung 3. Diese Bäume werden auch 2-3 Bäume genannt, weil jeder Knoten (bis auf die Blätter) entweder 2 oder 3 Kinder besitzt.

Es ist zweckmäßig sich vorzustellen, dass die l Schlüssel und die $l+1$ Verweise auf die Kinder von v wie in Abb. 4.18 innerhalb des Knotens p angeordnet sind:

v0	s1	v1	s2	v2	...	sl	vl

Abbildung 4.18: Ein Knoten v eines B-Baums.

Wir implementieren B-Bäume als verallgemeinerte Listenstruktur. Jeder innere Knoten w enthält die folgenden Informationen:

- $w.l$ – Anzahl der Schlüssel
- $w.key[1], \ldots, w.key[l]$ – Schlüssel $s_1, \ldots s_l$
- $w.info[1], \ldots, w.info[l]$ – Datenfelder zu Schlüsseln $s_1, \ldots, s_l$
- $w.child[0], \ldots, w.child[l]$ – Verweise auf Kinderknoten $v_0, \ldots, v_l$

Blätter markieren wir hier durch $w.l = 0$.

Eigenschaften von B-Bäumen

Behauptung 1: Die Anzahl der Blätter in einem B-Baum T ist um 1 größer als die Anzahl der Schlüssel in T.

Beweis: Induktion über die Höhe h von B-Bäumen. $h = 1$: Hat der Baum die Höhe 1, dann besteht er aus der Wurzel und k Blättern mit $2 \leq k \leq m$. Die Wurzel besitzt in diesem Fall laut Definition $k - 1$ Schlüssel.

$h \longrightarrow h + 1$ (Induktionsschluss): Sei T ein Baum der Höhe $h + 1$ und seien $T_1, \ldots, T_k$, $2 \leq k \leq m$, die k Teilbäume von T gleicher Höhe h und jeweils $n_1, \ldots, n_k$ Blättern, so besitzen sie nach Induktionsvoraussetzung $(n_1 - 1), \ldots, (n_k - 1)$ Schlüssel. Laut Definition muss die Wurzel $k - 1$ Schlüssel besitzen. Daraus errechnen sich für den Baum $\sum_{i=1}^{k} n_i$ Blätter und $\sum_{i=1}^{k}(n_i - 1) + (k - 1) = (\sum_{i=1}^{k} n_i) - 1$ Schlüssel.

Behauptung 2: Ein B-Baum der Ordnung m mit gegebener Höhe h hat die minimale Blattanzahl $n_{\min} = 2\lceil \frac{m}{2} \rceil^{h-1}$ und maximale Blattanzahl $n_{\max} = m^h$.

Beweis: Die minimale Blattanzahl wird erreicht, wenn die Wurzel 2 und alle anderen inneren Knoten nur $\lceil \frac{m}{2} \rceil$ Kinder besitzen, die maximale, wenn alle Knoten m Kinder besitzen.

Behauptung 3: Für die Höhe h eines B-Baumes der Ordnung m mit N Schlüsseln gilt:

$$h \leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left(\frac{N+1}{2} \right) \text{ und } h \geq \log_m (N+1)$$

Beweis: Die Anzahl der Blätter eines B-Baumes mit N Schlüsseln ist $N + 1$ und durch $n_{\min}$ bzw. $n_{\max}$ nach unten bzw. oben beschränkt.

Damit gilt auch für *B-Bäume die für balancierte Suchbäume typische Eigenschaft*, nämlich dass ihre Höhe logarithmisch in der Anzahl der gespeicherten Schlüssel beschränkt ist.

Suchen, Einfügen und Entfernen in B-Bäumen

Suchen: Das Suchen nach einem Schlüssel x in einem B-Baum der Ordnung m kann als natürliche Verallgemeinerung des von binären Suchbäumen bekannten Verfahrens aufgefasst werden.

Man beginnt bei der Wurzel p und sucht in p die Stelle i, die den kleinsten Schlüssel größer gleich x enthält. Falls diese existiert, und $p.key[i]$ ungleich x ist, dann sucht man im Knoten von Verweis $p.child[i-1]$ weiter. Falls dieser nicht existiert und wir in einem Blatt angekommen sind, wird „NULL" bzw. „nicht gefunden" zurückgegeben.

Eine mögliche Implementierung zeigt Algorithmus 24.

Algorithmus 24 Suche (p, x)

Eingabe: B-Baum mit Wurzel p; Schlüssel x
Ausgabe: Knoten mit Schlüssel x oder NULL, falls x nicht vorhanden ist
Variable(n): Zählvariable i

```
1: i = 1;
2: solange i ≤ p.l ∧ x > p.key[i] {
3:    i = i + 1;
4: }
5: falls i ≤ p.l ∧ x == p.key[i] dann {
6:    retourniere (p, i); // Schlüssel gefunden
7: }
8: falls p.l == 0 dann {
9:    retourniere NULL; // Blatt erreicht: Suche erfolglos
10: } sonst {
11:    retourniere Suche (p.child[i − 1], x); // rekursiv weitersuchen
12: }
```

Beachten Sie insbesondere folgenden Fall: Ist x größer als der größte Schlüssel s_l in p, so ergibt sich $i = l + 1$ und damit wird die Suche im Knoten von Verweis $p.child[l]$ fortgesetzt.

In dieser Implementierung wird ein Knoten jeweils linear durchsucht. Bei großem m ist es jedoch vorteilhaft, diese Suche als binäre Suche durchzuführen.

Einfügen: Um einen neuen Schlüssel x in einen B-Baum *einzufügen*, sucht man zunächst nach x. Da x im Baum noch nicht vorkommt, endet die Suche erfolglos in einem Blatt p_0, das die erwartete Position des Schlüssels repräsentiert. Sei p der Vorgänger von p_0 und $p.child[i]$ zeige auf p_0. Wir fügen zunächst Schlüssel x zwischen den Schlüsseln s_i und s_{i+1} in p ein und erzeugen ein neues Blatt.

Falls p nun zu viele Kinder bzw. Schlüssel besitzt, müssen wir p in zwei verschiedene Knoten aufspalten. Sind $s_1, \ldots, s_m$ die Schlüssel, so bildet man zwei neue Knoten, die jeweils die Schlüssel $s_1, \ldots, s_{\lceil \frac{m}{2} \rceil - 1}$ und $s_{\lceil \frac{m}{2} \rceil + 1}, \ldots, s_m$ enthalten, und fügt den mittleren Schlüssel

$s_{\lceil \frac{m}{2} \rceil}$ auf dieselbe Weise in den Vorgänger des Knotens p ein. Dieses Teilen eines übergelaufenen Knotens wird solange rekursiv wiederholt, bis ein Knoten erreicht ist, der noch nicht die Maximalzahl von Schlüsseln gespeichert hat, oder bis die Wurzel erreicht ist. Muss die Wurzel geteilt werden, dann erzeugt man eine neue Wurzel, welche die durch Teilung entstehenden Knoten als Kinder und den mittleren Schlüssel $s_{\lceil \frac{m}{2} \rceil}$ als einzigen Schlüssel hat.

Abbildung 4.19 zeigt ein Beispiel des Einfügens von Schlüssel „14" in den B-Baum von Abb. 4.17. Die Suche nach 14 endet erfolglos in dem Blatt vor Schlüssel 15. Zunächst wird Schlüssel 14 also in den Knoten p_0 zwischen den Schlüsseln 12 und 15 zusammen mit seinem neuen Blatt eingefügt (s. Abb. 4.19 (a)). Nun besitzt p_0 vier $(= m + 1)$ Kinder, muss also *aufgespalten* werden, was im nächsten Schritt geschieht: der mittlere Schlüssel wandert dabei nach oben und der Rest von p_0 wird in zwei Knoten aufgeteilt (s. Abb. 4.19 (b)). Nun besitzt der Vorgänger von p_0 wiederum zu viele Schlüssel und muss somit auch aufgeteilt werden; dadurch entsteht eine neue Wurzel, die den Schlüssel 7 enthält (s. Abb. 4.19 (c)). Der B-Baum ist also in seiner Höhe um 1 gewachsen.

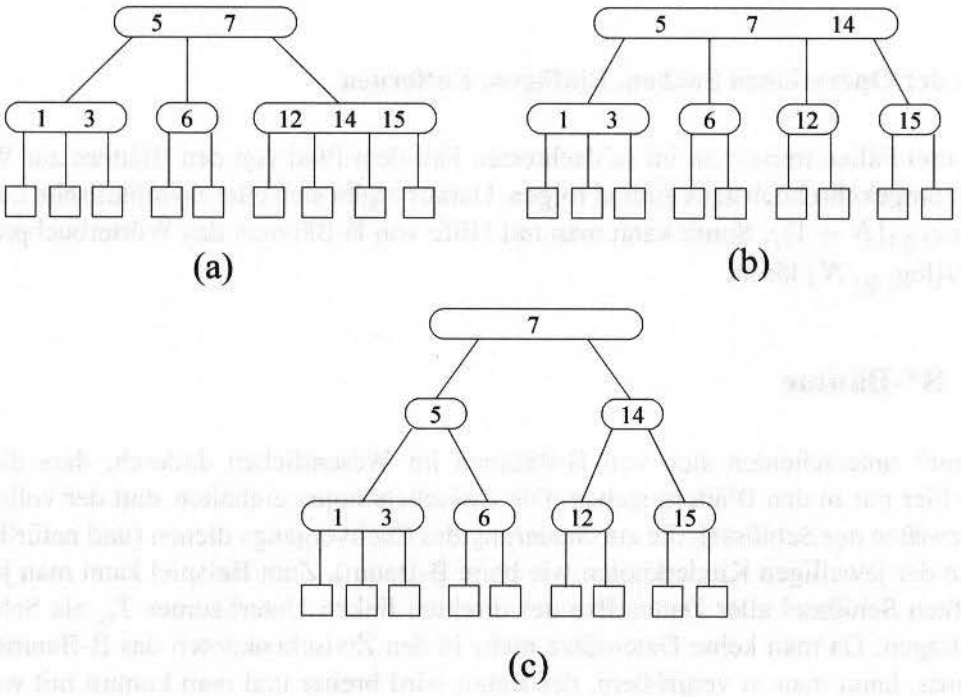


Abbildung 4.19: Einfügen von 14 erzeugt einen zunächst ungültigen B-Baum der Ordnung 3 (a), der dann durch Knotenspaltung (b) und (c) wieder korrigiert wird. Dabei wächst der Baum um eine Ebene.

Wichtig: **B-Bäume wachsen** im Unterschied zu binären Suchbäumen **immer nach oben**, d.h. nur durch Aufspaltung der Wurzel!

Entfernen: Zum Entfernen eines Schlüssels aus einem B-Baum der Ordnung m sucht man den Eintrag zunächst. Liegt dieser in einem inneren Knoten, der weitere innere Knoten als

Unterbäume besitzt, so reduziert man diese Situation auf das Entfernen eines Schlüssels in der tiefsten Ebene, indem der Schlüssel mit einem geeigneten Ersatzschlüssel vertauscht wird: In Frage kommen der kleinste im unmittelbar rechts anschließenden Unterbaum bzw. der größte im unmittelbar links anschließenden Unterbaum. (Dieses Vorgehen ist mit dem Löschen eines Knotens in einem natürlichen binären Suchbaum vergleichbar, wenn dieser Knoten zwei Nachfolger besitzt!) Wurde der Schlüssel aus einem Knoten in der tiefsten Ebene entfernt, so kann es vorkommen, dass dieser Knoten p nun „zu wenige" Schlüssel besitzt. Hierfür gibt es folgende Lösungsmöglichkeiten:

(a) Falls ein Geschwisterknoten (= Nachbarknoten derselben Ebene) ausreichend Schlüssel beinhaltet, übernimmt man Schlüssel von diesem. Zu beachten ist, dass dabei auch ein oder mehrere Einträge in den Elternknoten entsprechend angepasst werden müssen.

(b) Man verschmilzt p mit einem Geschwisterknoten. Das bedeutet wiederum, dass die Einträge im Elternknoten geeignet angepasst werden müssen.

Analyse der Operationen Suchen, Einfügen, Entfernen

In allen drei Fällen muss man im schlechtesten Fall dem Pfad von den Blättern zur Wurzel und/oder umgekehrt höchstens einmal folgen. Daraus ergibt sich eine asymptotische Laufzeit von $O(\log_{\lceil \frac{m}{2} \rceil}(N+1))$. Somit kann man mit Hilfe von B-Bäumen das Wörterbuchproblem in Zeit $O(\log_{\lceil \frac{m}{2} \rceil} N)$ lösen.

4.3.2 B*-Bäume

B*-Bäume[1] unterscheiden sich von B-Bäumen im Wesentlichen dadurch, dass die Datensätze hier nur in den Blättern stehen. Die Zwischenknoten enthalten statt der vollständigen Datensätze nur Schlüssel, die zur Steuerung des Suchvorgangs dienen (und natürlich die Adressen der jeweiligen Kinderknoten wie beim B-Baum). Zum Beispiel kann man jeweils den größten Schlüssel aller Datensätze des direkten linken Unterbaumes T_{v_i} als Schlüssel s_{i+1} eintragen. Da man keine Datensätze mehr in den Zwischenknoten des B-Baums speichern muss, kann man m vergrößern, der Baum wird breiter und man kommt mit weniger Ebenen aus. Die Zwischenknoten enthalten wie beim B-Baum mindestens $\lceil m/2 \rceil$ und höchstens m Nachfolger (Seitenadressen) und einen Schlüssel weniger als Nachfolger. Die Blätter sind anders eingeteilt: sie enthalten mindestens k^* und höchstens $2k^*$ Datensätze und keine Seitenadressen. Abbildung 4.20 zeigt ein Beispiel. Die Operationen sind sehr ähnlich wie beim B-Baum.

Suchen: Man muss im Unterschied zum B-Baum in jedem Fall bis zu einem Blatt gehen. Das erhöht die mittlere Anzahl von Schritten jedoch kaum.

[1] In der Literatur wird der Begriff des B*-Baums unterschiedlich verwendet. Für die hier beschriebene Datenstruktur ist diese Bezeichnung aber wahrscheinlich am weitesten verbreitet.

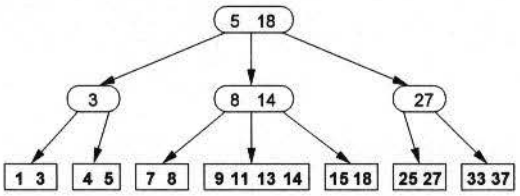


Abbildung 4.20: Ein B*-Baum mit $m = 3$ und $k^* = 2$.

Einfügen: Kleinere Unterschiede zum B-Baum ergeben sich dadurch, dass es nur einen trennenden Schlüssel (keine trennenden Datensätze) gibt.

Entfernen: Hier liegt der zu entfernende Datensatz immer in einem Blatt. Bei Operationen in Zwischenknoten ist zu beachten, dass dort keine Datensätze, sondern nur Schlüssel stehen.

Kapitel 5

Hashverfahren

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.

In etwa vergleichbar ist die grundlegende Idee der Hashverfahren mit einem Adressbüchlein, in dem man seine persönlichen Adressen verwaltet: Alle Einträge für Namen mit gleichem Anfangsbuchstaben befinden sich auf jeweils einer eigenen, sehr rasch zugreifbaren Seite. Innerhalb dieser Seite sind die Einträge aber normalerweise nicht weiter nach einem speziellen Kriterium geordnet. Da im Normalfall durch die Aufsplittung nach dem Anfangsbuchstaben auf einer Seite nicht sehr viele Einträge stehen, ist die Suche effizient möglich.

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 (Schlüssel: C-Identifier). Sie sind immer dann nützlich, wenn die Anzahl der Schlüssel nicht zu groß ist und wenn häufiger gesucht als eingefügt und entfernt wird.

Wir legen eine Tabellengröße m fest, die typischerweise etwas größer ist (rund 20%) als die Anzahl der einzufügenden Schlüssel n, und wählen eine Hashfunktion $h : U \to \{0, \ldots, m-1\}$. Die einzufügenden Schlüssel $I \in U$ werden in $T[h(I)]$ gespeichert. Abbildung 5.1 illustriert die Idee der Hashverfahren.

Der große Vorteil dieser Methode im Vergleich zu den bisherigen Suchmethoden ist, dass hier die durchschnittliche 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 $h(I_1)$ und $h(I_2)$ kollidieren (eine Kollision bedeutet $h(I_1) = h(I_2)$ für $I_1 \neq I_2$)

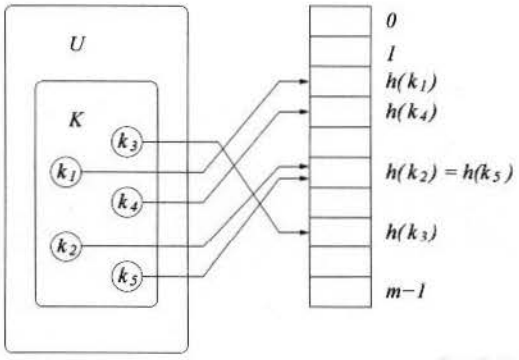


Abbildung 5.1: Hashing.

Datenstruktur: Die sogenannte *Hashtabelle* wird als ein Array $T[0], \ldots, T[m-1]$ der Größe m realisiert. Die Hashtabelle speichert m Einträge mit Hashadressen $0, \ldots, m-1$.

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 Kollisionsvermeidung bzw. Kollisionsbehandlung. Damit beschäftigen wir uns in den Abschnitten 5.2 und 5.3. Die Stichworte hierzu lauten Hashing mit Verkettung der Überläufer und Offene Hashverfahren.

5.1 Zur Wahl der Hashfunktion

Ziel: Die Hashadressen sollten möglichst gleich verteilt in $\{0, 1, \ldots, m-1\}$ sein. Hashfunktionen sollten auch Häufungen fast gleicher Schlüssel (z.B. i, $i+1$, $i+2$, j, $j+1$) möglichst gleichmäßig auf den Adressbereich streuen.

Generelle Annahme: Die Schlüssel k sind nichtnegative ganze Zahlen. Es ist immer möglich, Schlüssel so zu interpretieren, z.B:

Character Strings: Jeder Character hat im ASCII-Code eine Nummer im Bereich $\{0, \ldots, 127\}$, z.B.

$$\text{'p'} \cong 112, \quad \text{'t'} \cong 116, \quad \Rightarrow \quad \text{"pt"} \cong 112 \cdot 128 + 116 = 14452$$

bzw. allgemein für beliebig lange ASCII-Strings $(s_1, \ldots, s_l)$:

$$k = \sum_{i=1}^{l} 128^{l-i} \cdot \text{ord}(s_i) \tag{5.1}$$

Allerdings ist zu beachten, dass diese Methode für lange Strings zu sehr großen Zahlen führen kann!

5.1.1 Die Divisions-Rest-Methode

Die Hashfunktion der **Divisions-Rest-Methode** ist gegeben durch:

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

Eigenschaften

(1) Die Hashfunktion kann sehr schnell berechnet werden.

(2) Die richtige Wahl von m (Tabellengröße) ist hier 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.: $m = 2^7 - 1 = 127$:

$$pt = (112 \cdot 128 + 116) \bmod 127 = 14452 \bmod 127 = 101$$

$$tp = (116 \cdot 128 + 112) \bmod 127 = 14960 \bmod 127 = 101$$

Dasselbe passiert, wenn in einem längeren String zwei Buchstaben vertauscht werden. Letztlich wird in diesem Fall die Summe der beiden Buchstaben modulo m berechnet.

Eine gute Wahl für m ist eine Primzahl, die kein $r^i \pm j$ teilt und weit weg von einer Zweierpotenz ist. Diese Wahl hat sich in der Praxis gut bewährt.

Beispiel: Eine Hashtabelle soll ca. 550 Einträge aufnehmen, die Schlüssel sind *character strings*, interpretiert als 2^8-adische Zahlen. Eine gute Wahl wäre hier z.B. $m = 701$, da $2^9 = 512$ und $2^{10} = 1024$.

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:

$$k = (\ldots(s_1 \cdot 128 + s_2) \cdot 128 + s_3) \cdot 128 + \ldots + s_{l-1}) \cdot 128 + s_l$$

Es gilt:

$$\begin{aligned} k \bmod m \;=\; & (\ldots(s_1 \cdot 128 + s_2) \bmod m) \cdot 128 + s_3) \bmod m) \cdot 128 + \ldots + \\ & + s_{l-1}) \bmod m) \cdot 128 + s_l) \bmod m \end{aligned}$$

Durch die wiederholt vorgezogene modulo-Rechnung wird der Bereich immer möglichst rasch wieder eingeschränkt, und es treten keine Werte größer als $(m-1) \cdot 128 + 127$ auf.

5.1.2 Die Multiplikationsmethode

Die Hashfunktion der **Multiplikationsmethode** lautet:

$$\begin{aligned} h(k) &= \lfloor m(k \cdot A \bmod 1) \rfloor \\ &= \lfloor m \underbrace{(k \cdot A - \lfloor k \cdot A \rfloor)}_{\in [0,1)} \rfloor \end{aligned}$$

mit $0 < A < 1$. $(k \cdot A - \lfloor k \cdot A \rfloor)$ heißt auch der „gebrochene Teil" von $k \cdot A$).

Eigenschaften

(1) Die Wahl von m ist hierbei unkritisch.

(2) Gleichmäßige Verteilung für $U = \{1, 2, \ldots, n\}$ bei guter Wahl von A.

Gute Wahl für A

Irrationale Zahlen sind eine gute Wahl, denn:

Satz: 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 (Beweis: Vera Turan Sos [1957]).

Beste Wahl für A nach Knuth [1973]: Der goldene Schnitt

$$A = \Phi^{-1} = \frac{\sqrt{5}-1}{2} = 0.6180339887...$$

ist bekannt als der *goldene Schnitt* (siehe Abbildung 5.2).

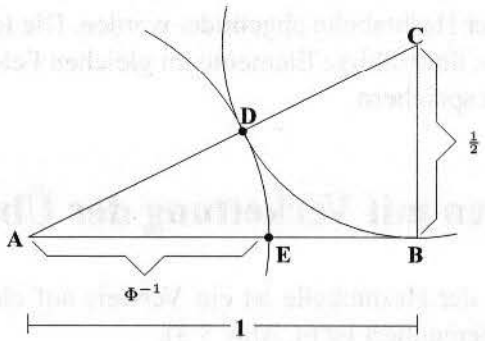


Abbildung 5.2: Der goldene Schnitt.

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. $x^2 = a(a-x)$. 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. In der Renaissance gab es eine Wiederbelebung.

Beispiel: (Dezimalrechnung) $k = 123\,456$, $m = 10\,000$, $A = \Phi^{-1}$

$$\begin{aligned} h(k) &= \lfloor 10\,000 \cdot (123\,456 \cdot 0.61803... \bmod 1) \rfloor \\ &= \lfloor 10\,000 \cdot (76\,300.0041151... \bmod 1) \rfloor \\ &= \lfloor 10\,000 \cdot 0.0041151... \rfloor \\ &= \lfloor 41.151... \rfloor \\ &= 41 \end{aligned}$$

Diskussion

- Eine gute Wahl für m wäre hier $m = 2^i$. Dann kann $h(k)$ effizient berechnet werden (eine einfache Multiplikation und ein bis zwei Shifts).

- Eine empirische Untersuchung ergab, dass die Divisions-Rest-Methode der Multiplikationsmethode überlegen ist.

In den folgenden Abschnitten diskutieren wir verschiedene Möglichkeiten der Kollisionsbehandlung und -vermeidung. Eine Kollision tritt dann auf, wenn zwei verschiedene Schlüssel auf die gleiche Adresse in der Hashtabelle abgebildet werden. Die Idee der Methode mit Verkettung der Überläufer ist es, überzählige Elemente im gleichen Feld der Hashtabelle einfach in einer verketteten Liste zu speichern.

5.2 Hashverfahren mit Verkettung der Überläufer

Methode: Jedes Element der Hashtabelle ist ein Verweis auf eine Überlaufkette, die als verkettete lineare Liste implementiert ist (s. Abb. 5.3).

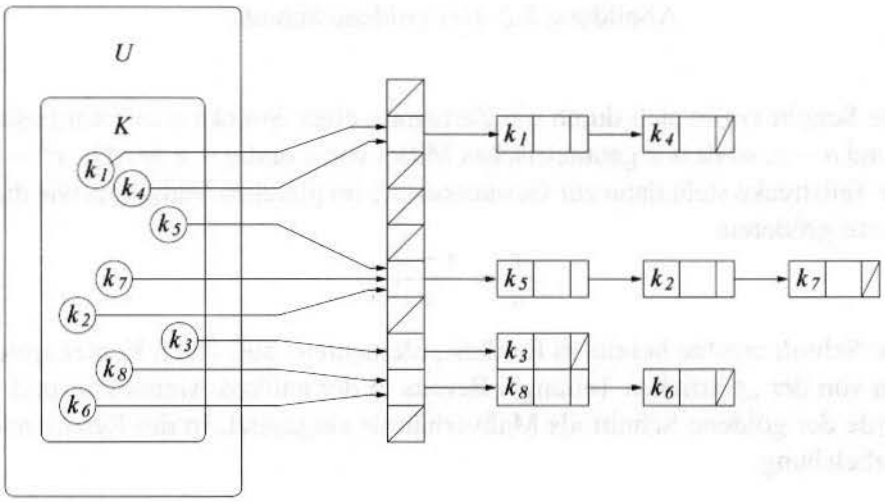


Abbildung 5.3: Hashing mit Verkettung der Überläufer.

Datenstruktur: Wir wählen als Datenstruktur eine einfach verkettete Liste ohne Dummy-Elemente, die durch *L.key, L.info* und *L.next* ansprechbar sind.

Pseudocode Hashing mit Verkettung der Überläufer Die Algorithmen 25, 26, 27 und 28 demonstrieren die Implementierung des Verfahrens im Pseudocode.

Analyse

Wir analysieren zunächst die Operation *Suchen*. Wir nehmen an, dass sich in der Hashtabelle n Schlüssel befinden.

Algorithmus 25 Initialisiere (**var** T, m)

Eingabe: Hashtabelle T; Elementanzahl m
Ausgabe: initialisierte Hashtabelle T
Variable(n): Index i

```
1: für i = 0, ..., m − 1 {
2:     T[i] = NULL;
3: }
```

Algorithmus 26 Einfügen (**var** T, p)

Eingabe: Hashtabelle T; einzufügendes Element p
Ausgabe: Hashtabelle T mit neu eingetragenem Element p
Variable(n): Hashindex pos

```
1: pos = hash(p.key);
2: p.next = T[pos];
3: T[pos] = p;
```

Algorithmus 27 Suchen (T, k)

Eingabe: Hashtabelle T; gesuchter Schlüssel k
Ausgabe: gesuchtes Element p

```
1: p = T[hash(k)];
2: solange p ≠ NULL ∧ p.key ≠ k {
3:     p = p.next;
4: }
5: retourniere p;
```

Algorithmus 28 Entfernen (**var** T, k)

Eingabe: Hashtabelle T; Schlüssel k des zu entfernenden Elementes (wir wissen bereits, dass ein Element mit dem gesuchten Schlüssel in T enthalten ist)
Ausgabe: Element mit dem Schlüssel k wurde aus T entfernt
Variable(n): Hashindex pos; Elemente p und q

```
1: pos = hash(k);
2: q = NULL;
3: p = T[pos];
4: solange p.key ≠ k {
5:     q = p;
6:     p = p.next;
7: }
8: falls q == NULL dann {
9:     T[pos] = T[pos].next;
10: } sonst {
11:     q.next = p.next;
12: }
```

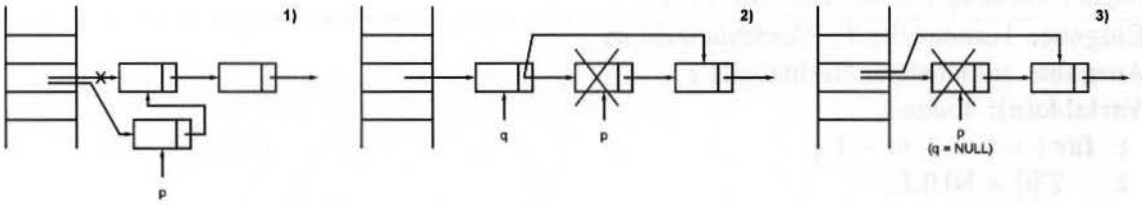


Abbildung 5.4: Das Einfügen und das Entfernen eines Elementes.

Definition: Wir nennen die durchschnittliche Anzahl von Elementen in einer Überlaufkette $\alpha = \frac{n}{m}$ den **Auslastungsfaktor** (Belegungsfaktor).

Worst-Case: $C_{worst}(n) = \Theta(n)$. In diesem Fall bildet die Hashfunktion alle Elemente auf denselben Platz ab.

Average-Case: Hier 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 Berechnung von $h(k)$ benötigt konstante Zeit.

Wir zählen die Anzahl der Schlüsselvergleiche.

Erfolglose Suche: offensichtlich $C_{avg}(n) = \alpha$ (die Listen sind im Durchschnitt α Elemente lang und müssen bis zum Ende durchlaufen werden).

Erfolgreiche Suche: Wir mitteln über alle n eingefügten Elemente, wobei zumindest immer ein Vergleich notwendig ist (dabei ist $\frac{i-1}{m}$ die durchschnittliche Länge einer Liste zu dem Zeitpunkt, als Element i in die Hashtabelle eingefügt wurde):

$$\begin{aligned} C_{avg}(n) &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) \\ &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1) \\ &= 1+\frac{1}{nm}\cdot\frac{n(n-1)}{2} \\ &= 1+\frac{n-1}{2m} \\ &= 1+\frac{n}{2m}-\frac{1}{2m} \\ &= 1+\frac{\alpha}{2}-\frac{1}{2m} \end{aligned}$$

Daraus kann man ablesen, dass die Operation *Suchen* 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 Analyse der Operation *Entfernen* bringt das gleiche Ergebnis. Die Operation *Einfügen* hingegen ist immer in Zeit $O(1)$ möglich wenn am Listenanfang eingefügt wird (und wir annehmen, dass die Hashfunktion in konstanter Zeit berechnet werden kann).

Diskussion

+ Belegungsfaktor von mehr als 1 ist möglich.
+ Echte Entfernungen von Einträgen sind möglich.
+ Eignet sich für den Einsatz mit Externspeichern (Hashtabelle im Internspeicher).
– Zu den Nutzdaten kommt der Speicherplatzbedarf für die Verweise.
– Es wird selbst dann Platz für Überläufer außerhalb der Hashtabelle benötigt, wenn in der Tabelle noch viele Plätze frei sind.

5.3 Offene Hashverfahren

Idee: Alle Elemente werden – im Gegensatz zum vorigen Verfahren – in der Hashtabelle gespeichert. Wenn ein Platz belegt ist, so werden in einer bestimmten Reihenfolge weitere Plätze ausprobiert. Diese Reihenfolge nennt man **Sondierungsreihenfolge**.

Dazu erweitern wir die Hashfunktion auf zwei Argumente:

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Die Sondierungsreihenfolge ergibt sich dann durch

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle.$$

Diese sollte eine Permutation von $\langle 0, 1, \ldots, m-1 \rangle$ sein.

Annahme: Die Hashtabelle enthält immer wenigstens einen unbelegten Platz.

Eigenschaften:

(1) Stößt man bei der Suche auf einen unbelegten Platz, so kann die Suche erfolglos abgebrochen werden.

(2) Wegen (1) wird nicht wirklich entfernt, sondern nur als „entfernt" markiert. Beim Einfügen wird ein solcher Platz als „frei", beim Suchen als „wieder frei" (d.h. verfügbar, aber früher schon einmal belegt) betrachtet.

(3) **Nachteil:** Wegen (2) ist die Suchzeit nicht mehr proportional zu $\Theta(1+\alpha)$, deshalb sollte man, sofern man viele Entfernungen vornehmen muss, die Methode der Verkettung der Überläufer vorziehen.

Sondierungsreihenfolgen

Ideal wäre das „uniform Hashing": Jeder Schlüssel erhält 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.

5.3.1 Lineares Sondieren

Gegeben ist eine normale Hashfunktion

$$h' : U \to \{0, 1, \ldots, m-1\}.$$

Wir definieren für $i = 0, 1, \ldots, m-1$

$$h(k,i) = (h'(k) + i) \bmod m.$$

Kritik

- Es gibt nur m verschiedene Sondierungsfolgen, da die erste Position die gesamte Sequenz festlegt:

$$h'(k), h'(k)+1, \ldots, m-1, 0, 1, \ldots, h'(k)-1.$$

- Lange belegte Teilstücke tendieren dazu, schneller zu wachsen als kurze. Dieser unangenehme Effekt wird **Primäre Häufungen** genannt.

Beispiel: $m = 8$, $h'(k) = k \bmod m$

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

Analyseergebnisse

Für die Anzahl der Sondierungen im Durchschnitt gilt für $(0 < \alpha = \frac{n}{m} < 1)$ (ohne Beweis):

Erfolglose Suche $\approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$

Erfolgreiche Suche $\approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$

Erweiterung: Konstante Schrittweite > 1

Eine Verallgemeinerung der Hashfunktion $h(k, i)$ mit linearem Sondieren verwendet eine zusätzliche Konstante $c \in \mathbb{N}$, $c \geq 1$, zur Steuerung der Schrittweite bei der Suche nach einem freien Platz in der Hashtabelle:

$$h(k, i) = (h'(k) + i \cdot c) \bmod m.$$

Bei der Verwendung einer Schrittweite $\neq 1$ ist aber sicherzustellen (zum Beispiel durch geeignete Wahl der Tabellengröße), dass auch wirklich alle Plätze der Hashtabelle sondiert werden. Primäre Häufungen können auf diesem Wege allerdings nicht reduziert oder gar vermieden werden.

5.3.2 Quadratisches Sondieren

Die Hashfunktion für **Quadratisches Sondieren** lautet:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Beispiel: $m = 8$, $h'(k) = k \bmod m$, $c_1 = c_2 = \frac{1}{2}$, gleiche Schlüssel 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

$$\begin{aligned} 14 \to 6 \quad &\to \quad 6 + \frac{1}{2}(1 + 1^2) \bmod 8 = 7 \\ &\to \quad 6 + \frac{1}{2}(2 + 2^2) \bmod 8 = 1 \end{aligned}$$

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

Kritik

Wie beim linearen Sondieren gibt es nur m verschiedene Sondierungsfolgen. Dieser Effekt wird **Sekundäre Häufungen** genannt.

Analyseergebnisse

Die Anzahl der Sondierungen im Durchschnitt beträgt (ohne Beweis):

$$\text{Erfolglose Suche} \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

$$\text{Erfolgreiche Suche} \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

Analyseergebnisse für „uniform hashing"

Hier ist die Anzahl der Sondierungen im Durchschnitt für $\alpha = \frac{n}{m} < 1$:

$$\text{Erfolglose Suche} \approx \frac{1}{1-\alpha}$$

$$\text{Einfügen} \approx \frac{1}{1-\alpha}$$

$$\text{Erfolgreiche Suche} \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

Übersicht über die Güte der Kollisionsstrategien

In der folgenden Tabelle wurden für alle diskutierten Kollisionsstrategien einige Werte der Analyseergebnisse für verschiedene Auslastungsfaktoren α berechnet.

α	Verkettung		offene Hashverfahren					
			lineares S.		quadr. S.		uniform hashing	
	erfolgreich	erfolglos	er	el	er	el	er	el
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	—	—	—	—	—	—

5.3.3 Double Hashing

Idee: Die Sondierungsreihenfolge hängt von einer zweiten Hashfunktion ab, die unabhängig von der ersten Hashfunktion ist.

Gegeben sind zwei Hashfunktionen

$$h_1, h_2 : U \to \{0, 1, \ldots, m-1\}.$$

Wir definieren für $i = 0, 1, \ldots, m-1$:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m.$$

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.

Bedingungen an h_2: Für alle Schlüssel k muss $h_2(k)$ relativ prim zu m sein :

$$ggT(h_2(k), m) = 1,$$

sonst wird die Tabelle nicht vollständig durchsucht. (Falls $ggT(h_2(k), m) = d > 1$, so wird nur $\frac{1}{d}$-tel durchsucht).

Zwei Vorschläge

(1) $m = 2^P$, $P \in \mathbb{N} \wedge P > 1$ (schlecht für Divisionsmethode), $h_2(k)$ immer ungerade

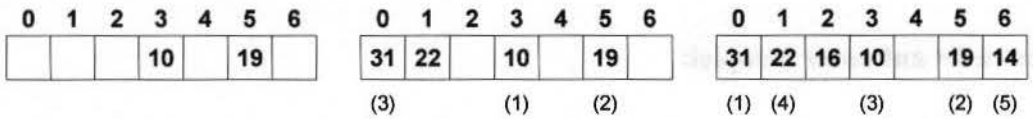
(2) m Primzahl, $0 < h_2(k) < m$, z.B.

$$\begin{aligned} h_1(k) &= k \bmod m \\ h_2(k) &= 1 + (k \bmod m') \end{aligned}$$

mit $m' = m - 1$ oder $m' = m - 2$.

Beispiel: $m = 7$, $h_1(k) = k \bmod 7$, $h_2(k) = 1 + (k \bmod 5)$

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! 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.

Idee: Wenn beim Einfügen eines Schlüssels k ein sondierter Platz j bereits belegt ist mit einem anderen Schlüssel $k' = T[j].key$, setze

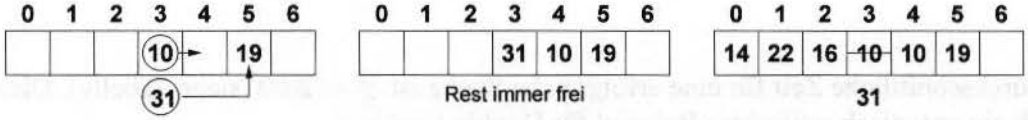
$$j_1 = (j + h_2(k)) \bmod m \quad \text{(normales Double Hashing)}$$
$$j_2 = (j + h_2(k')) \bmod m.$$

Ist Platz j_1 frei, dann kann der Schlüssel k an dieser Stelle in die Hashtabelle eingefügt werden ($T[j_1] = k$).

Ist hingegen Platz j_1 belegt, dafür aber j_2 frei, muss nicht weiter ein freier Platz für den Schlüssel k gesucht werden, denn k', der am sondierten Platz j die Kollision verursacht hat, kann in nur einem Schritt leicht auf Position j_2 verschoben werden: man trägt also k' in $T[j_2]$ ein und k in das nun frei gewordene Feld $T[j]$ (*nicht* $T[j_1]$).

Für den Fall, dass sowohl Platz j_1 als auch j_2 belegt sind, beginnt man mit einer neuen Iteration und setzt $j = j_1$.

Angewendet auf unser Beispiel:



Hier beträgt die durchschnittliche Zeit für eine erfolgreiche Suche nun $\frac{7}{6} = 1.17$ (siehe Tabelle).

k	10		19		31		22		14		16		
	2	+	1	+	1	+	1	+	1	+	1	=	7

Ein ausführlicheres Beispiel zur Verbesserung nach Brent:

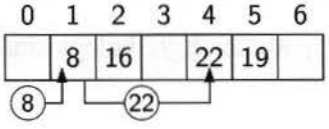
Das nun folgende, etwas ausführlichere Beispiel verdeutlicht im Detail das Einfügen mehrerer Schlüssel in eine Hashtabelle mittels Double-Hashing unter Verwendung der Verbesserung nach Brent.

Gegeben sei eine Hashtabelle T der Größe $m = 7$ sowie die beiden Hashfunktionen

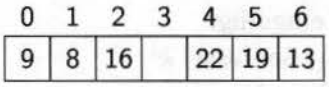
$$\begin{aligned} h_1(k) &= k \bmod m \\ h_2(k) &= 1 + (k \bmod 5). \end{aligned}$$

In die anfangs leere Hashtabelle werden nun die folgenden Schlüssel in der gegebenen Reihenfolge eingefügt: $\langle 16, 19, 22, 8, 13, 9, 18 \rangle$

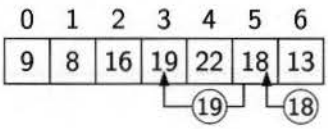
Hashfunktion	*Beschreibung*
$h_1(16) = 2$	$T[2]$ frei $\rightarrow T[2] = 16$
$h_1(19) = 5$	$T[5]$ frei $\rightarrow T[5] = 19$
$h_1(22) = 1$	$T[1]$ frei $\rightarrow T[1] = 22$
$h_1(8) = 1$	$T[1]$ besetzt (Schlüssel 22)
$1 + h_2(8) = 5$	Versuch Double-Hashing (DH): Schrittweite $h_2(8) = 4 \rightarrow T[5]$ besetzt (Schlüssel 19)
$1 + h_2(22) = 4$	Versuch Verbesserung nach Brent (VnB): Schrittweite $h_2(22) = 3 \rightarrow$ Schlüssel 22 kann auf freien Platz 4 verschoben werden $\rightarrow T[4] = 22, T[1] = 8$



Hashfunktion	*Beschreibung*
$h_1(13) = 6$	$T[6]$ frei $\rightarrow T[6] = 13$
$h_1(9) = 2$	$T[2]$ besetzt (Schlüssel 16)
$2 + h_2(9) = 0$	Versuch DH: $T[0]$ frei $\rightarrow T[0] = 9$



Hashfunktion	*Beschreibung*
$h_1(18) = 4$	$T[4]$ besetzt (Schlüssel 22)
$4 + h_2(18) = 1$	Versuch DH: $T[1]$ besetzt (Schlüssel 8)
$4 + h_2(22) = 0$	Versuch VnB: $T[0]$ besetzt (Schlüssel 9) $\rightarrow$ Schlüssel 22 kann nicht verschoben werden
$1 + h_2(18) = 5$	Versuch DH: $T[5]$ besetzt (Schlüssel 19)
$1 + h_2(8) = 5$	Versuch VnB: $T[5]$ besetzt (Schlüssel 19) $\rightarrow$ Schlüssel 8 kann nicht verschoben werden
$5 + h_2(18) = 2$	Versuch DH: $T[2]$ besetzt (Schlüssel 16)
$5 + h_2(19) = 3$	Versuch VnB: Schlüssel 19 kann auf freien Platz 3 verschoben werden $\rightarrow$ $T[3] = 19, T[5] = 18$



Beachten Sie bitte, dass dieses Beispiel die Verbesserung nach Brent demonstrieren sollte und deshalb – um eine hohe Anzahl an Kollisionen zu erreichen – die Hashtabelle bis zum letzten Platz gefüllt wurde. In der Praxis wird man diese Extremsituation versuchen zu vermeiden, obwohl das folgende Analyseergebnis zumindest in Bezug auf die erfolgreiche Suche auch bei hohem Füllgrad eine bemerkenswert gute Nachricht darstellt.

Analyseergebnis

Im Allgemeinen beträgt die Anzahl der Sondierungen im Durchschnitt:

Erfolglose Suche $\approx \frac{1}{1-\alpha}$ (wie uniform)

Erfolgreiche Suche < 2.5 (unabhängig von α für $\alpha \leq 1$).

Mit anderen Worten bewirkt die Verbesserung nach Brent, dass der durchschnittliche Aufwand einer erfolgreichen Suche selbst im Extremfall einer vollständig gefüllten Hashtabelle unter 2.5 Sondierungsschritten liegt.

Pseudocode

Algorithmus 29 Einfügen_Brent (**var** T, k)

Eingabe: Hashtabelle T; neuer Schlüssel k
Ausgabe: Element k wurde in T eingefügt
Variable(n): Hashindizes j, j_1, j_2; Schlüssel k'

```
1:  // Brents Variation von doppeltem Hashing
2:  j = hash1(k);
3:  solange T[j].status == used {
4:     k' = T[j].key;
5:     j1 = (j + hash2(k)) mod m;
6:     j2 = (j + hash2(k')) mod m;
7:     falls T[j1].status ≠ used ∨ T[j2].status == used dann {
8:        j = j1;
9:     } sonst {
10:       T[j].key = k;
11:       k = k';
12:       j = j2;
13:    }
14: }
15: T[j].key = k;
16: T[j].status = used;
```

Kapitel 6

Graphen

Viele Aufgabenstellungen lassen sich mit Hilfe graphentheoretischer Konzepte modellieren. *Graphen* bestehen aus Objekten und deren Beziehungen zueinander. Sie modellieren also diskrete Strukturen, wie z.B. Netzwerke oder Prozesse. Betrachten wir als Beispiel das Bahnnetz von Österreich. Die Objekte (sogenannte Knoten) können hier die Bahnhöfe (und eventuell Weichen) sein, die Beziehungen (sogenannte Kanten) die Schienen des Bahnnetzes. Belegt man die Kanten mit zusätzlichen Attributen (bzw. Gewichten), so kann man damit verschiedene Probleme formulieren. Bedeutet das Attribut einer Kante zwischen u und v die Fahrzeit eines Zuges von u nach v, so besteht das Problem der kürzesten Wege darin, die minimale Fahrzeit von einer beliebigen Stadt A zu einer beliebigen Stadt B zu ermitteln.

Andere Beispiele, die mit Hilfe von graphentheoretischen Konzepten gelöst werden können, sind Routenplanungsprobleme. Hier sollen z.B. Güter von mehreren Produzenten zu den Verbrauchern transportiert werden.

Im Bereich Produktionsplanung geht es u.a. darum, die auftretenden Teilaufgaben bestehenden Maschinen zuzuordnen, so dass die Herstellungszeit minimiert wird. Da hierbei die Reihenfolge der Abarbeitung der Teilaufgaben zentral ist, werden diese Probleme auch *Scheduling*-Probleme genannt.

Ein klassisches Graphenproblem ist das „Königsberger Brückenproblem", das Euler 1736 gelöst und damit die Theorie der Durchlaufbarkeit von Graphen gegründet hat. Die topologische Graphentheorie hat sehr von den Forschungen am „Vierfarbenproblem" profitiert: Die ursprüngliche Aufgabe ist, 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 ausreichen.

Im Bereich der chemischen Isomere ist es heutzutage von großer Bedeutung für die Industrie (im Rahmen der Entwicklung neuer künstlicher Stoffe und Materialien), folgende bereits 1890 von Caley gestellte Frage zu beantworten: Wie viele verschiedene Isomere einer

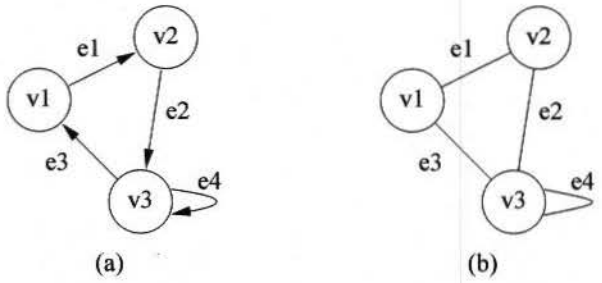


Abbildung 6.1: Beispiele eines (a) gerichteten und (b) ungerichteten Graphen.

bestimmten Zusammensetzung existieren? Caley hat damit die Abzähltheorie von Graphen begründet. Oftmals werden auch Beziehungsstrukturen bzw. 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 Betriebsabläufe und Geschäftsprozesse als Graphen modelliert (z.B. UML-Darstellung).

6.1 Definition von Graphen

Ein *Graph* ist ein Tupel (V, E), wobei V eine endliche Menge ist, deren Elemente *Knoten* (engl. *vertices, nodes*) genannt werden. Die Menge E besteht aus Paaren von Knoten. Sind diese geordnet ($E \subseteq V \times V$), spricht man von *gerichteten* Graphen, sind die Paare ungeordnet, spricht man von *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. *loop*). Abbildung 6.1(a) und 6.1(b) zeigen je ein Beispiel eines gerichteten und ungerichteten Graphen.

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 *Nachbarn*

Die eingehende Nachbarmenge eines Knotens $v \in V$ ist definiert als

$$N^-(v) = \{u \in V | (u, v) \in E\}$$

und die ausgehende Nachbarmenge als $N^+(v) = \{w \in V | (v, w) \in E\}$. Für ungerichtete Graphen gilt: $N^-(v) = N^+(v) =: N(v)$. Der (Eingangs- bzw. Ausgangs-)Knotengrad von $v \in V$ ist definiert als $d^-(v)$, $d^+(v)$ bzw. $d(v) = d^-(v) + d^+(v)$ und bezeichnet die Anzahl

seiner (ausgehenden bzw. eingehenden) inzidenten Kanten. Dabei wird eine Schleife (v, v) an Knoten v zweimal gezählt. Für ungerichtete Graphen gilt folgendes Lemma.

Lemma: In einem ungerichteten Graphen $G = (V, E)$ ist die Anzahl der Knoten mit ungeradem Grad gerade.

Beweis: Summiert man über alle Knotengrade, so zählt man jede Kante genau zwei Mal:

$$\sum_{v \in V} d(v) = \underbrace{2 \cdot |E|}_{gerade\ Zahl}$$

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.

6.2 Darstellung von Graphen im Rechner

Wir stellen im Folgenden zwei verschiedene Möglichkeiten vor, Graphen im Rechner zu speichern. Sie unterscheiden sich in Hinblick auf den benötigten Speicherplatz und die benötigte Laufzeit für die bei manchen Algorithmen auftretenden, immer wiederkehrenden Abfragen.

Wir nehmen in der Folge an, dass die Knotenmenge $V = \{v_1, \ldots, v_n\}$ und $n = |V|$.

(A) Speicherung in einer Adjazenzmatrix

Eine Möglichkeit ist, einen Graphen ohne Mehrfachkanten als Adjazenzmatrix zu speichern. Das sieht dann folgendermaßen aus:

M sei eine $n \times n$-Matrix mit Einträgen $\quad M[u, v] = \begin{cases} 1 & \text{falls } (u, v) \in E \\ 0 & \text{sonst} \end{cases}$

Für die Beispiele aus Abb. 6.1(a) und 6.1(b) ergeben sich die folgenden Matrizen:

	v_1	v_2	v_3
v_1	0	1	0
v_2	0	0	1
v_3	1	0	1

	v_1	v_2	v_3
v_1	0	1	1
v_2	1	0	1
v_3	1	1	1

Analyse typischer Abfragen:

Abfrage: Existiert die Kante (v, w)?	$\Theta(1)$
Iteration über alle Nachbarn eines Knotens	$\Theta(n)$
Platzverbrauch zum Speichern	$\Theta(n^2)$ selbst wenn der Graph *dünn* ist (d.h. viel weniger als n^2 Kanten enthält)

(B) Speicherung mit Hilfe von Adjazenzlisten

Eine Alternative (die am meisten verwendete) ist die Speicherung mittels Adjazenzlisten. Für jeden Knoten $v \in V$ existiert eine Liste der ausgehenden Kanten. Man kann dies folgendermaßen als einfach verkettete Listenstruktur umsetzen.

Für die Beispiele aus Abb. 6.1(a) und 6.1(b) ergeben sich hier folgende Listen:

1 : 2	1 : 2,3
2 : 3	2 : 1,3
3 : 1,3	3 : 1,2,3
gerichtet	ungerichtet

Als Realisierung hierfür bieten sich z.B. einfach verkettete lineare Listen an:

1 → 2 → .	1 → 2 → 3 → .
2 → 3 → .	2 → 1 → 3 → .
3 → 1 → 3 → .	3 → 1 → 2 → 3 → .
(a) gerichtet	(b) ungerichtet

Als alternative Umsetzung bieten sich Felder (Arrays) an: Hierbei gibt es einen Eintrag $index(v)$ für jeden Knoten, der angibt, an welcher Stelle in der Kantenliste (*edgelist*) die Nachbarliste von v beginnt. Für das Beispiel aus Abb.1(b) ungerichtet ergibt sich:

index[v]	1 | 3 | 5
edgelist[e]	2 | 3 || 1 | 3 || 1 | 2 | 3

Analyse typischer Abfragen:

Abfrage: Existiert die Kante (v, w)?	$\Theta(d(v))$
Iteration über alle Nachbarn eines Knotens	$\Theta(d(v))$
Platzverbrauch	$\Theta(\|E\| + \|V\|)$

Für die Effizienz von Graphenalgorithmen ist es wichtig, jeweils die richtige Datenstruktur zur Speicherung von Graphen zu wählen. Ist der Graph „dünn", d.h. enthält er relativ wenige Kanten, also $\Theta(|V|)$-viele Kanten, so sind Adjazenzlisten meist besser geeignet. Ist der Graph sehr „dicht", d.h. enthält er $\Theta(|V|^2)$-viele Kanten und sind viele Anfragen der Art „Existiert die Kante (u, v) ?" zu erwarten, dann ist die Darstellung als Adjazenzmatrix vorzuziehen.

6.3 Durchlaufen von Graphen

Die wichtigsten Verfahren zum Durchlaufen von Graphen sind *Tiefensuche* und *Breitensuche*. Wir konzentrieren uns auf die Tiefensuche.

6.3.1 Tiefensuche (Depth-First-Search, DFS)

Tiefensuche ist für folgende Graphenprobleme gut geeignet:

(1) Finde alle Knoten, die von s aus erreichbar sind.
Gegeben sei ein Knoten s. Bestimme alle Knoten v, für die es einen Pfad von s nach v gibt. Dabei ist ein *Pfad der Länge* k eine Folge $v_0, e_0, v_1, e_1, \ldots, v_{k-1}, e_{k-1}, v_k$ mit $e_i = (v_i, v_{i+1}) \in E$.
Ein *Weg* ist ein Pfad, bei dem alle Knoten und Kanten verschieden sind.
Ein Pfad ist ein *Kreis*, wenn $v_0 = v_k$.

(2) Bestimme die Zusammenhangskomponenten von G.
Ein ungerichteter Graph G heißt *zusammenhängend*, wenn für alle $u, v \in V$ gilt: Es existiert ein Pfad von u nach v.
Ein gerichteter Graph G heißt *stark zusammenhängend*, wenn für alle $u, v \in V$ gilt: Es existiert ein (gerichteter) Pfad von u nach v.
Daneben gibt es für gerichtete Graphen auch den Begriff des schwachen Zusammenhangs: ein solcher Graph heißt *schwach zusammenhängend* (oder auch nur *zusammenhängend*), wenn der zugrundeliegende ungerichtete Graph zusammenhängend ist.
Eine *Zusammenhangskomponente* ist ein maximal zusammenhängender Untergraph von G. Sind $G = (V, E)$ und $H = (W, F)$ Graphen, sodass $W \subseteq V$ und $F \subseteq E$, so heißt H *Untergraph* von G.

(3) Enthält G einen Kreis?

(4) Ist G 2-zusammenhängend?
Ein zusammenhängender Graph G ist *2-zusammenhängend*, wenn $G - \{v\}$ (das ist G nach Löschen des Knotens v und seiner inzidenten Kanten) für alle $v \in V$ zusammenhängend ist. (2-Zusammenhang ist z.B. bei Telefonnetzen wichtig; Stichwort: „Survivability".)

Die Idee des DFS-Algorithmus ist die folgende: Verlasse jeden Knoten so schnell wie möglich, um „tiefer" in den Graphen zu laufen.

1. v sei der letzte besuchte Knoten: markiere v

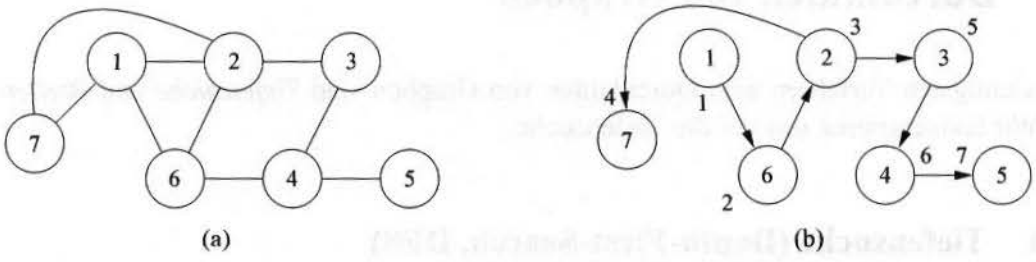


Abbildung 6.2: Ein Graph und ein möglicher DFS-Baum.

2. Sei (v, w) eine Kante und der Knoten w ist noch unmarkiert, dann: setze $v := w$ und gehe zu 1.

3. Sonst: wenn kein unmarkierter Nachbar von v mehr existiert, gehe zurück zum Vorgänger von v.

Wir betrachten als Beispiel den Graphen in Abb. 6.2(a). Abb. 6.2(b) zeigt eine mögliche Abarbeitungsreihenfolge des DFS-Algorithmus. Die kleinen Nummern neben den Knoten geben die Reihenfolge an, in der diese markiert wurden. Die gerichteten Kanten geben für jeden besuchten Knoten dessen eindeutigen Vorgänger und seine direkten Nachfolger an. Der Graph, der durch diese gerichteten Kanten definiert wird, ist der zu dem DFS-Lauf gehörige *DFS-Baum*.

Im Folgenden präsentieren wir mögliche Lösungen zu den oben genannten Problemen (1)–(3). Wir nehmen dabei an, dass der gegebene Graph ungerichtet ist. Wir überlassen es Ihnen, die dazugehörigen Varianten für gerichtete Graphen zu entwickeln.

Basisalgorithmus zur Tiefensuche

Algorithmus 30/31 realisiert die Grundidee der Tiefensuche. Hier werden alle Knoten $v \in V$ bestimmt, die von dem gegebenen Knoten $s \in V$ durch einen Pfad erreichbar sind. Der Knoten v ist von s aus erreichbar, wenn entweder $s = v$ oder wenn es einen Knoten u gibt mit $(s, u) \in E$ und v von u aus erreichbar ist. Nach Ausführung von Depth-First-Search1(s) gilt für alle $v \in V$: $markiert[v] = 1$ genau dann, wenn es einen Pfad von s nach v gibt.

Analyse der Laufzeit:

- Initialisierung: $\Theta(|V|)$.
- DFS(v) wird für jeden Knoten höchstens einmal aufgerufen und hat (ohne Folgekosten der dort gestarteten DFS-Aufrufe) Kosten $O(d(v))$.

- Insgesamt ergibt dies Kosten von:

$$\Theta(|V|) + \sum_{v \in V} O(d(v)) = O(|V|) + O(|E|) = O(|V| + |E|).$$

Dies ist asymptotisch optimal.

Algorithmus 30 Depth-First-Search1(G, s)

Eingabe: Graph $G = (V, E)$; Knoten $s \in V$
Ausgabe: alle Knoten in G, die von s aus erreichbar sind
Variable(n): Knoten v

```
1: Initialisierung: setze markiert[v] = 0 für alle v;
2: DFS1(G, s);
3: für alle Knoten v ∈ V {
4:    falls markiert[v] == 1 dann {
5:       Ausgabe von v;
6:    }
7: }
```

Algorithmus 31 DFS1(G, v)

Eingabe: Graph $G = (V, E)$; Knoten $v \in V$
Variable(n): Knoten w

```
1: // rekursiver Aufruf von DFS1 für alle noch nicht markierten Nachbarknoten von v
2: markiert[v] = 1;
3: für alle Knoten w aus N(v) {
4:    falls markiert[w] == 0 dann {
5:       DFS1(G, w);
6:    }
7: }
```

Bestimmung der Zusammenhangskomponenten

Algorithmus 32/33 realisiert einen Tiefensuche-Algorithmus zur Lösung von Problem (2) für ungerichtete Graphen. Ein Aufruf von DFS2$(G, v, compnum)$ besucht und markiert alle Knoten, die in der gleichen Zusammenhangskomponente wie v liegen, mit der aktuellen Komponentennummer *compnum*. Bleiben nach dem Lauf von DFS2$(G, v, compnum)$ noch unmarkierte Knoten übrig, erhöht sich die Komponentennummer *compnum* um eins, und DFS2$(G, w, compnum)$ wird aufgerufen; jeder in diesem Lauf entdeckte Knoten wird der neuen Komponente mit Nummer *compnum* zugeordnet.

Analyse der Laufzeit: Mit Hilfe der gleichen Argumentation wie für Algorithmus 30/31 folgt eine Laufzeit von $\Theta(|V| + |E|)$.

Algorithmus 32 Depth-First-Search2(G)

Eingabe: Graph $G = (V, E)$

Ausgabe: Markierungsfeld $markiert[v]$ enthöht für jeden Knoten $v \in V$ die ID der ihm zugeordneten Zusammenhangskomponente; Zahl $compnum$ enthält am Ende die Anzahl der Komponenten

```
1: Initialisierung: setze markiert[v] = 0 für alle v;
2: compnum = 0;
3: für alle v aus V {
4:    falls markiert[v] == 0 dann {
5:       compnum = compnum + 1;
6:       DFS2(G, v, compnum);
7:    }
8: }
```

Algorithmus 33 DFS2($G, v, compnum$)

Eingabe: Graph $G = (V, E)$; Knoten $v \in V$

Variable(n): Knoten w

```
1: // rekursiver Aufruf von DFS2 für alle noch nicht markierten Nachbarknoten von v
2: markiert[v] = compnum;
3: für alle Knoten w aus N(v) {
4:    falls markiert[w] == 0 dann {
5:       DFS2(G, w, compnum);
6:    }
7: }
```

Kreissuche in einem Graphen

Algorithmus 34/35 zeigt einen Tiefensuche-Algorithmus zur Lösung von Problem (3) für ungerichtete Graphen. Falls der betrachtete Nachbarknoten w von v in Algorithmus 35 noch unmarkiert ist, dann wird die Kante (v, w) in den DFS-Baum T aufgenommen. Um am Ende leichter den Kreis zu erhalten, wird v als direkter Vorgänger von w in $parent[w]$ gespeichert. Die weitere Durchsuchung wird bei w mit dem entsprechenden Aufruf DFS3 fortgesetzt.

Sobald v auf einen Nachbarn w trifft, der bereits markiert aber nicht sein direkter Vorgänger ist, wurde ein Kreis gefunden; eine entsprechende Kante (v, w) wird in B gespeichert. Der Graph G enthält also genau dann keinen Kreis, wenn $B = \emptyset$ ist. Ansonsten wird jeder im DFS-Lauf entdeckte Kreis durch eine Kante in der Menge B repräsentiert. Ein Detail ist noch, dass in Zeile 3 von DFS3 durch die zusätzliche Abfrage von $(w, v) \notin B$ sichergestellt wird, dass für jede Kante $(w, v) \in B$ in weiterer Folge nicht auch die entgegengesetzt gerichtete Kante (v, w) zu B hinzugefügt wird.

Achtung: Sie können sich leicht überlegen, dass nicht alle Kreise in G durch den beschriebenen Algorithmus gefunden werden. Machen Sie sich klar, dass dies kein Widerspruch zur obigen Aussage ist.

Mit Hilfe des DFS-Baumes (der hier sowohl in T als auch in *parent* gespeichert ist) kann man nun ganz einfach die Menge der gefundenen Kreise abrufen. Für jede Kante (v, w) in B erhält man den dazugehörigen Kreis, indem man so lange den eindeutigen Vorfahren (*parent*) von v im DFS-Baum T folgt, bis man auf w trifft. Dazu ist es notwendig, die Menge B als gerichtete Kantenmenge abzuspeichern, damit man Anfangsknoten v und Endknoten w unterscheiden kann. Ein Verfolgen der Vorfahren von w würde nämlich nicht zu v führen. (Anmerkung: Hier wäre es eigentlich nicht nötig gewesen, die Menge T zu speichern. Wir haben sie nur zur besseren Veranschaulichung eingeführt.)

Analyse der Laufzeit: Die gleiche Argumentation wie für Algorithmus 30 liefert die Laufzeit $\Theta(|V| + |E|)$.

Algorithmus 34 Depth-First-Search3(G)

Eingabe: Graph $G = (V, E)$
Ausgabe: Kreis in G, falls einer existiert
Variable(n): Kantenmenge B von Kanten, die einen Kreis schließen; Feld *parent* für Verweise auf Elternknoten, DFS-Baum T; Markierungsfeld *markiert*; Knoten $v \in V$

```
1: Initialisierung: setze markiert[v] = 0 für alle v;
2: B = ∅; T = ∅;
3: für alle v aus V {
4:    falls markiert[v] == 0 dann {
5:       parent[v] = NULL;
6:       DFS3(G, v, B, T, parent);
7:    }
8: }
9: falls B ≠ ∅ dann {
10:   Ausgabe „Kreis gefunden“;
11: }
```

Algorithmus 35 DFS3(G, v, **var** B, **var** T, **var** *parent*)

Eingabe: Graph $G = (V, E)$; Knoten v in G; Kantenmenge B von Kanten, die einen Kreis schließen; DFS-Baum T; Feld *parent* für Verweise auf Elternknoten, DFS-Baum T;
Variable(n): Knoten $w \in V$

```
1: markiert[v] = 1;
2: für alle Knoten w aus N(v) {
3:    falls markiert[w] == 1 ∧ parent[v] ≠ w ∧ (w, v) ∉ B dann {
4:       B = B ∪ {(v, w)};
5:    } sonst falls markiert[w] == 0 dann {
6:       T = T ∪ {(v, w)};
7:       parent[w] = v;
8:       DFS3(G, w, B, T, parent);
9:    }
10: }
```

6.4 Topologisches Sortieren

In vielen Anwendungen ist eine Halbordnung von Elementen durch eine binäre Relation gegeben, die unmittelbar durch einen zyklenfreien gerichteten Graphen $G = (V, E)$ dargestellt werden kann. Die Knoten V des Graphen entsprechen den Elementen und die Kanten $(u, v) \in E$ beschreiben Beziehungen wie „u kommt vor v“ oder „u ist kleiner als v“. Gesucht ist eine vollständige Ordnung (d.h. eine lineare Anordnung oder Nummerierung) der Elemente V, die mit der gegebenen Relation (E) verträglich ist. Diesen Vorgang nennt man *topologisches Sortieren*.

Beispiel: Anziehen von Kleidungsstücken

- Elemente: Hose, Mantel, Pullover, Socken, Schuhe, Unterhemd, Unterhose
- Relationen:
  - Unterhemd vor Pullover
  - Unterhose vor Hose
  - Pullover vor Mantel
  - Hose vor Mantel
  - Hose vor Schuhe
  - Socken vor Schuhe

Die Veranschaulichung dieser Beziehungen als sogenannter Relationsgraph zeigt Abbildung 6.3.

Eine topologische Sortierung beschreibt nun eine gültige (sequentielle) Folge der Kleidungsstücke um sie anzuziehen, beispielsweise:

(Unterhose, Unterhemd, Hose, Pullover, Socken, Schuhe, Mantel)

Formal sei die topologische Sortierung durch eine Abbildung $ord : V \to 1, \ldots, n$ mit $n = |V|$ notiert. Das heißt für unser Beispiel: ord(Unterhose) $= 1$, ord(Unterhemd) $= 2$ usw.

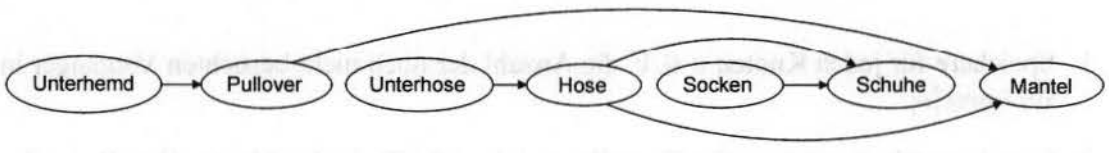


Abbildung 6.3: Beispiel eines Relationsgraphen

Allgemein kann es aber durchaus mehrere gültige Ordnungen geben, so würde in unserem Beispiel auch die Ordnung (Unterhemd, Pullover, Socken, Unterhose, Hose, Schuhe, Mantel) obige Beziehungen erfüllen.

Lemma: Es gibt zumindest eine gültige topologische Ordnung, wenn die Beziehungen zyklenfrei sind, d.h., der Relationsgraph kreisfrei ist.

Beweis: Dies kann man durch Induktion über die Knotenanzahl zeigen. Falls $|V| = 1$, dann existiert offensichtlich die topologische Sortierung, in der dem einen Element $u \in V$ $ord(u) = 1$ zugewiesen ist. Falls $|V| > 1$, so betrachtet man einen Knoten v mit $d^-(v) = 0$ (Eingangsgrad 0, d.h. ohne eingehende Kanten) und definiert $ord(v) = 1$. Auf Grund der Zyklenfreiheit muss ein solcher Knoten immer existieren. Danach entfernt man v, wodurch ein um einen Knoten verkleinerter Graph entsteht. An dessen topologische Sortierung wird v vorne angefügt.

Aus diesem Prinzip ergibt sich auch unmittelbar ein grundsätzlicher Ansatz zum topologischen Sortieren:

1. Suche einen Knoten $v \in V$ mit $d^-(v) = 0$
2. Füge v in der topologischen Sortierung als nächstes Element hinzu
3. Lösche v mit allen inzidenten Kanten aus dem Graphen
4. Solange der Graph nicht leer ist, gehe zu (1)

Es ist noch zu klären, wie jeweils ein Knoten $v \in V$ mit $d^-(v) = 0$ effizient gefunden werden kann. Eine Möglichkeit wäre, bei einem beliebigen Knoten zu beginnen und Kanten rückwärts zu verfolgen. Auf Grund der Zyklenfreiheit würde man hiermit nach spätestens $O(n)$ Schritten zu einem Knoten mit Eingangsgrad 0 kommen.

Effizienter ist es jedoch, den jeweils aktuellen Eingangsgrad zu jedem Knoten zu speichern und auf dem aktuellen Stand zu halten. Dann genügt es, anstatt einen Knoten aus G tatsächlich zu entfernen, die Eingangsgrade seiner direkten Nachfolger zu verringern. Um einen Knoten mit Eingangsgrad 0 schnell zu finden, verwalten wir die Menge aller Knoten mit aktuellem Eingangsgrad 0. Hiermit ergibt sich folgender verfeinerter Lösungsansatz, der in Algorithmus 36 noch konkreter dargestellt ist.

1. Speichere für jeden Knoten $v \in V$ die Anzahl der noch nicht besuchten Vorgänger in $inDegree[v]$
2. Speichere alle Knoten v, für die $inDegree[v] = 0$ gilt, in der Menge Q (z.B. implementiert als eine Queue)
3. Q enthält demnach alle Knoten, die als nächstes besucht werden dürfen.

4. Nimm einen Knoten u aus Q, entferne ihn aus Q
5. Speichere u in der topologischen Sortierung als nächstes Element
6. Reduziere $inDegree[v]$, für alle $(u, v) \in E$, um 1 und aktualisiere Q
7. Solange Q nicht leer ist gehe zu (4)

Algorithmus 36 Top-Sort(G, **var** ord)

```
1: ∀v ∈ V : inDegree[v] = 0;
2: für alle (u, v) ∈ E {
3:    inDegree[v] = inDegree[v] + 1;
4: }
5: Q = {v ∈ V | inDegree[v] = 0};
6: k = 1;
7: solange Q ist nicht leer {
8:    nimm ersten Knoten u aus Q; Q = Q \ {u};
9:    ord(u) = k;
10:   k = k + 1;
11:   für alle Knoten v ∈ N+(u) {
12:      inDegree[v] = inDegree[v] − 1;
13:      falls inDegree[v] == 0 dann {
14:         Q = Q ∪ {v};
15:      }
16:   }
17: }
18: falls k ≠ n + 1 dann {
19:    G ist nicht azyklisch und es existiert keine topologische Sortierung;
20: }
```

Dieser skizzierte Algorithmus zur topologischen Sortierung ist eine modifizierte *Breitensuche*. Bei der Breitensuche werden – im Gegensatz zur Tiefensuche – alle Nachfolger eines Knotens immer abgearbeitet, *bevor* deren weitere Nachfolger verfolgt werden.

Laufzeit

Sei $n = |V|$ die Anzahl der Elemente und $m = |E|$ die Anzahl der Beziehungen. Die Initialisierung von $inDegree$ und Q benötigt Laufzeit $\Theta(n+m)$. Die zentrale Schleife hat (im erfolgreichen Fall) $\Theta(n)$ Wiederholungen. Insgesamt wird darin jede Kante genau einmal inspiziert, das entsprechende Aktualisieren von $inDegree$ und Q benötigt nur eine konstante zusätzliche Zeit pro Kante. Somit ist auch die Gesamtlaufzeit des topologischen Sortierens $\Theta(n + m)$.

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
- 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.

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.

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 an Hand von Beispielen kennen lernen werden.

NP-schwierige Optimierungsprobleme treten in der Praxis häufiger auf. Für sie betrachten wir im Zuge dieser Lehrveranstaltung nur die *Enumeration* als ein so genanntes *exaktes Verfahren*, d.h. ein Verfahren, das immer optimale Lösungen liefert, aber wegen des exponentiell steigenden Zeitaufwands im Allgemeinen nur für kleinere Probleminstanzen verwendet werden kann. In der weiterführenden LVA Algorithmen und Datenstrukturen 2 wird dann genauer auf Lösungsansätze für NP-schwierige Optimierungsprobleme eingegangen. U.a. wird als ein weiteres exaktes Verfahren *Branch-and-Bound* vorgestellt, aber auch *Heuristiken* werden behandelt, die meist die optimale Lösung – mit oder ohne einer Gütegarantie – nur annähern.

7.1 Greedy-Algorithmen

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 beispielsweise im folgenden Problem – zur optimalen Lösung.

7.1.1 Minimale aufspannende Bäume

Ein Graph wird *Baum* genannt, wenn er kreisfrei (im ungerichteten Sinne) und zusammenhängend ist. Für einen Baum gilt: $|V| = |E| + 1$. Ferner gibt es in einem Baum jeweils

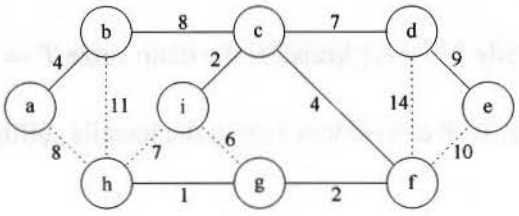


Abbildung 7.1: Die durchgezogenen Linien stellen einen MST des gezeigten Graphen dar.

genau einen ungerichteten Weg zwischen je zwei Knoten. Entfernt man eine Kante aus einem Baum, so zerfällt dieser in genau zwei Komponenten. Bei Wegnahme eines Knotens mit Grad $k \geq 2$ und seiner inzidenten Kanten zerfällt ein Baum in k Komponenten. Man kann sich überlegen, dass die obigen Aussagen jeweils äquivalente Definitionen eines Baumes sind.

Das MST-Problem:

Gegeben ist ein zusammenhängender und *gewichteter* (ungerichteter) Graph $G = (V, E)$ mit Kantengewichten $w_e = w_{uv} = w_{vu}$ für $e = (u, v) \in E$.
Gesucht ist ein *minimaler aufspannender Baum* (*Minimum Spanning Tree, MST*) in G, d.h. ein zusammenhängender, zyklenfreier Untergraph $G_T = (V, T)$ mit $T \subseteq E$, dessen Kanten alle Knoten aufspannen und für den $w(T) = \sum_{e \in T} w_e$ so klein wie möglich ist.

Abbildung 7.1 zeigt einen gewichteten Graphen G und einen minimalen aufspannenden Baum G_T von G. Das Gesamtgewicht des MST ist (bzw. dessen Kosten sind) 37. Beachten Sie, dass der MST nicht eindeutig sein muss. So kann z.B. in Abb. 7.1 die Kante (b, c) durch (a, h) ersetzt werden, ohne dass sich das Gewicht ändert. Eindeutigkeit des MST ist jedoch dann gegeben, wenn die Gewichte aller Kanten unterschiedlich sind.

Das MST-Problem taucht in der Praxis z.B. als Unterproblem beim Design von Kommunikationsnetzwerken auf.

7.1.2 Der Algorithmus von Kruskal

Kruskal schlug 1956 einen Algorithmus zur exakten Lösung des MST-Problems vor, der dem Greedy-Prinzip folgt.

Die Idee des Kruskal-Algorithmus lautet wie folgt:

- Sortiere die Kanten von E nach ihren Gewichten: $w_{e_1} \leq w_{e_2} \leq \ldots \leq w_{e_m}$
- $T = \emptyset$

- Für $i = 1, \ldots, m$: Falls $T \cup \{e_i\}$ kreisfrei ist, dann setze $T = T \cup \{e_i\}$.

Der Algorithmus ist *greedy*, weil er in jedem Schritt die jeweils „billigste" noch hinzufügbare Kante aufnimmt.

Korrektheit:

- kreisfrei? $\surd$
- zusammenhängend? $\surd$
- aufspannend? $\surd$
- minimal?
  Indirekte Annahme: T wäre nicht minimal.
  Dann existiert ein aufspannender Baum B mit $w(B) < w(T)$. Außerdem existiert eine Kante $e \in B$ mit $e \notin T$. Wir bauen nun T schrittweise zu B um; wenn wir dies schaffen, ohne eine teurere Kante durch eine billigere zu ersetzen, haben wir den erwünschten Widerspruch erreicht.
  $T \cup \{e\}$ enthält einen Kreis C. In C existiert eine Kante f mit $f \in T$ und $f \notin B$.
  Behauptung: $w(e) \geq w(f)$, denn sonst wäre e vor f angeschaut worden und zu T hinzugefügt worden. Denn das Hinzufügen von e zu dem aktuellen Unterbaum von T hätte keinen Kreis C' erzeugen können, da sonst $(C \cup C') \cap T$ und somit T einen Kreis enthalten würde. Wir setzen also $T' = T \cup \{e\} \setminus \{f\}$. T' ist weiterhin ein aufspannender Baum und es gilt $w(T') \geq w(T)$.
  Wir bauen T' weiter um bis $T' = B$ erreicht ist. $\surd$

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, für jede Abfrage den DFS-Algorithmus für $T \cup \{e\}$ aufzurufen. Dies würde eine Gesamtlaufzeit von $O(|V||E|)$ ergeben, da sich in T bis zu $|V|$ Kanten befinden können, was einer Laufzeit von $O(|V|)$ für einen DFS-Aufruf entspricht.

Im Folgenden werden wir sehen, wie wir für den zweiten Teil des Kruskal-Algorithmus, d.h. alles nach dem Sortieren der Kanten, mit Hilfe der Datenstruktur *Union-Find* eine fast lineare Laufzeit erhalten können und damit der Gesamtaufwand durch das Kantensortieren bestimmt wird ($= O(|E| \log |E|)$).

7.1.3 Implementierung von Kruskal mittels Union-Find

Die Idee besteht darin, eine Menge von Bäumen, die der aktuellen Kantenmenge T entspricht, iterativ zu einem Baum zusammenwachsen zu lassen („Wir lassen Wälder wachsen"). Man beginnt mit Bäumen, die jeweils nur aus einem Knoten des gegebenen Graphen

$G = (V, E)$ bestehen. Dann werden nach jeder Hinzunahme einer Kante (u, v) zur anfangs leeren Menge T die Bäume, in denen u und v liegen, zu einem Baum verbunden (*union*). Eine Hilfsfunktion liefert hierzu jeweils die Information, in welchem Baum ein Knoten liegt. Liegen u und v bereits im gleichen Baum, dann würde die Hinzunahme der Kante (u, v) zu einem Kreis führen.

Wir betrachten das folgende Beispiel zum Graphen aus Abb. 7.1:

Komponenten (Bäume)	nächste Kante	Hinzunahme zu T
$\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g\},\{h\},\{i\}$	(g,h)	$\surd$
$\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g,h\},\{i\}$	(c,i)	$\surd$
$\{a\},\{b\},\{c,i\},\{d\},\{e\},\{f\},\{g,h\}$	(f,g)	$\surd$
$\{a\},\{b\},\{c,i\},\{d\},\{e\},\{f,g,h\}$	(c,f)	$\surd$
$\{a\},\{b\},\{c,f,g,h,i\},\{d\},\{e\}$	(a,b)	$\surd$
$\{a,b\},\{c,f,g,h,i\},\{d\},\{e\}$	(g,i)	nein!

(g, i) wird nicht hinzugefügt, weil g und i bereits im gleichen Baum liegen, d.h. verbunden sind. Die Hinzunahme der Kante (g, i) würde also einen (verbotenen) Kreis erzeugen. Weiter geht es dann z.B. wie folgt:

Komponenten (Bäume)	nächste Kante	Hinzunahme zu T
$\{a,b\},\{c,f,g,h,i\},\{d\},\{e\}$	(c,d)	$\surd$
$\{a,b\},\{c,d,f,g,h,i\},\{e\}$	(h,i)	nein!
$\{a,b\},\{c,d,f,g,h,i\},\{e\}$	(b,c)	$\surd$
$\{a,b,c,d,f,g,h,i\},\{e\}$	(a,h)	nein!
$\{a,b,c,d,f,g,h,i\},\{e\}$	(d,e)	$\surd$

In diesem Schritt wurde ein kompletter MST erzeugt, daher werden die restlichen Kanten abgelehnt. Man kann also den Algorithmus bereits an dieser Stelle abbrechen, weil ohnehin nur mehr Kreise entstehen würden, wie die probeweise Weiterführung zeigt:

Komponenten (Bäume)	nächste Kante	Hinzunahme zu T
$\{a,b,c,d,e,f,g,h,i\}$	(e,f)	nein!
$\{a,b,c,d,e,f,g,h,i\}$	(b,h)	nein!
$\{a,b,c,d,e,f,g,h,i\}$	(d,f)	nein!

Im Folgenden betrachten wir die *Union-Find* Datenstruktur genauer.

Abstrakter Datentyp: Dynamische Disjunkte Menge (DDM)

Wir betrachten den abstrakten Datentyp DDM, der die *Union-Find* Datenstruktur beschreibt. Die Objekte sind eine Familie $S = \{S_1, S_2, \ldots, S_k\}$ disjunkter Teilmengen einer endlichen Menge V. Jede Teilmenge S_i hat einen Repräsentanten $s_i \in S_i$.
Die Operationen sind die folgenden: Seien $v, w \in V$,

- *makeset*(v): erzeugt eine neue Menge $\{v\}$; $S = S \cup \{\{v\}\}$; hierbei gilt die Annahme, dass $v \notin S_i$ für alle $i = 1, \ldots, k$.
- *union*(v, w): vereinigt die Mengen S_v und S_w, deren Repräsentanten v und w sind, zu einer neuen Menge $S_v \cup S_w$. Der Repräsentant von $S_v \cup S_w$ ist ein beliebiges Element aus $S_v \cup S_w$; $S = S \setminus \{S_v, S_w\} \cup \{S_v \cup S_w\}$.
- *findset*(v): liefert den Repräsentanten der (eindeutigen) Menge, die v enthält.

Mit Hilfe dieses abstrakten Datentyps können wir nun den Kruskal-Algorithmus genauer formulieren:

Algorithmus 37 Kruskal-Minimum-Spanning-Tree(G)

Eingabe: Graph $G = (V, E)$ mit Gewichten w_e für alle $e \in E$
Ausgabe: Kantenmenge T des Minimum Spanning Tree von G
Variable(n): Dynamische disjunkte Menge S

```
1: S = ∅; T = ∅; i = 1;
2: Sortiere Kanten: w(e_1) ≤ w(e_2) ≤ w(e_m);
3: für alle Knoten v aus V {
4:    makeset(v);
5: }
6: solange |T| < |V| − 1 {
7:    // Sei e_i = (u_i, v_i) die nächste Kante
8:    falls findset(u_i) ≠ findset(v_i) dann {
9:       T = T ∪ (u_i, v_i);
10:      union(findset(u_i), findset(v_i));
11:   }
12:   i = i + 1;
13: }
```

Man beachte, dass nicht unbedingt jede Kante von E betrachtet werden muss, sondern abgebrochen wird, wenn ein vollständiger Spannbaum erreicht wurde. An der Ordnung des Aufwands ändert dies jedoch nichts, da im Worst-Case die teuerste Kante benötigt wird.

Realisierung des Datentyps DDM:

Jede Menge S_i wird durch einen gerichteten Baum repräsentiert. Die Knoten des Baumes sind die Elemente der Menge. Jeder Knoten im Baum enthält einen Verweis auf seinen Vorgänger. Die Wurzel markieren wir dadurch, dass sie auf sich selbst zeigt. So besitzt jeder Knoten genau eine ausgehende Kante, und zur Speicherung der Bäume genügt ein einfaches Feld *parent*$[v]$. Abbildung 7.2 zeigt die Situation unseres Beispiels vor und nach dem Hinzufügen der Kante (c, f).

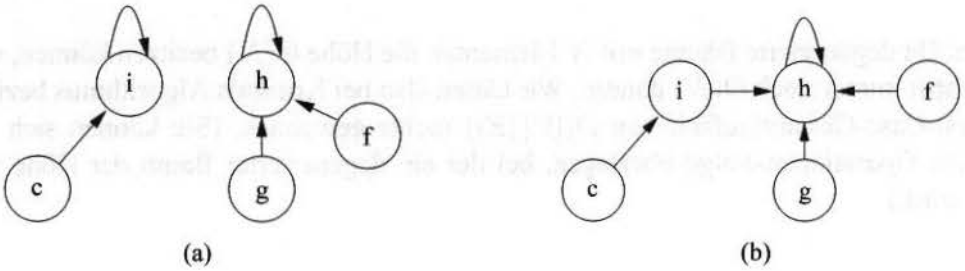


Abbildung 7.2: Dynamische disjunkte Mengen: (a) zwei gerichtete Bäume, die die Mengen $\{c, i\}$ und $\{g, h, f\}$ repräsentieren; (b) nach der Vereinigung der Mengen.

Algorithmus 38 makeset (v)

1: *parent*$[v] = v$

Algorithmus 39 union (v, w)

1: *parent*$[v] = w$

Algorithmus 40 findset (v)

1: $h = v$;
2: **solange** *parent*$[h] \neq h$ {
3: $h =$*parent*$[h]$;
4: }
5: retourniere h;

parent$[v]$ sieht danach (Abbildung 7.2) folgendermaßen aus:

v	a	b	c	d	e	f	g	h	i
parent$[v]$	a	b	i	d	e	h	h	h	h

Die Realisierung der Operationen der *Union-Find* Datenstruktur DDM ist nun nicht nur offensichtlich, sondern vor allem auch kurz und elegant, wie man an den Implementierungen Algorithmus 38, Algorithmus 39, und Algorithmus 40 sieht.

Analyse der Laufzeit:

- *makeset*: $O(1)$
- *union*: $O(1)$
- *findset*: $O(h(\text{Baum})) = O(N)$ nach N Union-Operationen

Problem: Da degenerierte Bäume mit N Elementen die Höhe $\Theta(N)$ besitzen können, würde jeder Schritt immer noch $O(N)$ dauern. Wir hätten also bei Kruskals Algorithmus bezüglich der Worst-Case-Gesamtlaufzeit von $O(|V|\,|E|)$ nichts gewonnen. (Sie können sich leicht selbst eine Operationen-Folge überlegen, bei der ein degenerierter Baum der Höhe $\Theta(N)$ erzeugt wird.)

Verbesserung 1: Vereinigung nach Größe oder Höhe:

Idee: Um zwei Bäume mit Wurzeln u und v zu vereinigen, macht man die Wurzel des Baumes mit kleinerer Größe (bzw. Höhe) zum direkten weiteren Nachfolger der Wurzel des Baumes mit größerer Größe (bzw. Höhe). Der Repräsentant des größeren bzw. höheren Baumes wird somit zum Repräsentanten der neuen Menge. Hierzu muss man sich zusätzlich zu jedem Baum dessen Größe bzw. Höhe in der Wurzel merken.

Lemma: Dieses Verfahren liefert Bäume mit Höhe $h \leq \log N$ nach N Operationen (ohne Beweis).

Folgerung: Die *findset*-Operation benötigt im Worst-Case $\Theta(\log N)$ Schritte.

Verbesserung 2: Methode der Pfadverkürzung (Kompressionsmethode):

Idee: Man erhält eine weitere Verkürzung der Pfade, und somit eine Laufzeiteinsparung der *findset*-Operation, wenn man beim Durchlauf jeder *findset*-Operation alle durchlaufenen Knoten direkt an die Wurzel anhängt. Bei der Ausführung von *findset* durchläuft man also den gleichen Pfad zweimal: das erste Mal, um die Wurzel zu finden und das zweite Mal, um den Vorgänger aller Knoten auf dem durchlaufenen Pfad auf die Wurzel zu setzen. Dies verteuert zwar die *findset*-Operation geringfügig, führt aber zu einer drastischen Reduktion der Pfadlängen, denn man kann das folgende Theorem zeigen:

Theorem (Amortisierte Analyse): Sei M die Anzahl aller *union*-, *makeset*-, *findset*-Operationen und N die Anzahl der Elemente in allen Mengen, $M \geq N$: Die Ausführung der M Operationen benötigt $\Theta(M \cdot \alpha(M, N))$ Schritte.

$\alpha(M, N)$ heißt die *inverse Ackermannfunktion*. Ihr Wert erhöht sich mit wachsendem M bzw. N nur extrem langsam. So ist $\alpha(M, N) \leq 4$ für alle praktisch auftretenden Werte von M und N (solange $M \leq 10^{80}$).

Damit ist die Ausführung einer *findset*-Operation durchschnittlich (gemittelt über alle M Operationen) innerhalb des Kruskal-Algorithmus in praktisch konstanter Zeit möglich. Dies führt zu einer praktisch linearen Laufzeit des zweiten Teils des Kruskal-Algorithmus, nachdem die Kanten sortiert worden sind. Der Gesamtaufwand wird nun durch das Kantensortieren bestimmt und ist somit $O(|E| \log |E|)$.

7.1.4 Der Algorithmus von Prim

Ein weiterer, exakter Algorithmus zur Bestimmung eines MST wurde von Prim 1957 vorgeschlagen. Dieser Ansatz arbeitet ebenfalls nach dem Greedy-Prinzip. Wir wollen diesen Algorithmus hier nur kurz zusammenfassen. Im Gegensatz zu Kruskals Algorithmus wird von einem Startknoten ausgegangen, und neue Kanten werden immer nur zu einem bestehenden Baum hinzugefügt:

Algorithmus 41 Prim-MST(G)

Eingabe: Graph $G = (V, E)$ mit Gewichten w_e für alle $e \in E$
Ausgabe: Kantenmenge T des Minimum Spanning Tree von G
Variable(n): Menge C der bereits „angeschlossenen" Knoten

1: Wähle einen beliebigen Startknoten $s \in V$;
2: $C = \{s\}$;
3: $T = \emptyset$;
4: **solange** $|C| \neq |V|$ {
5: Ermittle eine Kante $e = (u, v)$ mit $u \in C, v \in V \setminus C$ und minimalem Gewicht w_e;
6: $T = T \cup \{e\}$;
7: $C = C \cup \{v\}$;
8: }

Schritt (5), das Ermitteln einer günstigsten Kante zwischen dem bestehenden Baum und einem noch freien Knoten, kann effizient gelöst werden, indem alle noch freien Knoten $V \setminus C$ in einem Heap verwaltet werden. Die Sortierung erfolgt dabei nach dem Gewicht der jeweils günstigsten Kante, die einen freien Knoten mit dem aktuellen Baum verbindet.

Der Heap muss nach dem Hinzufügen jeder Kante aktualisiert werden. Konkret wird für jeden verbleibenden freien Knoten überprüft, ob es eine Kante zum gerade angehängten Knoten v gibt und diese günstiger ist als die bisher bestmögliche Verbindung zum Baum.

Unter Verwendung des beschriebenen Heaps ist der Gesamtaufwand des Algorithmus von Prim $O(|V|^2)$. Damit ist dieser Ansatz für dichte Graphen ($|E| = \Theta(|V|^2)$) etwas besser geeignet als jener von Kruskal. Umgekehrt ist Kruskals Algorithmus für dünne Graphen ($|E| \approx \Theta(|V|)$) effizienter.

Weiterführende Literatur

Das Minimum Spanning Tree Problem wird z.B. in den Büchern von Sedgewick und Cormen, Leiserson und Rivest (siehe Literaturliste (2) bis (4) auf Seite 14) ausführlich beschrieben. In letzterem Buch finden sich auch eine weitergehende Beschreibung zu Greedy-Algorithmen, eine genaue Definition und Behandlung der Ackermann-Funktion und ihrer Inversen, sowie Informationen zur amortisierten Analyse.

7.2 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 und das Acht-Damen-Problem.

7.2.1 Das 0/1-Rucksack-Problem

Gegeben: N Gegenstände g_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 Rucksackproblem lässt sich nun formal folgendermaßen definieren:

Maximiere

$$\sum_{i=1}^{N} c_i x_i,$$

wobei

$$x_i \in \{0,1\} \wedge \sum_{i=1}^{N} w_i x_i \leq K.$$

Beispiel: 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

Algorithmus 42 Enum $(z, xcost, xweight, x)$;

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: falls xweight ≤ K dann {
2:   falls xcost > maxcost dann {
3:     maxcost = xcost;
4:     bestx = x;
5:   }
6:   für i = z + 1, ..., N {
7:     x[i] = 1;
8:     Enum (i, xcost + c[i], xweight + w[i], x);
9:     x[i] = 0;
10:  }
11: }
```

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.

Ein Enumerationsalgorithmus für das 0/1-Rucksackproblem

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. Algorithmus 42) 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.

Analyse:

Korrektheit: Zeilen (6)-(9) enumerieren über alle möglichen Teilmengen einer N-elementigen Menge. Zeilen (1)-(5) sorgen dafür, dass nur zulässige Lösungen betrachtet werden und die optimale Lösung gefunden wird. $\surd$

Laufzeit: $O(2^N)$.

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 7.3.1 betrachten werden.

7.2.2 Das Acht-Damen-Problem

Wir wollen nun ein weiteres kombinatorisches Problem betrachten, das wir mit einer Enumeration aller Möglichkeiten lösen wollen. Das *Acht-Damen-Problem* ist ein klassisches Schachproblem, das von Bezzel 1845 formuliert wurde. Die Aufgabe verlangt, acht Damen auf einem Schachbrett so aufzustellen, dass keine Dame von einer anderen bedroht wird. Bedrohungen gehen dabei gemäß den Schachregeln in horizontaler, vertikaler und diagonaler Richtung von jeder Dame aus, wie Abbildung 7.3 zeigt.

Bei diesem Problem geht es daher im Gegensatz zu vielen anderen nicht darum, aus einer großen Menge gültiger Lösungen eine zu finden, die eine Zielfunktion maximiert oder minimiert, sondern nur darum, überhaupt eine gültige Lösung zu finden („*Constraint satisfaction*" Problem).

Wir wollen hier dieses Problem verallgemeinern und gehen von n Damen und einem $n \times n$ Spielfeld aus. Die einzelnen Spalten bezeichnen wir anstatt mit Buchstaben nun wie die Zeilen mit den Zahlen $1, \ldots, n$. Außerdem wollen wir nicht nur eine richtige Lösung, sondern alle erzeugen.

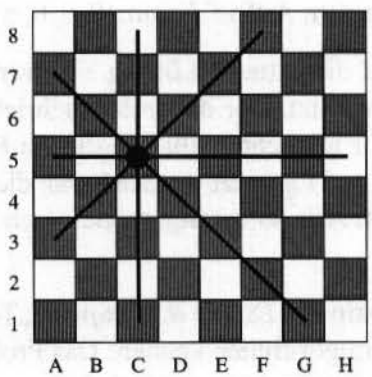


Abbildung 7.3: Bedrohte Felder einer Dame im Acht-Damen-Problem.

Ein naiver Ansatz, alle Möglichkeiten der Damenaufstellungen zu enumerieren, ist in Algorithmus 43 beschrieben. Dabei ist $D[i]$ ein Vektor, der die Position für jede Dame $i = 1, \ldots, n$ speichert.

Algorithmus 43 Naive-Enumeration(D, i)

Eingabe: aktuelle Stellung D; erste noch nicht platzierte Dame i
Ausgabe: komplette korrekte Stellungen

```
1: für D[i] = (1,1) bis (n,n) {
2:   falls i < n dann {
3:     Naive-Enumeration(D, i+1)
4:   } sonst {
5:     Überprüfe D und gib es aus, falls korrekt // Komplette Platzierung erreicht
6:   }
7: }
```

Ein Aufruf dieses rekursiven Algorithmus mit *Naive-Enumeration(D,1)* erzeugt alle Kombinationen von allen möglichen Platzierungen jeder einzelnen Dame auf eines der $n \times n$ Felder. Insgesamt werden daher $(n^2)^n = n^{2n}$ komplette Stellungen erzeugt und auf Korrektheit überprüft. Das Überprüfen auf Korrektheit erfordert aber nochmals einen Aufwand von $\Theta(n^2)$, womit der Gesamtaufwand $\Theta(n^{2n+2})$ ist!

Für $n = 8$ sind ca. $2,8 \cdot 10^{14}$ Stellungen zu erzeugen. Selbst wenn 1 Million Stellungen pro Sekunde überprüft werden könnten, würde das ca. 9 Jahre dauern. Wir sehen also, dass ein derart naives Vorgehen sehr teuer kommen würde.

Durch Berücksichtigung folgender Beobachtungen kann das Verfahren entscheidend verbessert werden:

1. In einer korrekten Lösung muss in jeder Zeile genau eine Dame stehen.
2. Ebenso muss in jeder Spalte genau eine Dame stehen.
3. Ein paarweises Vertauschen von Damen ändert nichts an einer Lösung.

Wir fixieren daher zunächst, dass Dame i immer in Zeile i zu stehen kommt. Die Spalten aller Damen müssen immer unterschiedlich sein. Daher können wir nun alle möglichen Stellungen, welche die oberen Bedingungen erfüllen, enumerieren, indem wir alle Permutationen der Werte $\{1, \ldots, n\}$ erzeugen. Wenn Π eine solche Permutation ist, stellt dann $\Pi[i]$ die Spalte für die Dame in der i-ten Zeile dar.

Mit Algorithmus 44 können wir alle Permutationen erzeugen und die entsprechenden Stellungen auf Richtigkeit überprüfen. F ist dabei die Menge aller bisher noch nicht verwendeten Spalten. Der Algorithmus wird mit *Alle-Permutationen*(Π,$\{1, \ldots, n\}$,1) gestartet, wobei Π nicht initialisiert zu sein braucht.

Algorithmus 44 Alle-Permutationen(Π, F, i)

Eingabe: aktuelle Teillösung Π; Menge F der noch nicht einer Dame zugewiesenen Spalten; erste noch nicht fixierte Position (Dame) i in Π

Ausgabe: komplette korrekte Stellungen

```
1: für alle s ∈ F {
2:    Π[i] = s;
3:    falls i < n dann {
4:       Alle-Permutationen(Π, F \ {s}, i + 1)
5:    } sonst {
6:       Überprüfe Π auf diagonales Schlagen und gib Stellung aus, falls korrekt
7:       // Komplette Stellung erreicht
8:    }
9: }
```

Nun werden „nur" mehr $n!$ Stellungen erzeugt und auf Korrektheit untersucht ($8! = 40320$). Die rekursiven Aufrufe können wie in Abbildung 7.4 gezeigt als Baum dargestellt werden.

Wir können das Verfahren weiter beschleunigen, indem wir so früh wie möglich ganze Gruppen von Permutationen erkennen, die keine gültige Lösungen darstellen, und diese erst gar nicht vollständig erzeugen. In Abbildung 7.4 würde das bedeuten, dass wir bei Konflikten in bereits fixierten Knoten alle weiteren Unterbäume „abschneiden".

Man bezeichnet allgemein ein solches Vorgehen als *begrenzte Enumeration* im Gegensatz zur *vollständigen Enumeration*.

Die folgende Variante des Algorithmus überprüft unmittelbar bei der Platzierung der i-ten Dame, ob diese nicht von einer bereits vorher platzierten bedroht wird (s. Algorithmus 45). Nur wenn dies nicht der Fall ist, wird mit der Rekursion fortgesetzt. Der Überprüfung auf diagonales Schlagen zweier Damen liegt die Idee zu Grunde, dass bei einer Bedrohung der Absolutbetrag der Spaltendifferenz gleich dem Absolutbetrag der Zeilendifferenz sein muss.

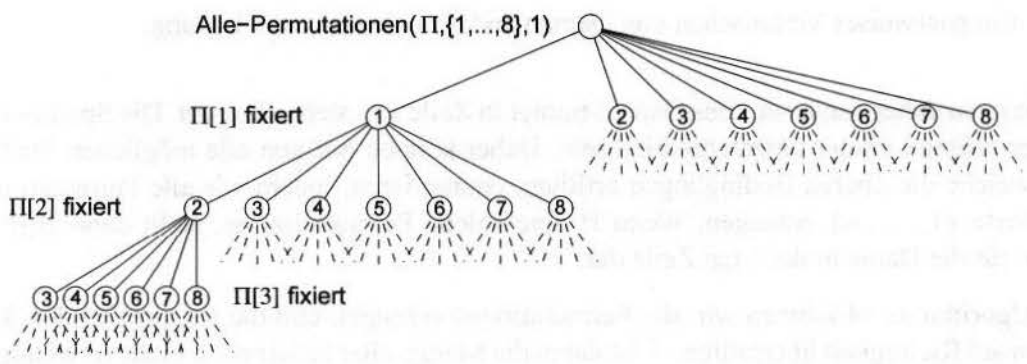


Abbildung 7.4: Rekursive Aufrufe in *Alle-Permutationen*.

Algorithmus 45 Begrenzte-Enumeration(Π, F, i)

Eingabe: aktuelle Teillösung Π; Menge F der noch nicht einer Dame zugewiesenen Spalten; erste noch nicht fixierte Position (Dame) i in Π

Ausgabe: komplette korrekte Stellungen

```
1:  für alle s ∈ F {
2:    Π[i] = s;
3:    j = 1;
4:    solange j < i und |Π[i] − Π[j]| ≠ i − j {
5:      j = j + 1
6:    }
7:    falls j = i dann {
8:      // keine Bedrohung
9:      falls i < n dann {
10:       Begrenzte-Enumeration(Π, F \ {s}, i + 1)
11:     } sonst {
12:       // Komplette, gültige Stellung erreicht
13:       Gib Stellung aus
14:     }
15:   }
16: }
```

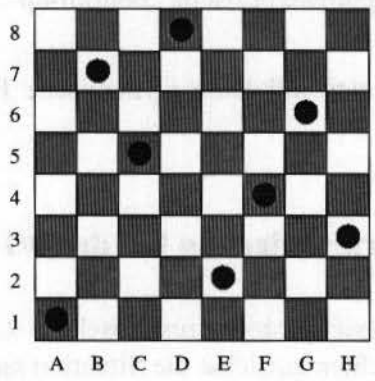


Abbildung 7.5: Eine Lösung zum Acht-Damen-Problem.

Die exakte Berechnung des Aufwands ist nun schwierig, in der Praxis aber kommt es bei $n = 8$ nun nur mehr zu 2057 rekursiven Aufrufen im Vergleich zu 69281 beim Enumerieren aller Permutationen. Eine richtige Lösung ist in Abbildung 7.5 dargestellt.

Eine weitere Verbesserung ist die Ausnutzung von Symmetrien. So können wir uns in der Enumeration bei der Platzierung der ersten Dame auf die Spalten $1, \ldots, n/2$ beschränken. Zu jeder dann gefundenen Lösung generieren wir automatisch gleich die um die Vertikale

gespiegelte Lösung:

$$\Pi'[i] = n - \Pi[i] + 1, \quad \forall i = 1, \ldots, n.$$

Der Aufwand der Enumeration sinkt dadurch um den Faktor 2. Auch horizontale Spiegelungen und Rotationen können ausgenutzt werden, um den Aufwand weiter zu senken.

7.3 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 mögliche Ergebnisse so ab, dass sie zur Lösung größerer Probleme verwendet werden können.

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 betrachten im Folgenden eine effiziente Dynamische Programmierung für das 0/1-Rucksackproblem.

7.3.1 Dynamische Programmierung für das 0/1-Rucksackproblem

Eine Möglichkeit, das 0/1-Rucksackproblem aus Abschnitt 7.2.1 in Teilprobleme zu zerlegen, ist die folgende: Wir betrachten zunächst die Situation nach der Entscheidung über die ersten l Gegenstände und merken uns für alle möglichen Gesamtwerte c diejenigen Lösungen, die jeweils das kleinste Gewicht erhalten. Wir definieren:

$$b_l(c) = \min \left\{ \sum_{i=1}^{l} w_i x_i \mid x_i \in \{0,1\} \wedge \sum_{i=1}^{l} c_i x_i = c \right\}$$

Für jedes $l = 1, \ldots, N$ erhalten wir somit ein Teilproblem.

Die Funktion $b_l(c)$ kann rekursiv folgendermaßen bestimmt werden:

$$b_l(c) = \min\{b_{l-1}(c), b_{l-1}(c - c_l) + w_l\} \text{ für } l = 2, \ldots, N,$$

$$\text{und } b_1(c) = \begin{cases} w_1 & \text{falls } c = c_1 \\ 0 & \text{falls } c = 0 \\ \infty & \text{sonst} \end{cases}$$

In Worten bedeutet dies: Wir können $b_l(c)$ aus den bereits bekannten Funktionswerten für kleinere l folgendermaßen berechnen: Im Entscheidungsschritt l wird entweder der Gegenstand l nicht in den Rucksack gepackt – in diesem Fall wird der Wert $b_l(c)$ von $b_{l-1}(c)$ einfach übernommen – oder Gegenstand l wird in den Rucksack gepackt. Das minimale Gewicht des neu gepackten Rucksacks mit Gegenstand l und Wert c erhalten wir dann als Summe aus dem minimalen Gewicht des Rucksacks vor dem Packen von l mit Wert gleich $c - c_l$ und dem Gewicht von Gegenstand l.

Problem: Effiziente Bestimmung aller $b_l(0), \ldots, b_l(c)$?
Lösung: Entscheide iterativ über alle Gegenstände $l = 1, \ldots, N$.

In unserem Beispiel sieht das folgendermaßen aus:

1. Entscheidung über den ersten Gegenstand a:

$$x_1 = \begin{cases} 0: & b_1(0) = 0 \\ 1: & b_1(3) = 3 \end{cases}$$

2. Entscheidung über den zweiten Gegenstand b:

$$x_2 = \begin{cases} 0: & b_2(0) = 0 \text{ und } b_2(3) = 3 \\ 1: & b_2(5) = 4 \text{ und } b_2(8) = 7 \end{cases}$$

Man sieht hier, dass $b_l(c)$ nicht für alle möglichen c wirklich berechnet wird, sondern nur für die relevanten Fälle. Man kann sich das sehr gut mit einem Entscheidungsbaum veranschaulichen (s. Abb. 7.6). Entscheidend ist, dass bei zwei Rucksäcken mit gleichen c-Werten derjenige mit dem kleinsten Gewicht behalten wird und der andere verworfen wird. Einträge mit $b_l(c) > K$ sind irrelevant und werden nicht betrachtet.

Dynamischer Programmierungsalgorithmus für das Rucksackproblem

Sei M_l die Menge aller Elemente $(S, c, b_l(c))$ in Stufe l, wobei S die Menge der jeweils eingepackten Gegenstände ist. Der Algorithmus *Dynamische Programmierung 0/1-Knapsack* berechnet M_l iterativ für alle $l = 1, \ldots, N$.

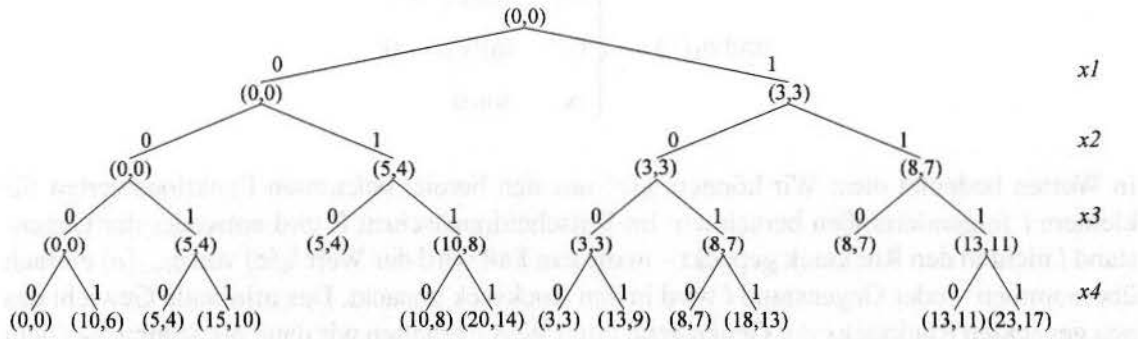


Abbildung 7.6: Teil des Entscheidungsbaumes bis $l = 4$ unseres Beispiels zur Dynamischen Programmierung; die Werte in Klammern bezeichnen das erreichte $(c, b_l(c))$.

Algorithmus 46 Dynamische Programmierung 0/1-Knapsack

Eingabe: N Gegenstände mit Gewichten $w_1, \ldots, w_N$ und Werten $c_1, \ldots, c_N$; Rucksack der Größe K

Ausgabe: Eingepackte Gegenstände S mit maximalem Wert unter allen Rucksäcken mit Gewicht kleiner gleich K

Variable(n): Mengen $M_0, \ldots, M_N$ aller Tripel $(S, c, b_l(c))$ in allen Stufen

1: Sei $M_0 = \{(\emptyset, 0, 0)\}$;
2: **für** $l = 1, \ldots, N$ {
3: Sei $M_l = \emptyset$
4: **für** jedes $(S, c, b) \in M_{l-1}$ {
5: $M_l = M_l \cup \{(S, c, b)\}$
6: **falls** $b + w_l \leq K$ **dann** {
7: $M_l = M_l \cup \{(S \cup \{l\}, c + c_l, b + w_l)\}$
8: }
9: }
10: Untersuche M_l auf Paare mit gleichem c und behalte für jedes solche Paar das Element mit kleinstem Gewicht
11: }
12: Optimale Lösung ist dasjenige Element (S, c, b) in M_N mit maximalem c.

Analyse:

Korrektheit: Es wird jeweils in Schritt l ($l = 1, \ldots, N$) unter allen möglichen Lösungen mit Wert c die „leichteste" behalten. Nach N Schritten muss die optimale Lösung dabei sein. $\surd$
Laufzeit: Die Laufzeit hängt von der Realisierung der Schritte (4)-(7) sowie (10) ab.

Um Schritt (10) effizient durchzuführen – siehe Algorithmus 47 – führen wir ein Array $Y[0, \ldots, c^l_{\max}]$ von Verweisen auf Tripel in M_l ein; dabei ist $c^l_{\max}$ der größte auftretende c-Wert in M_l. Wir gehen davon aus, dass alle c_i und c immer ganzzahlig sind.

Jede Menge M_l enthält höchstens c_{opt} Elemente, wobei c_{opt} den maximalen Lösungswert

Algorithmus 47 Beispiel für Realisierung von Schritt 10

Variable(n): Feld $Y[0,\ldots,c^l_{\max}]$ von Verweisen auf Tripel in M_l mit kleinsten Gewichten

```
1: Bestimme c^l_max unter den Elementen in M_l;
2: für c = 0,...,c^l_max {
3:    Y[c] = NULL;
4: }
5: für alle Tripel (S, c, b_l(c)) in M_l {
6:    falls Y[c] == NULL dann {
7:       Y[c] =Verweis auf (S, c, b_l(c)) in M_l
8:    } sonst {
9:       falls Y[c].b_l(c) ≤ b_l(c) dann {
10:         Entferne (S, c, b_l(c)) aus M_l
11:      } sonst {
12:         Entferne Y[c] aus M_l;
13:         Y[c] =Verweis auf (S, c, b_l(c)) in M_l
14:      }
15:   }
16: }
```

bezeichnet. Die Laufzeit der Schritte (4)-(7) hängt davon ab, wie man die Kopien der Menge S realisiert; kopiert man sie jedesmal, dann ergibt sich insgesamt eine Gesamtlaufzeit von $O(N^2 c_{\text{opt}})$, wobei c_{opt} den optimalen Lösungwert bezeichnet. Gibt man sich mehr Mühe bei der Organisation der Menge S, dann kann man auch Laufzeit $O(N c_{\text{opt}})$ erreichen.

Algorithmus 48 zeigt eine alternative Möglichkeit für eine Implementierung der Idee der Dynamischen Programmierung. Wir verwenden die folgende Datenstruktur:
Wir speichern $b_l(0),\ldots,b_l(c)$ im Feld $A[0,\ldots,c_m]$, wobei hier c_m eine obere Schranke für c ist. Im Feld $A_S[0,\ldots,c_m]$ werden die korrespondierenden Mengen von eingepackten Gegenständen (S) gespeichert. Für die Berechnung der nächsten Ebene wird das Feld A in ein Feld B und Feld A_S in ein Feld B_S kopiert, um die zur Ableitung benötigten Daten nicht vorzeitig zu überschreiben.

Für die Berechnung von c_m sortieren wir die Gegenstände absteigend nach ihrem Nutzen. Der Nutzen N_i eines Gegenstands i ist definiert als der relative Wert im Vergleich zum Gewicht: $N_i = c_i/w_i$. Intuitiv sind Gegenstände mit einem größeren Nutzen wertvoller als welche mit geringerem Nutzen, und sind deshalb vorzuziehen. (Jedoch führt ein einfacher Greedy-Algorithmus hier nicht zur Optimallösung.) Nun gibt es folgende Möglichkeiten, um c_m zu bestimmen:

1. Wir nehmen den Gegenstand i mit dem größten Nutzen p Mal, solange bis $p \cdot w_i > K$.
2. Wir multiplizieren den größten Nutzen N_i mit K.
3. Wir packen zunächst den Gegenstand mit größtem Nutzen, dann denjenigen mit zweitgrößtem Nutzen, usw. bis das Gesamtgewicht $\geq K$ ist.

Algorithmus 48 Rucksack Dynamische Programmierung 2

Eingabe: N Gegenstände mit Gewichten $w_1, \ldots, w_N$ und Werten $c_1, \ldots, c_N$; Rucksack der Größe K

Ausgabe: beste Lösung mit ihrem Gesamtwert und Gesamtgewicht

Variable(n): Feld $A[0, \ldots, c_m]$ von Gewichten und entsprechende Mengen von Gegenständen $A_S[0, \ldots, c_m]$ für Level l; Feld $B[0, \ldots, c_m]$ von Gewichten und entsprechende Mengen von Gegenständen $B_S[0, \ldots, c_m]$ für Level $l-1$

1: **für** $c = 0, \ldots, c_m$ {
2: $A[c] = B[c] = \infty$
3: }
4: $B[0] = 0$; $B_S[0] = \emptyset$;
5: **für** $l = 1, \ldots, N$ {
6: **für** $c = 0, \ldots, c_m$ {
7: **falls** $B[c] \neq \infty$ **dann** {
8: // Fall: $x[l] == 0$
9: **falls** $B[c] < A[c]$ **dann** {
10: $A[c] = B[c]$; $A_S[c] = B_S[c]$
11: }
12: // Fall: $x[l] == 1$
13: **falls** $B[c] + w_l < A[c + c_l] \wedge B[c] + w_l \leq K$ **dann** {
14: $A[c + c_l] = B[c] + w_l$; $A_S[c + c_l] = B_S[c] \cup \{l\}$;
15: }
16: }
17: }
18: // kopiere A nach B und A_S nach B_S
19: **für** $c = 0, \ldots, c_m$ {
20: $B[c] = A[c]$; $B_S[c] = A_S[c]$; $A[c] = \infty$
21: }
22: }
23: // suche beste Lösung
24: $c = c_m$;
25: **solange** $B[c] = \infty$ {
26: $c = c - 1$
27: }
28: retourniere $(B_S[c], c, B[c])$

Korrektheit: Auch hier wird jeweils die kleinste Lösung unter allen Werten gleich c nach l Schritten berechnet. Für $l = N$ ist also die Lösung des Rucksack-Problems korrekt.

Laufzeit: $O(N \cdot c_m)$.

Oberflächlich betrachtet sehen die Laufzeiten der beiden Dynamischen Programmierungsalgorithmen von $O(Nc_{\text{opt}})$ bzw. $O(Nc_m)$ polynomiell aus. Allerdings kann c_{opt} (bzw. c_m im

Allgemeinen sehr groß sein. Nur in dem Fall, dass c_{opt} durch ein Polynom in N beschränkt ist, besitzt der Algorithmus polynomielle Laufzeit. Da die Gesamtlaufzeit nicht nur von der Eingabegröße N, sondern auch von den Werten c_i, w_i und K abhängt, nennt man die Laufzeit *pseudopolynomielle Laufzeit*.

Zusammenfassend können wir festhalten, dass das Rucksackproblem mit Hilfe von Dynamischer Programmierung in pseudopolynomieller Zeit exakt lösbar ist. Da in der Praxis c_{opt} meist durch ein Polynom beschränkt ist, führt dies dann zu einer polynomiellen Laufzeit (obwohl das Problem im Allgemeinen NP-schwierig ist).

7.4 Kürzeste Pfade in einem Graphen

Ein weiteres klassisches Problem, das basierend auf dynamischer Programmierung effizient gelöst werden kann, ist die Suche nach einem kürzesten Pfad in einem gewichteten Graphen. Wir betrachten hierzu den von Edsger W. Dijkstra im Jahre 1959 veröffentlichten Algorithmus, der unter der Bezeichnung *Dijkstras Algorithmus* sehr bekannt ist.

Gegeben sei ein gerichteter Graph $G = (V, E)$ mit Bogenlängen $d_{u,v} > 0, \forall (u, v) \in E$, sowie ein Startknoten $s \in V$ (*source*) und ein Zielknoten $t \in V$ (*target*). Gesucht ist der kürzeste Pfad $P \subseteq E$ von Knoten s zu Knoten $t \in V$.

Die Länge eines Pfades P ist hierbei

$$d(P) = \sum_{(u,v) \in P} d_{u,v}.$$

Sei $dist[v]$ die Länge des bisher kürzesten gefundenen Pfades von s zu Knoten v, $\forall v \in V$. Dijkstras Algorithmus basiert auf folgendem rekursiven Zusammenhang:

$$dist[v] = \begin{cases} 0 & \text{für } v = s \\ \min_{\forall u \mid (u,v) \in E} \left(dist[u] + d_{u,v} \right) & \forall v \in V \setminus \{s\} \end{cases}$$

Die Grundidee ist, in einer Datenstruktur Q alle Knoten zu speichern, für die die kürzesten Pfade von s aus noch nicht bekannt sind. Anfangs sind das alle Knoten in V. In $pred[v]$ wird für jeden Knoten $v \in V$ ein Vorgänger (*predecessor*) gespeichert, um damit die kürzesten bisher gefundenen Pfade zu jedem Knoten zu repräsentieren. Nun wird iterativ jeweils ein Knoten v aus Q herausgenommen, der von s aus mit der kleinsten bisher bekannten Pfadlänge erreichbar ist. Anfangs ist $v = s$. Dieser Knoten v wird *besucht* und seine ausgehenden Kanten werden in Betracht gezogen um für die Nachbarknoten $N^+(v)$ gegebenenfalls kürzere als bisher bekannte Pfade (bzw. anfangs überhaupt gültige Pfade) zu finden. Wird irgendwann t besucht, dann terminiert das Verfahren. Algorithmus 49 zeigt Dijkstras Algorithmus in konkretem Pseudocode.

Algorithmus 49 Dijkstra$(G = (V, E), s, t)$

```
1:  für alle Knoten v ∈ V {
2:     dist[v] = ∞; // Distanz des kürzesten Kantenzuges von s nach v
3:     pred[v] = undef; // Vorgänger am kürzesten Pfad von s nach v
4:  }
5:  dist[s] = 0;
6:  Q = V;
7:  solange Q nicht leer {
8:     Finde u ∈ Q mit minimalem dist[u];
9:     Q = Q \ {u};
10:    falls dist[u] = ∞ dann {
11:       Abbruch: Es existiert kein Pfad von s nach t;
12:    }
13:    falls u = t dann {
          // Kürzester Pfad gefunden; gib Distanz und Pfad (rückwärts) aus;
14:       Ausgabe: dist[t];
15:       solange pred[u] ≠ undef {
16:          Ausgabe: u;
17:          u = pred[u];
18:       }
19:       Abbruch;
20:    }
21:    für alle Nachbarknoten v ∈ N+(u) {
22:       distalt = dist[u] + d_{u,v};
23:       falls dist[v] > distalt dann {
24:          dist[v] = distalt;
25:          pred[v] = u;
26:       }
27:    }
28: }
```

Abbildung 7.7 zeigt ein Beispiel, in dem der kürzeste Weg von H nach B in einem ungerichteten Graphen gesucht wird. Die Kantenbeschriftungen geben die Kantenlängen $d_{u,v}$ an, besuchte Knoten werden eingefärbt und die in Klammern stehenden Werte bei den Knoten repräsentieren die Längen $dist[v]$ der bisher bekannten kürzesten Pfade von s nach v.

Korrektheit

Die Korrektheit von Dijkstras Algorithmus ergibt sich aus der obigen Rekursionsgleichung und der Tatsache, dass immer ein noch nicht besuchter Knoten als nächster besucht wird, der auf kürzestem Weg erreicht werden kann. Der Pfad zu diesem Knoten muss daher bereits der tatsächlich kürzeste Pfad sein.