

GRUNDVORLESUNG

DATENSTRUKTUREN

WS 2001/02

von Ingo Wegener

Universität Dortmund
Lehrstuhl Informatik 2
D-44221 Dortmund

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt besonders für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

© Prof. Dr. Ingo Wegener, 1992.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ziele und Überblick	1
1.2	Literatur	4
1.3	Das Maxsummenproblem	6
1.4	Das MAXMIN-Problem	10
1.5	Registermaschinen	12
1.6	Komplexitätsmaße	14
1.7	Größenordnungen	17
2	Grundlegende Datenstrukturen	21
2.1	Motivation und Vorgehensweise	21
2.2	Arrays	21
2.3	Lineare Listen	25
2.4	Datenstrukturen für Mengen	34
2.5	Datenstrukturen für Bäume	36
2.6	Datenstrukturen für Graphen	39
2.7	Datenstrukturen für Intervalle	45
2.8	Datenstrukturen für Partitionen	48
2.9	Datenstrukturen für boolesche Funktionen	52
3	Dynamische Dateien	59
3.1	Vorbemerkungen	59
3.2	Hashing	60
3.3	Binäre Suchbäume	65
3.4	2-3-Bäume	71
3.5	Bayer-Bäume	81
3.6	AVL-Bäume	83
3.7	Skiplisten	90
4	Sortieren	96
4.1	Vorbemerkungen	96
4.2	Sortieren mit binärer Suche	97
4.3	Sortieren durch Mischen	98
4.4	Quicksort	100

4.5	Heapsort	104
4.6	Eine untere Schranke für allgemeine Sortiervverfahren	109
4.7	Bucketsort	113
4.8	Das Auswahlproblem	114
4.9	Sortieren auf Parallelrechnern	116
5	Entwurfsmethoden für Algorithmen	120
5.1	Vorbemerkungen	120
5.2	Greedy Algorithmen	120
5.3	Dynamische Programmierung	126
5.4	Der Algorithmus von Dijkstra	131
5.5	Branch-and-Bound Algorithmen	133
5.6	Eine allgemeine Analyse von divide-and-conquer Algorithmen	138
5.7	Matrixmultiplikation	139
5.8	Die schnelle Fouriertransformation	141
5.9	Nächste Nachbarn in der Ebene	146
5.10	Die Sweepline-Technik	148
5.11	Die Analyse von Spielbäumen	150
5.12	Randomisierte Suchheuristiken	153

Dieses Skript wurde von Alice Czerniejewski, Danny Rozynski und Marion Scheel mit dem Textsystem L^AT_EX geschrieben.

Für die Mitteilung aufgespürter Fehler sind wir immer dankbar.

1 Einleitung

1.1 Ziele und Überblick

Auf Folie 44 der Vorlesung „Programmierung“ (WS 1999/2000 oder WS 2000/01) steht folgendes Zitat von Ghezzi: „A software engineer must of course be a good programmer, be well-versed in data structures and algorithms, ...“. Und damit ist die Motivation für diese Vorlesung mit dem ausführlichen Titel „Datenstrukturen und Entwurfsmethoden für effiziente Algorithmen“ bereits beschrieben. Effizienz ist ein notwendiges (aber nicht hinreichendes) Kriterium für gute Software (und auch Hardware). Dies betrifft alle Bereiche innerhalb der Informatik, wie die Entwicklung von Informationssystemen, Betriebssystemen, verteilten Systemen oder Hardware, und alle Anwendungen der Informatik, wie in der Molekularbiologie, der Logistik, der Physik, der Chemie oder in den Ingenieurwissenschaften. Somit sind die Kenntnisse der Inhalte und Methoden dieser Vorlesung grundlegend für alle Studierenden der Informatik, unabhängig von der späteren Spezialisierung.

Naive Lösungen algorithmischer Probleme können „praktisch unrealisierbar“ sein, da die benötigten Ressourcen an Rechenzeit und/oder Speicherplatz nicht zur Verfügung stehen. Mit Hilfe des Einsatzes geeigneter Datenstrukturen und algorithmischer Methoden lassen sich viele algorithmische Probleme effizient lösen. Die Effizienz kann sich im praktischen Gebrauch erweisen und zuvor mit Experimenten belegt werden. Besser ist es jedoch, ein Produkt mit Gütegarantie herzustellen. Dies bedeutet den formalen Beweis, dass die Datenstruktur oder der Algorithmus das Gewünschte leistet (Korrektheitsbeweis), und die Abschätzung der benötigten Ressourcen (Analyse). Daher gehören zu dieser Vorlesung stets auch Korrektheitsbeweise und Analysen der benötigten Ressourcen. In der Vorlesung „Programmierung“ wurden Effizienzaspekte ausgespart. Die Datenstrukturen, die in der Vorlesung „Programmierung“ vorgestellt wurden, werden hier also unter dem Effizienzaspekt noch einmal beleuchtet. Effizienzanalysen haben häufig direkte Auswirkungen auf den Algorithmenentwurf. Sie decken nämlich Schwachstellen von Algorithmen auf und geben Hinweise darauf, wo sich Verbesserungen besonders lohnen.

Wie entwirft man nun für ein gegebenes Problem einen effizienten Algorithmus? Zunächst benötigen wir grundlegende Kenntnisse über das Gebiet, aus dem das Problem stammt. Dieses Wissen kann in späteren Spezialvorlesungen erlernt werden oder es wird direkt bei der Bearbeitung des Problems erworben. In dieser grundlegenden Vorlesung werden wir nur solche Probleme behandeln, für die derartige Spezialkenntnisse nicht erforderlich sind.

Darüber hinaus ist der Entwurf effizienter Algorithmen ein Handwerk, wobei Meisterleistungen nur mit viel Erfahrung, dem richtigen Gefühl für das Problem und einer Portion Intuition, manchmal auch Glück, erbracht werden. Ziel unserer Vorlesung muss es also sein, das notwendige Handwerkszeug bereitzustellen und dieses praktisch zu erproben.

Die Umsetzung effizienter Algorithmen erfordert schließlich noch den Einsatz passender Datenstrukturen. Wenn wir z. B. das Objekt b in einer Folge a_1, \dots, a_n von Objekten zwischen a_i und a_{i+1} einfügen wollen, hängt die Realisierung dieses Befehls sehr davon ab, wie wir die Daten verwalten. Können wir direkt auf a_i zugreifen oder müssen wir a_i erst suchen? Falls wir a_i suchen müssen, ist das nur auf die umständliche Weise, die

Folge von a_1 bis a_i zu durchlaufen, möglich? Können wir b direkt hinter a_i einfügen, oder müssen wir erst durch Verschiebung von a_{i+1}, \dots, a_n Platz für b schaffen? Passende effiziente Datenstrukturen bilden also das Kernstück aller effizienten Algorithmen.

Die von uns behandelten Probleme sind so ausgewählt, dass es sich einerseits um wichtige und interessante Probleme handelt und andererseits bei der Lösung dieser Probleme allgemeine Prinzipien und Methoden erlernt werden können. Da der zweite Aspekt auf lange Sicht der wichtigere ist, werden wir in Kapitel 1 mit zwei Problemen beginnen, deren Lösung sich nicht lohnen würde, wenn wir dabei nicht das allgemeine Vorgehen exemplarisch kennenlernen würden.

Unser Vorgehen soll unabhängig von speziellen Rechnern und Programmiersprachen sein. Moderne Programmiersprachen enthalten genügend große Anteile „imperativer Programmiersprachen“, um die Umsetzung der von uns behandelten Datenstrukturen und Entwurfsmethoden zu unterstützen. Die Akzeptanz neuer Methoden durch Informatikerinnen und Informatiker in der Praxis und darüber hinaus durch Programmiererinnen und Programmierer hängt wesentlich von der einfachen Verfügbarkeit dieser Methoden ab. Obwohl in zahlreichen Lehrbüchern und Skripten ausführlich beschrieben, wurden effiziente Datenstrukturen in der Praxis viel zu wenig eingesetzt. Inzwischen steht mit LEDA eine Programmbibliothek für die wichtigsten Datenstrukturen zur Verfügung. Wir wollen uns daher in dieser Vorlesung nicht damit aufhalten, Datenstrukturen zum wiederholten Mal zu programmieren.

In diesem einführenden Kapitel werden, wie schon angesprochen, zwei exemplarische Probleme ausführlich gelöst. Um einen Bezugspunkt für die Messung von Rechenzeit und Speicherplatz zu haben, wird das Modell der Registermaschinen als Rechnermodell vorgestellt. Schließlich werden grundlegende Komplexitätsmaße diskutiert und der Gebrauch der O -Notation wiederholt und gerechtfertigt.

In Kapitel 2 werden grundlegende Datenstrukturen wie Arrays, Stacks, Queues, Listen, Bäume und Graphen behandelt und auf einfache Probleme angewendet.

In Kapitel 3 konzentrieren wir uns auf dynamische Datenstrukturen. Dies sind Datenstrukturen, bei denen die zu speichernde Datenmenge nicht statisch vorgegeben ist, sondern sich während der Anwendung dynamisch ändert. Operationen wie das Einfügen und Entfernen von Daten müssen unterstützt werden. Die zwei wichtigsten Methodenklassen sind das Hashing und der Entwurf balancierter Suchbäume.

In Kapitel 4 wird das Problem, n Objekte zu sortieren, ausführlich behandelt. Sortieralgorithmen sind vermutlich die immer noch am häufigsten benutzten Unterprogramme. An diesem Problem werden wir exemplarisch zeigen, dass in der Umgebung paralleler Rechner andere Algorithmen effizient sind als in der Umgebung sequentieller Rechner.

Schließlich wenden wir uns in Kapitel 5 Entwurfsmethoden für effiziente Algorithmen zu. Dabei werden wir allgemeine Methoden kennenlernen und exemplarisch anwenden.

Bei vielen Optimierungsproblemen ist es naheliegend, die einzelnen Teile der Lösung lokal zu optimieren. Wir diskutieren, wann derartige greedy (gierige) Algorithmen zu optimalen oder zumindest guten Lösungen führen. Mit der dynamischen Programmierung wird der Gefahr begegnet, Teilprobleme wiederholt zu behandeln. Dagegen wird mit Branch-

and-Bound Methoden versucht, den Suchraum gezielt so zu durchsuchen, dass auf die Untersuchung großer Bereiche verzichtet werden kann, weil bereits klar ist, dass dort keine optimalen Lösungen liegen. Divide-and-Conquer ist eine wohlbekannte Entwurfsmethode für Algorithmen. Nach einer allgemeinen Analyse dieses Ansatzes wollen wir etwas ausgefallene Anwendungen kennenlernen: die Berechnung nächster Nachbarn in der Ebene, die Multiplikation quadratischer Matrizen und die FFT (Fast Fourier Transform), die in der Bild- und Signalverarbeitung grundlegend ist. Die Sweepline Technik, die viele Anwendungen in der algorithmischen Geometrie hat, wird mit einer Beispielanwendung vorgestellt. Die Methode des Backtracking ist sicherlich den meisten bekannt. Für eine effiziente Anwendung, z. B. in der Schachprogrammierung, ist es auch hier nötig, die Suche so zu gestalten, dass viele Bereiche des Suchraumes nicht näher betrachtet werden. Die Strategie des α - β -Pruning wird diskutiert. Abschließend wird eine Einführung in randomisierte Suchheuristiken gegeben, dazu gehören die randomisierte lokale Suche, Simulated Annealing, Tabu Search und evolutionäre Algorithmen.

Zur Erfolgskontrolle am Ende des Semesters werden die angestrebten Lernziele aufgelistet:

- Kenntnis elementarer Datenstrukturen, ihrer Eigenschaften, Vor- und Nachteile,
- Kenntnis wichtiger Entwurfsmethoden für effiziente Algorithmen,
- Kenntnis effizienter Algorithmen für grundlegende Probleme,
- Erfahrung in der Anwendung von Datenstrukturen und Entwurfsmethoden auf neue Probleme,
- Erfahrung in der Umsetzung von Datenstrukturen und Algorithmen in lauffähige Programme sowie Erfahrung im Einsatz von Programmbibliotheken,
- Kenntnis von Methoden, um die Effizienz von Datenstrukturen zu messen,
- Kenntnis von Methoden, um die Komplexität eines Problems abzuschätzen, d.h. zu zeigen, dass gewisse Ressourcen für die Lösung eines Problems notwendig sind.

Die Verwirklichung dieser Lernziele wird durch Vorlesung, Skript und Übungsgruppen unterstützt, aber nicht garantiert. Neben der Mitarbeit in der Vorlesung (Fragen stellen, mitdenken, mitschreiben, verschiedene Lösungen diskutieren) wird von den Teilnehmerinnen und Teilnehmern der Vorlesung mehr erwartet. Der Stoff der Vorlesung kann nicht durch einfaches Zuhören in der Vorlesung und Lesen im Skript verstanden werden. Insbesondere die komplexeren Sachverhalte können erst dann richtig verstanden werden, wenn sie in eigenen Worten aufgeschrieben worden sind. Dies kann durch kein noch so gutes Skript ersetzt werden. Hilfreich ist auch das Lesen von Lehrbüchern, da auf diese Weise der gleiche Sachverhalt von verschiedenen Standpunkten beleuchtet wird. Schließlich geht schon aus den Lernzielen hervor, dass der Zweck der Vorlesung verfehlt wird, wenn nicht Erfahrungen gesammelt werden. Diese können nur durch eigenständige Lösung von theoretischen und praktischen Übungsaufgaben erworben werden. Dabei ist eine Diskussion

der Aufgaben und von Lösungsansätzen in kleineren Arbeitsgruppen durchaus zu empfehlen. Allerdings ist es im Hinblick auf Seminare, die Diplomarbeit und die Berufspraxis unerlässlich, die Lösungen selbst zu formulieren, die Programme selber zu schreiben und die Lösungen in der Übungsgruppe selbst darzustellen. Die Leiterinnen und Leiter der Übungsgruppe sind nicht dazu da, Musterlösungen an die Tafel zu bringen.

1.2 Literatur

Die Vorlesung folgt keinem Lehrbuch. Jedes Lehrbuch, das sich mit Datenstrukturen und/oder effizienten Algorithmen befasst, ist eine sinnvolle Ergänzung zur Vorlesung. Daher dient die folgende Liste nur als Orientierungshilfe.

- A.V. Aho, J.E. Hopcroft, J.D. Ullman: The design and analysis of computer algorithms, Addison-Wesley, 1974
- A.V. Aho, J.E. Hopcroft, J.D. Ullman: Data structures and algorithms, Addison-Wesley, 1983
- R.K. Ahuja, T.L. Magnanti, J.B. Orlin: Network flows. Theory, algorithms and applications, Prentice-Hall, 1993
- G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi: Complexity and approximation, Springer, 1999
- J. Bentley: Programming pearls, Addison-Wesley, 1986
- T.H. Cormen, C.E. Leiserson, R.L. Rivest: Introduction to algorithms, MIT Press, 1990
- R.L. Graham, D.E. Knuth, O. Patashnik: Concrete mathematics. A foundation for computer science, Addison-Wesley, 1989
- D.S. Hochbaum (Hrsg.): Approximation algorithms for NP-hard problems, PWS Publishing Company, 1997
- D.E. Knuth: The art of computer programming, Band 1-3, Addison-Wesley, 1981
- D.C. Kozen: The design and analysis of algorithms, Springer, 1991
- J.v. Leeuwen (Hrsg.): Handbook of theoretical computer science, Vol.A: Algorithms and complexity, MIT Press, 1990
- S. Martello, P. Toth: Knapsack problems, Wiley, 1990
- K. Mehlhorn: Data structures and algorithms, Band 1-3, Springer, 1984
- R. Motwani, P. Raghavan: Randomized algorithms, Cambridge Univ. Press, 1995

- T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen, BI, 1990
- C.H. Papadimitriou, K. Steiglitz: Combinatorial optimization: algorithms and complexity, Prentice-Hall, 1982
- E.M. Reingold, J. Nievergelt, N. Deo: Combinatorial algorithms: theory and practice, Prentice-Hall, 1977
- R. Sedgewick: Algorithms, Addison-Wesley, 1983
- V.V. Vazirani: Approximation algorithms, Springer, 2000
- I. Wegener: Effiziente Algorithmen für grundlegende Funktionen, Teubner, 1989

Wer sich bisher um das Lesen englischsprachiger Literatur erfolgreich herumgedrückt hat, sollte dieses Manko in diesem Semester ausgleichen.

1.3 Das Maxsummenproblem

Das Maxsummenproblem besteht in folgender Aufgabe. Für eine Folge a_1, \dots, a_n ganzer Zahlen, für die wir auf a_i direkt zugreifen können, sei für $1 \leq i \leq j \leq n$ die Funktion f definiert durch

$$f(i, j) := a_i + \dots + a_j.$$

Es soll ein Paar (i, j) berechnet werden, für das $f(i, j)$ maximal ist.

Für dieses Problem ist mir keine Anwendung bekannt. Allerdings lassen sich grundlegende Vorgehensweisen unserer Vorlesung an Hand von vier Algorithmen zur Lösung dieses Problems veranschaulichen. Als Rechenschritte zählen wir die arithmetischen Operationen und Vergleiche. Der zusätzliche Verwaltungsaufwand wird für alle Algorithmen gering sein.

Algorithmus 1.3.1: (der naive Algorithmus)

Es sollen alle $f(i, j)$, $1 \leq i \leq j \leq n$, berechnet werden und dann der maximale f -Wert ermittelt werden. Offensichtlich genügen zur Berechnung von $f(i, j)$ genau $j - i$ Additionen. Zur Maximumbestimmung setzen wir $MAX := a_1 = f(1, 1)$. Jeder weitere f -Wert wird mit dem aktuellen MAX -Wert verglichen. Wenn ein f -Wert größer als MAX ist, ersetzt er MAX .

Wir kommen zur Analyse des Algorithmus. Die Zahl der Vergleiche $V_1(n)$ ist um 1 kleiner als die Zahl der Paare (i, j) . Es gibt genau j Paare (\cdot, j) . Also ist

$$V_1(n) = \sum_{1 \leq j \leq n} j - 1 = \frac{1}{2}n(n+1) - 1 = \frac{1}{2}n^2 + \frac{1}{2}n - 1.$$

Für die Zahl der Additionen $A_1(n)$ gilt nach unseren Vorüberlegungen

$$\begin{aligned} A_1(n) &= \sum_{1 \leq i \leq n} \sum_{i \leq j \leq n} (j - i) = \sum_{1 \leq i \leq n} \sum_{0 \leq k \leq n-i} k \\ &= \sum_{1 \leq i \leq n-1} \sum_{1 \leq k \leq i} k = \sum_{1 \leq i \leq n-1} \frac{1}{2}i(i+1) \\ &= \frac{1}{2} \left(\sum_{1 \leq i \leq n-1} i^2 + \sum_{1 \leq i \leq n-1} i \right) \\ &= \frac{1}{2} \left(\frac{1}{6}(n-1)n(2(n-1)+1) + \frac{1}{2}(n-1)n \right) \\ &= \frac{1}{6}n^3 - \frac{1}{6}n. \end{aligned}$$

Die Gesamtzahl der Operationen beträgt

$$T_1(n) := V_1(n) + A_1(n) = \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n - 1.$$

Algorithmus 1.3.2: (der normale Algorithmus)

So naiv wie der naive Algorithmus sollte eigentlich niemand sein, selbst vor der Vorlesung Datenstrukturen. Der naive Algorithmus berechnet $a_1 + a_2$ für $f(1, 2), f(1, 3), \dots, f(1, n)$, also $(n - 1)$ -mal. Allgemein ist es einfach, zur folgenden allgemeinen Erkenntnis zu gelangen:

$$f(i, j + 1) = f(i, j) + a_{j+1}.$$

Wenn wir die f -Werte $f(i, \cdot)$ für festes i nach aufsteigenden j -Werten berechnen, genügen also für alle $f(i, \cdot)$ -Werte genau $n - i$ Additionen. Es gilt nun

$$\begin{aligned} V_2(n) &= V_1(n) = \frac{1}{2}n^2 + \frac{1}{2}n - 1, \\ A_2(n) &= \sum_{1 \leq i \leq n} (n - i) = \sum_{1 \leq k \leq n-1} k = \frac{1}{2}n^2 - \frac{1}{2}n, \\ T_2(n) &= n^2 - 1. \end{aligned}$$

Die nächste Verbesserung greift auf eine allgemeine Entwurfsmethode für effiziente Algorithmen zurück. Um einen wesentlichen Fortschritt zu erzielen, dürfen wir nicht mehr alle f -Werte berechnen. Diese Erkenntnis liefert uns die Analyse des normalen Algorithmus! Wir greifen auf den Wahlspruch von Louis XI zurück, der manchmal auch Cäsar zugeschrieben wird: Divide et impera, neudeutsch: Divide and Conquer. Die Lösung eines großen Problems wird auf die Lösung mehrerer kleinerer Probleme zurückgeführt, wobei sich die Lösungen der kleinen Probleme effizient zu einer Lösung des großen Problems verbinden lassen. Die Methode führt typischerweise auf rekursive Algorithmen, da ja die kleineren Probleme wieder in noch kleinere Probleme zerlegt werden, usw. Daher ist es wichtig, dass die kleineren Probleme entweder vom gleichen Typ wie das Ausgangsproblem oder aber einfach zu lösen sind.

Algorithmus 1.3.3: (Divide-and-Conquer Algorithmus)

Der einfacheren Darstellung in diesem einführenden Beispiel wegen nehmen wir an, dass $n = 2^k$ eine Zweierpotenz ist. Wir zerlegen die Menge der zu untersuchenden Paare (i, j) , $1 \leq i \leq j \leq n$, in drei disjunkte Klassen:

- $1 \leq i \leq j \leq n/2$,
- $1 \leq i \leq n/2 < j \leq n$,
- $n/2 < i \leq j \leq n$.

Wenn wir die Probleme für die drei Klassen von Paaren (i, j) gelöst haben, d.h. jeweils auch den maximalen f -Wert kennen, genügen zwei Vergleiche, um den Gesamtsieger zu berechnen. Das erste und das dritte Problem sind vom gleichen Typ wie das Ausgangsproblem, nur hat die Zahlenfolge halbe Länge. Diese Probleme werden rekursiv mit dem

Divide-and-Conquer Ansatz gelöst. Was aber geschieht mit dem zweiten Problem? Hier denken wir uns eine effiziente, direkte Lösung aus. Für $1 \leq i \leq n/2 < j \leq n$ sei

$$g(i) := a_i + \dots + a_{n/2} \text{ und } h(j) = a_{n/2+1} + \dots + a_j.$$

Dann gilt offensichtlich

$$f(i, j) = g(i) + h(j)$$

und, um f zu maximieren, können wir beide Summanden einzeln maximieren. Dazu berechnen wir, analog zum normalen Algorithmus, in dieser Reihenfolge $g(n/2) = a_{n/2}$, $g(n/2 - 1), \dots, g(1)$ und danach den maximalen g -Wert. Es genügen $n/2 - 1$ Vergleiche und $n/2 - 1$ Additionen. Die Berechnung des maximalen h -Wertes verläuft auf analoge Weise. Mit einer Addition des maximalen g -Wertes und des maximalen h -Wertes erhalten wir den maximalen f -Wert für alle Paare (i, j) aus der zweiten Klasse. Das zweite Problem lässt sich also mit $n - 1$ Additionen und $n - 2$ Vergleichen, insgesamt $2n - 3$ Operationen lösen, obwohl die zweite Klasse $n^2/4$ Paare (i, j) enthält.

Wir kommen nun zur Analyse des Divide-and-Conquer Algorithmus. Zur Lösung des Problems lösen wir rekursiv zwei Probleme halber Größe, mit $2n - 3$ Operationen ein andersartiges Problem und berechnen aus diesen drei Lösungen mit 2 Vergleichen die Gesamtlösung. Der rekursive Algorithmus führt also zu einer Rekursionsgleichung für die Zahl der benötigten Operationen:

$$T_3(1) = 0, \quad T_3(2^k) = 2T_3(2^{k-1}) + 2 \cdot 2^k - 1.$$

Wie lösen wir nun derartige Gleichungen? Um ein Gefühl für diese Gleichungen zu bekommen, setzen wir für $T_3(2^{k-1})$ wieder die Rekursionsgleichung ein, usw.

$$\begin{aligned} T_3(2^k) &= 2T_3(2^{k-1}) + 2 \cdot 2^k - 1 \\ &= 4T_3(2^{k-2}) + 2(2 \cdot 2^{k-1} - 1) + 2 \cdot 2^k - 1 \\ &= 4T_3(2^{k-2}) + (2^{k+1} - 2) + (2^{k+1} - 1) \\ &= 8T_3(2^{k-3}) + (2^{k+1} - 4) + (2^{k+1} - 2) + (2^{k+1} - 1) \end{aligned}$$

Nun raten wir die Lösung der Rekursionsgleichung

$$T_3(2^k) = 2^l T_3(2^{k-l}) + \sum_{1 \leq i \leq l} (2^{k+1} - 2^{i-1}).$$

und verifizieren unsere Vermutung mit einem Induktionsbeweis. Wir benutzen, da wir $T_3(1)$ kennen, die spezielle Lösung für $l = k$.

$$\begin{aligned} T_3(2^k) &= 2^k T_3(1) + \sum_{1 \leq i \leq k} (2^{k+1} - 2^{i-1}) \\ &= 2k2^k - (2^k - 1) = (2k - 1)2^k + 1 \\ &= (2 \log n - 1)n + 1. \end{aligned}$$

Am Ende dieser Vorlesung sollte jede aktive Teilnehmerin und jeder aktive Teilnehmer in der Lage sein, in ähnlicher Ausgangssituation zu diesem Algorithmus und seiner Analyse zu kommen. Aber wir haben noch einen Trick im Zylinder.

Algorithmus 1.3.4: (Dynamische Programmierung)

Die allgemeine Strategie der dynamischen Programmierung werden wir erst am Ende der Vorlesung diskutieren. Hier wollen wir speziell auf unser Problem eingehen. Welche Informationen über das Teilproblem auf dem Bereich $[1, k - 1]$ benötigen wir, um mit Hilfe dieser Informationen und dem Wert von a_k das Teilproblem auf $[1, k]$ zu lösen und die Informationen bereit zu stellen, um Probleme auf noch größeren Bereichen lösen zu können? Die Menge möglicher Lösungen für den Bereich $[1, k]$ zerlegen wir nun disjunkt in zwei Mengen:

- $1 \leq i \leq j \leq k - 1$,
- $1 \leq i \leq j = k$.

Die erste Menge beschreibt ein Problem vom selben Typ und wir merken uns in der Variablen A_{k-1} den maximalen Wert für den Bereich $[1, k - 1]$. Der zweite Bereich ist wiederum einfacher zu behandeln, da alle erlaubten Lösungen mit dem Wert a_k enden. Den besten Wert für die zweite Menge merken wir uns in den Variablen B_k . Die Initialisierung der Werte ist einfach: $A_1 = a_1$ und $B_1 = a_1$. Zur Berechnung von $B_k, k \geq 2$, betrachten wir die Fälle $i < k$ und $i = k$. Für $i < k$ betrachten wir Bereiche $[i, k - 1]$ und zusätzlich a_k . Falls $i = k$ ist, ist der Wert a_k . Also ist

$$B_k = \max \{B_{k-1} + a_k, a_k\}$$

und kann mit zwei Operationen berechnet werden. Aufgrund unserer Aufteilung der Menge möglicher Lösungen folgt

$$A_k = \max \{A_{k-1}, B_k\}$$

und A_k kann mit einem Vergleich berechnet werden. Da diese Operationen für $k \in \{2, \dots, n\}$ durchgeführt werden, folgt

$$T_4(n) = 3n - 3.$$

Natürlich müssen wir uns die A - und B -Werte nur so lange merken, bis das nächste Wertepaar berechnet ist.

In der folgenden Tabelle haben wir die Ergebnisse noch einmal zusammengestellt.

n	naiv $\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n - 1$	normal $n^2 - 1$	Divide-and-Conquer $(2 \log n - 1)n + 1$	Dyn. Prog. $3n - 3$
$2^2 = 4$	19	15	13	9
$2^4 = 16$	815	255	113	45
$2^6 = 64$	45759	4095	705	189
$2^8 = 256$	2829055	65535	3841	765
$2^{10} = 1024$	179481599	1048575	19457	3069
$2^{15} = 32768$	$> 5 \cdot 10^{12}$	$\approx 10^9$	950273	98301

Tabelle 1.3.1

Die Rechenzeiten „richten sich“ nach den Größenordnungen (n^3 , n^2 , $n \log n$, n) und nicht nach den konstanten Vorfaktoren ($\frac{1}{6}$, 1, 2, 3). Es sollte sich das beruhigende Gefühl breitmachen, dass es sich lohnt, effiziente Algorithmen zu entwerfen. Die Zeit, um effizient eine Folge der Länge 2^{15} zu bearbeiten, ist kürzer als die Zeit, um eine Folge der Länge 2^{12} , 2^9 , bzw. 2^7 mit den Algorithmen Divide-and-Conquer, normal bzw. naiv zu bearbeiten.

1.4 Das MAXMIN-Problem

Das MAXMIN-Problem besteht in der folgenden Aufgabe. Für eine Folge a_1, \dots, a_n verschiedener Zahlen sollen die größte und kleinste bestimmt werden. Dieses Problem entspricht dem Problem, einen Turnierplan mit möglichst wenigen Spielen aufzustellen, bei dem unter der Annahme, dass immer der bessere Spieler gewinnt, der Champion und der Absteiger bestimmt werden.

Algorithmus 1.4.1:

1. Bestimme den Champion.
2. Bestimme unter allen Spielern, die in Schritt (1) noch nicht gewonnen haben, den Absteiger.

Wir wissen bereits, wie wir den Champion mit $n - 1$ Spielen bestimmen. Es ist klar, dass ein Spieler, der bereits ein Spiel gewonnen hat, nicht der Absteiger ist. Für die benötigte Zahl von Spielen ist es also entscheidend, wieviele Spieler in (1) nicht verloren haben. Gemäß Kapitel 1.3 lassen wir nach jedem Spiel den Gewinner auch im nächsten Spiel antreten, während der Verlierer in die Abstiegsrunde kommt. Wenn im ersten Spiel bereits der Champion antritt, bestreitet er alle $n - 1$ Spiele zur Ermittlung des Champions. Zur Ermittlung des Absteigers treten noch $n - 1$ Spieler an, und wir benötigen weitere $n - 2$ Spiele. In diesem schlechten Fall brauchen wir also $2n - 3$ Spiele. Wenn wir jedoch Glück haben, kommen wir mit $n - 1$ Spielen aus, dann nämlich, wenn die Spieler zufällig bezüglich ihrer Spielstärke vom Absteiger zum Champion sortiert sind.

Hierbei sehen wir schon, dass die Rechenzeit im schlechtesten Fall von der im besten Fall erheblich abweichen kann. In diesem Fall könnte uns die durchschnittliche Rechenzeit interessieren. Hier wollen wir jedoch die Zahl der Spiele im schlechtesten Fall minimieren.

Algorithmus 1.4.2:

1. Führe $\lfloor n/2 \rfloor$ Spiele mit verschiedenen Spielern durch.
2. Bestimme den besten unter den $\lfloor n/2 \rfloor$ Gewinnern der 1. Runde.
3. Bestimme den schlechtesten unter den $\lfloor n/2 \rfloor$ Verlierern der 1. Runde.
4. Falls n ungerade, lasse den Spieler, der an der 1. Runde nicht beteiligt war, gegen den Sieger aus der Championrunde (2) und den Verlierer der Abstiegsrunde (3) antreten.

Die Zahl der Spiele in Algorithmus 1.4.2 lässt sich leicht bestimmen.

1. $\lfloor n/2 \rfloor$
2. $\lfloor n/2 \rfloor - 1$
3. $\lfloor n/2 \rfloor - 1$
4. 0, falls n gerade, und 2, falls n ungerade.

Die Gesamtzahl der Spiele beträgt also $3n/2 - 2$, falls n gerade ist, und $3\lfloor n/2 \rfloor$, falls n ungerade ist. Damit beträgt die Zahl der Spiele in jedem Fall $n + \lfloor n/2 \rfloor - 2$. Wenn uns nun, auch bei längerem Nachdenken, kein besserer Turnierplan einfällt, möchten wir gerne mit dem Nachdenken aufhören, aber auch nicht von der Angst geplagt werden, dass anderen ein besserer Turnierplan einfällt. Unsere Ruhe finden wir erst wieder, wenn wir bewiesen haben, dass es keinen besseren Turnierplan geben kann.

Satz 1.4.3: Das MAXMIN-Problem kann mit $n + \lfloor n/2 \rfloor - 2$ Vergleichen, aber nicht mit weniger Vergleichen gelöst werden.

Beweis: Algorithmus 1.4.2 kommt mit $n + \lfloor n/2 \rfloor - 2$ Vergleichen aus. Für den Beweis der unteren Schranke, also den Beweis, dass weniger Vergleiche nicht ausreichen, arbeiten wir wieder mit der Sprache der Tennisturniere. Um den Champion und den Absteiger zu kennen, müssen wir von $n - 1$ Spielern wissen, dass sie nicht Champion sind, und von $n - 1$ Spielern wissen, dass sie nicht Absteiger sind. Nichtchampions müssen ein Spiel verloren haben, und Nichtabsteiger müssen ein Spiel gewonnen haben. Insgesamt benötigen wir $2n - 2$ Informationseinheiten.

Ein Spiel kann maximal 2 Informationseinheiten liefern. Wir teilen die Spieler in vier Kategorien:

1. ?-Spieler: Sie haben noch nicht gespielt.

2. $+-$ -Spieler: Sie haben schon gewonnen, aber noch nicht verloren.
3. $--$ -Spieler: Sie haben schon verloren, aber noch nicht gewonnen.
4. \pm -Spieler: Sie haben schon gewonnen und schon verloren.

Für \pm -Spieler haben wir bereits ausreichende Informationen, sie sollten nicht mehr spielen. Für die anderen Kombinationen überlegen wir uns, wieviele Informationseinheiten sie liefern.

1. $??$ -Spiele liefern auf jeden Fall 2 Informationseinheiten.
2. $+/+$ -Spiele und $-/-$ -Spiele liefern auf jeden Fall eine Informationseinheit.
3. $+/?$ - und $-/?$ -Spiele liefern mindestens eine Informationseinheit (+ gewinnt bzw. - verliert) und in glücklichen Fällen zwei Informationseinheiten.
4. $+/-$ -Spiele müssen keine Informationen liefern (+ gewinnt), können aber zwei Informationseinheiten liefern.

Da wir vorher nichts über die Spielstärke wissen, ist die Situation für uns so, als würde der Teufel den Spielausgang festlegen. Dieser muss uns nur in $??$ -Spielen zwei Informationseinheiten zugestehen. Jeder Spieler ist jedoch nur in einem Spiel ein $?$ -Spieler, danach ist er $+$ - oder $-$ -Spieler. Die maximale Zahl von $??$ -Spielen beträgt also $\lfloor n/2 \rfloor$. Um die restlichen $2n - 2 - 2\lfloor n/2 \rfloor = 2\lceil n/2 \rceil - 2$ Informationseinheiten zu erhalten, benötigen wir im worst case also mindestens $2\lceil n/2 \rceil - 2$ Spiele. Damit benötigen wir im worst case also mindestens $\lfloor n/2 \rfloor + 2\lceil n/2 \rceil - 2 = n + \lceil n/2 \rceil - 2$ Spiele. \square

Für das MAXMIN-Problem haben wir eine vollständige Lösung. Wir haben für einen Algorithmus bewiesen, dass er optimal ist. Dies ist leider die Ausnahme. Wir kennen zwar viele Algorithmen, von denen wir glauben, dass sie zumindest fast optimal sind. Der Beweis unterer Schranken ist allerdings in den meisten Fällen zu schwierig. Wir müssen dabei für alle denkbaren Algorithmen beweisen, dass sie bestimmte Ressourcen benötigen.

Wir sollten noch festhalten, dass wir mit den Begriffen Turnier, Champion und Absteiger, sowie der Kategorisierung der Spieler eine Fachsprache geschaffen haben, die es uns besonders leicht macht, über unser Problem nachzudenken. Die Einbettung eines Problems in die richtige Umgebung ist häufig der erste Schritt zur Problemlösung. Auf diese Weise wird assoziatives und damit kreatives Denken unterstützt.

1.5 Registermaschinen

In den beiden bisher behandelten Problemen haben wir auf zwar sinnvolle, aber dennoch willkürliche Weise festgelegt, welche Operationen wir zählen wollen. Die tatsächliche Entscheidung, wie Rechenschritte zu bewerten sind, hängt von der jeweiligen Situation, dem Problem, dem Algorithmus, dem Rechner und der verwendeten Programmiersprache ab.

Systematische Aussagen zur Komplexität und Effizienz von Algorithmen setzen dagegen ein Rechnermodell voraus, das einerseits nahe genug an realen Rechnern ist, aber andererseits maschinen- und programmiersprachenunabhängige Untersuchungen von Algorithmen ermöglicht. Das Modell der Registermaschinen (random access machine = RAM) ist der bisher gelungenste und allgemein anerkannte Kompromiss zwischen beiden Anforderungen.

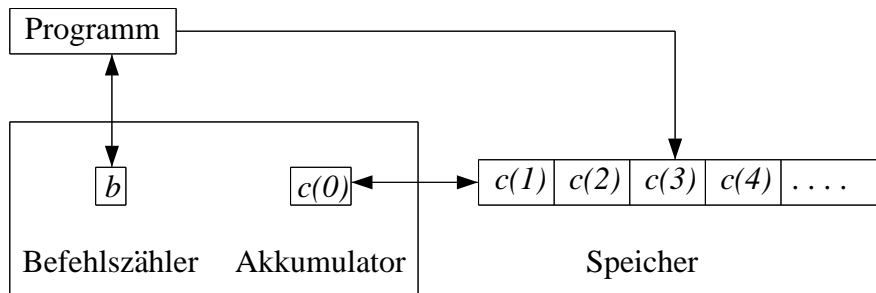


Abbildung 1.5.1: Aufbau einer RAM.

Die Inhalte des Befehlszählers, des Akkumulators und der Register sind natürliche Zahlen, die beliebig groß sein können. Die Anzahl der Register ist unbeschränkt, damit Programme für beliebige Eingabelängen laufen können. Die Register haben auf natürliche Weise Adressen. Ein Programm ist eine endliche Folge von Befehlen aus einer noch zu spezifizierenden Befehlsliste. Die Programmzeilen sind, mit 1 beginnend, durchnummeriert. Der Befehlszähler b startet mit dem Wert 1 und steht später auf der Nummer des auszuführenden Befehls. In den ersten Registern des Speichers steht zu Beginn die Eingabe, in den weiteren Registern ebenso wie im Akkumulator 0. Am Ende der Rechnung stehen die Ausgabedaten in vorher festgelegten Registern. Der Inhalt von Register i wird mit $c(i)$ (c = contents) bezeichnet.

Wir listen nun die zulässigen Befehle auf.

LOAD i :	$c(0) := c(i), b := b + 1.$
STORE i :	$c(i) := c(0), b := b + 1.$
ADD i :	$c(0) := c(0) + c(i), b := b + 1.$
SUB i :	$c(0) := \max\{c(0) - c(i), 0\}, b := b + 1.$
MULT i :	$c(0) := c(0) * c(i), b := b + 1.$
DIV i :	$c(0) := \lfloor c(0)/c(i) \rfloor, b := b + 1.$
GO TO j	$b := j.$
IF $c(0)? l$ GO TO j	$b := j$ falls $c(0)? l$ wahr ist, und $b := b + 1$ sonst. (Dabei ist $? \in \{=, <, \leq, >, \geq\}.$)
END	$b := b.$

Die Befehle CLOAD, CADD, CSUB, CMULT und CDIV entsprechen den Befehlen ohne den Präfix C (für Konstante), wobei auf der rechten Seite $c(i)$ durch die Konstante i

ersetzt wird. Die Befehle INDLOAD, INDSTORE, INDADD, INDSUB, INDMULT und INDDIV entsprechen den Befehlen ohne den Präfix IND (für indirekte Adressierung), wobei in den jeweiligen Zeilen $c(i)$ durch $c(c(i))$ ersetzt wird.

Es sollte klar sein, dass es zwar mühevoll, aber nicht schwierig ist, alle Algorithmen und Programme auf Registermaschinen zu übertragen. Ganze Zahlen lassen sich in zwei, rationale Zahlen in drei Registern darstellen. Arrays und lineare Listen können ebenso benutzt werden wie allgemeine if-Abfragen und Schleifen. Durch die ausschließliche Benutzung von GO-TO-Sprungbefehlen werden die Programme schwer verständlich. Wir werden Programme und Algorithmen daher nicht auf Registermaschinen übertragen (das wäre eine schlimme Strafe), sondern behalten dieses Modell als allgemeines Modell im Hinterkopf. Die Zahl der Rechenschritte einer RAM unterscheidet sich jeweils nur um einen konstanten Faktor von der Zahl der Rechenschritte konkreter Rechner. Daher werden wir häufig auch vor allem die Größenordnung der Anzahl der Rechenschritte betrachten, da der konstante Faktor vom Rechnermodell abhängt.

Üblicherweise wird das uniforme Kostenmaß verwendet, wobei jede Programmzeile mit Ausnahme des Befehls END eine Kosteneinheit verursacht. Dieses Maß ist gerechtfertigt, solange nicht mit sehr großen Zahlen gerechnet wird. Rechenschritte mit sehr großen Zahlen können viele normale Rechenschritte in sich „verstecken“. Dann ist das logarithmische Kostenmaß realistischer. Die Kosten eines Rechenschritts entsprechen der Zahlenlänge der dabei benutzten Zahlen.

Die Quintessenz dieser Betrachtungen ist die Folgende. Es gibt ein anerkanntes, aber unhandliches Rechnermodell für eine exakte Betrachtung der Rechenzeit und des Speicherplatzes. Rechenzeiten und Speicherplatzbedarf auf realen Rechnern unterscheiden sich untereinander und von dem Referenzmodell der Registermaschine. Wenn wir Rechenzeiten, die sich nur um kleine konstante Faktoren unterscheiden, als „im wesentlichen gleich“ akzeptieren, können wir von der Rechenzeit eines Algorithmus unabhängig vom betrachteten Rechnermodell reden. Unsere späteren Überlegungen sind also für alle gängigen Rechner gültig. Bei einer größeren Abhängigkeit vom benutzten Rechner wäre eine allgemeine Theorie von Datenstrukturen und Algorithmen nicht möglich. Wie wir mit „im wesentlichen gleichen“ Rechenzeiten formal umgehen, diskutieren wir in Kapitel 1.7.

1.6 Komplexitätsmaße

Definition 1.6.1: Die Laufzeit $T_P(x)$ eines (RAM-) Programmes P auf einer Eingabe x ist die Anzahl der bei Eingabe x auszuführenden Befehle (uniforme Zeitkomplexität). Bei der logarithmischen Zeitkomplexität $T_P^{\log}(x)$ werden jedem ausgeführten Befehl so viele Kosteneinheiten zugeordnet, wie in diesem Befehl Bits verarbeitet werden.

Definition 1.6.2: Der Platzbedarf $S_P(x)$ eines (RAM-) Programmes P auf einer Eingabe x ist die Anzahl der benutzten Register (uniformer Platzbedarf). Beim logarithmischen Platzbedarf $S_P^{\log}(x)$ werden für jedes benutzte Register die Bits der größten in ihm gespeicherten Zahl gezählt.

Eine gemeinsame Minimierung von Rechenzeit und Speicherplatz ist wünschenswert, aber nicht immer erreichbar. Es gibt Probleme, bei denen das kleinste erreichbare Produkt aus Rechenzeit und Speicherplatz beweisbar wesentlich größer als das Produkt aus der besten erreichbaren Rechenzeit und dem besten erreichbaren Platzbedarf ist. Es gibt dann einen sogenannten Trade-off zwischen Rechenzeit und Platzbedarf. Für die in dieser Vorlesung behandelten Datenstrukturen und Algorithmen wird fast immer nur wenig Speicherplatz benötigt. Wir konzentrieren uns daher auf die Laufzeit von Algorithmen. Da wir nie mit Zahlen arbeiten werden, die wesentlich größer als die Zahlen in der Eingabe und Ausgabe sind, beschränken wir uns auf die uniformen Komplexitätsmaße.

Schon für die meisten einfachen Algorithmen ist die geschlossene Berechnung von $T_P(x)$ für alle Eingaben x sehr kompliziert oder zu schwierig. Wir können T_P zwar mit einer leichten Modifikation des Programms P berechnen, jedoch ist dies wenig hilfreich. Wir möchten ja die Komplexität von P kennen, ohne P erst zu benutzen. Daher suchen wir nach aussagekräftigen und gleichzeitig prägnanten Kennzahlen für die Laufzeit eines Algorithmus.

Definition 1.6.3: Für eine Eingabe x sei $|x|$ die Länge der Eingabe (z. B. die Anzahl der in x enthaltenen Zahlen oder die Bitlänge von x). Die worst case Rechenzeit des Programms P auf Eingaben der Länge n ist definiert durch

$$T_P(n) := \sup\{T_P(x) \mid |x| = n, x \text{ Eingabe für } P\}.$$

Es sei q_n eine Wahrscheinlichkeitsverteilung auf der Menge der Eingaben der Länge n . Die average case (durchschnittliche) Rechenzeit des Programms P auf Eingaben der Länge n ist definiert durch

$$T_{P,q_n}(n) := \sum_{\substack{|x|=n, \\ x \text{ Eingabe für } P}} T_P(x) q_n(x).$$

Obwohl die Definition von $T_P(n)$ von allen Eingaben x der Länge n abhängt, kann es sehr wohl möglich sein, die worst case und average case Rechenzeit auszurechnen, ohne alle $T_P(x)$ auszurechnen.

Die worst case Rechenzeit gibt uns die Sicherheit, in keinem Fall eine größere Rechenzeit zu benötigen. Häufig ist sie wesentlich einfacher zu berechnen als die average case Rechenzeit und trifft dennoch die wesentliche Aussage. Für viele Algorithmen ist nämlich die Rechenzeit für alle Eingaben einer bestimmten Länge ungefähr gleich groß. Dies gilt für unsere Algorithmen für das Maxsummenproblem ebenso wie für den optimalen (für den worst case) Algorithmus für das MAXMIN-Problem. Dies ist jedoch für Algorithmus 1.4.1 nicht mehr der Fall. Wir werden in dieser Vorlesung noch wesentlich krassere Beispiele kennenlernen (Stichwort Quicksort). In vielen Anwendungen ist es sicher günstiger, im Durchschnitt eine geringe Rechenzeit zu haben, als sich nur für den ungünstigen Fall zu wappnen.

Die average case Rechenzeit ist aber leider kein einfach zu handhabendes Maß. Welche Wahrscheinlichkeitsverteilung ist der jeweiligen Situation angemessen? Aus Unkenntnis über die „wahre“ Verteilung wird häufig die Gleichverteilung gewählt. Wenn die wahre

Verteilung anders aussieht, ist die berechnete average case Rechenzeit ohne Aussagekraft. Hinzu kommt, dass es für die meisten nicht ganz einfachen Algorithmen bisher nicht gelungen ist, eine zufriedenstellende Analyse der average case Rechenzeit durchzuführen. Wir müssen uns daher im Allgemeinen mit der Berechnung oder sogar einer Abschätzung der worst case Rechenzeit zufrieden geben. Pessimisten können nicht enttäuscht werden.

Beispiel 1.6.4: Der vermutlich am häufigsten in Programmen benutzte Befehl ist die Addition von 1 zu einer Zahl i . Betrachten wir dafür die Taktzahl eines von Neumann Addierwerkes als Komplexitätsmaß. Diese ist offensichtlich um 1 größer als die Zahl der Einsen am Ende der Binärdarstellung von i .

Wir betrachten nun Zahlen i der Länge n , d.h. $0 \leq i \leq 2^n - 1$. Die worst case Rechenzeit beträgt offensichtlich $n + 1$ für die Eingabe $i = 2^n - 1$. Wir kommen zur Berechnung der average case Rechenzeit bei Gleichverteilung auf der Eingabemenge, d.h. jede Zahl i ist mit Wahrscheinlichkeit 2^{-n} die Eingabezahl. Sei $i = (i_{n-1}, \dots, i_0)$ die Binärdarstellung von i und sei k der kleinste Index mit $i_k = 0$. Es gibt 2^{n-k} Eingaben, die mit $(0, 1, \dots, 1)$ enden, wobei am Ende $k-1$ Einsen stehen. Für diese Eingaben benötigt das von Neumann Addierwerk k Takte. Hinzu kommt die Eingabe $i = 2^n - 1$, die $n + 1$ Takte benötigt. Die average case Rechenzeit beträgt also

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

Dabei gilt

$$\begin{aligned} \sum_{1 \leq k \leq n} 2^{n-k} k &= 2^{n-1} + 2 \cdot 2^{n-2} + 3 \cdot 2^{n-3} + \dots + n 2^{n-n} \\ &= 2^{n-1} + 2^{n-2} + 2^{n-3} + 2^{n-4} + \dots + 2^0 \\ &\quad + 2^{n-2} + 2^{n-3} + 2^{n-4} + \dots + 2^0 \\ &\quad + 2^{n-3} + 2^{n-4} + \dots + 2^0 \\ &\quad \dots \\ &\quad + 2^0 \\ &= (2^n - 1) + (2^{n-1} - 1) + (2^{n-2} - 1) + (2^{n-3} - 1) + \dots + (2^1 - 1) \\ &= 2^{n+1} - 2 - n \end{aligned}$$

Damit beträgt die average case Taktzahl

$$2^{-n} (2^{n+1} - 2 - n + (n + 1)) = 2 - 2^{-n}.$$

Es genügen durchschnittlich knapp 2 Takte, im worst case werden jedoch $n + 1$ Takte durchgeführt.

Im obigen Beispiel war es möglich, die average case Rechenzeit direkt, also ohne Tricks, auszurechnen. Dazu mussten wir aber die Summe aller $2^{n-k}k$, $1 \leq k \leq n$, berechnen. Häufig können die Rechnungen mit geeigneten Tricks vereinfacht werden. Auch das wollen wir hier am Beispiel sehen. Wir stellen nämlich folgendes fest. Wenn die Addition von 1 genau m Takte dauert, dann wird die Anzahl der Einsen in der Binärzahl um $2 - m$ größer (für $m > 2$ wird die Anzahl der Einsen also kleiner). Für die Zahl i , $0 \leq i \leq 2^n - 1$, sei m_i die zugehörige Anzahl der Takte. Wir sind an der Summe aller $m_i 2^{-n}$ interessiert. Es ist aber viel einfacher, die Summe aller $2 - m_i$ zu bilden. Wir starten nämlich mit 0 Einsen in der Zahl 0, addieren 2^n -mal 1 und enden mit einer Eins in der Zahl 2^n . Also ist

$$S = \sum_{0 \leq i \leq 2^n - 1} (2 - m_i) = 1,$$

$$\sum_{0 \leq i \leq 2^n - 1} m_i = 2 \cdot 2^n - 1$$

und

$$2^{-n} \sum_{0 \leq i \leq 2^n - 1} m_i = 2 - 2^{-n}.$$

Mit dem „richtigen“ Zugang zur Analyse haben wir Rechenarbeit eingespart. Bei komplizierten Algorithmen ist eine Analyse oftmals nur mit derartigen Tricks möglich. Aber auch diese Tricks fallen nicht vom Himmel, sondern folgen allgemeinen Methoden. Hier ist es die Methode der Potenzialfunktionen, die im Hauptstudium in der Vorlesung „Effiziente Algorithmen“ systematisch behandelt wird.

Wenn wir bei der Entwicklung eines Algorithmus innerhalb des Algorithmus eine Entscheidung treffen müssen, z.B. in welcher Reihenfolge wir Objekte behandeln wollen, wenn wir glauben, dass diese Entscheidung die Rechenzeit wesentlich beeinflussen kann, und wenn wir keine Idee haben, wie eine gute Entscheidung aussehen kann, dann ist es oftmals ratsam, die Entscheidung dem Zufall zu überlassen. Randomisierte Algorithmen können auf zufällige Bits zugreifen und den Algorithmusablauf von diesen Zufallsbits abhängig machen. Dann hängt die Rechenzeit selbst für eine feste Eingabe x vom Zufall ab, ist also eine Zufallsvariable T_x . Wenn die Wahrscheinlichkeit, dass $T_x = t$ ist, mit $\text{Prob}(T_x = t)$ bezeichnet wird, ist

$$E(T_x) = \sum_{0 \leq t < \infty} t \cdot \text{Prob}(T_x = t)$$

die erwartete Rechenzeit des Algorithmus für Eingabe x . Wir sind dann an randomisierten Algorithmen interessiert, deren worst case (bezogen auf Eingaben gleicher Länge) erwartete (bezogen auf die verwendeten Zufallsbits) Rechenzeit klein ist. Der Einsatz randomisierter Algorithmen ist inzwischen eine Selbstverständlichkeit geworden. Zum Entwurf und zur Analyse randomisierter Algorithmen sind grundlegende Kenntnisse der Wahrscheinlichkeitstheorie notwendig.

1.7 Größenordnungen

Es seien $t_1(n)$ und $t_2(n)$ die worst case Rechenzeiten von zwei Algorithmen A_1 und A_2 , die dasselbe Problem lösen. Wir wissen bereits, dass t_1 und t_2 „etwas, aber nicht sehr“

vom Rechnermodell abhängen. Wann können wir nun sagen, dass A_1 „im wesentlichen besser“ als A_2 ist? Zunächst einmal sind Rechenzeiten für „sehr kurze“ Eingaben nicht wesentlich. Darüber hinaus wollen wir „konstante Funktionen“ nicht so wichtig nehmen und t_1 und t_2 als „im wesentlichen gleich“ bezeichnen, wenn $t_1(n)/t_2(n)$ und $t_2(n)/t_1(n)$ durch eine Konstante nach oben beschränkt sind. In diesem Fall sprechen wir davon, dass t_1 und t_2 dieselbe Größenordnung oder Wachstumsordnung haben. Hierbei vergrößern wir natürlich die Sichtweise. Für konkrete Algorithmen und Eingabelängen kann ein Algorithmus, dessen Laufzeit eine größere Wachstumsordnung hat und der damit asymptotisch langsamer ist, durchaus besser sein. Für die Untersuchung der Größenordnung hat sich die so genannte O -Notation eingebürgert.

Definition 1.7.1: Es seien $f, g : \mathbb{N} \rightarrow \mathbb{R}_0^+$.

1. $f = O(g)$ (in Worten: f wächst nicht schneller als g), wenn gilt: $\exists c, n_0 \forall n \geq n_0 : f(n) \leq cg(n)$.
2. $f = \Omega(g)$ (f wächst mindestens so schnell wie g), wenn $g = O(f)$ ist.
3. $f = \Theta(g)$ (f und g sind von gleicher Größenordnung), wenn $f = O(g)$ und $g = O(f)$ gelten.
4. $f = o(g)$ (f wächst langsamer als g), wenn $f(n)/g(n)$ eine Nullfolge ist.
5. $f = \omega(g)$ (f wächst schneller als g), wenn $g = o(f)$ ist.

Gleichungsfolgen in der O -Notation müssen stets von links nach rechts gelesen werden. So sind

$$10n^2 + n = O(n^2) = o(n^3)$$

und

$$10n^2 + n = \Omega(n^2) = \omega(n)$$

sinnvoll, während

$$10n^2 + n = o(n^3) = O(n^2)$$

ebenso sinnlos ist wie

$$10n^2 + n = O(n^2) = \Omega(n^2).$$

Lemma 1.7.2:

1. $\forall k > 0 : n^k = o(2^n)$.
2. Es seien p_1 und p_2 Polynome vom Grad d_1 bzw. d_2 , wobei die Koeffizienten von n^{d_1} bzw. n^{d_2} positiv sind. Dann gilt
 - (a) $p_1 = \Theta(p_2) \Leftrightarrow d_1 = d_2$.
 - (b) $p_1 = o(p_2) \Leftrightarrow d_1 < d_2$.
 - (c) $p_1 = \omega(p_2) \Leftrightarrow d_1 > d_2$.

3. $\forall k > 0, \varepsilon > 0 : \log^k n = o(n^\varepsilon)$.
4. $2^{n/2} = o(2^n)$.

Beweis: Übungsaufgabe. Gleichzeitig ein Beweis, dass die Analysis-Vorlesung Anwendungen hat. \square

Wir merken uns, dass logarithmische Funktionen (wie $\log^k n$) langsamer wachsen als polynomielle Funktionen, zu denen wir hier selbst n^ε mit $\varepsilon > 0$ zählen. Bei verschiedenen Polynomen entscheidet der Grad über die Größenordnungen. Alle Polynome wachsen langsamer als die exponentiellen Funktionen.

Die folgenden Tabellen sollen für typische Rechenzeiten untermauern, dass die Größenordnung der Rechenzeit entscheidende Bedeutung hat. Die Leserin und der Leser sind aufgefordert, ihre Einsichten dadurch zu vertiefen, dass sie die Tabellen mit Rechenzeiten fortsetzen, die auch von 1 verschiedene Vorfaktoren und Summanden kleinerer Größenordnung haben. Wir nehmen an, dass ein Rechenschritt 0,001 Sekunden benötigt.

$T_p(n)$	Maximale Eingabelänge bei Rechenzeit		
	1Sek.	1Min.	1Std.
n	1000	60000	3600000
$n \log n$	140	4895	204094
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

Tabelle 1.7.1

Trotz unserer Betonung der herausragenden Rolle der Größenordnung von Rechenzeiten soll stets versucht werden, die Rechenzeiten so genau wie möglich zu analysieren.

Mit der folgenden Tabelle wollen wir ausdrücken, wie sich die Größenordnung der Rechenzeit bei technologischen Fortschritten auswirkt. Um wieviel können wir bei vorgegebener Rechenzeit die Eingabelänge vergrößern, wenn die Rechner zehnmal so schnell werden? In der folgenden Tabelle spielen übrigens konstante Faktoren für die Rechenzeit keine Rolle. Die folgenden Beziehungen wurden bei der Aufstellung der folgenden Tabelle benutzt: $\log(10p) = \log p + \log 10 \approx \log p$, $3,16^2 \approx 10$, $2,15^3 \approx 10$ und $2^{33} \approx 10$.

$T_p(n)$	Maximale Eingabelänge in Abhängigkeit der Technologie	
	alt	neu(d.h. 10-mal schneller)
n	p	$10p$
$n \log n$	p	$(\text{fast } 10)p$
n^2	p	$3,16p$
n^3	p	$2,15p$
2^n	p	$p + 3,3$

Tabelle 1.7.2

Tabelle 1.7.1 ist aus einem älteren Lehrbuch entnommen, als die Zeit 0,001 Sekunden für

einzelne Rechenschritte realistisch war. Mit Tabelle 1.7.2 können wir Tabelle 1.7.1 leicht aktualisieren. Wenn Rechner um den Faktor 10^m (der Wert von m hängt tatsächlich vom Rechner ab, ist aber eine Konstante) schneller geworden sind, ergeben sich die folgenden Werte: $10^m p$, fast $10^m p$, $(3,16)^m p$, $(2,15)^m p$ und $p + 3,3 \cdot m$.

Tabelle 1.7.2 macht einen strukturellen Unterschied zwischen den Rechenzeiten n , $n \log n$, n^2 , n^3 einerseits und 2^n andererseits deutlich. In den ersten vier Fällen wächst die zu bearbeitende Eingabelänge um einen konstanten Faktor, der vom Grad des Rechenzeitpolynoms abhängt. Im letzten Fall wächst die zu bearbeitende Eingabelänge nur um einen konstanten Summanden. Um wieviel muss die Rechengeschwindigkeit steigen, damit Probleme der Eingabelänge $p + 100$ bearbeitet werden können?

Definition 1.7.3: Sei $f : \mathbb{N} \rightarrow \mathbb{R}_0^+$.

1. f heißt polynomiell beschränkt (wächst polynomiell), wenn es ein Polynom p mit $f = O(p)$ gibt.
2. f wächst exponentiell, wenn es ein $\varepsilon > 0$ mit $f = \Omega(2^{n^\varepsilon})$ gibt.

Aufgrund der obigen Diskussion nennen wir Algorithmen höchstens dann effizient, wenn ihre Rechenzeit polynomiell wächst. In bestimmten Situationen schränken wir den Begriff „effizient“ weiter ein. So können $\Theta(n^3)$ -Algorithmen für das Maxsummenproblem nicht mehr effizient genannt werden. Sofern irgend möglich, sollten exponentielle Algorithmen vermieden werden.

2 Grundlegende Datenstrukturen

2.1 Motivation und Vorgehensweise

Datenstrukturen dienen zur Abspeicherung von Daten und zusätzlichen Informationen, um bestimmte Operationen zu unterstützen, also effizient zu ermöglichen. Sie werden hier isoliert vorgestellt, wobei sie in der Praxis wichtige Bestandteile von Algorithmen sind. Es ist daher stets nützlich, sich Anwendungssituationen für die vorgestellten Datenstrukturen zu überlegen. Genauso wichtig ist es, sich Situationen zu überlegen, in denen die behandelte Datenstruktur nicht geeignet ist. Natürlich lassen sich verschiedene Datenstrukturen auch kombinieren.

Wie bereits diskutiert, soll unsere Darstellung nicht auf eine bestimmte Programmiersprache zugeschnitten sein.

In der Praxis stellt ein Algorithmus Anforderungen auf, welche Operationen zu unterstützen sind. Die Analyse des Algorithmus führt oft auch zu Aussagen, welche Operationen wie oft auf was für Daten auszuführen sind. Dann wird im Reservoir vorhandener Datenstrukturen (z.B. LEDA) nach einer passenden Datenstruktur Ausschau gehalten. Gegebenenfalls wird eine Datenstruktur der Situation angepasst oder eine neue Datenstruktur entworfen, analysiert und in das bestehende Reservoir eingefügt. In der Lehre gehen wir anders vor. Wir lernen Teile des bestehenden Reservoirs mit ihren Eigenschaften kennen und diskutieren Beispielanwendungen.

2.2 Arrays

Unter einem Array (oder Feld) verstehen wir einen Speicherbereich mit einer festgelegten Zahl von n Speicherplätzen, in denen Daten eines bestimmten Typs abgelegt werden können. Wir stellen uns vor, dass im Rechner (z.B. einer Registermaschine) ein fortlaufender Speicherbereich für ein Array a reserviert ist, so dass auf das Datum $a(i)$ an Position i direkt (in Zeit $O(1)$) zugegriffen werden kann. Wir können auf Daten aber nur über die Positionen und nicht über Namen oder Eigenschaften direkt zugreifen. Damit folgt, dass lokale Operationen wie z.B.

- ersetze das Datum an Position i durch x
- vertausche die Daten an den Positionen i und j

in konstanter Zeit durchführbar sind. Dagegen ist die Operation

- entferne das Datum an Position j und schreibe es zwischen die Daten an den Positionen i und $i + 1$

für Arrays nicht lokal. Falls $i < j$ ist, müssen die Daten an den Positionen $i + 1, \dots, j$ verändert werden. In einem Array der Länge n können zwar Plätze unbelegt sein (ein Leerzeichen enthalten), aber eine Unterbringung von mehr als n Daten ist unmöglich.

Arrays werden vornehmlich für Datenmengen einer festen Größe verwendet. Derartige Datenstrukturen heißen *statisch*.

In vielen Anwendungen haben wir es mit sortierten Arrays zu tun. Die Daten stammen dann aus einer vollständig geordneten Menge und es gilt $a(1) < \dots < a(n)$. Dies erleichtert die Suche nach einem Datum erheblich. In einem unsortierten Array können wir die Suche nach x nur mit der Aussage „ x ist nicht im Array vorhanden“ beenden, wenn wir alle Arraypositionen abgesucht haben. Für die Suche in sortierten Arrays wollen wir die drei wichtigsten Suchstrategien beschreiben.

Lineare Suche: Es werden die Arrayplätze in der Reihenfolge $1, \dots, n$ untersucht. Die Suche kann erfolgreich abgebrochen werden, wenn x gefunden wurde, und erfolglos abgebrochen werden, wenn ein Datum $y > x$ gefunden wurde. Die worst case Suchzeit beträgt $\Theta(n)$ und wir „erwarten“ auch in den meisten Fällen eine lineare Suchzeit. Nur wenn es gute Gründe für die Annahme gibt, dass x in Bezug auf die Arrayobjekte klein ist, ist die lineare Suche effizient.

Binäre Suche: Wir vergleichen das Datum $y = a(\lceil n/2 \rceil)$ mit dem gesuchten Objekt. Falls $n = 1$, endet die Suche in jedem Fall. Ansonsten haben wir, falls $x = y$, x gefunden. Falls $x < y$ ist, fahren wir im Teilarray mit den Positionen $1, \dots, \lceil n/2 \rceil - 1$ rekursiv fort. Falls $x > y$ ist, fahren wir im Teilarray mit den Positionen $\lceil n/2 \rceil + 1, \dots, n$ rekursiv fort. Falls $n = 2^k - 1$ ist, haben die Teilarrays die Größe $2^{k-1} - 1$ und es folgt induktiv, dass k Vergleiche stets ausreichen. Falls $n > 2^k - 1$ ist, reichen nicht in jedem Fall k Vergleiche. Also beträgt die worst case Anzahl an Vergleichen $\lceil \log(n+1) \rceil$. Wenn wir vorab wissen, dass x im Array abgespeichert ist, sinkt diese Anzahl um 1. Andererseits kann eine Suche niemals nach weniger als $\lceil \log(n+1) \rceil - 1$ Vergleichen erfolglos abgebrochen werden. In den meisten Fällen ist die gute worst case Anzahl an Vergleichen dafür ausschlaggebend, binär zu suchen.

Geometrische Suche: Wenn wir vermuten, dass x bezogen auf die Arraydaten relativ klein ist, bildet die geometrische Suche einen guten Kompromiss zwischen linearer und binärer Suche. Es sei 2^k die größte Zweierpotenz, für die $2^k \leq n$ ist. Dann werden die Daten an den Positionen $2^0, 2^1, \dots, 2^k$ so lange mit x verglichen, bis x gefunden wurde oder ein Datum $y > x$ an einer Position 2^m gefunden wurde. Im zweiten Fall wird eine binäre Suche auf den Positionen $2^{m-1} + 1, \dots, 2^m - 1$ gestartet. Ist auch das Datum an Position 2^k kleiner als x , wird eine binäre Suche auf den Positionen $2^k + 1, \dots, n$ gestartet. Jede der beiden Phasen erfordert höchstens $\lceil \log n \rceil + 1$ Vergleiche. Die geometrische Suche braucht also niemals mehr als doppelt so viele Vergleiche wie die binäre Suche. Andererseits ist sie für Daten $x < a(2^m)$, m klein, sehr effizient. Dann ist die Anzahl der Vergleiche durch $2m + 1$ nach oben beschränkt. Die geometrische Suche schlägt die binäre Suche, falls $x < a(i)$ und $i \approx n^{1/2}$ ist.

In Kapitel 4.5 über Heapsort werden wir die hier diskutierten Ergebnisse über die drei Suchstrategien anwenden.

Viele Datenmengen sind nicht von vornherein „linear angeordnet“ wie eine Folge a_1, \dots, a_n . Mehrdimensionale Arrays oder mehrdimensionale Matrizen müssen erst in eine entsprechende Form gebracht werden. Eine (zweidimensionale) Matrix der Größe $n_1 \times n_2$

wird zeilenweise (oder spaltenweise) abgespeichert. Das Element an Matrixposition (i, j) erhält dann die Arrayposition $(i - 1)n_2 + j$, da $i - 1$ vollständige Zeilen und $j - 1$ Elemente der i -ten Zeile vor ihr abgespeichert werden müssen. Viele Programmiersprachen unterstützen selbst die Behandlung k -dimensionaler Arrays der Größe $n_1 \times \cdots \times n_k$ direkt. Aber irgend jemand hat die so genannte Speicherabbildungsfunktion beschreiben müssen. Wir betrachten die Arrayposition (i_1, \dots, i_k) . Vor dieser Position stehen

- $i_1 - 1$ vollständige $n_2 \times \cdots \times n_k$ -Arrays,
- weitere $i_2 - 1$ vollständige $n_3 \times \cdots \times n_k$ -Arrays, ...,
- weitere $i_k - 1$ Daten,

also steht das Element aus Arrayposition (i_1, \dots, i_n) in einem (eindimensionalen) Array an Position

$$\sum_{1 \leq j \leq k-1} (i_j - 1) \prod_{j < m \leq k} n_m + i_k.$$

In der Numerik spielen spezielle $n \times n$ -Matrizen, darunter Dreiecksmatrizen und Bandmatrizen, eine wichtige Rolle. Bei ihnen gibt es viele Positionen, von denen vorab bekannt ist, dass sie Nullen enthalten. Wir wollen diese Matrizen in einem Array abspeichern, dessen Länge die Anzahl der relevanten Positionen ist, also der Positionen, die eventuell von 0 verschieden sind. Um dann ein Matricelement zu bestimmen, benötigen wir die zugehörige Speicherabbildungsfunktion.

Definition 2.2.1: Eine $n \times n$ -Matrix M heißt untere Dreiecksmatrix, wenn $M(i, j) = 0$ für $i < j$ ist.

Eine untere Dreiecksmatrix der Größe $n \times n$ hat nur

$$1 + \cdots + n = n(n + 1)/2$$

relevante Positionen $(i, j), i \geq j$. Wir benutzen ein Array der Länge $n(n + 1)/2$ und speichern die relevanten Positionen zeilenweise ab. Das Element an Position (i, j) steht dann in dem eindimensionalen Array an Position

$$1 + \cdots + (i - 1) + j = i(i - 1)/2 + j.$$

Definition 2.2.2: Eine $n \times n$ -Matrix M heißt m -Bandmatrix, wenn $M(i, j) = 0$ für $|i - j| \geq m$ ist.

Abbildung 2.2.1 zeigt das typische Aussehen von Bandmatrizen.

Wir wollen die Speicherabbildungsfunktion im Fall $m < n/2$ beschreiben, da in diesem Fall die Speicherplatzersparnis größer ist. Es zeigt sich, dass selbst in so einfach aussehenden Situationen die Speicherabbildungsfunktion recht komplex wird. Wir betrachten die Zahl der relevanten Matricelemente in den einzelnen Zeilen. In den ersten m Zeilen wächst sie jeweils um 1, und zwar von m bis $2m - 1$. In den letzten m Zeilen fällt sie entsprechend

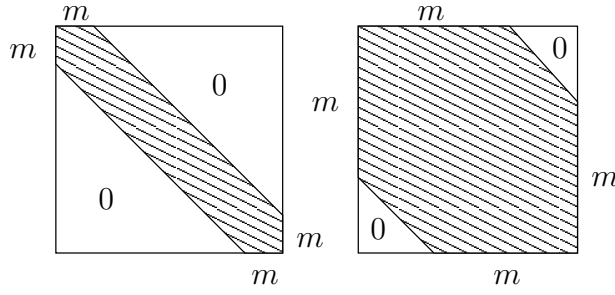


Abbildung 2.2.1: Bandmatrizen mit $m < n/2$ und $m > n/2$.

von $2m - 1, \dots, m$. Für alle anderen Zeilen gibt es $2m - 1$ essenzielle Positionen. Wir betrachten nun eine Position (i, j) mit $|i - j| < m$.

1.Fall: $i \leq m$.

Dann steht $M(i, j)$ an Position

$$(i - 1)m + 0 + 1 + \dots + (i - 2) + j = (i - 1)m + (i - 1)(i - 2)/2 + j.$$

2.Fall: $m < i < n - m + 1$.

Die ersten m Zeilen von M enthalten

$$m^2 + 1 + \dots + m - 1 = 3m^2/2 - m/2$$

relevante Positionen. Also steht $M(i, j)$ an Position

$$\begin{aligned} 3m^2/2 - m/2 + (i - m - 1)(2m - 1) + j - (i - m) \\ = 2im - m^2/2 - 2i - m/2 + j + 1. \end{aligned}$$

3.Fall: $i \geq n - m + 1$.

Die ersten $n - m$ Zeilen von M enthalten

$$3m^2/2 - m/2 + (n - 2m)(2m - 1) = 2nm - 5m^2/2 - n + 3m/2$$

relevante Positionen. Also steht $M(i, j)$ an Position

$$\begin{aligned} 2nm - 5m^2/2 - n + 3m/2 + (2m - 1) + \dots + (m + n - i + 1) + j - (i - m) \\ = 2nm - m^2/2 - n + 3m/2 - (m + n - i)(m + n - i + 1)/2 + j - i. \end{aligned}$$

Hier haben wir es bereits mit einem Trade-off zu tun. Es ist in jedem Anwendungsfall zu entscheiden, ob der Speicherplatzgewinn groß genug ist, dass sich die jeweilige Auswertung der Speicherabbildungsfunktion lohnt. Allerdings sollten wir nicht aus Bequemlichkeit auf die komprimierte Speicherung von Bandmatrizen verzichten. Das Aufstellen der Speicherabbildungsfunktion ist zwar mühevoll und langweilig, aber eine einmalige Arbeit, während der Gewinn bei sehr vielen Anwendungen auftreten kann.

Wenn das Matricelement $M(i, j)$ Abhängigkeiten zwischen Objekt i und Objekt j ausdrückt (z. B. Übergangswahrscheinlichkeiten bei der Simulation von Rechnern), haben die Matrizen oft sehr viele Nullen, die aber nicht wie bei Dreiecksmatrizen oder Bandmatrizen an vorgegebenen Positionen auftreten. Derartige Matrizen werden spärlich besetzt (sparse) genannt, ohne dass dieser Begriff präzise definiert wird. Arrays bilden für spärlich besetzte Matrizen eine Möglichkeit, diese komprimiert darzustellen. Allerdings wird dann das Auffinden von Matricelementen schwieriger. Wir diskutieren zwei Möglichkeiten.

Bei der Koordinatenmethode wird jedes Matricelement als Tripel $(i, j, M(i, j))$ aus Zeilenindex, Spaltenindex und Datum abgespeichert. Die von null verschiedenen Matricelemente werden zeilenweise gespeichert. Bei N von null verschiedenen Matricelementen genügt ein Array der Länge N , wobei jedes Arrayobjekt ein Tripel der oben beschriebenen Art ist.

Mit der zeilenweisen Abspeicherung haben wir die lexikographische Ordnung auf den Paaren (i, j) gewählt, d. h.

$$(i, j) < (i', j') : \Leftrightarrow i < i' \text{ oder } (i = i' \text{ und } j < j').$$

Um $M(i, j)$ zu berechnen, suchen wir in dem bezüglich (i, j) geordneten Array nach einem Tripel (i, j, \cdot) . Wenn wir ein derartiges Tripel finden, steht an der dritten Stelle $M(i, j)$. Wenn ein derartiges Tripel nicht abgespeichert ist, ist $M(i, j) = 0$. Wir haben die Wahl zwischen den beschriebenen Suchstrategien, wobei meistens die binäre Suche die beste Wahl ist.

Bei der Bitmusterdarstellung machen wir davon Gebrauch, dass wir in einem Maschinenwort der Länge w auch w Bits abspeichern können. Eine Matrix, deren Einträge nur Nullen und Einsen sind, kann also wesentlich kompakter abgespeichert werden als eine allgemeine Matrix M . Sei $M^*(i, j) = 0$, falls $M(i, j) = 0$, und $M^*(i, j) = 1$ sonst. Es wird M^* wie oben beschrieben abgespeichert. Dann werden die von null verschiedenen Matricelemente von M der Reihe nach zeilenweise abgespeichert. Zur Berechnung von $M(i, j)$ wird zuerst $M^*(i, j)$ angeschaut. Falls $M^*(i, j) = 0$, ist $M(i, j) = 0$. Ansonsten wird die Anzahl z aller $(i', j') \leq (i, j)$ mit $M^*(i', j') = 1$ berechnet. In dem Array, das die Einträge von M enthält, steht $M(i, j)$ dann an Position z .

Wir können hier einen typischen Trade-off-Effekt feststellen. Vorteile auf der einen Seite (Datenkompression, Einsparung von Speicherplatz) werden mit Nachteilen auf einer anderen Seite (komplexere Implementierung, insbesondere aber langsamerer Zugriff auf die Daten) erkaufte. In den Anwendungen muss abgewogen werden, was wichtiger ist.

2.3 Lineare Listen

Ein Array ist eine statische Datenstruktur zur Verwaltung von Folgen (a_1, \dots, a_n) . Bei vielen Anwendungen ist es nötig, Objekte oder Daten zu entfernen oder einzufügen. Lineare Listen unterstützen im Gegensatz zu Arrays diese Operationen und bilden daher eine *dynamische* Datenstruktur. Rechnerintern kann die Folge dann aber nicht mehr in der gegebenen Reihenfolge einen Speicherabschnitt belegen. Dies würde ja implizieren, dass beim Einfügen eines neuen Datums alle folgenden Daten verschoben werden müssen.

Stattdessen vermerken wir bei jedem Datum, wo wir das folgende Datum finden. Wir verwenden so genannte *Zeiger* auf das folgende Datum. Es ist klar, dass die Zeiger, die es ja in Arrays nicht gibt, zusätzlichen Speicherplatz beanspruchen. Die meisten Programmiersprachen unterstützen die Verwendung von Zeigern, so dass wir uns um die Details nicht mehr kümmern müssen. Wir wollen dennoch die Grundlagen des Umgangs mit Zeigern kurz ansprechen. Zu jedem Datum x gehört nun ein Zeiger p auf das folgende Datum, wobei nil einen Zeiger nach nirgendwo und daher das Ende der Liste beschreibt. Mit $p \uparrow$ bezeichnen wir das Objekt (y, q) , auf das p zeigt. Dann definieren wir

$$p \uparrow .dat := y \text{ und } p \uparrow .next := q.$$

Eine Liste L wird nun durch einen Zeiger p_L beschrieben, wobei $p_L \uparrow .dat$ das erste Listendatum ist und $p_L \uparrow .next$ auf das folgende Paar zeigt. Die schematische Darstellung findet sich in Abbildung 2.3.1.

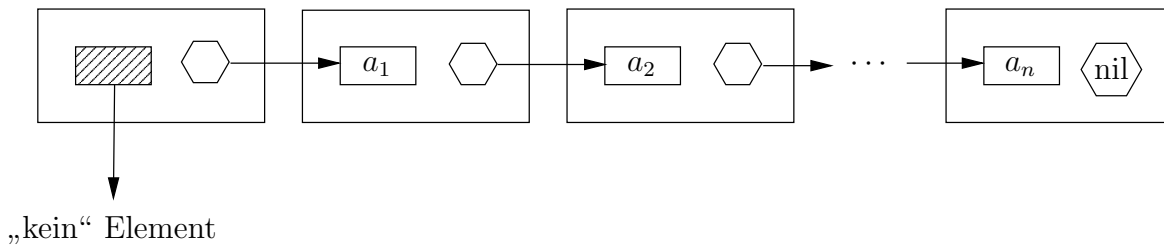


Abbildung 2.3.1: Darstellung einer linearen Liste.

Wie haben wir uns nun die rechnerinterne Abspeicherung einer Liste vorzustellen? Die Zeiger beschreiben Adressen, wo das nächste Paar aus Datum und Zeiger zu finden ist. Die Liste $L = (347, 18, 12, 6, 4711, 12)$, die an der Adresse 9 angesprochen wird, kann intern folgendermaßen abgespeichert sein.

1	2	3	4	5	6	7	8	9	10	11	12	13
18	12			347	4711	6		.			12	...
12	nil			1	2	6		5			7	...

Abbildung 2.3.2: Rechnerinterne Speicherung einer linearen Liste.

Der Befehl $new(p)$ sorgt dafür, dass der Zeiger p auf ein neues Paar zeigt, für das weder das Datum noch der Zeiger definiert ist. Garbage Collection Algorithmen müssen dafür sorgen, dass frei gewordene Speicherplätze eingesammelt und wieder vernünftig genutzt werden. Die Initialisierung einer Liste ist nun offensichtlich in linearer Zeit möglich. Zum Durchlauf einer linearen Liste ist eine while-Schleife mit dem Abbruchkriterium $p=nil$ geeignet.

Ein Hauptvorteil einer linearen Liste im Vergleich zu einem Array besteht darin, dass ein Objekt in konstanter Zeit hinter einem anderen Listenobjekt eingefügt werden kann. Andere Objekte müssen nicht verschoben werden. Dies wollen wir im Detail beschreiben.

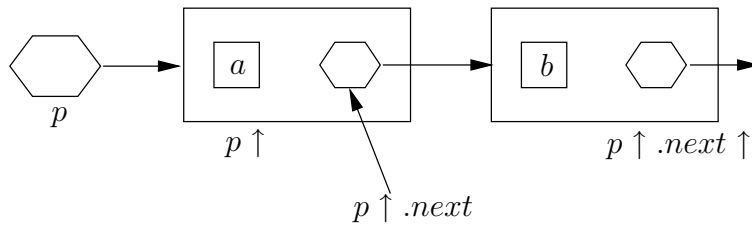


Abbildung 2.3.3: Eine Liste, in die ein neues Datum eingefügt werden soll.

Sei also p ein Zeiger auf ein Objekt, hinter das das Element x eingefügt werden soll. Die Ausgangssituation ist in Abbildung 2.3.3 dargestellt.

Folgende Befehlsfolge löst die Aufgabe, wobei q eine Variable vom Typ Zeiger ist.

- (1) $q := p \uparrow .next$
- (2) $new(p \uparrow .next)$
- (3) $(p \uparrow .next) \uparrow .dat := x$
- (4) $(p \uparrow .next) \uparrow .next := q$

Die Abbildungen 2.3.4, 2.3.5 und 2.3.6 zeigen die Situationen nach den Befehlen (1), (2) und (4).

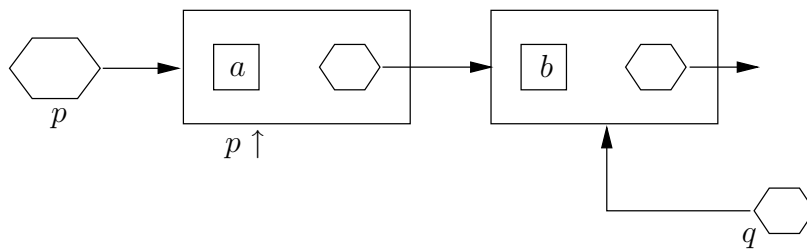


Abbildung 2.3.4: Die Situation nach Befehl (1).

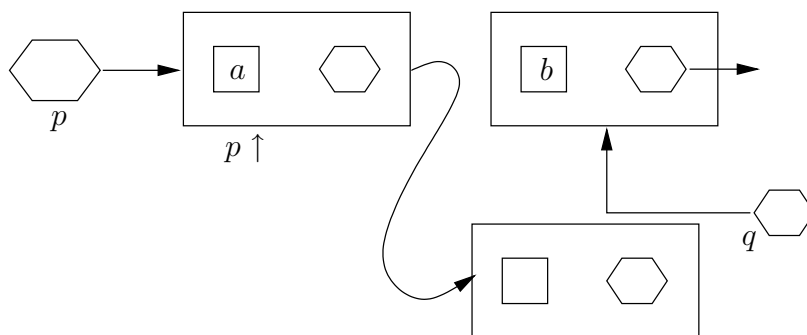


Abbildung 2.3.5: Die Situation nach Befehl (2).

Wir machen uns die Wirkung des Programms noch einmal an der rechnerinternen Ab-
speicherung deutlich. Sei p der Zeiger auf das Paar $(18, 12)$, also auf die Adresse 1. Sei

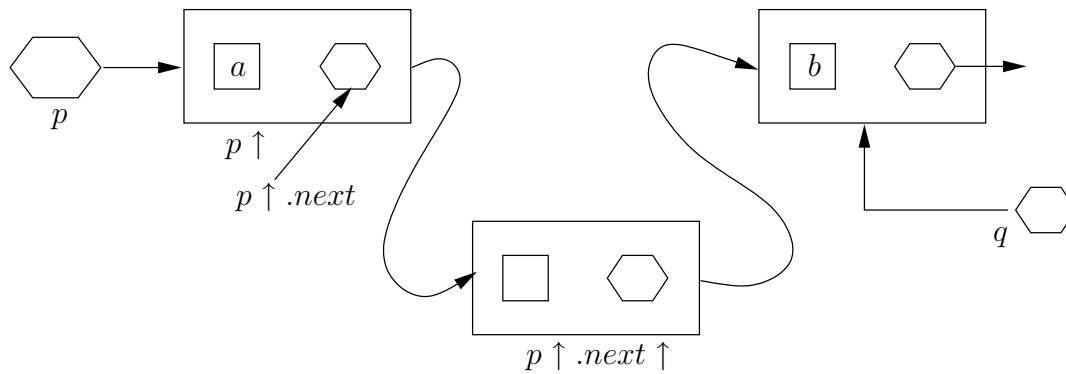


Abbildung 2.3.6: Die Situation nach Befehl (4).

$x = 93$. Mit Befehl (1) merken wir uns in q die Adresse 12. Befehl (2) stellt uns z. B. die Adresse 8 als freie Adresse zur Verfügung, wobei nun an Position 1 der Eintrag (18, 12) durch (18, 8) ersetzt wird. Die Befehle (3) und (4) speichern an der Adresse 8 den Eintrag (93, 12).

Bevor wir untersuchen, wie wir die wichtigsten Operationen auf linearen Listen ausführen können, beschreiben wir Erweiterungen von linearen Listen um zusätzliche Informationen. Diese Zusatzinformationen können in einigen Anwendungen sehr nützlich sein. Da diese Informationen ebenfalls stets aktualisiert werden müssen, sollten sie aber auch nur benutzt werden, wenn sie auch gebraucht werden.

- Zusätzliche Zeiger auf das letzte Datum der Liste.
- Zusätzliche Zeiger auf das jeweilige Vorgängerdatum, dies führt zu doppelt verketeten Listen.
- Zusätzliche Variable, die die aktuelle Länge der Liste beschreibt.

Bei der Behandlung der Operationen auf Listen führen wir nun auch gleich einen Vergleich mit Arrays durch.

- 1.) Initialisierung. Eine leere Liste kann in konstanter Zeit initialisiert werden. Das Einfügen der Folge (a_1, \dots, a_n) in eine leere Liste kostet Zeit $\Theta(n)$. Bei Arrays muss n vorab bekannt sein, dann auch Zeit $\Theta(n)$.
- 2.) Suche das Datum an Position p . Liste: $\Theta(p)$. Array: $\Theta(1)$.
- 3.) Suche Datum x . Liste der Länge l : $\Theta(p)$, falls Datum an Position p steht, $\Theta(l)$, wenn die Liste x nicht enthält. Suchen in geordneten Listen können erfolglos abgebrochen werden, wenn ein größeres Datum gefunden wird. Array: $O(n)$, bei geordneten Arrays mit binärer Suche: $O(\log n)$.
- 4.) Einfügen von Datum y hinter dem Datum x nach erfolgreicher Suche nach x . Liste: $\Theta(1)$. Diese Operation wird in Arrays nicht unterstützt.

- 5.) Einfügen von Datum y vor Datum x nach erfolgreicher Suche nach x . Diese Operation wird von Listen und Arrays nicht unterstützt. Bei Listen könnte man sich bei der Suche den Zeiger auf das Vorgängerdatum merken, doppelt verkettete Listen ermöglichen diese Operation in Zeit $\Theta(1)$.
- 6.) Entfernen von x nach erfolgreicher Suche. Auch diese Operation wird von Listen nicht direkt unterstützt. Bei der Suche muss der Zeiger auf das Vorgängerdatum gespeichert werden, um eine Zeit von $\Theta(1)$ zu ermöglichen. Doppelt verkettete Listen: $\Theta(1)$. Arrays unterstützen diese Operation nicht.
- 7.) Ersetze x durch y nach erfolgreicher Suche. Listen und Arrays: $\Theta(1)$.
- 8.) Bestimme Nachfolger. Listen und Arrays: $\Theta(1)$.
- 9.) Bestimme Vorgänger. Listen: $O(l)$, doppelt verkettete Listen und Arrays: $\Theta(1)$.
- 10.) Bestimme Anfangsdatum. Listen und Arrays: $\Theta(1)$.
- 11.) Bestimme letztes Datum oder Zeiger auf dieses Datum. Listen: $\Theta(l)$, bei Extrazeigern: $\Theta(1)$, Arrays: $\Theta(1)$.
12. Bestimme Länge. Listen: $\Theta(l)$, bei Verwaltung einer Extravariablen: $\Theta(1)$, bei Arrays im Allgemeinen fest vereinbart.
- 13.) Konkatenation, die Erzeugung der Folge $(a_1, \dots, a_n, b_1, \dots, b_m)$ aus den Folgen (a_1, \dots, a_n) und (b_1, \dots, b_m) . Listen: $\Theta(n)$, bei Extrazeigern auf das letzte Listenelement: $\Theta(1)$. Arrays unterstützen diese Operation nicht.
- 14.) Aufspaltung der Liste (a_1, \dots, a_n) in zwei Listen (a_1, \dots, a_m) und (a_{m+1}, \dots, a_n) nach erfolgreicher Suche nach a_m . Listen: $\Theta(1)$. Diese Operation wird von Arrays von der Idee her nicht unterstützt, sie ist aber in Zeit $\Theta(1)$ durchführbar.
- 15.) Verwaltung von Stacks. Listen: jede Operation in Zeit $\Theta(1)$, da alle Operationen lokal den Listenanfang betreffen. Arrays sind hierfür nicht gedacht, bei Vereinbarung einer Maximalgröße können sie verwendet werden, um Stackoperationen in Zeit $\Theta(1)$ zu ermöglichen.
- 16.) Verwaltung von Queues. Da Einfügungen am Ende stattfinden, sollten Listen mit Extrazeiger auf das Listende benützt werden. Sie ermöglichen jede Queueoperation in Zeit $\Theta(1)$. In Arrays könnte eine ringförmige Nummerierung verwendet werden. Mit zwei Variablen, die auf das aktuelle erste und letzte Datum zeigen, und bei Einhaltung einer Maximalgröße ist Zeit $\Theta(1)$ pro Queueoperation möglich.

Listen und Arrays unterstützen also teilweise verschiedene Operationen. Wenn wir effizient nach Daten suchen wollen und Daten effizient neu einfügen und entfernen wollen, sind weder Listen noch Arrays geeignet. Datenstrukturen, die diese drei zentralen Operationen gemeinsam unterstützen, werden in Kapitel 4 vorgestellt.

Als Anwendung linearer Listen wollen wir das topologische Sortieren behandeln.

Definition 2.3.1: Die Relation \leq ist eine vollständige (partielle) Ordnung auf M , wenn (1)–(4) ((2)–(4)) gelten.

- (1) $\forall x, y \in M: x \leq y$ oder $y \leq x$.
- (2) $\forall x \in M: x \leq x$.
- (3) $\forall x, y \in M: x \leq y$ und $y \leq x \Rightarrow x = y$.
- (4) $\forall x, y, z \in M: x \leq y$ und $y \leq z \Rightarrow x \leq z$.

Es gibt viele praktische Beispiele von partiellen Ordnungen, die keine vollständigen Ordnungen sind. So ist die Potenzmenge einer endlichen Menge bezüglich der Teilmengenrelation partiell, aber nicht vollständig geordnet. Gleiches gilt im Allgemeinen für die Relation \leq auf der Menge der für ein Projekt notwendigen Arbeiten. Es sei $a \leq b$, wenn $a = b$ ist oder a beendet sein muss, bevor b begonnen wird.

Sei M eine n -elementige Menge und \leq eine partielle Ordnung auf M . Unter einer topologischen Sortierung verstehen wir eine vollständige Ordnung auf M , also eine Aufzählung m_1, \dots, m_n der Elemente aus M , so dass für $i > j$ nicht $m_i \leq m_j$ gilt. In unserem zweiten Beispiel ist dies eine Reihenfolge, in der die Arbeiten durchgeführt werden können.

Lemma 2.3.2: Jede partiell geordnete endliche Menge lässt sich topologisch sortieren.

Beweis: Es genügt, die Existenz eines minimalen Elementes z in M zu zeigen. Dabei heißt z minimal, wenn es kein y mit $y \neq z$ und $y \leq z$ gibt. Wir können dann $m_1 = z$ wählen und anschließend $M - \{z\}$ topologisch sortieren.

Für $y \in M$ gilt, dass y minimal ist oder es ein $d(y) \in M$ mit $d(y) \neq y$ und $d(y) \leq y$ gibt. Wenn M kein minimales Element hat, ist $d(y)$ für alle $y \in M$ wohldefiniert. Für ein beliebiges $y \in M$ bilden wir die Kette $y, d(y), d^2(y) := d(d(y)), \dots, d^n(y)$. Da diese Kette $n + 1$ Elemente hat und M nur n Elemente enthält, gibt es Zahlen i und j mit $0 \leq i < j \leq n$ und $d^i(y) = d^j(y)$. Sei $x := d^i(y)$ und $k := j - i$. Dann ist $x = d^k(x)$ und $k \geq 1$. Wegen der Transitivität partieller Ordnungen gilt

$$x = d^k(x) \leq d(x) \leq x.$$

Wegen der Eigenschaft (3) für partielle Ordnungen folgt $x = d(x)$ im Widerspruch zur Definition von d . Also muss M mindestens ein minimales Element haben. \square

Eine partielle Ordnung \leq auf $M = \{x_1, \dots, x_n\}$ kann durch n Listen $L(1), \dots, L(n)$ beschrieben werden. Dabei enthält $L(i)$ genügend viele j mit $x_i \leq x_j$, um die partielle Ordnung vollständig zu beschreiben. Aufgrund der Transitivität von \leq muss k nicht in $L(i)$ stehen, wenn $j \in L(i)$ und $k \in L(j)$ ist. Außerdem nehmen wir an, dass i nicht in $L(i)$ enthalten ist, da diese Information redundant ist. Es sei l die Summe der Listenlängen.

Ein naiver Algorithmus zur topologischen Sortierung geht folgendermaßen vor. Indem alle Listen durchlaufen werden, wird festgestellt, welche j in keiner Liste stehen. Dies

sind die Indizes aller minimalen Elemente, die an den Anfang der topologischen Sortierung geschrieben werden können. Dann werden die Listen $L(j)$ für die minimalen Elemente „eliminiert“. Der Algorithmus kann dann auf der Restmenge analog fortfahren. Es ist leicht, sich Beispiele auszudenken, für die dieser Algorithmus eine Rechenzeit von $\Theta(n^2 + nl)$ benötigt.

Mit Hilfe der uns bereits bekannten Datenstrukturen können wir die Rechenzeit auf $O(n + l)$ drücken. Dies ist die optimale Größenordnung, da die Eingabelänge l und die Ausgabelänge n ist.

Algorithmus 2.3.3: Eingabe: $L(1), \dots, L(n)$, die die betrachtete partielle Ordnung wie oben beschrieben darstellen.

- (1) Initialisierung: Es wird ein Array a der Länge n mit $a(i) = 0, 1 \leq i \leq n$, initialisiert. Es wird eine leere Queue Q initialisiert.
- (2) Preprocessing: Alle Listen werden einmal durchlaufen und dabei wird in $a(i)$ abgespeichert, in wie vielen Listen i eingetragen ist.
- (3) Wir durchlaufen a und schreiben alle k mit $a(k) = 0$ in die Queue Q . Damit enthält Q die Indizes aller minimalen Elemente. In Zukunft soll Q die Elemente enthalten (wir identifizieren in der Beschreibung Elemente und ihre Indizes), die, nachdem die bereits in die Ausgabe geschriebenen Elemente entfernt wurden, minimal sind.
- (4) While $Q \neq \emptyset$ do:
Entferne das vorderste Element j aus Q und schreibe x_j in die Ausgabe. Für jedes i in $L(j)$ verringere $a(i)$ um 1. Wenn $a(i)$ den Wert 0 bekommt, füge i in die Queue Q ein.

Satz 2.3.4: Algorithmus 2.3.3 löst das Problem des topologischen Sortierens in Zeit $O(n + l)$.

Beweis: Es ist offensichtlich, dass alle Elemente x_i mit $a(i) = 0$ minimal sind. Für die anderen Elemente x_j ist aber $a(j)$ nicht unbedingt die Anzahl der Elemente x_k mit $x_k \leq x_j$ und $k \neq j$. Unsere Problembeschreibung impliziert ja zusätzliche \leq -Beziehungen, die sich aus der Transitivität von \leq ergeben. Dies sind aber Elemente x_m , so dass für ein k gilt: $x_m \leq x_k$ und $j \in L(k)$. Wenn also alle Elemente x_k , so dass $j \in L(k)$ ist, in die Ausgabe geschrieben worden sind, sind alle Elemente x_m mit $x_m \leq x_j$ in die Ausgabe geschrieben worden. Da Q als Queue und damit nach dem FIFO-Prinzip (first in first out) verwaltet wird, ist der Algorithmus korrekt.

Die Laufzeiten von $O(n)$, $O(n + l)$ und $O(n)$ für die ersten drei Schritte sind offensichtlich. Auch der vierte Schritt braucht nur Zeit $O(n + l)$, da jedes Element genau einmal in die Queue aufgenommen wird, genau einmal entfernt wird und jede Liste genau einmal durchlaufen wird. \square

Beispiel 2.3.5: $n = 9$, $l = 12$. Die gegebenen Listen und die berechneten a -Werte sind in Abbildung 2.3.7 dargestellt.

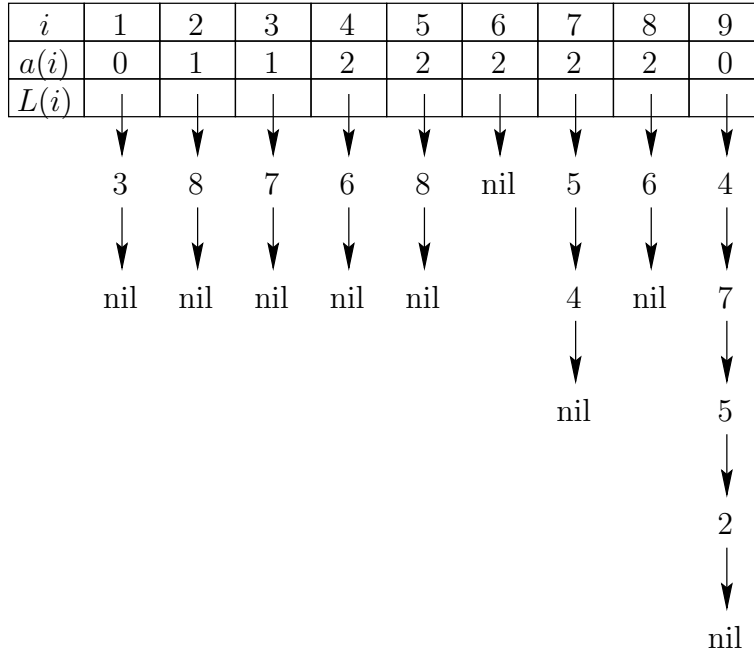


Abbildung 2.3.7: Die Information nach Schritt 2 von Algorithmus 2.3.3.

Da $a(1) = a(9) = 0$ und $a(i) \neq 0$ sonst, wird $Q = (1, 9)$ in Schritt (3).

Output x_1 , durchlaufe $L(1)$, $a(3) := 0$, $Q := (9, 3)$;

Output x_9 , durchlaufe $L(9)$, $a(4) := 1$, $a(7) := 1$, $a(5) := 1$, $a(2) := 0$, $Q := (3, 2)$;

Output x_3 , durchlaufe $L(3)$, $a(7) := 0$, $Q := (2, 7)$;

Output x_2 , durchlaufe $L(2)$, $a(8) := 1$, $Q := (7)$;

Output x_7 , durchlaufe $L(7)$, $a(5) := 0$, $a(4) := 0$, $Q := (5, 4)$;

Output x_5 , durchlaufe $L(5)$, $a(8) := 0$, $Q := (4, 8)$;

Output x_4 , durchlaufe $L(4)$, $a(6) := 1$, $Q := (8)$;

Output x_8 , durchlaufe $L(8)$, $a(6) := 0$, $Q := (6)$;

Output x_6 , durchlaufe $L(6)$, $Q := \emptyset$, STOP.

Lineare Listen ermöglichen auch eine weitere Alternative zur Darstellung spärlich besetzter Matrizen M . Die i -te Zeile von M wird als lineare Liste $L(i)$ angelegt. Das erste Objekt sei $(i, 0, 0)$, um die Zeilennummer anzuzeigen, es folgen die von null verschiedenen Matrixelemente dieser Zeile als Tripel $(i, j, M(i, j))$. Die Matrix ist als Array der Listen $L(i)$ angelegt. Natürlich würde es ausreichen, in $L(i)$ Paare $(j, M(i, j))$ abzuspeichern. In Kürze werden wir eine Erweiterung der Datenstruktur sehen, bei der die zusätzliche Information i wichtig ist.

Beispiel 2.3.6:

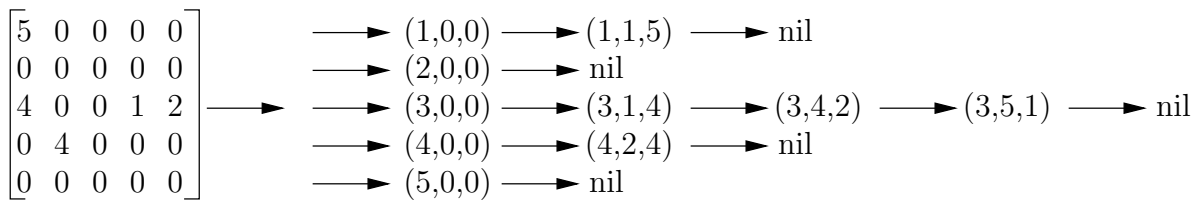


Abbildung 2.3.8: Die Listendarstellung einer spärlich besetzten Matrix.

Wir wollen nun zwei spärlich besetzte Matrizen A und B addieren. Die Zeilen werden einzeln addiert. Zunächst wird für die Matrix $C = A + B$ der Zeilenkopf $(i, 0, 0)$ angelegt. In A und B werden die i -ten Zeilen durchlaufen. Wir nehmen an, dass wir $(i, j, A(i, j))$ und $(i, k, B(i, k))$ erreicht haben.

1.Fall: $j < k$.

Das Element $(i, j, A(i, j))$ wird an die Ergebnisliste angehängt. In der A -Liste wird der Nachfolger aufgesucht.

2.Fall: $j > k$.

Das Element $(i, k, B(i, k))$ wird an die Ergebnisliste angehängt. In der B -Liste wird der Nachfolger aufgesucht.

3.Fall: $j = k$.

Es wird $s := A(i, j) + B(i, k)$ berechnet. Falls $s \neq 0$, wird (i, j, s) an die Ergebnisliste angehängt. In jedem Fall wird in beiden Listen der Nachfolger aufgesucht.

Erreichen wir ein Zeilenende, wird der Rest der anderen Zeile an die Ergebnisliste kopiert und gestoppt. Erreichen wir beide Zeilenenden, wird gestoppt. Stoppen bedeutet, dass die Ergebniszeile mit einem nil-Zeiger abgeschlossen wird.

Wenn A und B $n \times m$ -Matrizen mit a bzw. b von null verschiedenen Elementen sind, kann C mit dem obigen Algorithmus in Zeit $O(n + a + b)$ berechnet werden. Der Algorithmus wird aus nahe liegenden Gründen auch Reißverschlussverfahren genannt.

Diese Darstellung der Matrizen ist ungeeignet, wenn wir Matrizen sowohl addieren als auch multiplizieren wollen. Dann benutzen wir für die Zeilen und für die Spalten lineare Listen, wobei jedes Tripel $(i, j, A(i, j))$ nur einmal dargestellt wird, und daher zwei Zeiger auf jedes Tripel zeigen. Jedes Tripel erhält nun auch zwei Zeiger, $znext$ für den Zeilennachfolger und $snext$ für den Spaltennachfolger. Genau genommen haben wir aus linearen Listen bereits eine komplexere Datenstruktur konstruiert. Addition und Multiplikation spärlich besetzter Matrizen in dieser Darstellung werden in den Übungen behandelt. Wir begnügen uns hier mit der Darstellung der Matrix aus Beispiel 2.3.6.

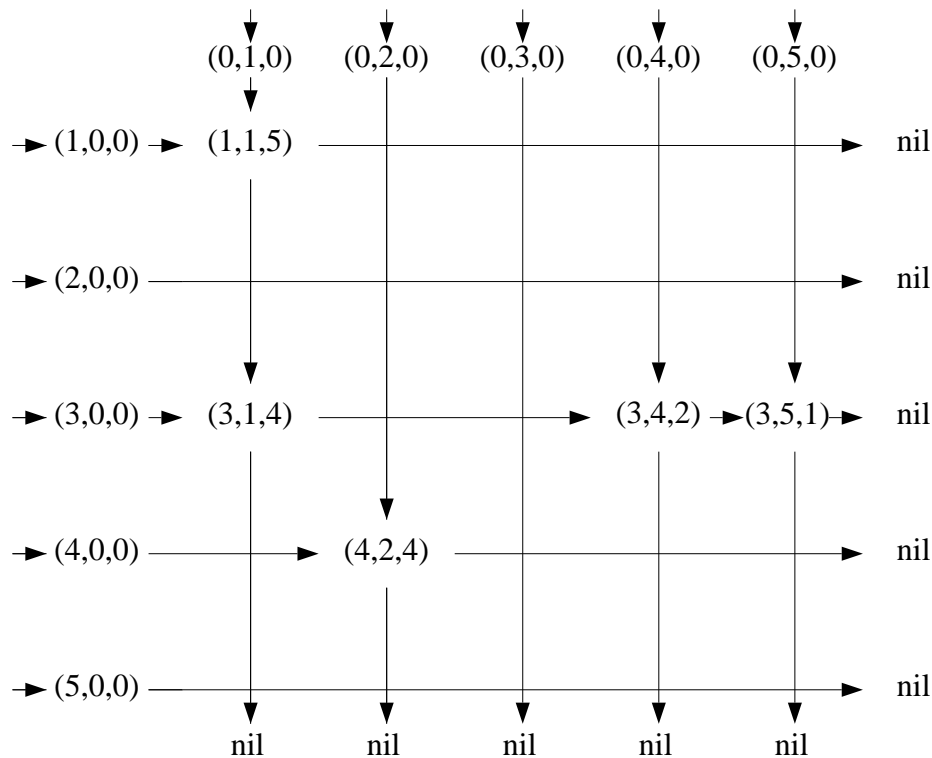


Abbildung 2.3.9: Eine spärlich besetzte Matrix mit Zeilen- und Spaltenlisten.

2.4 Datenstrukturen für Mengen

Für die Darstellung von Mengen unterscheiden wir im Wesentlichen zwei Datenstrukturen: die Bitvektordarstellung (oder Darstellung durch charakteristische Vektoren) und die Listendarstellung.

Die Bitvektordarstellung ist für ein vorgegebenes Universum $U = \{1, \dots, n\}$ besonders geeignet. Wir setzen voraus, dass wir nur Mengen $M \subseteq U$ darstellen müssen. Allerdings sollten die betrachteten Mengen in der Regel nicht sehr viel kleiner als die Grundmenge U sein. Anders ausgedrückt: Die Menge U sollte nicht zu groß sein. Wir stellen M durch ein Array a der Länge n dar. Dabei gilt

$$a(i) = 1 \Leftrightarrow i \in M, \text{ und } a(i) = 0 \text{ sonst.}$$

Mengenoperationen, die nur ein Element betreffen, sind in konstanter Zeit ausführbar:

- Ist $i \in M$? Antwort: $a(i)$.
- Füge i zu M hinzu. $a(i) := 1$.
- Entferne i aus M . $a(i) := 0$.

Die anderen gängigen Mengenoperationen sind in linearer Zeit ausführbar, allerdings linear bezüglich n , der Mächtigkeit der Grundmenge. Die Arrays a_1 und a_2 mögen M_1 und M_2 darstellen:

- $M = M_1 \cup M_2$. $a(i) := a_1(i) \vee a_2(i)$ für $1 \leq i \leq n$.
- $M = M_1 \cap M_2$. $a(i) := a_1(i) \wedge a_2(i)$ für $1 \leq i \leq n$.
- $M = M_1 - M_2$. $a(i) := a_1(i) \wedge (\neg a_2(i))$ für $1 \leq i \leq n$.
- $M = M_1 \Delta M_2$ (symmetrische Differenz). $a(i) := a_1(i) \oplus a_2(i)$ für $1 \leq i \leq n$.

Wie aber schon in Kapitel 2.2 diskutiert, passen w Bits in ein Maschinenwort. Im Allgemeinen sind die booleschen Operationen \wedge , \vee , \neg , \oplus auf ganzen Wörtern in konstanter Zeit durchführbar. Die Ausführungszeit der letzten Operationen fällt dann auf $\lceil n/w \rceil$. Die erste Gruppe von Operationen bleibt in Zeit $O(1)$ ausführbar, wenn auf jedes Bit des Wortes in konstanter Zeit zugegriffen werden kann. Das Bit $a(i)$ steht dann im $\lceil i/w \rceil$ -ten Wort an der Position $i \bmod w$, wenn die Positionen mit $0, \dots, w-1$ durchnummeriert sind.

Wenn das Universum, aus dem die betrachteten Elemente stammen dürfen, sehr groß ist oder zumindest viel größer als die typischerweise betrachteten Mengen, bietet sich eine Darstellung jeder Menge als Liste ihrer Elemente an. Wir unterscheiden ungeordnete Listen und geordnete Listen. Die Frage „ $x \in M$?“ benötigt in beiden Fällen im worst case Zeit $\Theta(|M|)$. Nach erfolgreicher Suche können Elemente in Zeit $\Theta(1)$ entfernt werden und nach erfolgloser Suche kann ein Element in beiden Fällen in Zeit $\Theta(1)$ eingefügt werden. Die Vereinigung zweier Mengen M_1 und M_2 kann in ungeordneten Listen in Zeit $\Theta(1)$ durchgeführt, wenn wir Zeiger auf die Listenenden verwalten. Allerdings kommen dann Elemente aus M_1 und M_2 in der Vereinigung doppelt vor, was vermieden werden sollte.

Dann müssen bei der Vereinigung von ungeordneten Listen alle Elemente aus M_1 in M_2 gesucht werden (oder umgekehrt). Wir sehen keine Möglichkeit, eine Rechenzeit von $\Theta(|M_1| \cdot |M_2|)$ zu vermeiden. Bei geordneten Listen können wir dagegen das in Kapitel 2.3 beschriebene Reißverschlussverfahren einsetzen und die Rechenzeit auf $O(|M_1| + |M_2|)$ senken.

	Bitvektor	ungeordnete Liste	geordnete Liste
$i \in M?$	$O(1)$	$O(l)$	$O(l)$
INSERT	$O(1)$	$O(1)$	$O(1)$
DELETE	$O(1)$	$O(1)$	$O(1)$
Vereinigung	$O(n)$ bzw. $O(n/w)$	$O(l_1 l_2)$	$O(l_1 + l_2)$
Durchschnitt	$O(n)$ bzw. $O(n/w)$	$O(l_1 l_2)$	$O(l_1 + l_2)$
Differenz	$O(n)$ bzw. $O(n/w)$	$O(l_1 l_2)$	$O(l_1 + l_2)$
Symm. Diff.	$O(n)$ bzw. $O(n/w)$	$O(l_1 l_2)$	$O(l_1 + l_2)$

Tabelle 2.4.1: Rechenzeiten für Operationen auf Mengen.

Wir fassen die Ergebnisse zusammen, wobei wir mit l_1 und l_2 die Längen der Listen L_1 und L_2 bezeichnen. Bei den Operationen INSERT und DELETE nehmen wir an, dass die erfolglose bzw. erfolgreiche Suche bereits durchgeführt worden ist und der Zeiger auf das gefundene Objekt gespeichert wurde. Die Größe der Grundmenge sei n und die Wortlänge w .

2.5 Datenstrukturen für Bäume

Die Behauptung, dass es wohl kein Teilgebiet der Informatik gibt, in dem Bäume keine wesentliche Rolle spielen, ist nicht übertrieben.

Wir beginnen mit Definitionen und Notationen. Unter vielen Möglichkeiten beschränken wir uns zunächst auf gewurzelte Bäume mit Kanten, die von der Wurzel weg gerichtet sind. Ein derartiger Baum T besteht aus einer endlichen Knotenmenge V , häufig $V = \{1, \dots, n\}$, und einer Kantenmenge $E \subseteq V \times V - \{(i, i) \mid i \in V\}$. Es gibt genau einen Knoten r , die Wurzel von T , für den es keine Kante $(\cdot, r) \in E$ gibt. Für alle Knoten $v \neq r$ gibt es genau einen Knoten w mit $(w, v) \in E$. Also haben Bäume $|V| - 1$ Kanten. Falls $(w, v) \in E$, heißt w Elter von v und v Kind von w . Sind v und $v' \neq v$ Kinder von w , heißen sie Geschwister. Ein Knoten v heißt Nachfolger von v_0 (und dann heißt v_0 Vorgänger von v), wenn es $v_1, \dots, v_m = v \in V$ gibt, so dass $(v_0, v_1), (v_1, v_2), \dots, (v_{m-1}, v_m) \in E$ sind. Diese Kanten beschreiben einen Weg (oder Pfad) der Länge m von v_0 zu v_m . Um wirklich einen Baum zu erhalten, müssen wir noch fordern, dass es für jeden Knoten $v \in V$ einen Weg von r zu v gibt.

Mit $\text{ind}(v)$ bezeichnen wir die Anzahl der $w \in V$ mit $(w, v) \in E$. In Bäumen ist $\text{ind}(v) = 1$, falls $v \neq r$, und $\text{ind}(r) = 0$. Mit $\text{outd}(v)$ bezeichnen wir die Anzahl der Kinder von v . Knoten ohne Kinder heißen Blätter.

Die Tiefe eines Knotens v in T ist die Länge des eindeutigen Weges von der Wurzel r zu v . Die Tiefe des Baumes ist die maximale Tiefe eines Knotens, also die maximale Tiefe eines Blattes.

Ein Baum ist outd- k -beschränkt, wenn kein Knoten mehr als k Kinder hat. Ein Baum heißt k -är, wenn alle inneren Knoten, das sind die Knoten, die nicht Blätter sind, genau k Kinder haben. Dabei steht ternär für 3-är, binär für 2-är und unär für 1-är (lineare Liste). Gerade für binäre Bäume ist die Bezeichnungsweise manchmal schlampig. Outd-2-beschränkte Bäume werden häufig als binäre Bäume bezeichnet. Ein vollständiger k -ärer Baum der Tiefe d ist ein k -ärer Baum, dessen Blätter alle Tiefe d haben. Dieser Baum hat dann

$$1 + k + \dots + k^d = (k^{d+1} - 1)/(k - 1)$$

Knoten.

Häufig werden die Kinder eines Knotens vollständig geordnet, so dass wir für $1 \leq i \leq \text{outd}(v)$ vom i -ten Kind von v sprechen können. Auch die Bäume heißen dann geordnet. In binären Bäumen sprechen wir auch vom linken (ersten) und rechten (zweiten) Kind. Dies bezieht sich natürlich auf die graphische Darstellung von Bäumen. Jeder Knoten $v \in V$ definiert den Teilbaum $T(v)$, der aus v und seinen Nachfolgern besteht.

Sei $k = \text{outd}(v)$ und seien v_1, \dots, v_k die Kinder von v , dann zerfällt $T(v)$ in die Wurzel v und die Teilbäume $T(v_1), \dots, T(v_k)$.

Häufig kommt es darauf an, die Knoten eines Baumes in einer passenden Reihenfolge zu durchlaufen (traversieren). Dabei werden vier wichtige Reihenfolgen unterschieden.

Definition 2.5.1: Es sei $T = (V, E)$ ein geordneter Baum mit Wurzel $r(T)$ und den Kindern v_1, \dots, v_k der Wurzel. Dabei ist $k = 0$ möglich. Die folgenden Ordnungen werden rekursiv definiert.

- (1) Postorder : $\text{post}(T) := \text{post}(T(v_1)), \dots, \text{post}(T(v_k)), r(T)$.
- (2) Preorder : $\text{pre}(T) := r(T), \text{pre}(T(v_1)), \dots, \text{pre}(T(v_k))$.
- (3) Inorder : $\text{in}(T) := \text{in}(T(v_1)), r(T), \text{in}(T(v_2)), \dots, \text{in}(T(v_k))$.
- (4) Levelorder : $\text{lev}(T) := r(T), \text{lev}(T(v_1)), r(T), \text{lev}(T(v_2)), \dots, r(T), \text{lev}(T(v_k)), r(T)$.

Bevor wir zu den Datenstrukturen für Bäume kommen, beschreiben wir Operationen, die häufig unterstützt werden sollen.

PARENT(x, T): Berechne den Elter von x in T . Falls x Wurzel von T ist, soll die Antwort nil lauten.

CHILD(x, i, T): Berechne das i -te Kind von x in T . Falls x weniger als i Kinder hat, soll die Antwort nil lauten.

LCHILD(x, T): Dies ist der Befehl CHILD($x, 1, T$).

RCHILD(x, T): Berechne das letzte Kind von x in T .

ROOT(T): Berechne die Wurzel von T .

CONCATENATE(x, T_1, \dots, T_m): Erzeuge einen Baum T mit Wurzel x und m Kindern v_1, \dots, v_m , so dass $T(v_1) = T_1, \dots, T(v_m) = T_m$ ist.

DEPTH(x, T): Berechne die Tiefe von x in T .

DEPTH(T): Berechne die Tiefe von T .

SIZE(T): Berechne die Zahl der Knoten in T .

Wir diskutieren mehrere Datenstrukturen.

(1) Die Knoten werden in einem Array abgespeichert. Zu jedem Knoten wird der Elter angegeben. Hierbei kann eine Array- oder Zeigerdarstellung gewählt werden. Zusätzlich können ROOT(T) und SIZE(T) abgespeichert werden. Offensichtlich kann die PARENT-Anfrage in Zeit $\Theta(1)$ beantwortet werden. Die Wurzel finden wir, wenn sie nicht direkt abgespeichert ist, indem wir von einem beliebigen Knoten den PARENT-Zeigern folgen, bis wir auf einen nil-Zeiger stoßen. Dies kann Zeit DEPTH(T) dauern. Auf ähnliche Weise können wir DEPTH-Anfragen beantworten. Der Befehl CONCATENATE kann in Zeit $\Theta(m)$ bearbeitet werden, wenn die Wurzeln von T_1, \dots, T_m gefunden wurden. Entscheidender Nachteil dieser Datenstruktur ist es, dass wir, um die Kinder von x zu erhalten, für alle Knoten den Elter betrachten müssen.

(2) Wieder wird ein Array der Knoten benutzt. Für den Knoten i wird eine geordnete Liste seiner Kinder verwaltet. Hier werden die Probleme umgedreht. Nun ist es schwierig, den Elter eines Knotens zu berechnen. Damit ist auch die Berechnung der Tiefe eines Knotens schwierig.

(3) Kombination aus (1) und (2). Die Vorteile werden auf Kosten des Speicherplatzes kombiniert.

(4) Häufig ist die größtmögliche Zahl k von Kindern in einem Baum vorab bekannt. Dann können wir die Listen aus (2) und (3) durch Arrays mit dem bekannten Vorteil des direkten Zugriffs ersetzen. Für jeden Knoten wird ein Array der Länge $k + 1$ eingerichtet. An der letzten Stelle steht der Elter, auf den ersten k Positionen stehen die Kinder in geordneter Reihenfolge, wobei jeweils auch nil-Einträge erlaubt sind. Zusätzlich verwalten wir $\text{ROOT}(T)$ und $\text{SIZE}(T)$. Die Operationen PARENT , CHILD , LCHILD , ROOT und SIZE sind in Zeit $\Theta(1)$ durchführbar, CONCATENATE in Zeit $\Theta(m) = O(k)$. Für $\text{DEPTH}(x, T)$ ist Zeit $O(\text{DEPTH}(T))$ ausreichend. Die Operation RCHILD ist mit binärer Suche in Zeit $O(\log k)$ ausführbar. Wenn für jeden Knoten die Zahl der Kinder vermerkt wird, sinkt die Zeit auf $\Theta(1)$.

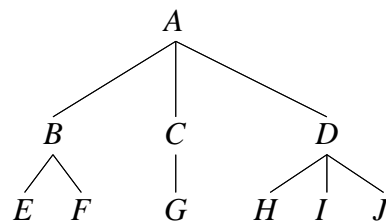
Wie gut ist die Speicherplatzausnutzung? In (1) wird kein Platz verschenkt. In (2) und (3) werden für die Zeiger $\Theta(n)$ Speicherplätze zusätzlich verbraucht. Wie viele der $n(k + 1)$ Arrayplätze der Datenstruktur (4) werden durch nil belegt? Da es $n - 1$ Kanten gibt und jede Kante zweimal dargestellt wird, gibt es $2n - 2$ von nil verschiedene Einträge und $n(k + 1) - (2n - 2) = nk - n + 2$ nil-Einträge. Der Anteil dieser Einträge beträgt

$$\frac{nk - n + 2}{n(k + 1)} = 1 - \frac{2n - 2}{n(k + 1)} = 1 - \frac{2}{k + 1} + \frac{2}{n(k + 1)} > 1 - \frac{2}{k + 1}.$$

Für $k = 10$ sind also mehr als 81,8 % der Einträge nil-Einträge. Eine fast 100%ige Speicherplatzausnutzung wird für $k = 1$ erreicht. Dann haben wir es aber mit doppelt verketteten Listen und nicht mit Baumstrukturen zu tun. Für $k = 2$ liegt die Speicherplatzverschwendung bei 33 %, und binäre Bäume sind schon „richtige“ Bäume, da sie sich verzweigen können. Daher versuchen wir, beliebige Bäume durch binäre Bäume zu simulieren.

(5) Sei T ein beliebiger Baum und v ein Knoten in T . Für v ist ein Array der Länge 3 vorgesehen, in dem der Elter, das linkeste Kind und das in der linearen Ordnung der Geschwister auf v folgende „Geschwist“ abgespeichert werden.

Beispiel 2.5.2:



$A : \text{nil}, B, \text{nil}.$ $F : B, \text{nil}, \text{nil}.$
 $B : A, E, C.$ $G : C, \text{nil}, \text{nil}.$
 $C : A, G, D.$ $H : D, \text{nil}, I.$
 $D : A, H, \text{nil}.$ $I : D, \text{nil}, J.$
 $E : B, \text{nil}, F.$ $J : D, \text{nil}, \text{nil}.$

Abbildung 2.5.1: Ein Baum und seine Simulation durch einen binären Baum.

Von den 30 Arrayplätzen sind 12 mit nil belegt. Die Operationen PARENT und LCHILD kosten Zeit $\Theta(1)$. Gleiches gilt für ROOT und SIZE, wenn die zugehörigen Variablen zusätzlich verwaltet werden. CONCATENATE kann in Zeit $\Theta(m)$ durchgeführt werden, und DEPTH hat die gleiche Zeitkomplexität wie in den Datenstrukturen (1), (3) und (4). Lediglich die Operationen CHILD und RCHILD sind teurer als in Datenstruktur (4). Für $\text{CHILD}(\cdot, i, T)$ ist Zeit $O(i)$ zu veranschlagen und für $\text{RCHILD}(x, T)$ Zeit $O(\text{outd}(x))$.

Wir stellen die Rechenzeiten für die wichtigsten Operationen und die Datenstrukturen (1)–(5) zusammen. Die Operationen ROOT und SIZE benötigen jeweils Zeit $O(1)$, wenn wir diese Parameter dynamisch mit verwalten.

	(1)	(2)	(3)	(4)	(5)
PARENT	$\Theta(1)$	$O(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
CHILD	$O(n)^*$	$\Theta(i)$	$\Theta(i)$	$\Theta(1)$	$\Theta(i)$
LCHILD	—	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
RCHILD	—	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\text{outd}(x))$
CONCATENATE	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
DEPTH	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

* Hier ist keine Ordnung auf den Kindern gegeben, es wird also eine Liste aller Kinder ausgegeben.

Tabelle 2.5.1: Operationen auf Bäumen und Rechenzeiten für verschiedene Datenstrukturen.

Es kommt also ganz auf die Anwendung an, welche Datenstruktur bevorzugt werden sollte.

2.6 Datenstrukturen für Graphen

Graphen gehören zu den wichtigsten Hilfsmitteln der Informatik, da sie Beziehungen zwischen je zwei Objekten ausdrücken können.

Auch hier beginnen wir mit Definitionen und Notationen. Ein ungerichteter Graph $G = (V, E)$ besteht aus einer endlichen Knotenmenge V und einer Kantenmenge E , wobei eine Kante $e = \{v, w\}$ eine zweielementige Teilmenge von V ist. Die Kante verbindet die Knoten v und w . Wenn Verwechslungen ausgeschlossen sind, schreiben wir auch $e = (v, w)$. Ein gerichteter Graph $G = (V, E)$ besteht aus einer endlichen Knotenmenge V und einer Kantenmenge E , wobei eine Kante $e = (v, w)$ ein geordnetes Paar von Knoten ist. Die Kante geht von v zu w .

Zwei Knoten heißen adjazent, wenn zwischen ihnen eine Kante verläuft. Ein Knoten und eine Kante heißen inzident, wenn der Knoten auf der Kante liegt. In ungerichteten Graphen ist der Grad $d(v)$ eines Knotens gleich der Zahl der Kanten, mit denen er inzidiert. In gerichteten Graphen ist der Ingrad $\text{ind}(v)$ (Outgrad $\text{outd}(v)$) eines Knotens gleich der Zahl der Kanten (\cdot, v) ((v, \cdot)). In jedem Fall setzen wir $n = |V|$ und $m = |E|$.

Direkte Nachfolger vom Knoten v sind die Knoten w , für die $(v, w) \in E$ ist. Ist w direkter Nachfolger von v , ist v direkter Vorgänger von w . Eine Folge v_0, \dots, v_k ist ein Weg der Länge k im Graphen G , falls $(v_{i-1}, v_i) \in E$ für $1 \leq i \leq k$ gilt. Mit unserer schlampigen Schreibweise für ungerichtete Graphen gilt diese Definition für gerichtete und ungerichtete Graphen. Für jeden Knoten v gibt es stets den Weg der Länge 0 von v nach v . Ein Weg heißt einfach, wenn höchstens Anfangs- und Endpunkt gleich sind. Ein einfacher Weg heißt Kreis, wenn Anfangs- und Endpunkt übereinstimmen, wobei wir Wege der Länge 0 und für ungerichtete Graphen Wege der Länge 2 ausschließen. Ein Graph heißt azyklisch, wenn er keinen Kreis enthält. Ein ungerichteter Graph heißt zusammenhängend, wenn es für jedes Knotenpaar (v, w) einen Weg zwischen v und w gibt. Ein gerichteter Graph heißt stark zusammenhängend, wenn es für jedes Knotenpaar (v, w) einen Weg von v nach w und einen Weg von w nach v gibt. Wir betrachten die folgenden Relationen auf den Knotenmengen von Graphen.

G ungerichtet: $v \approx w: \Leftrightarrow \exists$ Weg zwischen v und w .

G gerichtet: $v \approx w: \Leftrightarrow \exists$ Weg von v nach w und \exists Weg von w nach v .

Offensichtlich sind diese Relationen Äquivalenzrelationen, d. h., es gelten die folgenden Eigenschaften:

- (1) $\forall v \in V: v \approx v$.
- (2) $\forall v, w \in V: v \approx w \Rightarrow w \approx v$.
- (3) $\forall u, v, w \in V: u \approx v, v \approx w \Rightarrow u \approx w$.

Damit zerfällt V eindeutig in disjunkte Äquivalenzklassen von Knotenmengen, diese heißen in ungerichteten Graphen Zusammenhangskomponenten und in gerichteten Graphen stark zusammenhängende Komponenten.

Wir betrachten drei grundlegende Datenstrukturen für Graphen.

(1) Inzidenzmatrix: Es handelt sich um eine $n \times m$ -Matrix I . Es ist $I(i, j) \in \{0, 1\}$ und $I(i, j) = 1$ genau dann, wenn der i -te Knoten (bei gegebener Nummerierung) auf der j -ten Kante (bei gegebener Nummerierung) liegt. Da Kanten nur zwei Knoten enthalten,

haben wir es mit einer spärlich besetzten Matrix zu tun. Für gerichtete Graphen setzen wir $I(i, j) = 1$ ($I(i, j) = 2$), wenn der i -te Knoten Anfangsknoten (Endknoten) der j -ten Kante ist. Diese Datenstruktur erfordert recht großen Speicherplatz und wird nur in Spezialfällen benutzt.

(2) Adjazenzmatrix. Es handelt sich um eine $n \times n$ -Matrix A . Es ist $A(i, j) \in \{0, 1\}$ und $A(i, j) = 1$ genau dann, wenn es in gerichteten Graphen eine Kante vom i -ten zum j -ten Knoten und in ungerichteten Graphen eine Kante zwischen dem i -ten und dem j -ten Knoten gibt. Für ungerichtete Graphen ist A symmetrisch, und es genügt, die Elemente unterhalb der Hauptdiagonalen abzuspeichern. In jedem Fall ist der Speicherplatzbedarf $\Theta(n^2)$. Die Frage, ob zwei Knoten durch eine Kante verbunden sind, kann in Zeit $\Theta(1)$ beantwortet werden. Der Grad eines Knotens kann in Zeit $\Theta(n)$ berechnet werden. Allerdings benötigt jeder Algorithmus, der sich alle Kanten anschaut, Zeit $\Omega(n^2)$.

Für viele Graphen gilt $m = o(n^2)$. Es wäre dann schön, Algorithmen mit Laufzeit $O(n+m)$ zu haben.

(3) Adjazenzlisten. Es handelt sich um ein Array der Länge n für die Knoten. Für jeden Knoten existiert ein Zeiger auf eine Liste, die alle direkten Nachfolger des Knotens enthält. Der Speicherplatzbedarf beträgt $\Theta(n+m)$. Ob eine Kante (i, j) existiert (wir identifizieren im folgenden V mit $\{1, \dots, n\}$), kann nun allerdings nicht mehr in konstanter Zeit entschieden werden, die Zeit beträgt $\Theta(l_i)$, wenn l_i die Länge der i -ten Adjazenzliste ist. Die Knotengrade können nun schneller berechnet werden, $d(v)$ in Zeit $\Theta(d(v))$, $\text{outd}(v)$ ebenfalls in Zeit $\Theta(\text{outd}(v))$. Für die Berechnung von $\text{ind}(v)$ ist eine Zeit von $\Theta(m)$ erforderlich, wenn nicht auch Adjazenzlisten für eingehende Kanten vorliegen. Wenn die Knotengrade häufiger benötigt werden, sollten sie vorab berechnet und abgespeichert werden.

In Ruhe betrachtet sind die Adjazenzlisten genau die komprimierte Darstellung der spärlich besetzten Adjazenzmatrix (siehe Kapitel 2.2). Für Graphen mit vielen Kanten sind also Adjazenzmatrizen und für Graphen mit wenigen Kanten Adjazenzlisten vorzuziehen.

Viele, vielleicht sogar die meisten effizienten Graphalgorithmen greifen auf eine der beiden wichtigsten Methoden zum Traversieren von Graphen zurück: Depth-First-Search (DFS) und Breadth-First-Search (BFS). Die Tiefensuche (DFS) liefert dabei noch eine nützliche Klassifikation der Kanten und eine Nummerierung der Knoten.

Wir entwerfen zunächst einen DFS-Algorithmus für ungerichtete Graphen. Vom ersten Knoten suchen wir nach allen erreichbaren Knoten. Dabei wird der Weg zunächst soweit wie möglich in die Tiefe verfolgt, bevor ein Backtracking erfolgt. Den Knoten werden DFS-Nummern zugeordnet. Können keine weiteren Knoten gefunden werden, wird die Suche von einem bisher noch nicht besuchten Knoten neu gestartet. Die Kantenmenge wird dabei disjunkt in die Menge der Baumkanten (Treekanten) T und die Menge der Rückwärtskanten (Backkanten) B zerlegt. Die T -Kanten bilden einen Wald, also eine disjunkte Menge von Bäumen, wobei die Bäume die Zusammenhangskomponenten von G darstellen. Die Kanten in B und T werden gerichtet.

Algorithmus 2.6.1: DFS für ungerichtete Graphen $G = (V, E)$, gegeben durch Knotenarray und Adjazenzlisten $\text{Adj}(v)$.

- (1) Initialisierung: $i := 0$, $T := \emptyset$, $B := \emptyset$, für $x \in V$: $\text{num}(x) := 0$.
- (2) Für $x \in V$: *if* $\text{num}(x) = 0$ *then* $\text{DFS}(x, 0)$.

Es soll $\text{DFS}(x, v)$ die Tiefensuche von x aus beschreiben, wobei x von v aus erreicht wurde. Falls x der erste Knoten einer Zusammenhangskomponente ist, wird $v = 0$ gesetzt.

$\text{DFS}(v, u)$

- (1) $i := i + 1$, $\text{num}(v) := i$.
- (2) Für $w \in \text{Adj}(v)$:
if $\text{num}(w) = 0$ *then* $T := T \cup \{(v, w)\}$, $\text{DFS}(w, v)$
else if $\text{num}(w) < \text{num}(v)$ *and* $w \neq u$ *then* $B := B \cup \{(v, w)\}$.

Wir analysieren den DFS-Algorithmus. Für jeden Knoten v gibt es genau einen $\text{DFS}(v, \cdot)$ -Aufruf. Zeile (2) des Hauptprogramms sichert, dass es mindestens einen Aufruf gibt. Da $\text{DFS}(v, \cdot)$ nur aufgerufen wird, wenn $\text{num}(v) = 0$ ist, und in $\text{DFS}(v, \cdot)$ sofort $\text{num}(v)$ auf einen von Null verschiedenen Wert gesetzt wird, kann es nur einen $\text{DFS}(v, \cdot)$ -Aufruf geben. Da alle Adjazenzlisten einmal durchlaufen werden, ist die Laufzeit $\Theta(n + m)$. Jede Kante (v, w) wird zweimal betrachtet, da $v \in \text{Adj}(w)$ und $w \in \text{Adj}(v)$. Wir zeigen, dass jede Kante genau einen Eintrag in T oder B verursacht. Dazu untersuchen wir $\text{DFS}(w, u)$, wenn $v \in \text{Adj}(w)$ gefunden wird.

1.Fall: Es ist $\text{num}(v) = 0$. Dann wurde $\text{DFS}(v, \cdot)$ noch nicht aufgerufen, (w, v) wird T -Kante. Danach wird $\text{DFS}(v, w)$ aufgerufen. Nun gilt $0 \neq \text{num}(w) < \text{num}(v)$. Bei Behandlung von $w \in \text{Adj}(v)$ wird also (v, w) weder in T noch in B eingefügt.

2.Fall: Es ist $\text{num}(v) \neq 0$. Falls $u = v$, wurde (v, w) T -Kante. Ansonsten wird genau eine der Kanten (v, w) oder (w, v) B -Kante, je nachdem, welcher Knoten die kleinere DFS-Nummer hat.

Nachdem (\cdot, w) T -Kante wird, erhält w sofort seine DFS-Nummer. Also bildet $G^* = (V, T)$ einen Wald. Innerhalb von $\text{DFS}(v, 0)$ werden genau die mit v zusammenhängenden Knoten gefunden. Wir fassen unsere Ergebnisse zusammen.

Satz 2.6.2: Der DFS-Algorithmus erzeugt für ungerichtete Graphen $G(V, E)$ in $\Theta(n + m)$ Schritten eine Nummerierung der Knoten und eine Einteilung der Kanten in die disjunkten Kantenmengen T und B . Der Graph $G^* = (V, T)$ ist ein Wald, dessen Bäume den Zusammenhangskomponenten von G entsprechen. Zwischen zwei $\text{DFS}(\cdot, 0)$ -Aufrufen wird jeweils eine Zusammenhangskomponente von G nummeriert.

Satz 2.6.3: Ungeriichte Graphen können in Zeit $\Theta(n + m)$ darauf getestet werden, ob sie einen Kreis enthalten.

Beweis: Wir benutzen den DFS-Algorithmus. Falls $B = \emptyset$ ist, ist G ein Wald und damit kreisfrei. Kanten können nicht zwischen zwei Bäumen des T -Waldes verlaufen. Jede B -Kante schließt also einen Kreis in einem T -Baum. \square

Für gerichtete Graphen verläuft der DFS-Algorithmus ähnlich. Nun wird jede Kante nur noch einmal betrachtet. Falls $(v, w) \in E$ und $(w, v) \notin E$, ist $w \in \text{Adj}(v)$ und $v \notin \text{Adj}(w)$. Der Algorithmus benutzt für jeden Knoten v den Hilfsparameter $\alpha(v)$, der mit 0 initialisiert wird. Dieser Wert wird beibehalten, bis $\text{DFS}(v, \cdot)$ aufgerufen wird. Während der Abarbeitung von $\text{DFS}(v, \cdot)$ ist $\alpha(v) = 1$ und danach ist $\alpha(v) = 2$. Die Kantenmenge wird nun disjunkt eingeteilt in die Menge der Baumkanten (Treekanten) T , die Menge der Vorwärtskanten (Forwardkanten) F , die Menge der Rückwärtskanten (Backkanten) B und die Menge der Querverbindungskanten (Crosskanten) C . Es werde die Kante (v, w) betrachtet.

- (1) $\text{num}(w) = 0$. Dann wird (v, w) T -Kante.
- (2) $\text{num}(w) \neq 0$, $\text{num}(w) > \text{num}(v)$. Dann wird (v, w) F -Kante. Nach Voraussetzung $\text{num}(w) > \text{num}(v)$ wurde $\text{DFS}(w, \cdot)$ später als $\text{DFS}(v, \cdot)$ aufgerufen, aber $\text{DFS}(v, \cdot)$ wurde noch nicht beendet. Daher gibt es einen T -Weg von v zu w . Die Kante (v, w) ist im T -Baum vorwärts gerichtet.
- (3) $\text{num}(w) \neq 0$, $\text{num}(w) < \text{num}(v)$, $\alpha(w) = 1$. Dann wird (v, w) B -Kante. Da $\text{DFS}(w, \cdot)$ noch nicht beendet und $\text{num}(w) < \text{num}(v)$ ist, wurde v innerhalb von $\text{DFS}(w, \cdot)$ erstmals erreicht. Daher gibt es einen T -Weg von w zu v . Die Kante (v, w) ist im T -Baum rückwärts gerichtet.
- (4) $\text{num}(w) \neq 0$, $\text{num}(w) < \text{num}(v)$, $\alpha(w) = 2$. Dann wird (v, w) C -Kante. $\text{DFS}(w, \cdot)$ ist bereits abgearbeitet, während $\text{DFS}(v, \cdot)$ noch aktiv ist. Also gibt es keinen Weg von w zu v . Außerdem gibt es keinen T -Weg von v zu w . In T stehen v und w also in keiner Beziehung. Die Kante (v, w) führt entweder von einem Baum zu einem früher konstruierten Baum oder innerhalb eines Baumes zu einem früher konstruierten Teilbaum.

Satz 2.6.4: Der DFS-Algorithmus erzeugt für gerichtete Graphen $G = (V, E)$ in $\Theta(n+m)$ Schritten eine Nummerierung der Knoten und die Einteilung der Kanten in die disjunkten Mengen T , F , C und B .

Beispiel 2.6.5: $G = (V, E)$ sei durch Adjazenzlisten gegeben.

$a \rightarrow c; b \rightarrow h, i; c \rightarrow d, e, g; d \rightarrow a, g; e \rightarrow g; f \rightarrow c, e, h; g \rightarrow d; h \rightarrow a, i; i \rightarrow f$.

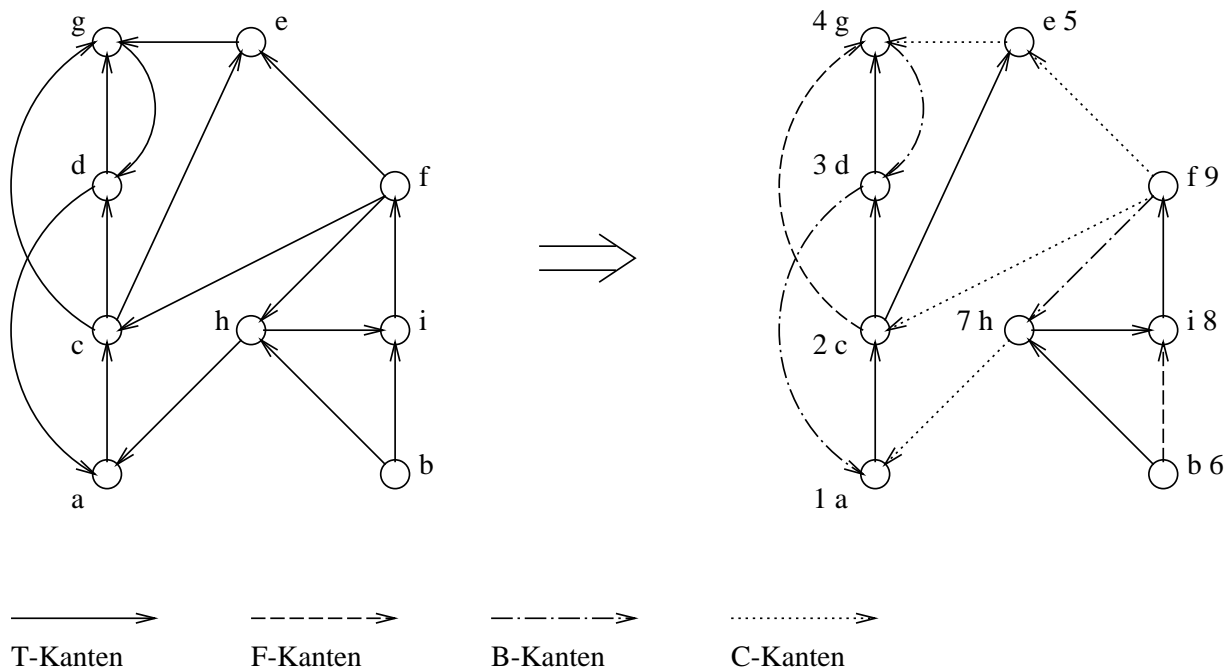


Abbildung 2.6.1: Ein Beispiel für den DFS-Algorithmus auf gerichteten Graphen

Es ist nützlich sich zu überlegen, wie eine ungerichtete Kante $\{v, w\}$ bzw. eine gerichtete Kante (v, w) im DFS-Algorithmus bearbeitet wird. Wir legen das Zeitintervall des $\text{DFS}(v, \cdot)$ -Aufrufs und den Zeitpunkt der Verarbeitung der Kante $\{v, w\}$ bzw. (v, w) durch den senkrechten Strich in Abbildung 2.6.2 fest. Aufgrund der rekursiven Struktur des DFS kann $\text{DFS}(w, \cdot)$ nur vor, nach oder innerhalb des $\text{DFS}(v, \cdot)$ -Aufrufs stattfinden oder den $\text{DFS}(v, \cdot)$ -Aufruf zeitlich umfassen.

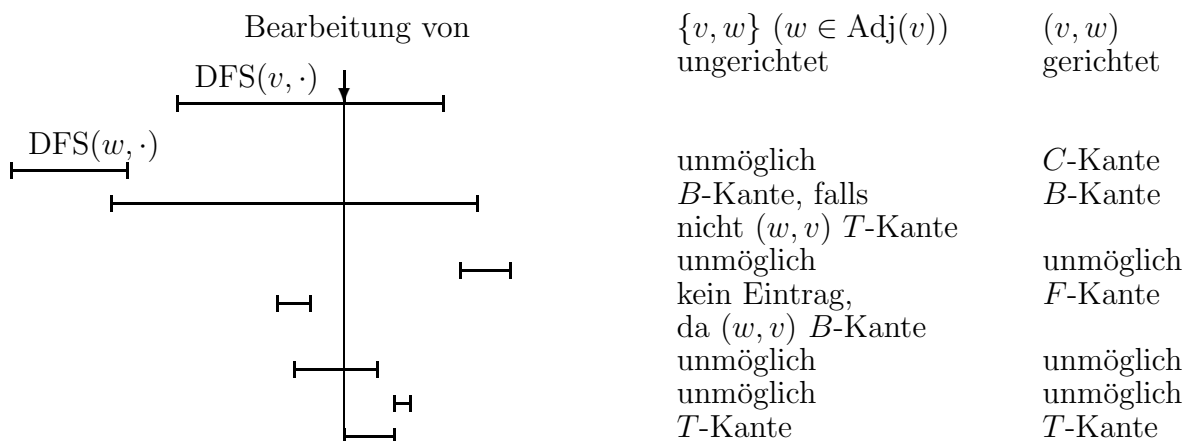


Abbildung 2.6.2: Illustration des zeitlichen Ablaufs des DFS-Algorithmus.

Im Gegensatz zur Tiefensuche wird beim BFS-Ansatz zunächst in die Breite gesucht. Beim

DFS wird (implizit durch die Rekursion) ein Stack benutzt, beim BFS eine Queue. Bei der Behandlung des Knotens v wird zunächst $\text{Adj}(v)$ vollständig durchlaufen. Dabei werden die noch nicht zuvor gefundenen Knoten in die Queue eingefügt. Die Knoten werden gemäß der Reihenfolge in der Queue abgearbeitet. Auch hier ist die Laufzeit $\Theta(n+m)$. Es wird keine Kanteneinteilung erzeugt. Der Vorteil des BFS-Ansatzes besteht darin, dass die Knoten in der Reihenfolge ihres Abstandes von v erreicht werden. An Stelle einer expliziten Beschreibung des BFS-Algorithmus, wenden wir die Breitensuche auf den Graphen aus Beispiel 2.6.5 an.

Beispiel 2.6.6: Wir betrachten den Graphen aus Beispiel 2.6.5.

$\text{BFS}(a) = 1$, $Q = (c)$, Q steht für die queue.

$\text{BFS}(c) = 2$, $Q = (d, e, g)$.

$\text{BFS}(d) = 3$, $Q = (e, g)$. Der Knoten a ist bereits besucht und wird daher nicht in Q aufgenommen.

$\text{BFS}(e) = 4$, $Q = (g)$. $\text{BFS}(g) = 5$, $Q = \emptyset$.

Neuer Start in Knoten b .

$\text{BFS}(b) = 6$, $Q = (h, i)$. $\text{BFS}(h) = 7$, $Q = (i)$.

$\text{BFS}(i) = 8$, $Q = (f)$. $\text{BFS}(f) = 9$, $Q = \emptyset$.

Alle Knoten sind besucht. Die Knoten haben die folgenden Abstände von a : $c \rightarrow 1$, $d \rightarrow 2$, $e \rightarrow 2$, $g \rightarrow 2$, b, h, i und f sind von a aus nicht erreichbar.

Zu Beginn dieses Abschnitts haben wir Graphen als Datenstrukturen vorgestellt, die binäre Beziehungen ausdrücken. Wenn auch Beziehungen zwischen größeren Knotenmengen beschrieben werden sollen, werden Graphen zu Hypergraphen verallgemeinert. Eine Hyperkante ist eine Teilmenge V' der Knotenmenge V und drückt eine gemeinsame Beziehung zwischen allen Knoten in V' aus. Wir wollen hier Hypergraphen nicht weiter betrachten.

2.7 Datenstrukturen für Intervalle

Wir wollen eine Datenstruktur für folgende Problemstellung entwerfen. Für $i \in \{1, \dots, n\}$ gibt es einen Zahlenwert $w(i)$. Ausgehend von diesen Werten soll in der Preprocessing Phase eine Datenstruktur aufgebaut werden, so dass für Intervalle $[i, j] = \{i, \dots, j\}$ der Wert $w(i, j) := w(i) + \dots + w(j)$ schnell berechnet werden kann (Query Zeit). In manchen Anwendungen (siehe Kapitel 5) ist es auch nötig, einen Wert $w(i)$ zu ändern (Update).

Wir können die $w(i)$ -Werte in einem Array abspeichern, was einen geringen Speicherplatzbedarf von n bedeutet. Die Berechnung von $w(i, j)$ ist dann in Zeit $\Theta(j - i + 1)$ möglich. Dies führt für große Intervalle zu einer linearen Rechenzeit. Der Wert von $w(i)$ kann in konstanter Zeit geändert werden.

Andererseits können wir vorab eine obere Dreiecksmatrix bilden, die an Stelle (i, j) den Wert $w(i, j)$ enthält. Wie in Kapitel 2.2 diskutiert, kann diese Matrix in einem Array der Länge $\Theta(n^2)$ abgespeichert werden. Dieser große Speicherplatzbedarf ermöglicht offensichtlich eine schnelle Zugriffszeit von $\Theta(1)$ auf $w(i, j)$. Wenn wir den Wert von $w(\lfloor n/2 \rfloor)$ ändern wollen, müssen die Werte für $\Theta(n^2)$ Intervalle geändert werden.

Beide naiven Datenstrukturen haben gravierende Nachteile. Wir suchen nach einer Datenstruktur, die weit weniger als quadratischen Speicherplatz und weit weniger als lineare Zugriffszeit erfordert. Segmentbäume (segment trees) bieten einen vernünftigen Kompromiss.

An der Wurzel des Segmentbaumes repräsentieren wir das Intervall $[1, n]$. Ein Knoten $[l, r]$ wird Blatt, wenn $l = r$ ist. Ansonsten erhält er zwei Kinder, die $[l, \lfloor (l + r)/2 \rfloor]$ und $[\lfloor (l + r)/2 \rfloor + 1, r]$ repräsentieren. Am Knoten $[l, r]$ wird $w(l, r)$ abgespeichert. Abbildung 2.7.1 zeigt einen Segmentbaum für $n = 20$.

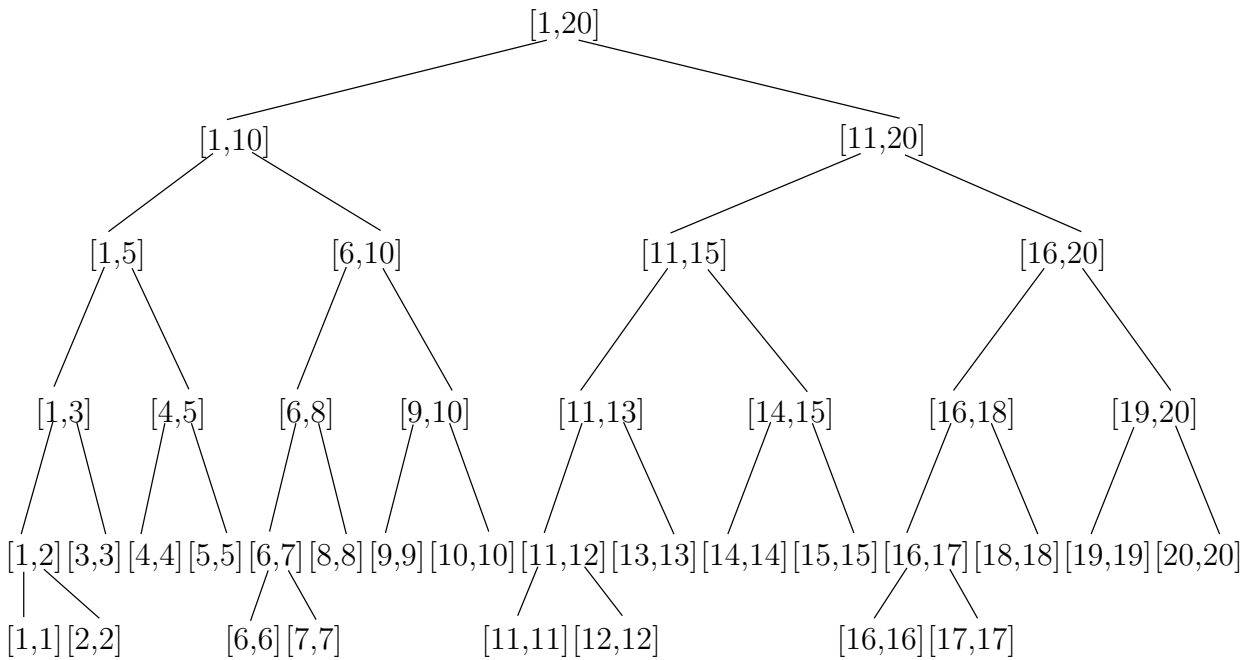


Abbildung 2.7.1: Der Segmentbaum für $n = 20$.

Da wir die Intervalle stets halbieren, beträgt die Tiefe des Segmentbaums $\lceil \log n \rceil$. Die Änderung von $w(i)$ führt zur Änderung der maximal $\lceil \log n \rceil + 1$ Werte auf dem Weg von der Wurzel zum Blatt $[i, i]$. Wir wollen nun ein vorgegebenes Intervalle $[i, j]$ in möglichst wenige disjunkte Intervalle zerlegen, die im Segmentbaum vorhanden sind und effizient aufgesucht werden können. Wenn wir diese Intervalle aufgesucht haben, können wir aus den zugehörigen Informationen $w(i, j)$ berechnen. Wenn wir im Segmentbaum auf das Intervall $[l, r]$, zu Beginn $[l, r] = [1, n]$, stoßen, gibt es vier Möglichkeiten.

- (1) $i \leq l \leq r \leq j$.
Die Information über $[l, r]$ wird benutzt. STOP.
- (2) $j \leq \lfloor (l + r)/2 \rfloor$.
Es genügt, im linken Teilbaum weiter zu suchen.
- (3) $i > \lfloor (l + r)/2 \rfloor$.
Es genügt, im rechten Teilbaum weiter zu suchen.

(4) $i \leq \lfloor (l+r)/2 \rfloor$, $j > \lfloor (l+r)/2 \rfloor$ und nicht (1).

Wir haben einen Gabelungspunkt erreicht und müssen in beiden Teilbäumen weiter suchen.

Zunächst deutet nichts auf die Zeiteffizienz dieses Verfahrens hin. Warum sollten wir nicht viele Gabelungspunkte finden und daher gezwungen sein, fast den ganzen Baum zu durchsuchen? Wenn es keinen Gabelungspunkt gibt, dann suchen wir nur höchstens $\lceil \log n \rceil + 1$ Knoten im Segmentbaum auf. Ansonsten ist der Weg bis zum ersten Gabelungspunkt $[l, r]$ eindeutig. Im linken Teilbaum erreichen wir den Knoten $[l, \lfloor (l+r)/2 \rfloor]$, und wir wissen, dass $j > \lfloor (l+r)/2 \rfloor$ ist. Unser Suchintervall $[i, j]$ ragt also rechts über alle im linken Teilbaum abgespeicherten Intervalle hinaus. Wenn wir in diesem Teilbaum den Knoten $[l', r']$ aufsuchen, gilt $j > r'$. Wenn wir für diesen Knoten auch im linken Teilbaum suchen, gilt $i \leq \lfloor (l' + r')/2 \rfloor$, d.h. $[i, j]$ überdeckt das rechte Kind $[\lfloor (l' + r')/2 \rfloor + 1, r']$. Der Knoten $[l', r']$ ist also kein echter Gabelungspunkt, da das rechte Kind zum Fall (1) gehört und direkt zur Ausgabe beiträgt. Insgesamt suchen wir also weniger als $4\lceil \log n \rceil$ Knoten auf, von denen weniger als $2\lceil \log n \rceil$ zur Lösung beitragen.

Diese Betrachtungen werden in Abbildung 2.7.2 veranschaulicht.

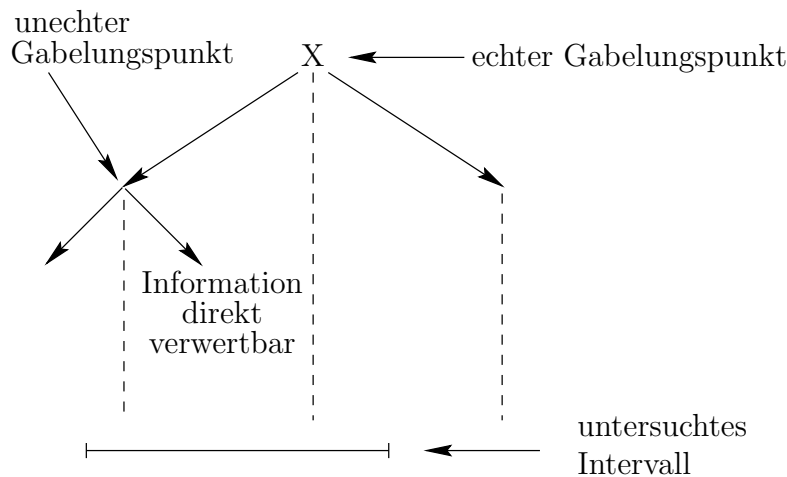


Abbildung 2.7.2: Echte und unechte Gabelungspunkte bei einer Abfrage in einem Segmentbaum.

Der Segmentbaum kann leicht in Zeit $\Theta(n)$ aufgebaut werden und, da er $2n - 1$ Knoten hat, auf Platz $\Theta(n)$ abgespeichert werden. Wir fassen unsere Ergebnisse zusammen.

Satz 2.7.1: Segmentbäume auf $1, \dots, n$ können in Zeit $\Theta(n)$ aufgebaut werden (Preprocessing Time) und haben $\Theta(n)$ Platzbedarf. Die Information über ein Intervall kann in Zeit $O(\log n)$ berechnet werden (Query Time).

Es ist zu beachten, dass die Zeit für das Preprocessing nur einmal anfällt und dann sehr viele Fragen beantwortet werden können.

Auch hier fassen wir die Resultate zusammen.

	alle $w(i)$	alle Intervalle	Segmentbaum
Speicherplatz	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$
Preprocessing	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$
Update $w(i)$	$\Theta(1)$	$O(n^2)$	$\Theta(\log n)$
Query $w(i, j)$	$O(n)$	$\Theta(1)$	$O(\log n)$

Tabelle 2.7.1: Rechenzeiten für die verschiedenen Datenstrukturen.

Wie sehr oft hat die Datenstruktur, die den besten Kompromiss darstellt, für keine Operation die beste Rechenzeit.

2.8 Datenstrukturen für Partitionen

Es seien n Objekte, o. B. d. A. mit $1, \dots, n$ bezeichnet, gegeben und zu Beginn in n elementigen Mengen, ebenfalls o. B. d. A. mit $1, \dots, n$ bezeichnet, organisiert. Folgende Operationen sollen unterstützt werden.

FIND(x), $1 \leq x \leq n$: Bestimme den Namen der Menge, in der sich x befindet.

UNION(A, B), wobei A und B aktuelle Mengennamen sind: Vereinige die Mengen A und B . Der Name für die neue Menge kann beliebig gewählt werden, ohne dass zwei Mengen denselben Namen erhalten. Da jedes Objekt nur in einer Menge sein darf, werden A und B bei ihrer Vereinigung vernichtet. Die neue Menge darf also auch mit A oder B bezeichnet werden.

Da jedes Objekt stets nur in einer Menge enthalten ist, bilden die Mengen jeweils eine Partition von $\{1, \dots, n\}$ und wir haben es mit Datenstrukturen für Partitionen zu tun, die die Befehle FIND und UNION unterstützen.

FIND-Befehle können zu verschiedenen Zeitpunkten verschiedene Antworten haben. Da jeder UNION-Befehl die Zahl aktueller Mengen um 1 verringert, kann eine Befehlsfolge nur $n - 1$ UNION-Befehle enthalten. Dann sind alle Objekte in einer Menge zusammengefasst. Damit die Mengennamen in einem Array verwaltet werden können, sollten als Mengennamen nur Elemente aus $\{1, \dots, n\}$ benutzt werden. In diesem Kapitel stellen wir Datenstrukturen vor, die UNION- und FIND-Befehle unterstützen. Eine Anwendung wird in Kapitel 5 vorgestellt.

Wir diskutieren zunächst zwei triviale Datenstrukturen. Wir können ein Array der Objekte anlegen und für jedes Objekt den Mengennamen verwalten. FIND-Befehle kosten Zeit $\Theta(1)$, UNION-Befehle Zeit $\Theta(n)$, da wir alle Elemente einer Menge nur finden können, indem wir das gesamte Array durchlaufen. Wir können auch ein Array der Mengen verwalten und dazu Listen mit den dazugehörigen Objekten und jeweils einem Extrazeiger auf das Listenende. UNION-Befehle kosten Zeit $\Theta(1)$, aber FIND-Befehle Zeit $O(n)$. Wir stellen zwei Datenstrukturen, die beide Befehlstypen unterstützen, vor.

Die erste Datenstruktur unterstützt besonders FIND-Befehle. Wir benutzen ein Array R der Länge n , wobei $R(i)$ den Namen der Menge enthält, in der i liegt. Wenn wir uns für die UNION-Befehle stets alle Elemente der zu vereinigenden Mengen ansehen, lassen sich

für $n - 1$ UNION-Befehle Kosten von $\Theta(n^2)$ im worst case nicht vermeiden. Daher werden wir für jede aktuelle Menge A eine Liste $L(A)$ der Elemente in A verwalten. Außerdem soll $s(A)$ die Größe von A beschreiben. Wir wollen nun als neuen Namen für die Vereinigung von A und B den Namen der größeren der beiden Mengen benutzen. Dann muss R nur für die Elemente der kleineren Menge geändert werden.

Die Initialisierung der Datenstruktur ist in Zeit $\Theta(n)$ möglich. Das Ergebnis von $\text{FIND}(x)$ ist einfach $R(x)$. Damit kostet ein FIND -Befehl nur konstante Zeit. UNION-Befehle werden mit folgendem Algorithmus ausgeführt.

Algorithmus 2.8.1: $\text{UNION}(A, B)$

- (1) Teste, ob $s(A) \leq s(B)$ ist. Falls ja, gehe zu (2) sonst zu (3).
- (2) Durchlaufe $L(A)$, für $x \in L(A)$ setze $R(x) := B$ und hänge $L(B)$ an das Ende von $L(A)$. Die neue Liste erhält den Namen $L(B)$. Setze $s(B) := s(A) + s(B)$.
- (3) Analog zu (2) mit vertauschten Rollen von A und B .

Offensichtlich beträgt die Laufzeit von Algorithmus 2.8.1 $\Theta(\min\{s(A), s(B)\})$.

Satz 2.8.2: Mit Hilfe der listenorientierten Datenstruktur kann jede Folge von $n - 1$ UNION- und m FIND-Befehlen in Zeit $O(n \log n + m)$ ausgeführt werden.

Beweis: Nach unseren Vorüberlegungen genügt es zu zeigen, dass die UNION-Befehle in Zeit $O(n \log n)$ ausführbar sind.

Es seien A_1, \dots, A_{n-1} die kleineren Mengen bei den UNION-Befehlen. Die Rechenzeit beträgt dann $O(s(A_1) + \dots + s(A_{n-1}))$. Die Abschätzung dieser Summe ist schwierig. Zwar wissen wir, dass $s(A_1) = 1$ ist, andererseits kann $s(A_{n-1}) = \lfloor n/2 \rfloor$ sein. Wir können uns für jedes $i \geq n/2$ Befehlsfolgen überlegen, so dass $s(A_i) \geq \lfloor n/4 \rfloor$ ist. Andererseits können nicht alle $s(A_i), i \geq n/2$, mindestens $\lfloor n/4 \rfloor$ sein. Dies ist eine nicht untypische Situation. Wir können jedes $s(A_i)$ durch seinen Maximalwert abschätzen und erhalten dann die richtige Schranke $O(n^2)$, die aber weit von der wahren Rechenzeit entfernt ist. Man spricht von der amortisierten Rechenzeit, wenn man die Rechenzeit einer Befehlsfolge betrachtet. Diese kann viel kleiner als das Produkt aus der Anzahl der Befehle und der worst case Rechenzeit aller Befehle und sogar viel kleiner als die Summe der worst case Rechenzeiten für die einzelnen Befehle sein. Wie aber können wir dann die amortisierte Rechenzeit abschätzen?

Ein wesentlicher Trick besteht in der so genannten Buchhaltermethode, mit der wir die entstehenden Kosten, also die Rechenzeiten, auf andere Kostenträger so umbuchen, dass die Summe der Kosten hinterher einfacher zu berechnen ist. Als neue Kostenträger wählen wir hier die Objekte $1, \dots, n$. Für den Summanden $s(A_i)$ werden die Kosten so auf die Objekte aus A_i umbucht, dass jedes $x \in A_i$ eine Kosteneinheit erhält. Wieviele Kosteneinheiten kann x erhalten? Das Objekt x startet in einer Menge der Größe 2^0 . Immer wenn x eine Kosteneinheit erhält, verdoppelt sich die Größe der Menge, die x enthält, mindestens. Dies liegt daran, dass die A_i -Mengen die kleineren (genauer: nicht größeren)

bei den UNION-Befehlen sind. Erhält x insgesamt $a(x)$ Kosteneinheiten, ist x am Ende in einer mindestens $2^{a(x)}$ -elementigen Menge. Also ist $2^{a(x)} \leq n$ und $a(x) \leq \lfloor \log n \rfloor$. Damit folgt $s(A_1) + \dots + s(A_n) \leq n \lfloor \log n \rfloor$ und die Behauptung. \square

Es lässt sich leicht zeigen, dass die im Beweis von Satz 2.8.2 gezeigte obere Schranke bis auf einen Faktor $1/2$ optimal ist. Dazu betrachte man einen balancierten Baum mit n Blättern und eine Folge von UNION-Befehlen, die die Mengen gemäß des Baumes von den Blättern zur Wurzel vornimmt.

Die zweite Datenstruktur unterstützt besonders UNION-Befehle. Dazu werden die Mengen durch Bäume dargestellt. Jeder Knoten repräsentiert ein Element der Menge, an der Wurzel steht der Name der Menge und die Größe der Menge. Es wird die erste Datenstruktur für Bäume verwendet. Die Knoten stehen in einem Array und zu jedem Knoten gehört ein Zeiger auf seinen Elter. Auf die Elemente kann also direkt zugegriffen werden, ebenso auf die Mengen, genauer auf die Wurzeln der Bäume, die die Mengen darstellen.

UNION(A, B): Greife auf die Wurzeln der Bäume, die A und B darstellen, zu. Bestimme die kleinere Menge, o.B.d.A. sei dies A . Die Wurzel von A erhält nun die Wurzel von B als Elter. Der Name der neuen Menge ist B und es wird $s(B)$ durch $s(A) + s(B)$ ersetzt. UNION-Befehle sind also in konstanter Zeit durchführbar. Indem wir die kleineren Mengen in die größeren Mengen einhängen, versuchen wir zu erreichen, dass keine Bäume mit langen Wegen entstehen.

FIND(x): Starte in x und laufe den eindeutigen Weg von x zur Wurzel des Baumes, in dem x liegt. Dort steht der Name der Menge, zu der x gehört.

Lemma 2.8.3: Bei der baumorientierten Datenstruktur haben die Bäume durch $\lfloor \log n \rfloor$ beschränkte Tiefe. FIND-Befehle können also in Zeit $O(\log n)$ durchgeführt werden.

Beweis: Wir zeigen durch Induktion über die Tiefe d , dass jeder erzeugte Baum, dessen Tiefe mindestens d beträgt, mindestens 2^d Knoten enthält. Daraus folgt sofort die Behauptung.

Für $d = 0$ ist die Behauptung erfüllt. Sei nun T ein möglicher Baum, der unter allen Bäumen mit Mindesttiefe d die kleinste Knotenzahl hat, wobei $d \geq 1$ ist. T entstand dadurch, dass T_1 in T_2 eingehängt wurde. Also ist $s(T_1) \leq s(T_2) < s(T)$. Nach Wahl von T ist die Tiefe von T_1 höchstens $d - 1$. Wäre die Tiefe kleiner als $d - 1$, müsste T_2 bereits Tiefe d haben, damit T Tiefe d hat. Dies und $s(T_2) < s(T)$ steht im Widerspruch zur Konstruktion von T . Also hat T_1 Tiefe $d - 1$ und damit mindestens 2^{d-1} Knoten. Es folgt $s(T) = s(T_1) + s(T_2) \geq 2s(T_1) \geq 2^d$. \square

Bemerkenswert ist der Ansatz dieses Beweises. Wir suchen eigentlich unter den entstandenen Bäumen mit n Knoten einen mit maximaler Tiefe. Statt dessen suchen wir unter allen entstehenden Bäumen mit Tiefe d einen mit minimaler Knotenzahl und errechnen aus dem Ergebnis das eigentlich gewünschte Ergebnis. Wir erhalten nun für $n - 1$ UNION-Befehle und m FIND-Befehle die Schranke $O(n + m \log n)$ für die Rechenzeit.

In vielen Fällen ist m größer als n . Dann verschenken wir die bei der Ausführung von FIND-Befehlen gewonnene Information. Mit Hilfe der Pfadkomprimierung (path compression) speichern wir die gewonnene Information.

Während $\text{FIND}(x)$ speichern wir den gesamten Suchpfad ab und lassen, nachdem wir die Wurzel des Baumes gefunden haben, alle Knoten auf dem Suchpfad auf diese Wurzel zeigen. Die Kosten des gerade ausgeführten FIND-Befehls wachsen um einen konstanten Faktor, spätere FIND-Befehle können viel effizienter ausgeführt werden. Je teurer ein FIND-Befehl ist, desto größer ist auch die potenzielle zukünftige Einsparung. Wir stellen die Technik der Pfadkomprimierung in Abbildung 2.8.1 dar.

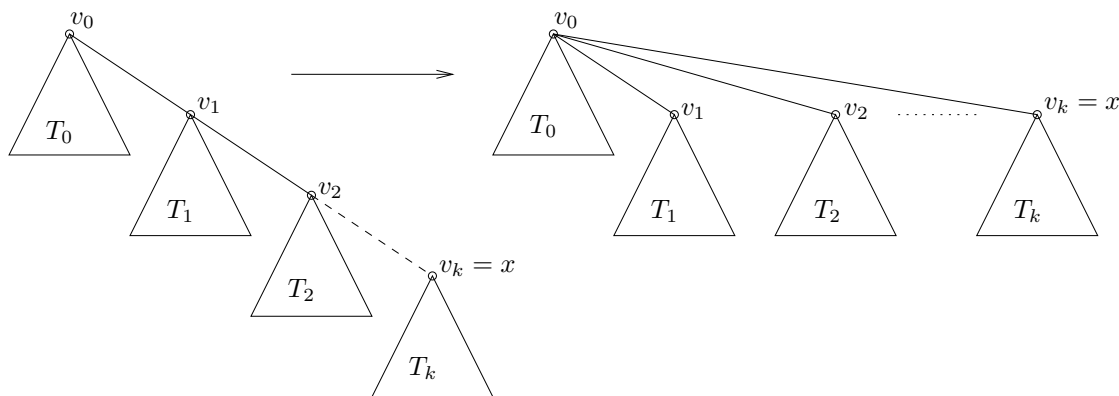


Abbildung 2.8.1: Die Technik der Pfadkomprimierung.

Die Abschätzung der Kosten für m FIND-Befehle bleibt späteren Vorlesungen vorbehalten. Wir stellen nur das Ergebnis vor.

Definition 2.8.4: Die Funktion Zweierturm Z ist rekursiv definiert durch

$$Z(0) := 1 \text{ und } Z(i) := 2^{Z(i-1)},$$

dazu sei $\log^* n := \min\{k | Z(k) \geq n\}$.

Wir können $\log^* n$ folgendermaßen interpretieren. Es ist die kleinste Anzahl von Anwendungen von \log auf n , so dass eine Zahl entsteht, die nicht größer als 1 ist. Es gilt

$$Z(1) = 2, \quad Z(2) = 4, \quad Z(3) = 16, \quad Z(4) = 65.536, \quad Z(5) = 2^{65.536}.$$

Damit ist $\log^* n \leq 5$ für $n \leq 2^{65.536}$.

Satz 2.8.5: Mit Hilfe der baumorientierten Datenstruktur und der Technik der Pfadkomprimierung kann jede Folge von $n - 1$ UNION- und m FIND-Befehlen in Zeit $O((n + m) \log^* n)$ ausgeführt werden.

2.9 Datenstrukturen für boolesche Funktionen

Bisher haben wir so einfache Objekte wie Mengen, Bäume, Graphen, Intervalle und Partitionen behandelt. Wir kommen nun zu Funktionen mit endlichem Definitions- und Bildbereich, also nach entsprechender Codierung zu booleschen Funktionen $f: \{0, 1\}^n \rightarrow \{0, 1\}^m$, die wir in die Koordinatenabbildungen f_1, \dots, f_m mit $f = (f_1, \dots, f_m)$ zerlegen können. Eine boolesche Funktion mit einem Output, also $f: \{0, 1\}^n \rightarrow \{0, 1\}$ kann wiederum mit der Menge $f^{-1}(1)$ identifiziert werden. Also haben wir es wieder nur mit Mengen zu tun?! Ja und dennoch ist die Situation eine ganz andere. Bisher haben wir Mengen behandelt, bei denen es denkbar ist, die Mengen durch Aufzählung ihrer Elemente zu beschreiben. Die Menge $f^{-1}(1)$ ist jedoch eine Teilmenge der 2^n -elementigen Menge $\{0, 1\}^n$ und hat typischerweise zu viele Elemente, um sie aufzuzählen. Hier müssen wir auch feststellen, dass es 2^{2^n} boolesche Funktionen $f: \{0, 1\}^n \rightarrow \{0, 1\}$ gibt und jede Darstellung im Durchschnitt mindestens 2^n Bits braucht. Also können wir nur auf Datenstrukturen hoffen, die für viele (wichtige) Funktionen eine kurze Darstellung haben und zusätzlich wichtige Operationen auf booleschen Funktionen unterstützen.

Als Anwendungsbeispiel sei die Hardwareverifikation genannt. Ein neuer Prozessor verwendet einen neuen Dividierer und es soll verifiziert werden, dass er für alle Eingaben (in der Hardware haben die Eingaben feste Länge) dieselben Ergebnisse liefert wie der alte Prozessor. Wir kommen auf dieses Beispiel später zurück.

Da wir Funktionen stets durch Graphen mit Zusatzinformationen darstellen, soll G_f die Darstellung von f bezüglich einer Datenstruktur bezeichnen. Die folgende Liste von Operationen soll unterstützt werden.

- 1.) Auswertung: Für G_f und $a \in \{0, 1\}^n$ soll $f(a)$ berechnet werden. Es ist wohl nicht nötig, diese Operation zu motivieren.
- 2.) Gleichheitstest: Für G_f und G_g soll entschieden werden, ob $f(a) = g(a)$ für alle $a \in \{0, 1\}^n$ gilt. Dies ist durch das Verifikationsbeispiel motiviert.
- 3.) Synthese: Für G_f und G_g und eine binäre boolesche Operation \otimes wie AND, OR, oder EXOR soll G_h , eine Darstellung von $h := f \otimes g$, berechnet werden. Es ist im Hardwareentwurf üblich, komplexere Funktionen aus einfacheren zusammenzusetzen. Dieser Prozess soll nachgeahmt werden. So kann eine Hardwarebeschreibung ausgehend von den Eingabevariablen Stück für Stück in eine Darstellung gemäß der gewählten Datenstruktur überführt werden.
- 4.) Erfüllbarkeitstest: Für G_f soll entschieden werden, ob es eine Eingabe $a \in \{0, 1\}^n$ mit $f(a) = 1$ gibt. Dies ist motiviert, da Synthese und Erfüllbarkeitstest einen Gleichheitstest ermöglichen. Offensichtlich sind f und g gleich, wenn $f \oplus g$ ($\oplus = \text{EXOR}$) nicht erfüllbar ist.
- 5.) Ersetzung durch Funktionen: Für G_f, G_g und eine Variable $x_i, 1 \leq i \leq n$, soll eine Darstellung G_h für $h := f|_{x_i=g}$, definiert durch

$$h(a_1, \dots, a_n) := f(a_1, \dots, a_{i-1}, g(a_1, \dots, a_n), a_{i+1}, \dots, a_n),$$

berechnet werden. Häufig wird eine Variable als „Stellvertreter“ für die Ausgabe eines anderen Hardwaremoduls verwendet und muss schließlich durch die Funktion, für die sie stellvertretend steht, ersetzt werden. Im Spezialfall, dass g eine Konstante Funktion ist, sprechen wir von der Ersetzung durch Konstanten. Dieser Spezialfall ist wichtig, da der allgemeinere Fall auf den Spezialfall und die Synthese zurückgeführt werden kann. Es gilt nämlich mit der Shannon-Zerlegung für boolesche Funktionen

$$f_{|x_i=g} = (g \wedge f_{|x_i=1}) + (\bar{g} \wedge f_{|x_i=0}).$$

6.) Abstraktion oder Quantifizierung: Für G_f und eine Variable $x_i, 1 \leq i \leq n$, soll

$$(\exists x_i)f := f_{|x_i=0} + f_{|x_i=1}$$

oder

$$(\forall x_i)f := f_{|x_i=0} \wedge f_{|x_i=1}$$

berechnet werden. Wenn die Spezifikation eines Hardwaremoduls gewisse Ausgaben verbietet (z. B. die Ausgabe 0, falls die Ausgabe als Divisor in einer folgenden Rechnung auftritt), muss nachgewiesen werden, dass für alle Eingaben bestimmte Ausgaben nicht erzeugt werden. Hier ist schon nach Definition klar, dass diese Operationen auf die Ersetzung durch Konstanten und Synthese zurückgeführt werden können.

Eine der kompaktesten Darstellungen boolescher Funktionen bilden Schaltkreise. Sie werden durch Bausteinlisten (Gatterlisten, gate lists, net lists) beschrieben. Dies ist eine Aufzählung B_1, \dots, B_m von Bausteinen, wobei jeder Baustein B_i ein Tripel aus der anzuwendenden binären booleschen Funktion und den beiden Vorgängern ist. Als Vorgänger von B_i sind die Bausteine B_1, \dots, B_{i-1} , die Variablen x_1, \dots, x_n und die Konstanten 0 und 1 erlaubt. Die an B_i dargestellte Funktion ergibt sich durch Auswertung der zugehörigen Funktion auf die an den Vorgängern dargestellten Funktionen. Schaltkreise sind gerichtete azyklische Graphen mit Knoten für $0, 1, x_1, \dots, x_n, B_1, \dots, B_m$ und Kanten von den Vorgängern auf den jeweiligen Baustein. Innerhalb der Datenstruktur Schaltkreis haben die Objekte, also die booleschen Funktionen, keine eindeutige Darstellung. Dies macht den Gleichheitstest ja erst zu einer spannenden Aufgabe. Gleichzeitig führt das zu einer weiteren wichtigen Operation.

7.) Minimierung: Für G_f finde eine Darstellung minimaler Größe für f innerhalb der betrachteten Datenstruktur.

Schaltkreise bilden nicht nur eine kompakte, sondern auch eine natürliche Darstellung von booleschen Funktionen. Sie sind originäre Bestandteile unserer Rechner. Als Hardware realisiert ist aber nur noch die Auswertung interessant. Wir wollen die betrachteten Operationen für Schaltkreise analysieren.

Die Auswertung ist in Zeit $O(m)$ möglich, da wir die Ergebnisse an den einzelnen Bausteinen sequenziell berechnen können. Dies gilt für die softwaremäßige Simulation des

Schaltkreises. Hardware erlaubt eine Zeitersparnis durch Parallelverarbeitung. Für zwei Schaltkreise ist die Synthese in linearer Zeit möglich. Die Schaltkreise werden kopiert, wobei sie auf dieselben Inputs zugreifen. An einem letzten Baustein vom Typ \otimes , der als Vorgänger die Bausteine erhält, an denen f und g dargestellt werden, wird dann $h = f \otimes g$ dargestellt. Die Ersetzung durch Konstanten ist einfach. Alle x_i verlassenden Kanten werden durch Kanten, die die entsprechende Konstante verlassen, ersetzt. Dennoch sind Schaltkreise nur als statische Datenstruktur für die wiederholte Auswertung geeignet und nicht als dynamische Datenstruktur. Der Grund ist, dass für die Operationen Erfüllbarkeitstest, Gleichheitstest und Minimierung keine effizienten Algorithmen bekannt sind. (Es gibt sogar sehr gute Gründe, warum es für diese Operationen keine effizienten Algorithmen geben kann, aber das gehört in andere Vorlesungen.)

Die am häufigsten eingesetzte dynamische Datenstruktur für boolesche Funktionen ist die Darstellung durch OBDDs (ordered binary decision diagrams).

Definition 2.9.1: Ein π -OBDD zur Variablenordnung $\pi = (x_1, \dots, x_n)$ ist ein gerichteter azyklischer Graph $G = (V, E)$ mit Zusatzinformationen. Die Knotenmenge enthält Senken (ohne Nachfolger) und innere Knoten. Jede Senke ist mit einer booleschen Konstanten markiert. Jeder innere Knoten ist mit einer Variablen $x_i, 1 \leq i \leq n$, markiert und hat zwei ausgehende Kanten, von denen eine mit 0 und die andere mit 1 markiert ist. Der Endpunkt einer einen x_i -Knoten verlassenden Kante darf nur ein x_j -Knoten mit $j > i$ oder eine Senke sein. (Der Graph zerfällt in $n + 1$ Schichten, den x_1 -Knoten, \dots , den x_n -Knoten und den Senken und alle Kanten sind „abwärts“ gerichtet.) Jeder Knoten v in einem OBDD stellt eine boolesche Funktion f_v dar. Der Wert $f_v(a), a \in \{0, 1\}^n$, ist folgendermaßen definiert. Der Auswertungsweg startet an v , an x_j -Knoten wird die a_j -Kante gewählt, bis eine Senke s erreicht wird. Dann ist $f_v(a)$ gleich der Markierung von s .

Da die Variablen einer booleschen Funktion keine festgelegte Ordnung haben, erhalten wir für verschiedene Variablenordnungen, z. B. $\pi = (x_1, x_2, x_3)$ und $\pi' = (x_3, x_1, x_2)$, verschiedene Datenstrukturen. Daher spricht man von π -OBDDs, wenn die Variablenordnung π gewählt wurde. Die Wahl einer geeigneten Variablenordnung ist ein wichtiges und schwieriges Problem. Eine Funktion kann bezüglich π lineare Größe haben und bezüglich π' exponentielle Größe erfordern. Operationen wie die Synthese sind nur effizient durchzuführen, wenn die betrachteten OBDDs dieselbe Variablenordnung respektieren. Dieses Problem wollen wir hier nicht vertiefen, sondern nur über π -OBDDs mit der natürlichen Variablenordnung $\pi = (x_1, \dots, x_n)$ diskutieren.

Wir kommen nun zu den Operationen auf π -OBDDs. Die Länge des Auswertungsweges ist durch n beschränkt. Daher ist die Auswertung unabhängig von der Größe des π -OBDDs in Zeit $O(n)$ möglich. Die Ersetzung durch Konstanten ist ebenfalls einfach. Betrachten wir den Fall, dass x_i durch c ersetzt werden soll. Dann erhalten wir die richtige Ausgabe, wenn wir die Auswertung so abändern, dass wir an x_i -Knoten stets die c -Kante wählen. Dies können wir erreichen, indem wir alle Kanten auf x_i -Knoten direkt auf deren c -Nachfolger zeigen lassen und die x_i -Knoten eliminieren. Dies ist mit einem DFS-Durchlauf des π -OBDDs in linearer Zeit möglich. Der Erfüllbarkeitstest für die an v dargestellte Funktion

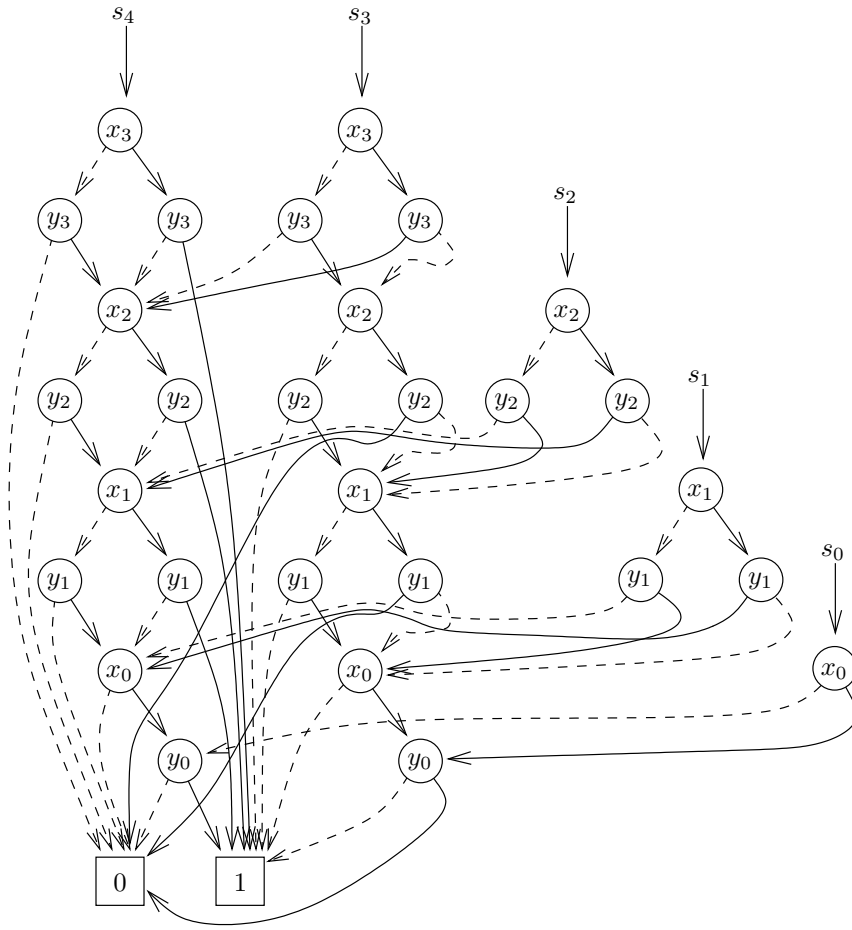


Abbildung 2.9.1: Ein OBDD für die Addition von 4-Bit-Zahlen.
Gestrichelte Kanten sind 0-Kanten, durchgezogene Kanten 1-Kanten.

ist äquivalent zur Frage, ob es einen gerichteten Weg von v zu einer mit 1 markierten Senke gibt. Da auf jedem Weg jede Variable nur einmal „getestet“ wird, gibt es für jeden graphentheoretischen Weg eine Eingabe, für die dieser Weg der Auswertungsweg ist. Also kann der Erfüllbarkeitstest mit einem an v startenden DFS-Durchlauf durchgeführt werden.

Wir kommen nun zur Synthese. Es seien also G_f, G_g und \otimes gegeben, sowie v^* und w^* die Knoten, an denen f und g dargestellt werden. Wie erhalten wir $f(a) \otimes g(a)$? Wir durchlaufen die Auswertungswege in G_f und G_g und verknüpfen die Ergebnisse. Die Idee besteht nun darin, G_f und G_g gleichzeitig zu durchlaufen. Dazu bilden wir eine Art Kreuzprodukt von $G_f = (V_f, E_f)$ und $G_g = (V_g, E_g)$. Das Ergebnis $G_h = (V_h, E_h)$ sei folgendermaßen definiert. Es ist $V_h := V_f \times V_g$. Sei $(v, w) \in V_h$ und seien v_0, v_1, w_0, w_1 die entsprechenden Nachfolger, wenn die Knoten keine Senken sind. Dann unterscheiden wir die folgenden Fälle:

- 1.) v ist x_i -Knoten und w ist x_i -Knoten. Dann wird (v, w) x_i -Knoten mit dem 0-

Nachfolger (v_0, w_0) und dem 1-Nachfolger (v_1, w_1) .

- 2.) v ist x_i -Knoten und w ist x_j -Knoten. Falls $i < j$, wird (v, w) x_i -Knoten mit dem 0-Nachfolger (v_0, w) und dem 1-Nachfolger (v_1, w) . Falls $i > j$, wird (v, w) x_j -Knoten mit dem 0-Nachfolger (v, w_0) und dem 1-Nachfolger (v, w_1) .
- 3.) v ist x_i -Knoten und w ist eine Senke. Dann wird (v, w) x_i -Knoten mit dem 0-Nachfolger (v_0, w) und dem 1-Nachfolger (v_1, w) . Der Fall, dass v eine Senke und w ein x_i -Knoten ist, wird analog behandelt.
- 4.) v ist eine c_1 -Senke und w eine c_2 -Senke. Dann wird (v, w) eine $(c_1 \otimes c_2)$ -Senke.

Da für OBDDs $G = (V, E)$ nach Definition $|E| < 2|V|$ ist, bezeichnen wir mit $|G| := |V|$ die Größe des OBDDs. Mit der obigen Definition erhalten wir ein π -OBDD G_h mit $|G_h| = |G_f| \cdot |G_g|$. Wir müssen zeigen, dass G_h am Knoten (v^*, w^*) wirklich $h = f \otimes g$ darstellt. Dazu zeigen wir allgemein, dass G_h am Knoten (v, w) die Funktion $f_v \otimes g_w$ darstellt. Dies wird über Induktion bezüglich der Knotennummern gezeigt, wobei wir die Knoten in umgekehrter topologischer Sortierung behandeln. Wir beginnen also bei den Knoten (v, w) , für die v und w Senken sind. Dann gilt die Behauptung per Definition. Wir betrachten nun einen x_i -Knoten (v, w) und die zugehörige Funktion $h_{(v,w)}$. Es wird behauptet, dass $h_{(v,w)} = f_v \otimes g_w$ ist. Es genügt, die Behauptung für die beiden Subfunktionen (Kofaktoren) $h_{(v,w)|x_i=0}$ und $h_{(v,w)|x_i=1}$ zu zeigen, da dies eine vollständige Fallunterscheidung ist. Betrachten wir o.B.d.A. den Fall $x_i = 0$. Es wird $h_{(v,w)|x_i=0}$ am 0-Nachfolger von (v, w) dargestellt. Nach Induktionsvoraussetzung und Konstruktion wird dort in allen Fällen $(f_v)_{|x_i=0} \otimes (g_w)_{|x_i=0} = (f_v \otimes g_w)_{|x_i=0}$ dargestellt. Dies folgt für x_i -Knoten v und/oder x_i -Knoten w direkt. Ist z. B. w ein x_j -Knoten, dann ist $j > i$ und g_w kann nicht von x_i essenziell abhängen, da unterhalb von x_j -Knoten kein x_i -Knoten erlaubt ist. Gleiches gilt, falls w eine Senke ist. In diesen Fällen ist $g_w = g_{w|x_i=0}$ und die Induktionsbehauptung folgt ebenfalls.

Die Synthese ist also effizient durchführbar, aber die OBDD-Größe wächst bei mehreren Syntheseschritten schnell. Wenn wir nur h darstellen wollen, sind alle Knoten, die von (v^*, w^*) nicht erreichbar sind, überflüssig und können eliminiert werden. Bei π -OBDDs mit mehr als 10^6 Knoten ist es essenziell, dass wir derartige Knoten gar nicht erst erzeugen. Dazu erzeugen wir ein π -OBDD G_h für h in einem DFS-Ansatz. Wir starten mit (v^*, w^*) und benutzen einen Stack für die Knoten von G_h , die wir erzeugt haben, für die wir aber noch keine Nachfolger konstruiert haben. Dies impliziert also einen DFS-artigen Aufbau des π -OBDDs für h . Solange noch ein Knoten auf dem Stack ist, erzeugen wir, wie bei der Bildung des Kreuzproduktes beschrieben, die Nachfolger dieses Knotens und legen sie auf dem Stack ab. Hierbei müssen wir aber aufpassen. Bei einem DFS-Durchlauf durch einen Graphen wird bei jedem erreichten Knoten überprüft, ob er zum ersten Mal oder zum wiederholten Mal erreicht wird. Nur im ersten Fall wird von ihm aus die Suche fortgesetzt. Hier haben wir es mit der DFS-Konstruktion von G_h , einem π -OBDD auf einer Teilmenge von $V_f \times V_g$, zu tun. Daher legen wir eine Tabelle, die so genannte „computed-table“, aller bereits besuchten $(v, w) \in V_f \times V_g$ an. Auf diese Weise kann beim wiederholten

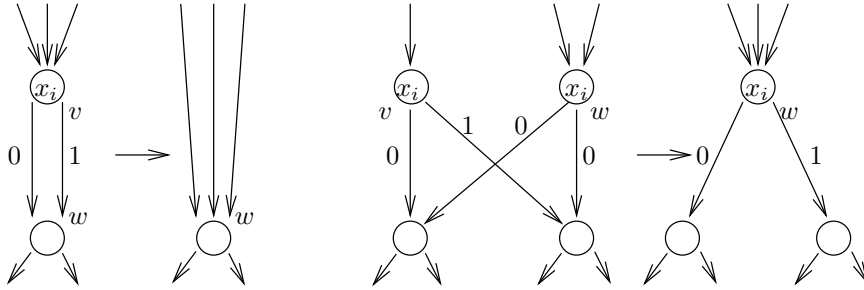


Abbildung 2.9.2: Eliminationsregel und Verschmelzungsregel für π -OBDDs.

Erreichen von (v, w) die Erzeugung eines neuen Knotens durch eine Kante auf den bereits vorhandenen Knoten ersetzt werden.

Zusätzlich gibt es zwei einfache Regeln zur Verkleinerung von π -OBDDs (siehe Abbildung 2.9.2.):

- Eliminationsregel: Zeigen die 0-Kante und die 1-Kante von einem Knoten v auf denselben Knoten w , können alle Knoten auf v durch Kanten auf w ersetzt und der Knoten v kann eliminiert werden.
- Verschmelzungsregel: Haben zwei Knoten v und w dieselbe Markierung x_i , denselben 0-Nachfolger und denselben 1-Nachfolger, können alle Kanten auf v durch Kanten auf w ersetzt und der Knoten v kann eliminiert werden.

Es ist offensichtlich, dass wir auf diese Weise von jedem Knoten aus (eventuell von w statt von v aus) und jede Eingabe dieselbe Senke erreichen.

Diese Regeln verändern also nicht die Menge der dargestellten Funktionen und verkleinern π -OBDDs um jeweils einen Knoten. Sie können also stets eingesetzt werden. Wir zitieren hier nur ein zentrales Ergebnis.

Satz 2.9.2: Es seien v_1, \dots, v_m Knoten in einem π -OBDD G , an denen f_1, \dots, f_m dargestellt werden. Wenn in G alle Knoten von mindestens einem der Knoten v_1, \dots, v_m erreichbar sind und wenn in G weder die Eliminationsregel noch die Verschmelzungsregel anwendbar ist, dann ist G das kleinste π -OBDD, das f_1, \dots, f_m darstellt und G ist bis auf die Benennung der Knoten eindeutig.

Wir haben es bisher verhindert, nicht erreichbare Knoten zu konstruieren. Wenn wir nun zusätzlich die beiden Reduktionsregeln, also die Eliminationsregel und die Verschmelzungsregel, „sofort“ anwenden, erzeugen wir „automatisch“ π -OBDDs minimaler Größe. Hierin liegt die eigentliche Stärke der Datenstruktur π -OBDD. Es wird die Konstruktion unnötig großer π -OBDDs vermieden. Wir integrieren nun die Anwendung der Reduktionsregeln in den Syntheseprozess. Immer wenn die an einem Knoten (v, w) startende DFS-Konstruktion abgeschlossen ist, überprüfen wir, ob (v, w) überflüssig ist. Aufgrund der DFS-Konstruktion ist die bisher einzige Kante auf (v, w) bekannt und kann gegebenenfalls einen neuen Endpunkt erhalten. Ob die Eliminationsregel anwendbar ist, ist auf

triviale Weise feststellbar. Für die Verschmelzungsregel verwenden wir eine zweite Tabelle, die so genannte unique-Tabelle. In ihr sind alle bestätigten Knoten, also die Knoten, für die die DFS-Konstruktion abgeschlossen ist, als Tripel aus Markierung, 0-Nachfolger und 1-Nachfolger eingetragen. Wenn wir unseren neuen Knoten auch als Tripel beschreiben, kann die Verschmelzungsregel genau dann angewendet werden, wenn dieses Tripel bereits in der unique-table vorhanden ist.

Nun bekommt die computed-table eine zusätzliche Aufgabe. Da ein Knotenpaar (v, w) besucht, aber eliminiert oder verschmolzen sein kann, muss diese Tabelle nun den Verweis enthalten, welcher Knoten und welches Tripel im neu konstruierten π -OBDD (v, w) repräsentiert. Dies ist nötig, damit Kanten auf (v, w) auf den Repräsentanten von (v, w) zeigen können. Beide Tabellen müssen die Operationen Suchen und Einfügen unterstützen. Datenstrukturen, die dieses tun, lernen wir in Kapitel 3 kennen. Die unique-table enthält niemals mehr Einträge als das erzeugte π -OBDD, während die computed-table viel mehr Einträge haben kann. Dieses Problem greifen wir in Kapitel 3 wieder auf. Hier soll darauf hingewiesen werden, was passiert, wenn wir die computed-table gegebenenfalls verkürzen, indem wir Einträge eliminieren. Dann werden DFS-Konstruktionen für Knotenpaare eventuell wiederholt gestartet. Das kostet Extrazeit, wobei Platz bei der computed-table gespart wurde. Das Ergebnis bleibt jedoch korrekt, da die Reduktionsregeln die doppelt erzeugten Knoten am Ende mit ihren Doppelgängern verschmelzen.

Natürlich können die Reduktionsregeln auch bei einer Ersetzung durch Konstanten eingesetzt werden. Wir haben also nun effiziente Algorithmen für die Auswertung, den Erfüllbarkeitstest, die Ersetzung durch Konstanten und die Synthese mit integrierter Minimierung. Wir haben schon vorab beschrieben, wie wir daraus Algorithmen für den Gleichheitstest, die Ersetzung durch Funktionen und die Abstraktion von Variablen erhalten.

Obwohl die Synthese effizient durchzuführen ist, kann das Ergebnis eine Größe von $\Theta(|G_f| \cdot |G_g|)$ haben. Damit kann eine Folge von n Syntheseschritten, ausgehend von einem π -OBDD, das mit $n + 2$ Knoten $0, 1, x_1, \dots, x_n$ darstellt, zu einem π -OBDD exponentieller Größe führen. Dies ist aber unvermeidlich, da es Funktionen gibt, die sich in Schaltkreisen linearer Größe, aber nur in OBDDs exponentieller Größe darstellen lassen. Mit diesen Bemerkungen schließen wir diese kurze Einführung in das viel umfassendere Gebiet OBDDs ab.

3 Dynamische Dateien

3.1 Vorbemerkungen

Eine Datei heißt dynamisch, wenn wir in sie neue Daten einfügen können und aus ihr Daten entfernen können. Wir nehmen an, dass ein Datum aus einem Schlüssel k und dem eigentlichen Datensatz besteht. Die Schlüssel stammen aus einem gegebenen Universum U . Es gibt für jeden Schlüssel k zu keinem Zeitpunkt mehr als einen Datensatz, der unter k abgespeichert ist.

Dynamische Dateien müssen also mindestens die folgenden Operationen unterstützen:

- MAKEDICT: Erzeugt eine leere Datei (dictionary, Wörterbuch).
- SEARCH (k): Sucht nach einem unter dem Schlüssel k abgespeicherten Datensatz und gibt ihn gegebenenfalls aus.
- INSERT (k, x): Fügt den Datensatz x unter dem Schlüssel k in die Datei ein. Falls es einen Datensatz mit Schlüssel k bereits gibt, wird dieser überschrieben.
- DELETE (k): Entfernt k und den unter k abgespeicherten Datensatz, falls vorhanden.

Zu den weiteren wünschenswerten Operationen gehören:

- MIN (MAX): Findet den Datensatz, der unter dem kleinsten (größten) Schlüssel abgespeichert ist.
- LIST: Listet die Datensätze, geordnet nach ihren Schlüsseln, auf.
- CONCATENATE (D_1, D_2, D): Nur definiert, falls alle Schlüssel in D_1 kleiner als alle Schlüssel in D_2 sind. Es wird eine neue Datei D gebildet, die die Daten aus D_1 und D_2 enthält.
- SPLIT (D, k, D_1, D_2): Aus der Datei D werden die Dateien D_1 und D_2 gebildet, wobei D_1 alle Datensätze aus D enthält, die unter Schlüsseln $k' \leq k$ abgespeichert sind, und D_2 die restlichen Datensätze aus D enthält. Es wird teilweise angenommen, dass D unter dem Schlüssel k einen Datensatz enthält.

In diesem Kapitel werden verschiedene Realisierungen dynamischer Dateien diskutiert. Wir beginnen in Kapitel 3.2 mit Verfahren unter der Bezeichnung Hashing. In den hier vorgestellten einfachen Varianten kann die worst case Rechenzeit linear sein (und dieses Verhalten können wir ja auch mit Arrays oder Listen erreichen). Allerdings ist das durchschnittliche und das „typische“ Verhalten viel besser. Wenn wir nun sehr viele Operationen auf den dynamischen Dateien durchführen, können wir mit einer sehr guten Gesamtrechenzeit rechnen. Im Kapitel 3.3 betrachten wir binäre Suchbäume. Dabei stellen wir fest,

dass es wünschenswert ist, die Baumtiefe zu kontrollieren. Dies gelingt mit 2-3-Bäumen (Kapitel 3.4) und AVL-Bäumen (Kapitel 3.6). Bayer-Bäume (Kapitel 3.5) bilden eine Verallgemeinerung von 2-3-Bäumen, die für die automatische Datenverwaltung in Rechnern eine zentrale Rolle spielen. Alle bis dahin behandelten Datenstrukturen arbeiten deterministisch. Der moderne Trend geht zu randomisierten Datenstrukturen und Algorithmen. Diese erreichen typischerweise nicht das gleiche worst case Verhalten, aber mit überwältigender Wahrscheinlichkeit (bezogen auf die im Algorithmus benutzten Zufallsbits) erreichen sie sehr gute Rechenzeiten. Randomisierte Algorithmen sind oft einfacher zu beschreiben und zu implementieren und sie sind dabei oft effizienter als deterministische Lösungen. Ihre Analyse verlangt allerdings den Umgang mit (jedoch elementaren) Methoden der Wahrscheinlichkeitstheorie. Skiplisten (Kapitel 3.7) bilden eine randomisierte Realisierung einer dynamischen Datei. Da die Datensätze bei der Manipulation der dynamischen Dateien nicht wichtig sind, identifizieren wir im folgenden Schlüssel, Daten und Datensätze.

3.2 Hashing

Das Universum U , aus dem die Schlüssel stammen können, ist meistens sehr groß. Das Adressfeld auf Überweisungsformularen hat eine Länge von 27. Wenn wir die 26 üblichen Buchstaben, Bindestriche, Kommas und Leerstellen zulassen, erhalten wir bereits ein Universum der Größe $29^{27} \geq 3 \cdot 10^{39}$.

Eine Hashfunktion h ist eine Abbildung $h: U \rightarrow \{0, \dots, M-1\}$. Für $x \in U$ ist $h(x)$ die Adresse, unter der x gespeichert werden soll. Gute Hashfunktionen sollen die folgenden Forderungen erfüllen:

- h soll einfach auszuwerten sein,
- h soll gut streuen.

Beim geschlossenen Hashing sollen die Daten in einem Array der Länge M mit den Adressen $0, \dots, M-1$ abgespeichert werden. Dabei soll M nicht wesentlich größer als die zu erwartende Maximalzahl gleichzeitig zu speichernder Daten sein. Damit ist typischerweise $M \ll |U|$. Dann kann h nicht injektiv sein. Es sollen aber ungefähr gleich viele Daten aus dem Universum auf jede Adresse abgebildet werden. Da wir vorher die abzuspeichernden Daten nicht kennen, sind allerdings Kollisionen nicht auszuschließen. Dann müssen Daten x und x' mit $x \neq x'$ und $h(x) = h(x')$ abgespeichert werden. Kollisionen sind selbst bei ideal streuenden Hashfunktionen, also $||h^{-1}(i)| - |h^{-1}(j)|| \leq 1$, und weit weniger als M zu speichernden Daten sehr wahrscheinlich. Dies ist das berühmte Geburtstagsparadoxon. Es seien n Daten zufällig aus U gewählt. Wir berechnen die Wahrscheinlichkeit, dass die Daten durch h , falls $\text{Prob}(h(x) = j) = 1/M$ ist, alle verschieden abgebildet werden. Diese Wahrscheinlichkeit beträgt

$$\frac{M-1}{M} * \dots * \frac{M-n+1}{M}$$

und ist für $M = 365$ und $n = 23$ nur noch 0,493 oder für $n = 50$ nur 0,03. Dies führt zu dem Schluss, dass wir mit Kollisionen leben müssen.

Als Hashfunktion für Universen $U = \{0, \dots, u - 1\}$ haben sich die Modulfunktionen

$$h(x) \equiv x \pmod{M}$$

bewährt. Falls M eine Zweierpotenz ist, werden durch die $(\text{mod } M)$ -Funktion nur die vorderen Bits gestrichen. Dies kann bei strukturierten Daten zu überdurchschnittlich vielen Kollisionen führen. Daher wird M häufig als Primzahl gewählt.

Wir nehmen an, dass die Auswertung von h in konstanter Zeit möglich ist. Zunächst behandeln wir dynamische Dateien, die nur das Suchen nach Daten und das Einfügen von Daten unterstützen. Diese beiden Operationen genügen bei der unique-table und der computed-table in Kapitel 2.9. Um mit Kollisionen fertig zu werden, legen wir in Abhängigkeit von $h(x)$ eine Reihenfolge fest, in der die Speicherplätze ausprobiert werden. Eine Suche wird erfolgreich beendet, wenn der gesuchte Schlüssel gefunden wurde, und erfolglos abgebrochen, wenn wir bei der Suche auf einen leeren Speicherplatz stoßen.

Als Beispiel nehmen wir $M = 19$ und $h(x) = 7$ an. Alle Rechnungen erfolgen $(\text{mod } M)$, falls nicht anders angegeben. Die wichtigsten Strategien zur Kollisionsbehandlung sind:

- Lineares Sondieren (linear probing): Es werden die Speicherplätze in der Reihenfolge $h(x) + i, 0 \leq i \leq M - 1$, betrachtet, im Beispiel in der Reihenfolge 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 0, 1, 2, 3, 4, 5, 6.
- Quadratisches Sondieren (quadratic probing): Es werden die Speicherplätze in der Reihenfolge $h(x), h(x) + 1^2, h(x) - 1^2, h(x) + 2^2, h(x) - 2^2, \dots$ betrachtet. Wenn wie in unserem Beispiel $M \equiv 3 \pmod{4}$ ist, ist gesichert, dass in den ersten M Schritten alle Speicherplätze betrachtet werden. Diese zahlentheoretische Aussage werden wir nicht beweisen. Im Beispiel erhalten wir die folgende Reihenfolge 7, 8, 6, 11, 3, 16, $-2 \equiv 17$, $23 \equiv 4$, $-9 \equiv 10$, $32 \equiv 13$, $-18 \equiv 1$, $43 \equiv 5$, $-29 \equiv 9$, $56 \equiv 18$, $-42 \equiv 15$, $71 \equiv 14$, $-57 \equiv 0$, $88 \equiv 12$, $-74 \equiv 2$.
- Multiplikatives Sondieren (add to hash): Hier ist die Reihenfolge $i \cdot h(x), 1 \leq i \leq M - 1$. Hierbei wird $h(x) \neq 0$ vorausgesetzt. Falls M Primzahl ist, werden alle Speicherplätze erreicht. Ansonsten gäbe es Zahlen i, j mit $1 \leq i < j \leq M - 1$ und

$$i \cdot h(x) \equiv j \cdot h(x) \pmod{M}.$$

Dann wäre $(j - i) \cdot h(x)$ ein Vielfaches von M . Dies ist, da $1 \leq j - i \leq M - 1$ und $1 \leq h(x) \leq M - 1$ ist, für Primzahlen M unmöglich. In unserem Beispiel ist die Reihenfolge 7, 14, $21 \equiv 2$, $28 \equiv 9$, $35 \equiv 16$, $42 \equiv 4$, $49 \equiv 11$, $56 \equiv 18$, $63 \equiv 6$, $70 \equiv 13$, $77 \equiv 1$, $84 \equiv 8$, $91 \equiv 15$, $98 \equiv 3$, $105 \equiv 10$, $112 \equiv 17$, $119 \equiv 5$, $126 \equiv 12$.

- Doppeltes Hashing (double hashing): es werden zwei Hashfunktionen $h_1(x) = x \bmod p$ und $h_2 = (x \bmod q) + 1$ für verschiedene Primzahlen $p, q > M$ verwendet und dann $(\bmod M)$ die Speicherplätze $h_1(x) + i \cdot h_2(x), 0 \leq i \leq M - 1$, betrachtet. Es sei beispielsweise $p = 67, q = 43$ und $x = 89$. Dann ist $h_1(x) = 22$ und $h_2(x) = 4$. Die Speicherplätze werden in folgender Reihenfolge betrachtet: $22 \equiv 3, 7, 11, 15, 0, 4, 8, 12, 16, 1, 5, 9, 13, 17, 2, 6, 10, 14, 18$.

Was unterscheidet diese Strategien? Dazu definieren wir Primärkollisionen und Sekundärkollisionen. Bei einer Primärkollision ist $h(x) = h(x')$. Dagegen ist bei einer Sekundärkollision $h(x) \neq h(x')$, aber bei den Suchen, die bei $h(x)$ und $h(x')$ starten, werden früh gemeinsame Plätze erreicht. Die Wahrscheinlichkeit von Primärkollisionen ist bei allen gut streuenden Hashfunktionen gleich groß. Dies gilt nicht für Sekundärkollisionen. Das lineare Sondieren neigt besonders stark zu Klumpenbildungen. Wenn für $M = 19$ die 8 Positionen 2, 5, 6, 9, 10, 11, 12 und 17 belegt sind, erhalten wir bei einer ideal streuenden Hashfunktion die folgenden Wahrscheinlichkeiten, einen Platz zu belegen:

- $\frac{5}{19}$ für den Platz 13 (dieser Platz wird belegt, falls $h(x) \in \{9, 10, 11, 12, 13\}$),
- $\frac{3}{19}$ für Platz 7,
- $\frac{2}{19}$ für jeden der Plätze 3 und 18,
- $\frac{1}{19}$ für jeden der Plätze 0, 1, 4, 8, 14, 15, 16.

Große Klumpen führen zu langen Suchzeiten, wenn wir im vorderen Teil des Klumpens starten. Die anderen Strategien zur Kollisionsbehandlung gestalten die Suche für verschiedene $h(x)$ -Werte unterschiedlich. Dies führt zu besserem Verhalten.

Für das lineare Sondieren konnte für großes M gezeigt werden, dass bei einer 95%-igen Auslastung der Datei eine erfolgreiche Suche im Durchschnitt 10,5 Speicherplätze und eine erfolglose Suche 200,5 Speicherplätze betrachtet. Derartige Analysen sind für die anderen Strategien noch schwieriger. Um die möglichen Grenzen auszuloten, wird ein Idealfall betrachtet.

Beim idealen Hashing wird angenommen, dass alle $\binom{M}{n}$ Konfigurationen, bei denen n von M Speicherplätzen belegt sind, dieselbe Wahrscheinlichkeit $1/\binom{M}{n}$ haben. Wir beginnen unsere Analyse mit der Suche nach einem nicht abgespeicherten Schlüssel.

Wie groß ist dann die Wahrscheinlichkeit, dass beim Einfügen des $(n + 1)$ -ten Datums genau r Plätze getestet werden müssen, bis ein freier Platz gefunden wird? Dieses Ereignis tritt bei folgenden Konstellationen ein. Die ersten $r - 1$ getesteten Plätze sind besetzt, der r -te Platz ist frei. Auf den restlichen $M - r$ Plätzen können die weiteren $n - (r - 1)$ besetzten Plätze beliebig verteilt sein, dafür gibt es $\binom{M-r}{n-(r-1)}$ Möglichkeiten. Wenn P_r die Wahrscheinlichkeit ist, dass r Tests durchgeführt werden, gilt

$$\begin{aligned} P_r &= \binom{M-r}{n-(r-1)} / \binom{M}{n} \text{ für } 1 \leq r \leq n+1 \text{ und} \\ P_r &= 0 \text{ für } r > n+1. \end{aligned}$$

Die erwartete Anzahl E durchzuführender Tests beträgt

$$\begin{aligned} E &= \sum_{1 \leq r \leq n+1} r P_r = \sum_{1 \leq r \leq n+1} r \binom{M-r}{n-(r-1)} / \binom{M}{n} \\ &= \sum_{1 \leq r \leq n+1} r \binom{M-r}{M-n-1} / \binom{M}{n} \quad \left(\text{da } \binom{a}{b} = \binom{a}{a-b} \right). \end{aligned}$$

Nun ersetzen wir r durch $(M+1) - (M+1-r)$. Da die Summe aller P_r natürlich 1 ist, erhalten wir

$$E = M+1 - \sum_{1 \leq r \leq n+1} (M+1-r) \binom{M-r}{M-n-1} / \binom{M}{n}.$$

Indem wir die Binomialkoeffizienten ausführlich hinschreiben, folgt sofort

$$(M+1-r) \binom{M-r}{M-n-1} = (M-n) \binom{M-r+1}{M-n}.$$

Also gilt

$$E = M+1 - (M-n) \binom{M}{n}^{-1} \sum_{1 \leq r \leq n+1} \binom{M-r+1}{M-n}.$$

Für die Auswertung der Summe hilft uns folgende Formel

$$\sum_{a|b \leq a \leq c} \binom{a}{b} = \binom{c+1}{b+1}.$$

Diese Gleichung wiederum lässt sich kombinatorisch beweisen. Auf der rechten Seite steht die Anzahl der Möglichkeiten, $b+1$ Elemente aus $\{1, \dots, c+1\}$ auszuwählen. Die linke Seite beschreibt die $(b+1)$ -elementigen Teilmengen von $\{1, \dots, c+1\}$ auf andere Weise. Es sei $a+1$ das größte Element einer solchen Menge. Dann ist $a \in \{b, \dots, c\}$. Die restlichen b Elemente können auf $\binom{a}{b}$ verschiedene Weisen aus $\{1, \dots, a\}$ gewählt werden. Also stimmen die rechte und die linke Seite der Gleichung überein. Es folgt

$$\begin{aligned} E &= M+1 - (M-n) \binom{M}{n}^{-1} \binom{M+1}{M-n+1} \\ &= M+1 - (M-n) \binom{M}{n}^{-1} \binom{M+1}{n} \\ &= M+1 - (M-n) \frac{M+1}{M-n+1} \\ &= (M+1) \left(1 - \frac{M-n}{M-n+1} \right) = \frac{M+1}{M-n+1}. \end{aligned}$$

Wir kommen nun zur erfolgreichen Suche. Wenn wir das Datum suchen, das wir als k -tes Datum eingefügt haben, sind die Kosten genauso groß wie beim Einfügen dieses Datums, also durchschnittlich $(M+1)/(M-k+2)$. Wenn jedes der n Daten mit Wahrscheinlichkeit $1/n$ gesucht wird, beträgt die durchschnittliche Suchzeit

$$\begin{aligned} E' &= \frac{1}{n} \sum_{1 \leq k \leq n} \frac{M+1}{M-k+2} \\ &= \frac{M+1}{n} \left(\frac{1}{M-n+2} + \frac{1}{M-n+3} + \cdots + \frac{1}{M+1} \right) \\ &= \frac{M+1}{n} (H(M+1) - H(M-n+1)). \end{aligned}$$

Hierbei ist $H(m) = 1 + \frac{1}{2} + \cdots + \frac{1}{m}$ die harmonische Reihe.

Was können wir über $H(m)$ aussagen? Aus der Betrachtung Riemannscher Summen folgt

$$\int_1^{n+1} x^{-1} dx \leq H(n) \leq \int_1^n x^{-1} dx + 1$$

und damit

$$\ln(n+1) \leq H(n) \leq \ln n + 1.$$

Also gilt

$$E' \approx \frac{M+1}{n} \ln \frac{M+1}{M-n+1}.$$

Hieraus lässt sich errechnen, dass beim idealen Hashing für großes M und einer 95%-igen Auslastung der Datei eine erfolgreiche Suche im Durchschnitt 3,15 Speicherplätze und eine erfolglose Suche im Durchschnitt 20 Speicherplätze betrachtet. Die möglichen Einsparungen gegenüber dem linearen Sondieren sind also beträchtlich. Experimente zeigen, dass das doppelte Hashing dem idealen Hashing nahe kommt.

Eine Abart des geschlossenen Hashings besteht darin, an jeder Arrayposition Platz für c Schlüssel zu schaffen. Die Kollisionsbehandlung tritt erst in Kraft, wenn alle c Plätze belegt sind. Die gängige Implementierung der computed-table bei der π -OBDD-Synthese nimmt diese Idee auf ($c = 4$ ist ein gängiger Wert) und verzichtet auf eine Kollisionsbehandlung. Ist die gewählte Arrayposition vollständig belegt, wird die am längsten nicht betrachtete Information überschrieben. Die Datei wird fehlerhaft, die Konsequenzen für die π -OBDD-Synthese haben wir bereits in Kapitel 2.9 diskutiert.

Wenn wir Daten auch entfernen, stoßen wir auf ein neues Problem. Wir können eine Suche an einem freien Platz nicht ohne weiteres als erfolglos abbrechen. Der Platz könnte ja beim Einfügen unseres Datums besetzt gewesen sein. Wir müssen dann markieren, ob diese Stelle jemals belegt war. Wenn dann nach einer bestimmten Zeit praktisch alle Plätze diese Markierung tragen, führen erfolglose Suchen stets zur Betrachtung aller Speicherplätze. Geschlossenes Hashing unterstützt die Operation DELETE also nur, wenn wir für jeden Schlüssel stets wissen, ob unter ihm eine Information abgespeichert ist.

Ansonsten bildet offenes Hashing eine Alternative. An jedem Speicherplatz wird nun eine Liste angelegt. Das Datum x wird auf jeden Fall am Platz $h(x)$ abgespeichert. Natürlich

kosten die Zeiger in den Listen (M nil-Zeiger und n echte Zeiger) zusätzlich Speicherplatz. Allerdings ist auch eine Auslastung mit mehr als M Daten möglich.

SEARCH(x): Berechne $h(x)$. Durchsuche die Liste am Platz $h(x)$, bis das Datum gefunden oder das Ende der Liste erreicht wird. Die Kosten sind bei erfolgloser Suche proportional zur Listenlänge.

INSERT(x) (nach erfolgloser Suche): Füge x vorne in die Liste am Platz $h(x)$ ein, Kosten $\Theta(1)$.

DELETE(x) (nach erfolgreicher Suche): Bei der Suche merkt man sich den Zeiger auf x . Dann kann das Datum in Zeit $\Theta(1)$ aus der Liste entfernt werden.

Entscheidend ist also die Listenlänge. Da alle M Listen zusammen n Daten enthalten, ist die durchschnittliche Listenlänge n/M . Eine erfolglose Suche in einer zufällig gewählten Liste spricht also durchschnittlich $1 + n/M$ Zeiger an. Bei einer erfolgreichen Suche enthält die Liste mit Sicherheit das gesuchte Objekt. Die durchschnittliche Länge dieser Liste beträgt also $1 + (n - 1)/M$.

Das gesuchte Objekt steht bei Listenlänge l mit Wahrscheinlichkeit $1/l$ an jeder der l Positionen. Im Durchschnitt müssen dann $(l + 1)/2$ Daten betrachtet werden. Für $l = 1 + (n - 1)/M$ ergibt sich der Wert $1 + (n - 1)/(2M)$.

Für $n = M$ (bei geschlossenem Hashing wäre das eine Auslastung von 100%) betragen die erwarteten Suchzeiten also 2 (bei erfolgloser Suche) und knapp $3/2$ (bei erfolgreicher Suche).

Konkrete Daten können eine uns unbekannte Struktur haben. Diese Struktur kann dazu führen, dass es sehr viele Primärkollisionen gibt. Ein Ausweg aus diesem Dilemma besteht darin, eine Klasse von Hashfunktionen zu betrachten und für jede Anwendung eine Hashfunktion zufällig aus der Klasse zu wählen. Es sind Klassen von Hashfunktionen bekannt, die für jede Schlüsselmenge U mit großer Wahrscheinlichkeit zu einem guten Verhalten führen. Auch hier behebt Randomisierung ein praktisch wichtiges Problem. Wir wollen diesen Aspekt hier nicht vertiefen.

3.3 Binäre Suchbäume

Im Folgenden nehmen wir an, dass das Universum eine vollständig geordnete Menge ist und dass Vergleiche zwischen zwei Elementen aus dem Universum effizient durchführbar sind. Dies ist in der Praxis keine wesentliche Einschränkung.

Definition 3.3.1: Ein binärer Baum, in dessen Knoten Daten aus einem vollständig geordneten Universum gespeichert sind, heißt Suchbaum, wenn für jeden Knoten v und seinen Inhalt x gilt, dass alle im linken (rechten) Teilbaum von v gespeicherten Daten kleiner (größer) als x sind.

In manchen Anwendungen ist es wichtig, dass alle Daten in den Blättern gespeichert sind. Dann speichert man in jedem inneren Knoten das größte im linken Teilbaum gespeicherte Datum. Die zugehörigen Modifikationen sind einfach und werden hier nicht vorgestellt.

Abbildung 3.3.1 zeigt einen binären Suchbaum:

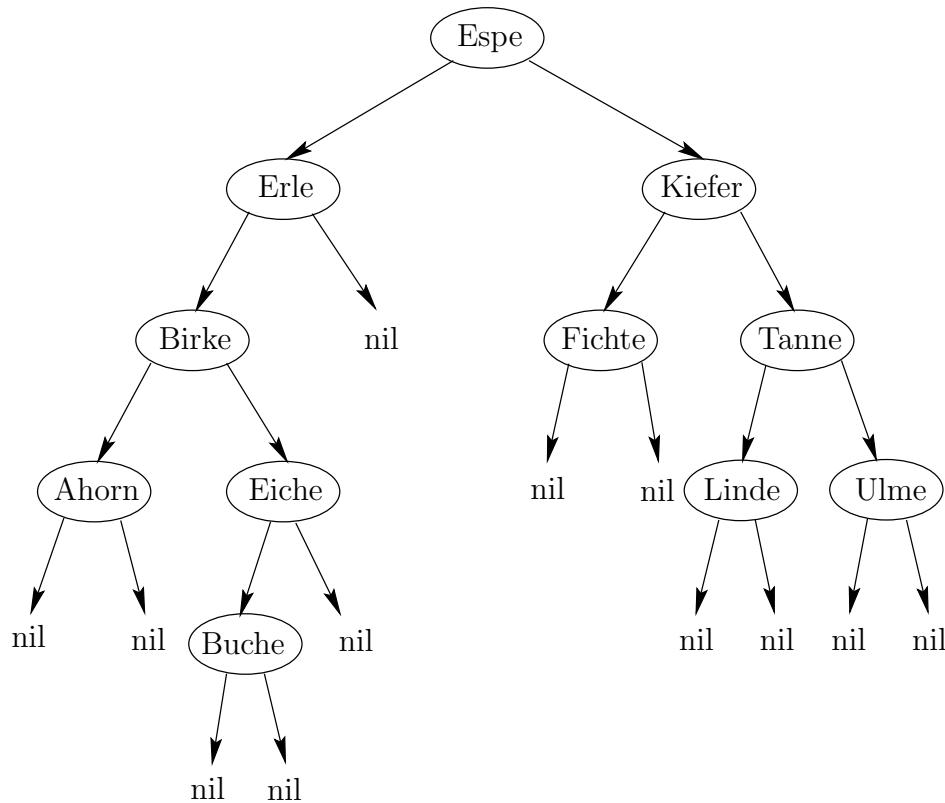


Abbildung 3.3.1: Ein binärer Suchbaum.

Die Suchprozedur ist nun einfach zu beschreiben.

SEARCH(x): Starte an der Wurzel. Wird ein Knoten mit Datum y erreicht, werden x und y verglichen.

1. $x = y$, erfolgreiche Suche.
2. $x < y$, suche linken Nachfolger auf. Falls dieser nicht existiert, erfolglose Suche.
3. $x > y$, suche rechten Nachfolger auf. Falls dieser nicht existiert, erfolglose Suche.

Die Güte eines Suchbaumes hängt offensichtlich davon ab, wie gut er balanciert ist. INSERT(x) (nach erfolgloser Suche): Der erreichte nil-Zeiger zeigt nun auf einen Knoten, in dem x abgespeichert wird und der zwei nil-Zeiger erhält.

Abbildung 3.3.2 zeigt den entstehenden Suchbaum, wenn die Wörter eins, zwei, ..., zehn in dieser Reihenfolge in einen leeren Suchbaum eingefügt werden. Wir verwenden die übliche lexikographische Ordnung auf Wörtern. Aus Platzgründen werden die nil-Zeiger nicht gezeigt.

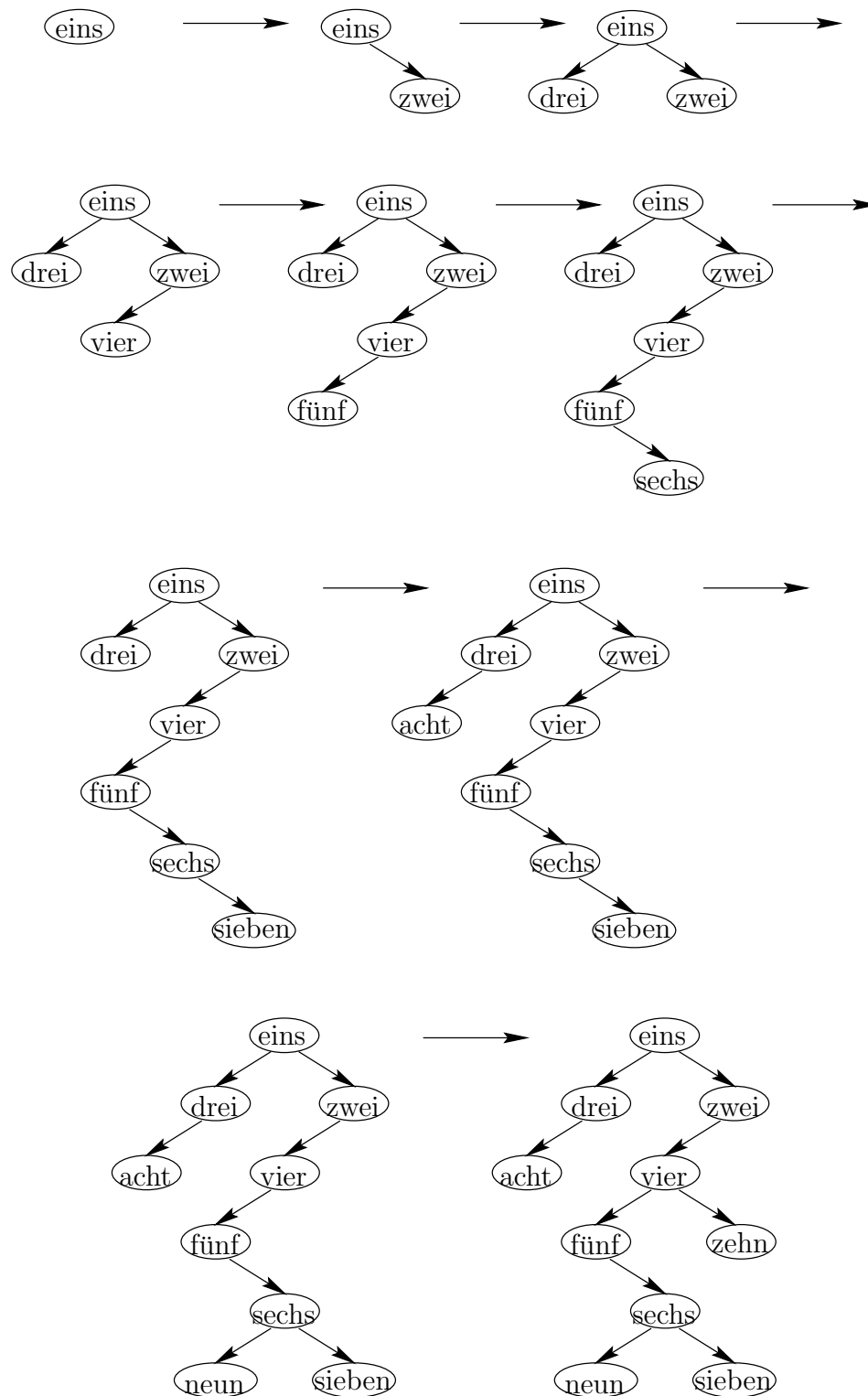


Abbildung 3.3.2: Die Entstehung eines binären Suchbaumes durch das Einfügen von Informationen in einen leeren Baum.

Nach dem Einfügen von „sieben“ war der Baum ziemlich entartet, hat sich aber dann noch etwas „erholt“. Wir vergleichen nun für diesen binären Baum, einen optimal balancierten Suchbaum und eine lineare Liste die durchschnittlichen Suchzeiten bei erfolgreicher Suche.

	worst case	average case
Baum Abb. 3.3.2	6	$\frac{1}{10}(1 + 2 + 2 + 3 + 3 + 4 + 4 + 5 + 6 + 6) = 3,6$
opt. Baum	4	$\frac{1}{10}(1 + 2 + 2 + 3 + 3 + 3 + 3 + 4 + 4 + 4) = 2,9$
lin. Liste	10	$\frac{1}{10}(1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10) = 5,5$

Tabelle 3.3.1: Ein Vergleich von Suchzeiten.

Inwieweit ist dieses Beispiel typisch? Wir analysieren allgemein die Situation, in der wir n Daten in einen leeren Baum einfügen, wobei alle $n!$ Reihenfolgen die gleiche Wahrscheinlichkeit haben. Sei $R_i(n)$ die erwartete Suchzeit für das i -kleinste Datum und $R(n) = \frac{1}{n}(R_1(n) + \dots + R_n(n))$ die durchschnittliche Suchzeit bei erfolgreicher Suche. Sei $S_j(n)$ die durchschnittliche Suchzeit bei erfolgreicher Suche, wenn das j -kleinste Datum an der Wurzel steht. Nach dem Satz über bedingte Wahrscheinlichkeiten gilt

$$R(n) = \frac{1}{n}(S_1(n) + \dots + S_n(n)).$$

Wenn das i -kleinste Datum an der Wurzel steht, ist der linke Teilbaum ein binärer Suchbaum auf den $i-1$ kleinsten Daten, unter denen jede Reihenfolge gleich wahrscheinlich ist, und der rechte Teilbaum ein binärer Suchbaum auf den $n-i$ größten Daten, unter denen jede Reihenfolge gleich wahrscheinlich ist. In ihnen ist die erwartete Suchzeit $R(i-1)$ bzw. $R(n-i)$, und sie werden mit Wahrscheinlichkeit $\frac{i-1}{n}$ bzw. $\frac{n-i}{n}$ erreicht. In jedem Fall muss die Wurzel getestet werden. Also gilt $R(1) = 1$ und

$$R(n) = 1 + \frac{1}{n} \sum_{1 \leq i \leq n} \left(\frac{i-1}{n} R(i-1) + \frac{n-i}{n} R(n-i) \right).$$

Wir setzen $T(n) = nR(n)$ und erhalten

$$T(n) = n + \frac{1}{n} \sum_{1 \leq i \leq n} (T(i-1) + T(n-i)).$$

Wie können wir diese Rekursionsgleichung auflösen? Dies ist nicht ganz einfach. Wir werden die Rechnung trotzdem durchführen, da die verwendeten Tricks oft angewendet werden können. In der Rekursionsgleichung kommen die Werte $T(n), \dots, T(1)$ vor. Unser erstes Ziel ist es, eine Rekursionsgleichung für $T(n)$ zu erhalten, in der nur $T(n-1)$ vorkommt. Dies erreichen wir durch Betrachtung von $nT(n) - (n-1)T(n-1)$. Die Faktoren führen dazu, dass die Terme $T(n-2), \dots, T(1)$ dieselben Vorfaktoren erhalten und in der Differenz verschwinden. Genauer:

$$nT(n) = n^2 + \sum_{1 \leq i \leq n} (T(i-1) + T(n-i)) = n^2 + 2 \cdot (T(1) + \dots + T(n-1))$$

$$(n-1)T(n-1) = (n-1)^2 + 2 \cdot (T(1) + \dots + T(n-2))$$

und

$$nT(n) - (n-1)T(n-1) = 2n-1 + 2 \cdot T(n-1).$$

Hieraus folgt

$$nT(n) - (n+1)T(n-1) = 2n-1.$$

Hierbei stört uns, dass der T -Wert mit der kleineren Eingabe $n-1$ den größeren Faktor $n+1$ hat. Dies können wir „ändern“, indem wir durch $n(n+1)$ dividieren. Es folgt

$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2n-1}{n(n+1)}.$$

Jetzt ist es naheliegend, $Z(n) = T(n)/(n+1)$ zu betrachten. Dann ist

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

Hier hilft wieder ein simpler Rechentrick. Offensichtlich ist

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

und, da $Z(0) = 0$ ist, folgt

$$\begin{aligned} Z(n) &= 2 \sum_{1 \leq i \leq n} \frac{i}{i} - 2 \sum_{1 \leq i \leq n} \frac{i}{i+1} - \sum_{1 \leq i \leq n} \frac{1}{i} + \sum_{1 \leq i \leq n} \frac{1}{i+1} \\ &= 2n - 2n + 2 \sum_{1 \leq i \leq n} \frac{1}{i+1} - 1 + \frac{1}{n+1} \\ &= 2 \sum_{1 \leq i \leq n} \frac{1}{i} - 2 + \frac{2}{n+1} - 1 + \frac{1}{n+1} \\ &= 2 \cdot H(n) - 3 + \frac{3}{n+1}. \end{aligned}$$

Dabei haben wir wie in Kapitel 3.1 die harmonische Reihe mit $H(n)$ bezeichnet.

Es folgt

$$T(n) = (n+1)Z(n) = 2(n+1)H(n) - 3(n+1) + 3$$

und

$$R(n) = T(n)/n = 2 \cdot \frac{n+1}{n} \cdot H(n) - 3 \cdot \frac{n+1}{n} + \frac{3}{n}.$$

Aus Kapitel 3.1 kennen wir bereits die Ungleichung $\ln(n+1) \leq H(n) \leq \ln n + 1$. Also ist

$$R(n) = 2 \cdot \ln n - O(1) = (2 \ln 2) \cdot \log n - O(1) \approx 1,386 \cdot \log n.$$

Satz 3.3.2: Bei Einfügung von n zufällig gewählten Daten in einen leeren Baum entsteht ein binärer Suchbaum, bei dem die durchschnittliche Suchzeit für erfolgreiches Suchen $(2 \ln 2) \log n - O(1) \approx 1,386 \log n$ beträgt.

Im besten Suchbaum ist die durchschnittliche Suchzeit für erfolgreiches Suchen $\log n - O(1)$. Zufällig gebildete Suchbäume sind also nicht weit von optimalen Suchbäumen entfernt. Ist die Datenliste jedoch sortiert, entsteht eine lineare Liste. In den meisten Anwendungen entstehen die Daten nicht in unstrukturierter Reihenfolge. Daher werden wir in den späteren Unterkapiteln balancierte Suchbäume untersuchen.

DELETE(x) (nach erfolgreicher Suche): Wir kennen dabei den Zeiger auf den Knoten v , der x enthält.

- (1) v ist Blatt. Setze den Zeiger auf v auf nil. Der Knoten v kann freigegeben werden.
- (2) v hat ein Kind. Setze den Zeiger auf v nun auf das Kind von v . Der Knoten v kann freigegeben werden.
- (3) v hat zwei Kinder. Suche im Suchbaum das größte Datum y , das kleiner als x ist. Dazu gehe zum linken Kind und dann stets zum rechten Kind, bis ein nil-Zeiger erreicht wird. Die Daten x und y werden ausgetauscht. Danach wird x entfernt. Der Knoten, in dem x sich nun befindet, hat kein rechtes Kind. Also sind wir in Fall (1) oder (2).

Bevor wir die Korrektheit dieses Algorithmus beweisen, führen wir die drei Fälle an dem Baum aus Abbildung 3.3.2 vor (siehe Abbildung 3.3.3).

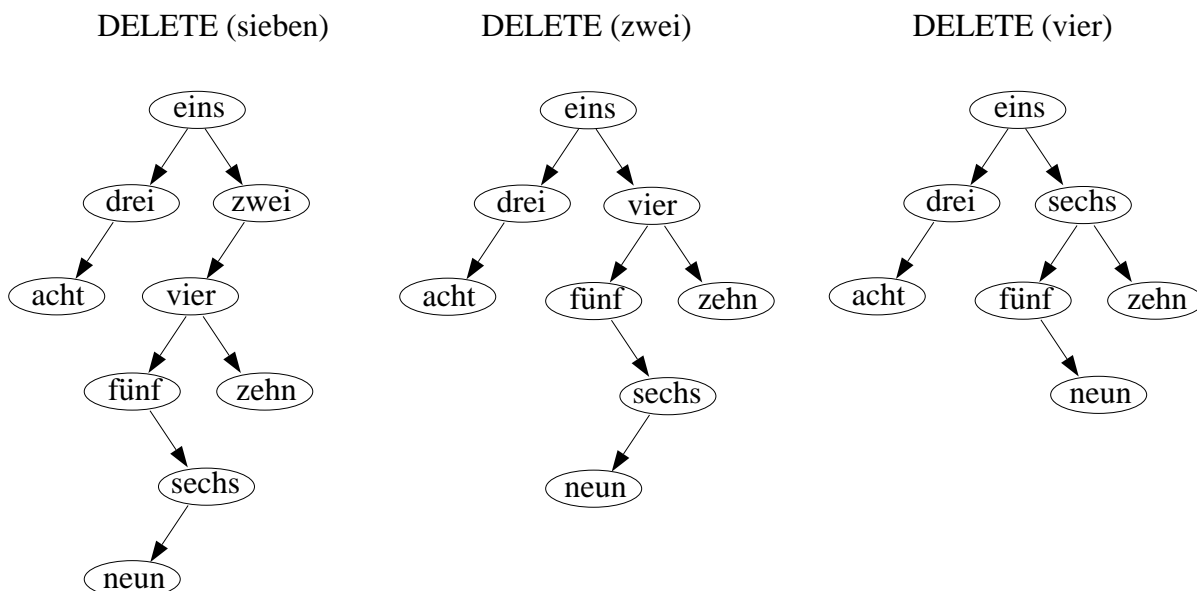


Abbildung 3.3.3: Das Entfernen von Daten aus einem binären Suchbaum.

Die Korrektheit ist im Fall (1) offensichtlich. Im Fall (2) umfasst der Zeiger auf v ein Intervall von Daten. Dieser Teilbaum hat eine Wurzel, die nur ein Kind hat. Lässt man diese Wurzel weg, ist der Rest wieder ein Suchbaum, der das gleiche Intervall, nur ohne das Datum x , beschreibt. Dieser Teil bleibt, wenn v linkes Kind seines Vorgängers war, auf der linken Seite des Vorgängers von v und damit auf der richtigen Seite. Im Fall (3) sind die Daten x und y in der vollständig geordneten Liste der Daten im Suchbaum nach Konstruktion benachbart. Die Vertauschung von x und y bringt also nur dieses Paar in Unordnung. Das Datum y steht zu allen anderen Daten in der richtigen Beziehung. Nachdem x entfernt worden ist, ist die lokale Unordnung beseitigt und alles in Suchbaumordnung.

3.4 2–3–Bäume

Die weiteren Bemühungen gehen nun dahin, den worst case zu verbessern, also zu sichern, dass keine Bäume mit großer Tiefe entstehen. Dazu sollen die Bäume ziemlich gut balanciert sein. Die Klasse der 2–3–Bäume geht auf Hopcroft (1970) zurück.

Definition 3.4.1: Ein Baum heißt 2–3–Baum, wenn die folgenden Eigenschaften erfüllt sind.

1. Jeder Knoten enthält ein oder zwei Daten.
2. Knoten mit einem Datum haben zwei Zeiger, wobei im linken (rechten) Teilbaum nur kleinere (größere) Daten als x enthalten sind.
3. Knoten mit zwei Daten x und y , wobei $x < y$ ist, haben drei Zeiger auf Teilbäume. Die Daten im ersten Teilbaum sind kleiner als x , im zweiten Teilbaum größer als x und kleiner als y , im dritten Teilbaum größer als y .
4. Die Zeiger, die einen Knoten verlassen, sind entweder alle nil-Zeiger oder alle keine nil-Zeiger. Im ersten Fall heißt der Knoten Blatt.
5. Alle Blätter haben gleiche Tiefe.

Zunächst einmal ist nicht klar, dass es für jedes n 2–3–Bäume mit n Daten gibt. Da wir jedoch die Operation INSERT auf der Klasse der 2–3–Bäume implementieren werden, folgt auch die Existenz derartiger Bäume. Jeder Knoten muss Platz für 2 Daten und 3 Zeiger haben. Da der Knoten unter Umständen nur 1 Datum und 2 Zeiger enthält, wird gegenüber binären Suchbäumen Speicherplatz verschenkt. Dafür ist es relativ leicht zu zeigen, dass 2–3–Bäume geringe Tiefe haben. Es ist allerdings zu bedenken, dass in einem Knoten ein zu suchendes Datum eventuell mit zwei Daten verglichen werden muss.

Lemma 3.4.2: Es sei d die Tiefe eines 2–3–Baumes mit n Daten. Dann gilt

$$\lceil \log_3(n+1) \rceil - 1 \leq d \leq \lfloor \log_2(n+1) \rfloor - 1.$$

Beweis: Die größte Datenzahl auf $d+1$ Ebenen, also bei Tiefe d , wird von vollständigen ternären Bäumen erreicht. Die Zahl der Daten beträgt dann $2(1 + 3 + \dots + 3^d) = 3^{d+1} - 1$. Sei nun d die Tiefe eines 2–3–Baumes mit n Daten. Dann gilt

$$3^{d+1} - 1 \geq n \text{ und } d \geq \lceil \log_3(n+1) \rceil - 1.$$

Die kleinste Datenzahl auf $d+1$ Ebenen wird von vollständigen binären Bäumen erreicht. Die Zahl der Daten beträgt dann $1 + 2 + \dots + 2^d = 2^{d+1} - 1$. Sei nun d die Tiefe eines 2–3–Baumes mit n Daten. Dann gilt

$$2^{d+1} - 1 \leq n \text{ und } d \leq \lfloor \log_2(n+1) \rfloor - 1.$$

□

Wir wollen nun SEARCH, INSERT und DELETE so implementieren, dass ihre Laufzeit proportional zur Baumtiefe und damit $O(\log n)$ ist.

SEARCH(x): Wir starten an der Wurzel. Es werde der Knoten v mit dem Datum y und eventuell dem Datum z erreicht. Falls x mit einem Datum in v übereinstimmt, wird die Suche erfolgreich beendet. Es sei $y < z$, falls z existiert. Falls $x < y$ ist, wird der erste oder linke Teilbaum aufgesucht. Falls $y < x$ ist und z nicht existiert, wird der zweite und damit rechte Teilbaum aufgesucht. Falls $y < x < z$ ist, wird der zweite oder mittlere Teilbaum aufgesucht. Falls $z < x$ ist, wird der dritte oder rechteste Teilbaum aufgesucht. Wird ein nil-Zeiger erreicht, wird die Suche erfolglos beendet. In jedem Fall wird der Suchpfad auf einem Stack abgespeichert.

INSERT(x) (nach erfolgloser Suche): Wir sind auf den nil-Zeiger gestoßen, wo x „eigentlich hingehört“. Zum besseren Verständnis werden wir im Folgenden an den Endzeigern in unseren Baumausschnitten die Bereiche angeben, auf die dieser Zeiger deutet. Diese Bereiche kommen in den Programmen oder den Datenstrukturen nicht vor. Der Knoten, von dem der gefundene nil-Zeiger ausgeht, enthält y und eventuell $y' > y$. Es seien z und z' die Daten im Baum, die in der vollständig geordneten Liste direkt vor bzw. direkt hinter y' bzw. y stehen. Wir bilden einen Knoten mit dem Datum x und zwei Zeigern auf die Bereiche (v, x) und (x, w) , wobei dies die Nachbarn in der geordneten Liste mit x sind.

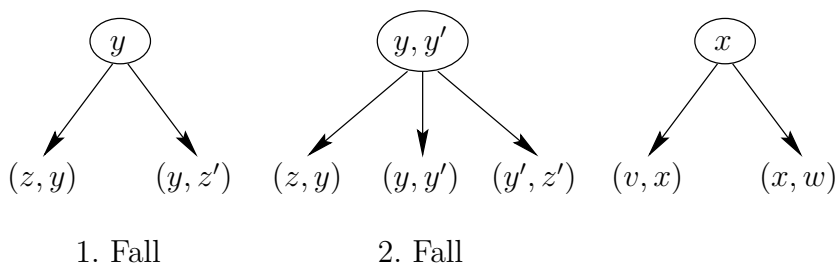


Abbildung 3.4.1: Die Ausgangssituationen für das Einfügen von x .

Wir betrachten den 1. Fall. Falls $x < y$, sind wir auf den (z, y) -Zeiger gestoßen. Der Bereich (z, y) wird in die Bereiche (z, x) und (x, y) aufgeteilt. Das Datum x wird im

Knoten mit Datum y vorne eingefügt. Der Knoten enthält drei Zeiger auf (z, x) , (x, y) und (y, z') . Falls $x > y$, sind wir auf den (y, z') -Zeiger gestoßen. Dieser Bereich wird aufgeteilt, das Datum x wird in den Knoten hinten eingefügt, der Knoten hat drei Zeiger auf (z, y) , (y, x) und (x, z') . In beiden Fällen erhalten wir einen 2-3-Baum. Wir betrachten die Zeigeroperationen genauer. Der Zeiger, auf den wir bei der Suche gestoßen sind, fällt weg, die beiden Zeiger an x werden übernommen. In beiden Unterfällen erhalten wir so die richtigen Zeiger.

Im 2. Fall ist $x < y$, $y < x < y'$ oder $y' < x$. Die beiden Zeiger an x zeigen auf folgende Bereiche. (z, x) und (x, y) , (y, x) und (x, y') bzw. (y', x) und (x, z') . In jedem Fall ersetzen wir den Zeiger, den wir auf dem Suchpfad erreicht haben, durch die beiden Zeiger an x . Wenn wir gedanklich x im Knoten an der passenden Stelle einfügen, erhalten wir einen Knoten mit drei Daten und vier Zeigern. Bei Verallgemeinerung von Definition 3.4.1 hat dieser Knoten die Suchbaumeigenschaften. Es sind alle Blätter auf dem gleichen Niveau, nur „platzt“ ein Blatt. Daher beenden wir unser Gedankenspiel. Wir bilden drei Knoten, der mit dem mittleren Datum zeigt auf die beiden anderen Knoten, der mit dem kleineren Datum erhält die ersten beiden Zeiger des geplatzten Knotens und der mit dem großen Datum die restlichen beiden Zeiger des geplatzten Knotens. Der Zeiger auf den früheren Knoten mit y und y' zeigt nun auf den Knoten mit dem mittleren Datum. Ein Blick auf alle drei Unterfälle zeigt, dass die Suchbaumeigenschaften erfüllt sind.

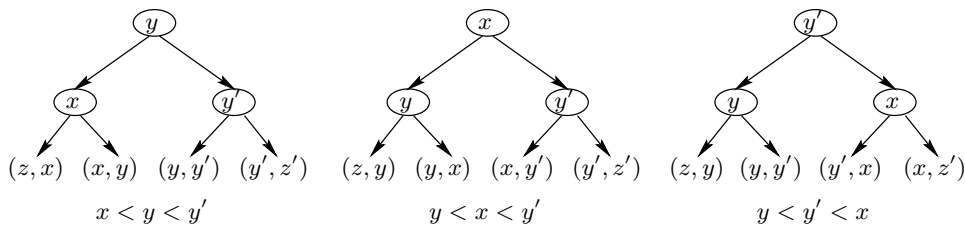


Abbildung 3.4.2: Das Einfügen von x in einen Knoten mit zwei Daten.

Leider haben wir nur ein Problem durch ein anderes ersetzt. Der Teilbaum mit der Wurzel y (x bzw. y') in den drei Fällen von Abbildung 3.4.2 ist um eine Ebene zu tief.

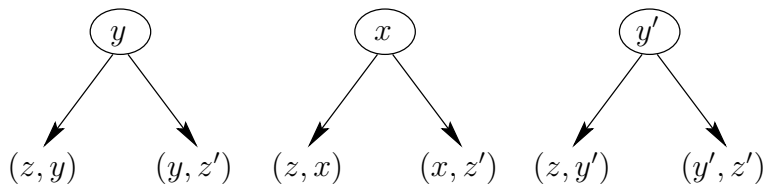
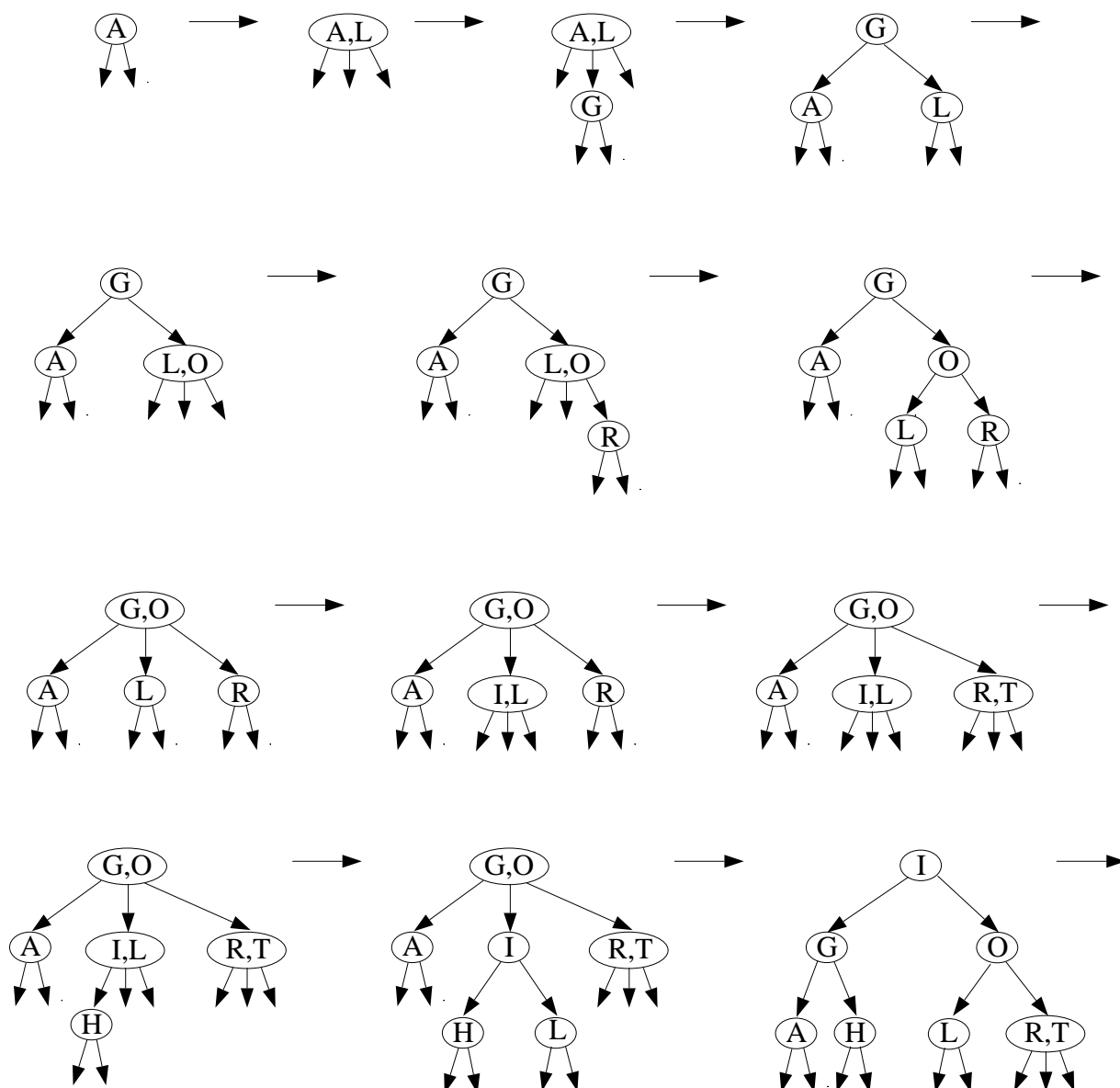


Abbildung 3.4.3: Eine Abstraktion der Situation in Abbildung 3.4.2.

Abbildung 3.4.3 ist eine Abstraktion von Abbildung 3.4.2 und beschreibt wieder die Ausgangssituation. Als wir den neuen Knoten x mit dem Zeiger auf die Bereiche (v, x) und (x, w) gebildet hatten und den Zeiger, den wir auf dem Suchpfad gefunden hatten, durch einen Zeiger auf x ersetzt hatten, war der Teilbaum mit der Wurzel um eine Ebene zu

tief. Wir nehmen also den nächsten Knoten und Zeiger vom Stack, der unseren Suchpfad enthält, und machen analog weiter, bis wir einmal im 1. Fall landen oder die Wurzel verarbeitet haben. Im letzteren Fall ist der Teilbaum, der um eine Ebene zu tief ist, der ganze Baum. Dann ist der Baum aber nicht um eine Ebene zu tief, sondern die Tiefe des Baumes ist um 1 gewachsen. 2–3-Bäume wachsen also über die Wurzel. Offensichtlich ist die Rechenzeit für INSERT proportional zur Tiefe des Baumes.

In Abbildung 3.4.4 fügen wir die Daten *A*, *L*, *G*, *O*, *R*, *I*, *T*, *H*, *M*, *U*, *S* in dieser Reihenfolge in einen leeren 2–3-Baum ein.



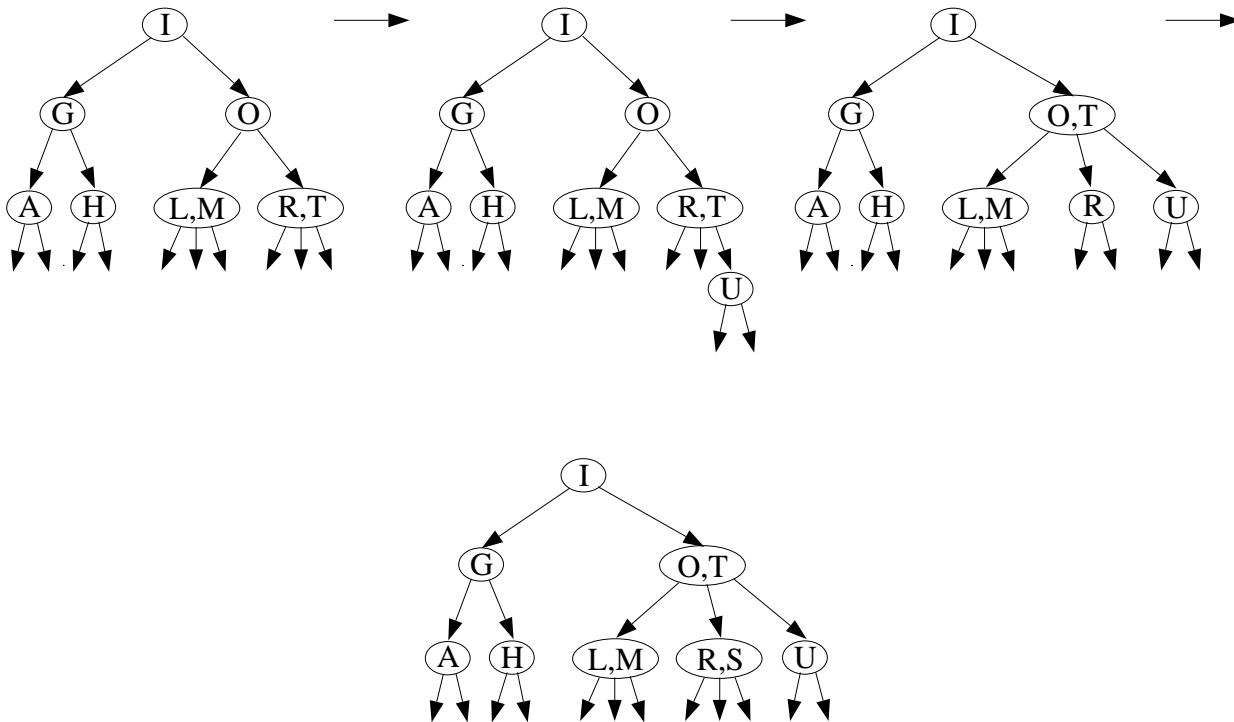


Abbildung 3.4.4: Der Aufbau eines 2–3–Baumes durch Einfügen von Informationen.

DELETE(x) (nach erfolgreicher Suche):

- (1) Wenn x nicht in einem Blatt steht, wird das größte Datum y mit $y < x$ aufgesucht. Falls x einziges oder kleinstes Datum in seinem Knoten ist, wird zunächst der erste Zeiger und sonst der zweite Zeiger gewählt. Danach wird immer der letzte Zeiger gewählt, bis ein nil-Zeiger erreicht wird. Dann ist y das größte Datum in dem zuletzt besuchten Knoten. Auch dieser Pfad wird auf dem Stack für den Suchpfad abgespeichert. Es werden x und y ausgetauscht. Nun muss ein Datum in einem Blatt gelöscht werden. Die Korrektheit dieses Vorgehens folgt analog zu Kapitel 3.3.
- (2) Enthält der Knoten mit Datum x noch ein weiteres Datum, werden x und ein nil-Zeiger entfernt, und wir sind fertig. Ansonsten enthält der Knoten nur x . Der Knoten wird freigegeben, der Zeiger auf diesen Knoten wird nil-Zeiger. Wir sind noch nicht fertig, da der „Teilbaum“, auf den dieser nil-Zeiger weist, eine Ebene zu hoch hängt.
- (3) Wir beschreiben eine allgemeinere Situation, von der die in (2) im zweiten Fall konstruierte Situation ein Spezialfall ist. Es gibt einen Zeiger p , der auf einen Teilbaum (eventuell nil) zeigt, dessen Blätter und nil-Zeiger jeweils auf einer Ebene liegen, aber eine Ebene höher als die restlichen Blätter und nil-Zeiger. Wir betrachten den Knoten v , von dem p ausgeht. Wenn es v nicht gibt, hängt der gesamte Baum eine Ebene zu hoch. Dann ist aber die Tiefe des Baumes um 1 gefallen. 2–3–Bäume schrumpfen also auch über die Wurzel. In diesem Fall können wir ebenfalls stoppen. Wir nehmen also an, dass v existiert. Den Knoten v verlässt mindestens ein weiterer

Zeiger. Wir wählen einen zu p benachbarten Zeiger q . O.B.d.A. sei q rechts von p , der andere Fall verläuft analog. Die Zeiger p und q werden in v durch das Datum x getrennt. Der Zeiger q zeigt auf den Knoten w .

1. Fall: w enthält zwei Daten y und z mit $y < z$ und daher drei Zeiger q_1, q_2, q_3 . Wir ersetzen x durch y . Die beiden Zeiger, die y nun in v umrahmen, zeigen auf Knoten mit dem Datum x bzw. z . Die beiden Knoten erhalten von links nach rechts die Zeiger p, q_1, q_2 und q_3 .

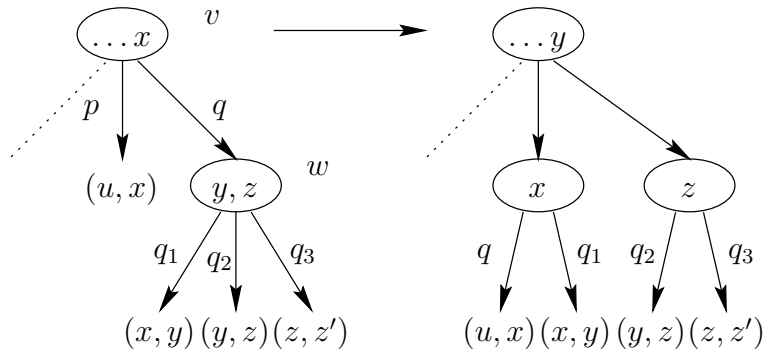


Abbildung 3.4.5: Die Rebalancierung durch Datenrotation.

Wir beachten, dass die Zeiger weiterhin auf die passenden Suchbereiche zeigen. In diesem Fall können wir stoppen, da wir wieder einen 2-3-Baum erhalten haben.

2. Fall: w enthält nur ein Datum y und daher zwei Zeiger q_1 und q_2 .

1. Unterfall: v enthält neben x noch ein Datum x' , o.B.d.A. $x' < x$. Wir bilden einen neuen Knoten mit x und y . Dadurch enthält v nur noch x' , der erste Zeiger bleibt unverändert, der zweite Zeiger ersetzt p und q und zeigt auf den neuen Knoten, der p, q_1 und q_2 als Zeiger enthält. Wir beachten, dass die Zeiger weiterhin auf die passenden Suchbereiche zeigen. Wieder können wir stoppen.

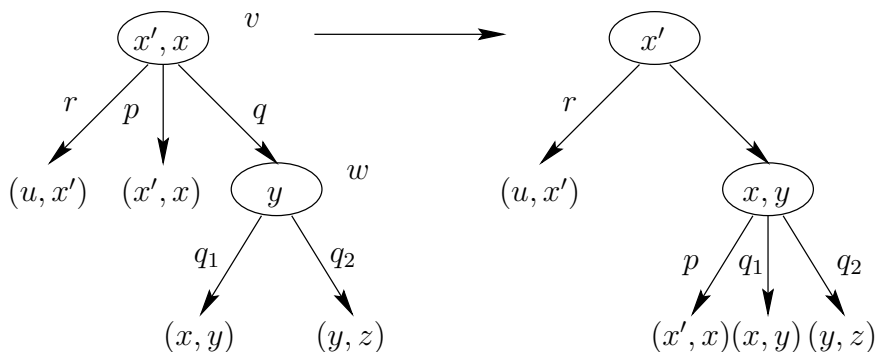


Abbildung 3.4.6: Eine weitere Rebalancierung durch Datenrotation.

2. Unterfall: v enthält nur x . Wir bilden einen neuen Knoten mit den Daten x und y , sowie mit den Zeigern p , q_1 , q_2 . Der bisher auf v gerichtete Zeiger zeigt nun auf den neuen Knoten.

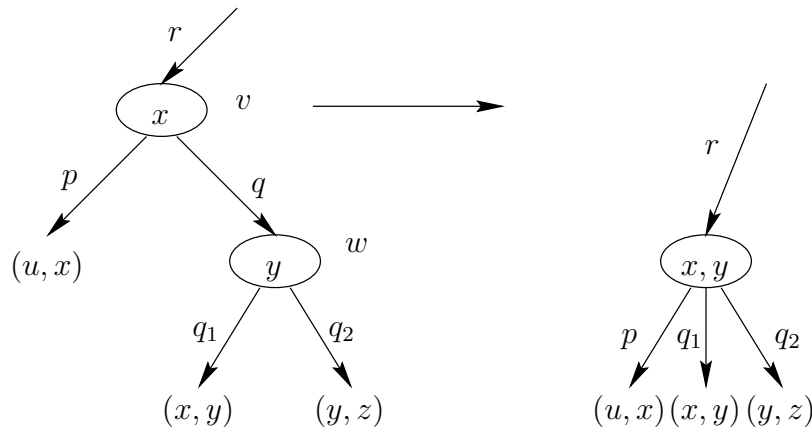
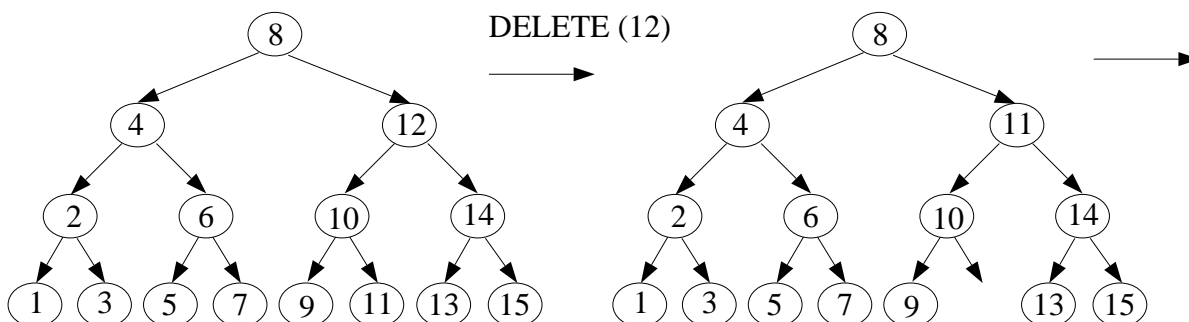


Abbildung 3.4.7: Ein Verschmelzungsschritt bei der Rebalancierung.

Es ist klar, dass die Zeiger weiterhin auf die passenden Suchbereiche zeigen. Wir können nicht stoppen, da der Teilbaum, auf den r zeigt, eine Ebene zu wenig hat. Wir sind in der in (3) beschriebenen Situation und fahren analog mit dem Zeiger r und dem Knoten, von dem dieser Zeiger kommt, fort.

Offensichtlich ist die Rechenzeit von $\text{DELETE}(x)$ proportional zur Tiefe des Baumes. Wir haben in (3) einen beliebigen Nachbarn gewählt. Wenn es zwei Nachbarn gibt und einer ein Datum und der andere zwei Daten enthält, ist es günstiger, den Nachbarn mit zwei Daten zu wählen, da wir dann in den 1. Fall kommen und auf jeden Fall stoppen können.

In Abbildung 3.4.8 wird die DELETE -Prozedur beispielhaft durchgeführt. Bei freier Wahl wählen wir den Nachbarn mit zwei Daten.



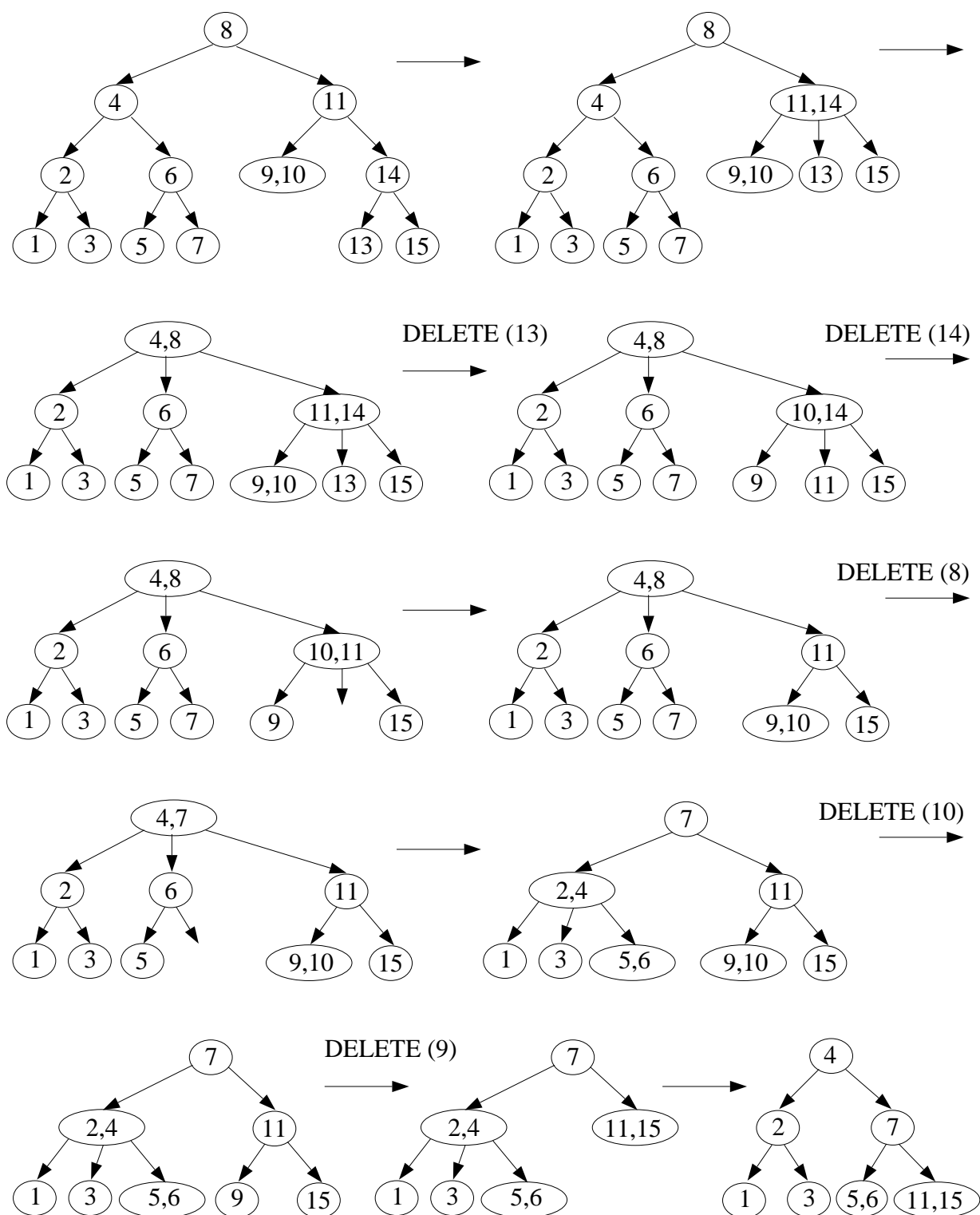


Abbildung 3.4.8: Beispiele zur Entfernung von Daten aus 2-3-Bäumen.

Satz 3.4.3: Die Operationen SEARCH, INSERT und DELETE können in 2-3-Bäumen in Zeit $O(\log n)$ ausgeführt werden.

2–3-Bäume haben den großen Vorteil, weitere Operationen zu unterstützen. Dafür ist es jedoch günstiger, wenn alle Daten in den Blättern gespeichert sind. In den inneren Knoten speichern wir das größte Datum des linken Teilbaumes und, falls vorhanden, das größte Datum des mittleren Teilbaumes. Außerdem verwalten wir für jeden Baum T seine Tiefe $d(T)$. Wir diskutieren hier nicht die Modifikationen, die für die Operationen SEARCH, INSERT und DELETE nötig sind.

CONCATENATE(T_1, T_2, T): Konstruiere aus den 2–3-Bäumen T_1 und T_2 , wobei alle Daten in T_1 kleiner als alle Daten in T_2 sind, den 2–3-Baum T , der die Daten aus T_1 und T_2 enthält.

1. Fall: $d(T_1) = d(T_2)$. Der Baum T besteht aus einer Wurzel, die als neuer Knoten das größte Datum aus T_1 enthält. Dieser Knoten enthält T_1 und T_2 als Teilbäume. Da das größte Datum aus T_1 gebraucht wird, verwalten wir das größte Datum jedes Baumes extra. Das größte Datum in T ist das größte Datum in T_2 . Mit dieser erweiterten Datenstruktur genügt für CONCATENATE in diesem Fall Zeit $O(1)$.

2. Fall: $d(T_1) > d(T_2)$ (der 3. Fall $d(T_1) < d(T_2)$ verläuft analog). Wir bilden einen Baum (nicht 2–3-Baum) T'_2 , der als Wurzel einen Knoten ohne Datum enthält. Dieser Knoten erhält (vorerst) nur einen Zeiger, der auf die Wurzel von T_2 zeigt. In T_1 gehen wir $d(T_1) - d(T_2) - 1$ Schritte, wobei wir immer den rechtesten Zeiger benutzen. Wir erreichen dann einen Knoten v .

1. Unterfall: v enthält nur ein Datum und zwei Zeiger.

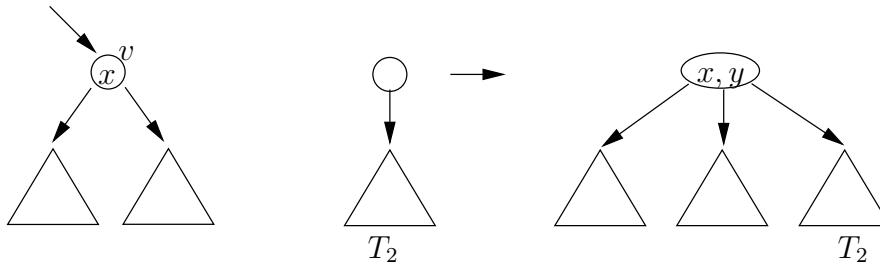


Abbildung 3.4.9: Illustration eines Unterfalls bei CONCATENATE.

Dabei ist x die Information aus v und y sei das größte Datum aus T_1 , das wir an der Wurzel von T_1 finden. Wir verschmelzen v und den Knoten ohne Daten wie in Abbildung 3.4.9. Das größte Datum des Baumes T , dessen Wurzel die alte Wurzel von T_1 ist, ist gleich dem größten Datum aus T_2 .

2. Unterfall: v enthält zwei Daten und drei Zeiger.

Es sei z das größte Datum aus T_1 . Wir konstruieren den neuen Baum wie in Abbildung 3.4.10 gezeigt. Der Teilbaum, dessen Wurzel y enthält, hat nur leider die Eigenschaft, dass alle Blätter eine Ebene zu tief liegen. Diese Situation kennen wir aber bereits aus dem INSERT-Algorithmus. Auf gleiche Weise wie dort konstruieren wir aus unserem Baum einen 2–3-Baum.

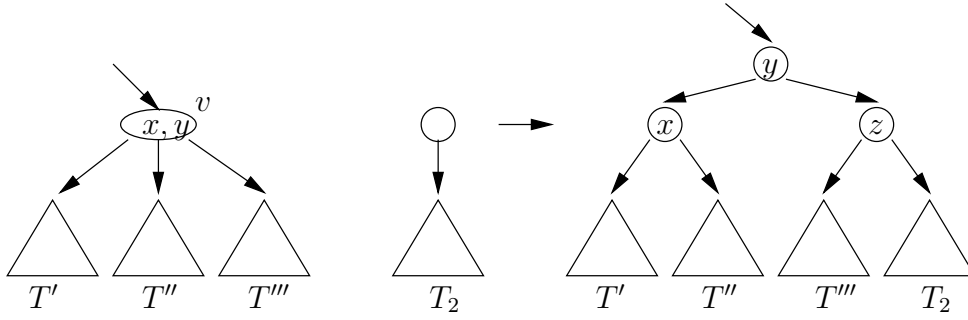


Abbildung 3.4.10: Illustration eines anderen Unterfalls bei CONCATENATE.

Die Rechenzeit im 2. Fall beträgt $O(d(T_1) - d(T_2))$.

Satz 3.4.4: Die Prozedur CONCATENATE kann für 2–3-Bäume T_1 und T_2 in Zeit $O(|d(T_1) - d(T_2)| + 1)$ durchgeführt werden.

SPLIT(T, a, T_1, T_2): Aus einem 2–3-Baum T , der a enthält, sollen zwei 2–3-Bäume konstruiert werden, wobei T_1 alle Daten $x \leq a$ und T_2 alle Daten $x > a$ enthält.

Zunächst führen wir SEARCH(a) durch und finden a in einem Blatt. Auf diesem Weg speichern wir folgende Informationen ab: den Suchpfad, alle Kinder von Knoten auf dem Suchpfad, die links vom Suchpfad liegen, und alle Kinder von Knoten auf dem Suchpfad, die rechts vom Suchpfad liegen. Die Daten des Blattes, das a enthält, werden ebenfalls abgespeichert, wobei a als links vom Suchpfad definiert wird.

Mit \odot bzw. $\boxed{\circ}$ haben wir in Abbildung 3.4.11 die abgespeicherten Knoten gekennzeichnet, die rechts bzw. links vom Suchpfad liegen. T_1 ist nun die Konkatenation aller Bäume, deren Wurzeln mit $\boxed{\circ}$ bezeichnet sind, gleiches gilt für T_2 und \odot . Wir behandeln aus Analogiegründen nur die Konstruktion von T_1 . Mit T_1^1, \dots, T_1^m bezeichnen wir die Teilbäume, aus denen wir T_1 zusammensetzen. So wie wir sie abgespeichert haben, gilt

$$d(T_1^i) \geq d(T_1^{i+1}).$$

Wir fangen vernünftigerweise mit den kleinen Bäumen an und lösen das Problem wie folgt:

Für $i = m - 1, \dots, 1$: CONCATENATE(T_1^i, T_1^{i+1}, T_1^i).

Schließlich ist $T_1 = T_1^1$.

Die Korrektheit dieses Verfahrens ist unmittelbar klar. Wir kommen zur Analyse der Rechenzeit. Wir zeigen durch Induktion über d' die folgende Behauptung. Die Konkatenation aller T_1^i mit $d(T_1^i) \leq d'$ ergibt einen 2–3-Baum der Tiefe $d' + 1$ oder kleiner. Für $d' = 0$ gilt dies offensichtlich, da wir höchstens zwei Knoten haben. Nun sei die Behauptung für d' korrekt. Wir erhalten also aus den Bäumen, deren Tiefe durch d' beschränkt ist, einen Baum, dessen Tiefe durch $d' + 1$ beschränkt ist. In der T_1^i -Folge gibt es nach Konstruktion höchstens zwei Bäume mit Tiefe $d' + 1$. Die Konkatenation von drei Bäumen, deren Tiefe durch $d' + 1$ beschränkt ist, führt zu einem Baum, dessen Tiefe durch $d' + 2$ beschränkt ist.



```

S := 0.45; P := 0.1; CPRINT := 0; n := 0.0; P := 0.0; T := T + 0.1; T := 0.1;

```

da eine Seite wesentlich mehr als drei Schlüsselwörter enthalten kann. Bayer-Bäume, oft kurz als B-Bäume bezeichnet, sind Verallgemeinerungen von 2-3-Bäumen, die auf diese Situation zugeschnitten sind.

Definition 3.5.1: Ein Baum heißt B-Baum der Ordnung m , wenn die folgenden Eigenschaften erfüllt sind.

1. Jeder Knoten mit Ausnahme der Wurzel enthält mindestens $\lceil m/2 \rceil - 1$ Daten. Jeder Knoten enthält höchstens $m - 1$ Daten. Die Daten sind sortiert.
2. Knoten mit k Daten x_1, \dots, x_k haben $k + 1$ Zeiger, die auf die Bereiche $(\cdot, x_1), (x_1, x_2), \dots, (x_{k-1}, x_k), (x_k, \cdot)$ zeigen.
3. Die Zeiger, die einen Knoten verlassen, sind entweder alle nil-Zeiger oder alle echte Zeiger.
4. Alle Blätter haben gleiche Tiefe.

Bemerkung 3.5.2: 2-3-Bäume sind B-Bäume der Ordnung 3.

Die Speicherplatzausnutzung in allen Knoten mit Ausnahme der Wurzel beträgt mindestens 50%. Die Sonderrolle der Wurzel ermöglicht B-Bäume mit weniger als $\lceil m/2 \rceil - 1$ Daten.

Bemerkung 3.5.3: Die Tiefe eines B-Baumes der Ordnung m mit n Daten liegt im Intervall $[\log_m(n + 1) - 1, \log_{\lceil m/2 \rceil}((n + 1)/2)]$.

Diese Bemerkung lässt sich analog zu Lemma 3.4.2 beweisen, wobei die Sonderrolle der Wurzel beachtet werden muss.

Satz 3.5.4: Die Operationen SEARCH, INSERT und DELETE lassen sich auf B-Bäumen T einer konstanten Ordnung m so implementieren, dass nur auf $O(d(T))$ Seiten je $O(1)$ -mal zugegriffen werden muss.

Beweis: Die Operation SEARCH muss nicht im Detail beschrieben werden. Bei Betrachtung des Knotens v wird x in die Daten x_1, \dots, x_k mit binärer Suche einsortiert. Dies erfordert ungefähr $\log m$ Vergleiche. Da die Baumtiefe ungefähr $\log_m n$ ist, kommen insgesamt wieder ungefähr $\log n$ Vergleiche zusammen. Wieder wird der Suchpfad abgespeichert.

INSERT(x) (nach erfolgloser Suche): Der gefundene nil-Zeiger zeigt auf einen neuen Knoten mit dem Datum x und zwei nil-Zeigern. Allgemein haben wir einen Zeiger auf einen Knoten mit einem Datum y , so dass die Blätter des zugehörigen Teilbaumes eine Ebene zu tief liegen. Falls nun der Knoten, zu dem dieser Zeiger gehört, nicht voll ist, also weniger als $m - 1$ Daten enthält, wird y dort eingefügt. Der Knoten erhält ein neues Datum, verliert den Zeiger auf y und erhält die beiden von y ausgehenden Zeiger, die den passenden Suchbereich abdecken. Falls der Elter voll ist, also $m - 1$ Daten enthält, haben wir es inklusive y mit m Daten zu tun. Die Daten seien $z_1 < \dots < z_m$. Wir bilden daraus drei Knoten und einen Teilbaum. Die Wurzel enthält $z_{\lceil m/2 \rceil}$ und zwei Zeiger auf

den linken Knoten mit $z_1, \dots, z_{\lceil m/2 \rceil - 1}$ und den rechten Knoten mit $z_{\lceil m/2 \rceil + 1}, \dots, z_m$, also $m - \lceil m/2 \rceil = \lfloor m/2 \rfloor \geq \lceil m/2 \rceil - 1$ Daten. Wir hatten m Zeiger, die den Elterknoten verlassen haben, und zwei Zeiger, die den Knoten mit Datum y verlassen haben. Diese beiden Zeiger ersetzen den Zeiger auf y . Diese $m + 1$ Zeiger werden von links nach rechts auf die beiden neuen Knoten verteilt, $\lceil m/2 \rceil$ für den linken Knoten und $\lfloor m/2 \rfloor + 1$ für den rechten Knoten. Die Suchbereiche passen zu den Zeigern. Wir gehen einen Schritt auf dem Suchpfad zurück und fahren dort analog fort. Wenn wir die Wurzel aufgespalten haben, sind wir fertig, da in der neuen Wurzel ein Datum erlaubt ist.

DELETE(x) (nach erfolgreicher Suche x): Wenn x nicht in einem Blatt abgelegt ist, wird x mit dem größten Datum $y < x$ vertauscht. Dieses Datum ist mit Sicherheit dann in einem Blatt abgelegt. Wir müssen also wieder nur die Situation betrachten, in der der Knoten v , der x enthält, ein Blatt ist.

Wir entfernen x und den zugehörigen nil-Zeiger. Wenn der Knoten dann noch genügend Daten enthält, sind wir fertig. Hat ein direkter Nachbar mindestens $\lceil m/2 \rceil$ Daten, genügt eine einfache Datenrotation, vergleiche Abbildung 3.5.1.

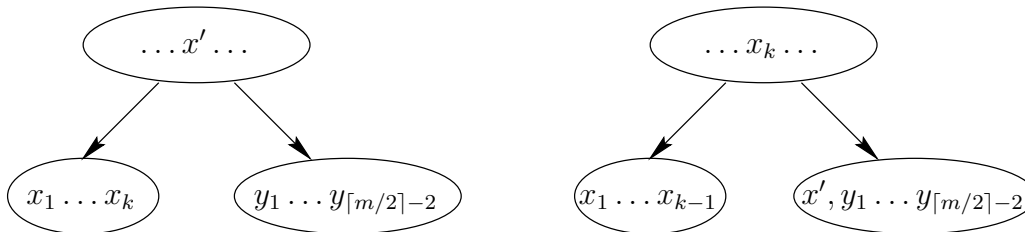


Abbildung 3.5.1: Eine Datenrotation.

Hierbei gibt der linke Nachbar auch seinen rechtesten Zeiger an seinen Nachbarn ab. Die symmetrische Situation lässt sich analog behandeln.

Ansonsten hat jeder direkte Nachbar $\lceil m/2 \rceil - 1$ Daten, wir betrachten o.B.d.A. den linken Nachbar. Es sei x' das Datum im Vorgänger, das die beiden betrachteten Knoten trennt. Wir ersetzen die beiden Zeiger auf die beiden Nachbarn durch einen Zeiger auf einen neuen Knoten mit den Daten $x_1, \dots, x_{\lceil m/2 \rceil - 1}$ des linken betrachteten Knoten, x' aus dem Vorgängerknoten und den Daten $y_1, \dots, y_{\lceil m/2 \rceil - 2}$ aus dem rechten betrachteten Knoten, zusammen $2\lceil m/2 \rceil - 2 \leq m - 1$ Daten. Der Knoten erhält die $\lceil m/2 \rceil + \lceil m/2 \rceil - 1$ Zeiger der beiden betrachteten Knoten.

Damit haben wir das Problem an den Vorgänger weitergegeben, aus dem nun x' entfernt werden muss. In der Wurzel ist das Entfernen eines Datums stets möglich. Im Extremfall bekommt die Wurzel 0 Daten und einen Zeiger auf die echte Wurzel des neuen Baumes. \square

3.6 AVL-Bäume

Wir kehren zu dynamischen Dateien, bei denen die Daten im Kernspeicher gehalten werden, zurück. 2–3-Bäume sind sehr effizient und erlauben außer den Standardoperationen

eine Reihe weiterer nützlicher Operationen. Einziger Nachteil ist die nur zu 50% gesicherte Speicherplatzausnutzung. Dies ist bei binären Suchbäumen anders. Adelson-Velskii und Landis (1962) haben gut balancierte binäre Suchbäume vorgestellt, die nach ihnen AVL-Bäume genannt werden.

Definition 3.6.1: Ein binärer Suchbaum heißt AVL-Baum, wenn für jeden Knoten gilt, dass sich die Tiefe des linken Teilbaumes und die Tiefe des rechten Teilbaumes um maximal 1 unterscheiden. Für den Knoten v mit den Teilbäumen T_l und T_r ist $\text{bal}(v) := d(T_l) - d(T_r)$ der Balancegrad von v .

In AVL-Bäumen gilt also $\text{bal}(v) \in \{-1, 0, +1\}$.

Satz 3.6.2: Es sei d die Tiefe eines AVL-Baumes mit n Daten. Dann gilt

$$\lceil \log(n+1) \rceil - 1 \leq d \leq \frac{1}{(\log 5)/2 - 1} \log n \approx 1,44 \log n.$$

Beweis: Die untere Schranke gilt sogar für alle binären Suchbäume.

Auch wenn sich die Tiefen der beiden Teilbäume jedes Knotens nur um 1 unterscheiden dürfen, heißt das nicht, dass alle Blätter nur auf zwei Ebenen liegen. Abbildung 3.6.1 illustriert diese Beobachtung.

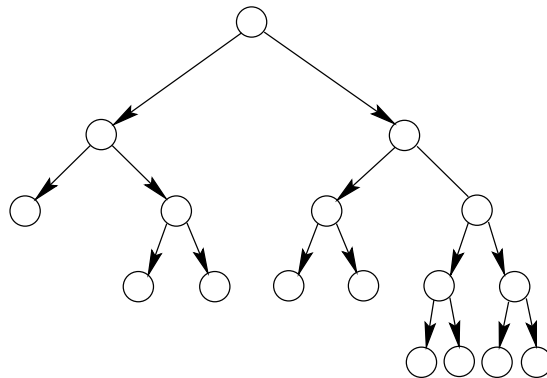


Abbildung 3.6.1: Ein AVL-Baum mit Blättern auf drei Ebenen.

Es ist nicht einfach, die Tiefe eines AVL-Baumes mit n Daten direkt nach oben abzuschätzen. Wir berechnen daher zunächst die minimale Anzahl $A(d)$ von Knoten eines AVL-Baumes der Tiefe d .

Offensichtlich ist $A(0) = 1$ und $A(1) = 2$. Für $d \geq 2$ muss ein AVL-Baum der Tiefe d eine Wurzel haben und ein Teilbaum muss Tiefe $d-1$ und damit mindestens $A(d-1)$ Knoten haben. Damit der Balancegrad der Wurzel im richtigen Bereich liegt, muss der andere Teilbaum mindestens Tiefe $d-2$ und damit mindestens $A(d-2)$ Knoten haben. Andererseits können wir auch einen AVL-Baum konstruieren, indem wir als Teilbäume AVL-Bäume der Tiefe $d-2$ und $d-1$ mit minimaler Knotenzahl benutzen. Es folgt für $d \geq 2$

$$A(d) = 1 + A(d-1) + A(d-2).$$

Wir erhalten also fast die Rekursionsgleichung für die Fibonaccizahlen. Es gilt

$$A(d) = \text{Fib}(d+2) - 1.$$

Diese Beziehung beweisen wir durch Induktion über d . Es ist $A(0) = 1 = 2 - 1 = \text{Fib}(2) - 1$ und $A(1) = 2 = 3 - 1 = \text{Fib}(3) - 1$. Schließlich gilt nach Induktionsvoraussetzung

$$A(d) = 1 + A(d-1) + A(d-2) = 1 + \text{Fib}(d+1) - 1 + \text{Fib}(d) - 1 = \text{Fib}(d+2) - 1.$$

Sei nun T ein AVL-Baum mit n Daten und Tiefe d . Dann ist $\text{Fib}(d+2) - 1 = A(d) \leq n$. Folgende Gleichung für die Fibonaccizahlen ist bekannt:

$$\text{Fib}(k) = \frac{1}{\sqrt{5}} \left[\left(\frac{\sqrt{5}+1}{2} \right)^{k+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{k+1} \right] \geq \frac{1}{\sqrt{5}} \left[\left(\frac{\sqrt{5}+1}{2} \right)^{k+1} - 1 \right]$$

Aus $\text{Fib}(d+2) \leq n+1$ folgt also

$$\left(\frac{\sqrt{5}+1}{2} \right)^{d+3} \leq \sqrt{5}(n+1) + 1$$

und

$$d+3 \leq \frac{1}{\log((\sqrt{5}+1)/2)} \log(\sqrt{5}(n+1) + 1).$$

Hieraus folgt nach Betrachtung der „kleineren“ Terme die Behauptung. \square

Wir kommen nun zu den Operationen SEARCH, INSERT und DELETE. Die Operation SEARCH verläuft wie in allgemeinen binären Suchbäumen. Dies gilt zunächst auch für INSERT und DELETE. Allerdings kann sich der Balancegrad aller Knoten auf dem Pfad von der Wurzel bis zu dem Knoten, wo die Veränderung stattfindet, um 1 verändert haben. Dies müssen wir korrigieren und, falls der Balancegrad +2 oder -2 wurde, eine Rebalancierung durch Rotation herbeiführen.

Wir betrachten zunächst Einfügungen. Wir starten unsere Betrachtungen an dem neuen Knoten mit dem neuen Datum. Dieser Knoten erhält den Balancegrad 0. Wir werden den Suchpfad rückwärts durchlaufen, bis die Rebalancierung abgeschlossen ist. Im Allgemeinen erreichen wir den Knoten v . O.B.d.A. nehmen wir an, dass das rechte Kind x von v auf dem Suchpfad liegt (der andere Fall kann analog behandelt werden). In diesem Fall, das wird sich zeigen, wird die Tiefe des rechten Teilbaumes um 1 gewachsen sein. Dies gilt am Anfang, wenn wir den Elter des neuen Knoten erreichen. Wir unterscheiden die drei möglichen Werte von $\text{bal}(v)$.

1. Fall: $\text{bal}(v) = 1$. Wir korrigieren den Balancegrad und setzen $\text{bal}(v) = 0$. Da die Tiefe des Teilbaumes mit Wurzel v nicht verändert wurde, kann die Rebalancierung beendet werden.

2. Fall: $\text{bal}(v) = 0$. Wir korrigieren den Balancegrad und setzen $\text{bal}(v) = -1$. Da die Tiefe des Teilbaumes mit Wurzel v um 1 gewachsen ist, ist die Rebalancierung nur abgeschlossen, wenn v Wurzel des Baumes ist. Ansonsten setzen wir die Rebalancierung am Elter von v fort.

3. Fall: $\text{bal}(v) = -1$. Der Knoten ist außer Balance, aktuell gilt $\text{bal}(v) = -2$.

Insbesondere ist das rechte Kind x von v Wurzel eines Teilbaumes der Tiefe $d \geq 1$. Wir bezeichnen mit w den Nachfolger von x auf dem Suchpfad. Sowohl der Teilbaum mit Wurzel x als auch der Teilbaum mit Wurzel w ist in der Tiefe um 1 gewachsen. Wir unterscheiden nun, ob w rechtes oder linkes Kind von x ist.

1. Unterfall: w ist rechtes Kind von x .

In diesem Fall kommt das Problem „von außen“ und es reicht eine einfache Rotation, in unserem Fall (x ist rechtes Kind von v und w rechtes Kind von x) eine Linksrotation, vergleiche Abbildung 3.6.2.

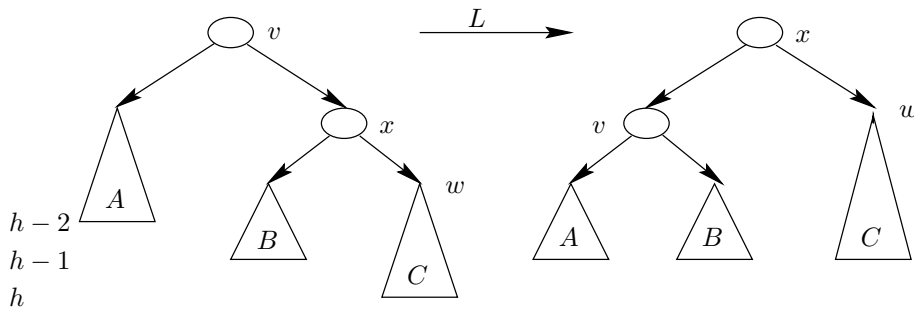


Abbildung 3.6.2: Eine Linksrotation.

Da die Einfügung eines Datums nur die Tiefe eines Teilbaumes erhöht, liegen die tiefsten Blätter von B auf Ebene $h - 1$. Es müssen zwei Balancewerte neu gesetzt werden:

$$\text{bal}(x) := 0 \text{ und } \text{bal}(v) := 0.$$

Da die Tiefe des betrachteten Teilbaumes genauso groß wie vor der Einfügung ist, ist die Rebalancierung abgeschlossen.

2. Unterfall: w ist linkes Kind von x .

In diesem Fall kommt das Problem „von innen“ und es wird eine doppelte Rotation ausgeführt, in unserem Fall (x ist rechtes Kind von v und w linkes Kind von x) eine Rechts-Links-Rotation.

In Abbildung 3.6.3 ist angenommen worden, dass $d(C) = d(B) + 1$ ist. In diesem Fall sind die Balancewerte vor der Doppelrotation

$$\text{bal}(v) = -2, \text{bal}(x) = 1, \text{bal}(w) = -1.$$

Die neuen Werte sind

$$\text{bal}(v) := 1, \text{bal}(x) := 0, \text{bal}(w) := 0.$$

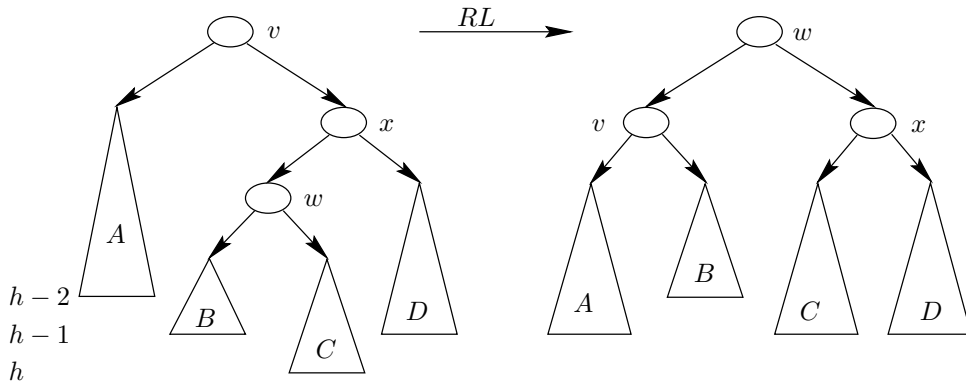


Abbildung 3.6.3: Eine Rechts-Links-Rotation.

Da die Tiefe des betrachteten Teilbaumes genauso groß wie vor der Einfügung ist, ist die Rebalancierung abgeschlossen. Natürlich kann auch $d(B) = d(C) + 1$ sein. Dann wird analog vorgegangen. Aus den alten Balancewerten

$$\text{bal}(v) = -2, \text{bal}(x) = 1, \text{bal}(w) = 1$$

wird

$$\text{bal}(v) := 0, \text{bal}(x) := -1, \text{bal}(w) := 0.$$

Wiederum ist die Rebalancierung abgeschlossen. Es gibt noch den Sonderfall, dass w der neu eingefügte Knoten ist. Dann sind alle Teilbäume A, B, C und D leer. Aus den alten Balancewerten

$$\text{bal}(v) = -2, \text{bal}(x) = 1, \text{bal}(w) = 0$$

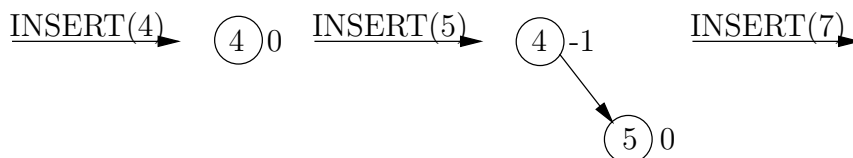
wird

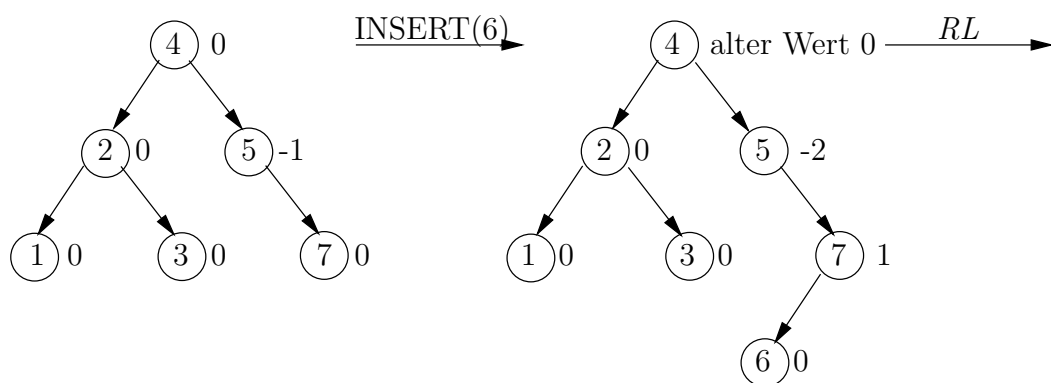
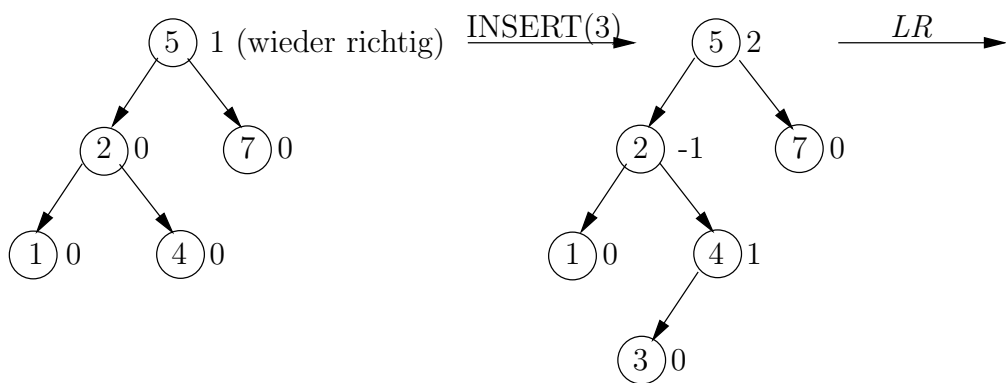
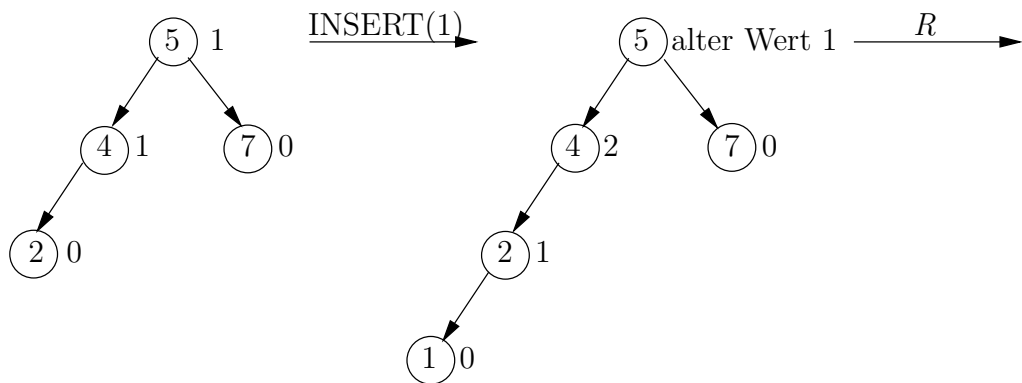
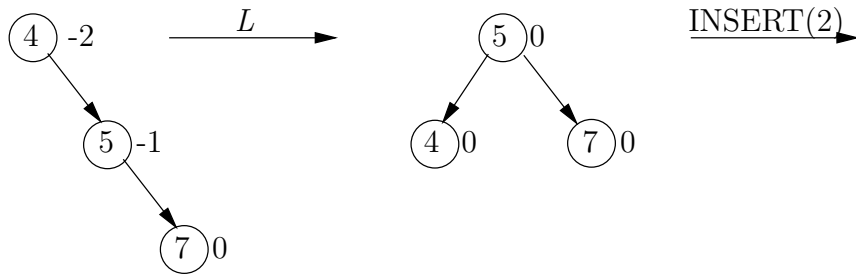
$$\text{bal}(v) := 0, \text{bal}(x) := 0, \text{bal}(w) := 0.$$

Auch in diesem Fall ist die Rebalancierung abgeschlossen. Nur bei der Berechnung der neuen Balancewerte müssen wir aufpassen. Welcher der drei „Unter-Unter-Fälle“ eintritt, erkennen wir am alten Wert von $\text{bal}(w)$.

Bei der Einfügung kann es nötig sein, den ganzen Suchpfad zurückzulaufen. Die Rebalancierung ist allerdings spätestens nach der ersten Rotation oder Doppelrotation abgeschlossen.

In Abbildung 3.6.4 werden die Daten 4, 5, 7, 2, 1, 3, 6 in einen leeren AVL-Baum eingefügt. Bei dieser Reihenfolge treten alle vier Typen von Rotationen auf. Neben den Knoten wird der Balancegrad vermerkt.





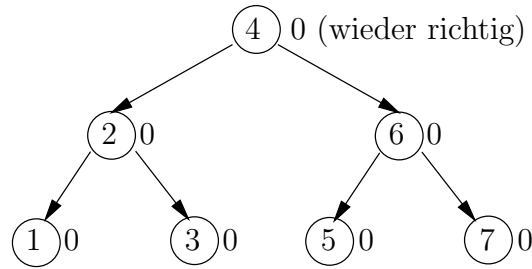


Abbildung 3.6.4: Der Aufbau eines AVL-Baumes durch Einfügungen.

Wir kommen nun zur Operation DELETE. Es sei ein Datum, o.B.d.A. im linken Teilbaum von v , entfernt worden und die Tiefe dieses Teilbaumes um 1 gesunken. War $\text{bal}(v) = 1$, ist nun $\text{bal}(v) = 0$ und die Tiefe des Baumes ist um 1 gesunken. Die Rebalancierung muss am Elter fortgesetzt werden. War $\text{bal}(v) = 0$, ist nun $\text{bal}(v) = -1$. Da die Tiefe des Baumes mit Wurzel v unverändert geblieben ist, ist die Rebalancierung abgeschlossen. Es bleibt also der Fall zu betrachten, dass $\text{bal}(v) = -1$ war und nun $\text{bal}(v) = -2$ ist. Sei x das rechte Kind von v und, falls existent, w das linke Kind von x .

Auch hier unterscheiden wir, ob der rechte Teilbaum von x zwei Ebenen tiefer als der linke Teilbaum von v endet (das Problem kommt „von außen“, wobei in diesem Fall der linke Teilbaum von x die gleiche Tiefe wie der rechte haben darf) oder ob nur der linke Teilbaum von x zwei Ebenen tiefer als der als der linke Teilbaum von v endet (das Problem kommt „von innen“).

Wenn das Problem „von außen“ kommt, genügt eine einfache Rotation, hier eine Linksrotation. Die Situation ist in Abbildung 3.6.5 dargestellt, wobei die gestrichelte Linie im Baum B andeutet, dass dieser Baum eine Ebene weniger haben kann.

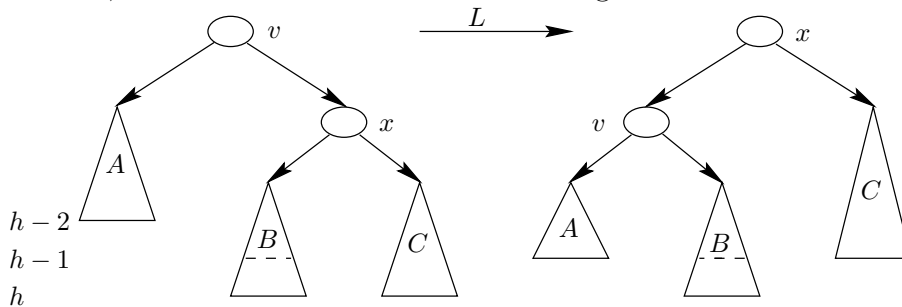


Abbildung 3.6.5: Der Fall einer einfachen Rotation bei der Prozedur DELETE.

Für die Änderung der Balancegrade gibt es zwei Fälle, abhängig von der Tiefe von B :

- B endet auf Ebene h , alte Werte: $\text{bal}(v) = -2, \text{bal}(x) = 0$, neue Werte: $\text{bal}(v) := -1, \text{bal}(x) := 1$. Die Tiefe des betrachteten Teilbaumes ist die gleiche wie vor der Operation DELETE. Die Rebalancierung ist abgeschlossen.
- B endet auf Ebene $h - 1$, alte Werte: $\text{bal}(v) = -2, \text{bal}(x) = -1$, neue Werte: $\text{bal}(v) := 0, \text{bal}(x) := 0$. Die Tiefe des betrachteten Teilbaumes ist gegenüber der

Situation vor der Operation DELETE um 1 gesunken. Die Rebalancierung muss am Elter (falls existent) fortgesetzt werden.

Wenn das Problem „von innen“ kommt, existiert w und wir führen eine Doppelrotation, hier eine Rechts-Links-Rotation, durch. Die Situation ist in Abbildung 3.6.6 dargestellt, wobei die gestrichelten Linien in den Bäumen B und C andeuten, dass einer dieser Bäume eine Ebene weniger haben kann.

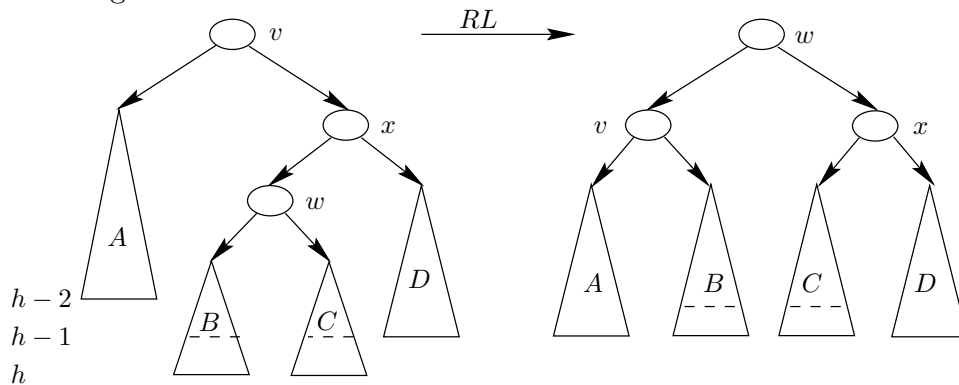


Abbildung 3.6.6: Der Fall einer Doppelrotation bei der Prozedur DELETE.

Für die Änderung der Balancegrade gibt es drei Fälle, abhängig von den Tiefen von B und C :

- B endet auf Ebene h und C auf Ebene $h - 1$, alte Werte: $\text{bal}(v) = -2$, $\text{bal}(x) = 1$, $\text{bal}(w) = 1$, neue Werte: $\text{bal}(v) := 0$, $\text{bal}(w) := 0$, $\text{bal}(x) := -1$.
- B und C enden auf Ebene h , alte Werte: $\text{bal}(v) = -2$, $\text{bal}(x) = 1$, $\text{bal}(w) = 0$, neue Werte: $\text{bal}(v) := 0$, $\text{bal}(x) := 0$, $\text{bal}(w) := 0$.
- B endet auf Ebene $h - 1$ und C auf Ebene h , alte Werte: $\text{bal}(v) = -2$, $\text{bal}(x) = 1$, $\text{bal}(w) = -1$, neue Werte: $\text{bal}(v) := 1$, $\text{bal}(x) := 0$, $\text{bal}(w) := 0$.

In allen drei Unterfällen ist die Tiefe des betrachteten Teilbaumes um 1 gesunken und die Rebalancierung muss am Elter (falls existent) fortgesetzt werden.

Bei der Prozedur DELETE kann es also vorkommen, dass auf jeder Ebene eine (einfache oder doppelte) Rotation erforderlich ist. Insgesamt haben wir folgenden Satz bewiesen.

Satz 3.6.3: Die Operationen SEARCH, INSERT und DELETE können in AVL-Bäumen in Zeit $O(\log n)$ ausgeführt werden.

3.7 Skiplisten

Wir haben nun eine Reihe von Datenstrukturen kennengelernt, mit denen sich dynamische Dateien effizient verwalten lassen. Nur 2-3-Bäume unterstützen die gesamte Liste von Operationen, wobei die Rebalancierung den konstanten Faktor für die Rechenzeit ziemlich erhöht. Skiplisten werden eine effiziente Unterstützung aller Operationen zulassen, wenn

wir uns damit zufrieden geben, dass mit sehr kleiner Wahrscheinlichkeit das Verhalten schlecht ist. Skiplisten sind randomisierte Datenstrukturen, die wahrscheinlichkeitstheoretischen Aussagen beziehen sich also auf die verwendeten Zufallsexperimente und nicht auf eine angenommene Wahrscheinlichkeitsverteilung auf der Menge der möglichen Daten.

Lineare Listen unterstützen das Einfügen und Entfernen von Daten, wenn die zugehörige Stelle und die sie umgebenden Zeiger bekannt sind. Die Suche nach einem Datum benötigt jedoch auch im Durchschnitt lineare Zeit. Dies lässt sich mit Hilfslisten vermeiden. Nehmen wir als Beispiel eine Liste mit 15 Datensätzen sowie einem Listenanfang und einem Listeneende (siehe Abbildung 3.7.1).

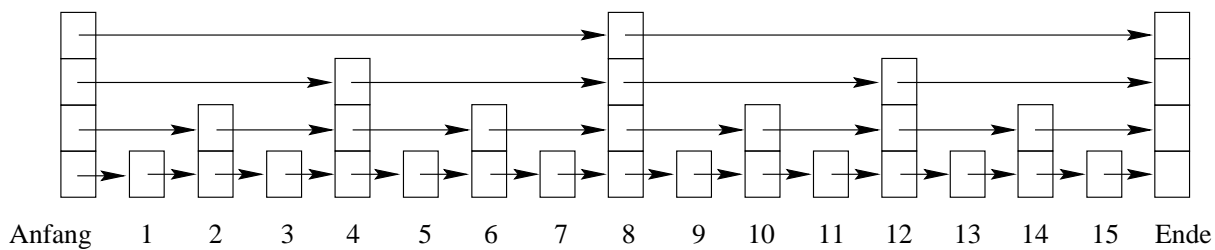


Abbildung 3.7.1: Eine Datei mit einer Liste und Hilfslisten.

Wir suchen zunächst in der obersten Liste. Wenn wir ein Datum finden, das zu groß ist, steigen wir in der Listenhierarchie eine Stufe herunter und suchen dort. Im Grunde ahmen wir die binäre Suche in Arrays nach. Die Zahl der Zeiger ist mit ungefähr $2n$ nur etwa doppelt so groß wie in linearen Listen, und die Suchdauer beträgt $O(\log n)$. Änderungen werden in dieser Datei allerdings nicht mehr unterstützt. Wie sollen wir bei Einfügungen und Entfernungen von Daten die regelmäßige Struktur der Listen und Hilfslisten aufrechterhalten?

Hier nehmen wir Zufallsexperimente zu Hilfe. Für jedes neu einzufügende Element wird seine „Höhe“, genauer die Zahl der Listen, in der das Datum abgespeichert wird, ausgewürfelt. Dazu werfen wir eine Münze (natürlich benutzen wir einen Pseudozufallsgenerator), bis sie das erste Mal auf Kopf fällt. Wenn dies nach h Münzwürfen der Fall ist, erhält das Datum die Höhe h . Die Wahrscheinlichkeit für die Höhe h beträgt somit 2^{-h} und die erwartete Höhe

$$\sum_{1 \leq h < \infty} h 2^{-h} = 2 .$$

Wenn wir die maximale Höhe aller Daten mit verwalten, also die größte Höhe, auf der der Anfangszeiger nicht auf das Listeneende zeigt, können wir unsere Suche stets in der höchsten, also am dünnsten besetzten Liste starten. In der folgenden Skipliste stehen im Gegensatz zu Abbildung 3.7.1 auch die abgespeicherten Daten. Es wird sich bei der Analyse als hilfreich erweisen, wenn wir die unterste Liste als Liste auf Level 0 betrachten. Bei Höhe h gibt es dann die Ebenen $0, \dots, h-1$. Ein Datum mit Höhe i ist in den Listen auf den Ebenen $0, \dots, i-1$ vertreten.

Die Bezeichnung Skipliste ergibt sich aus der Tatsache, dass wir in Listen auf größerer Höhe viele Daten auslassen oder überspringen („skippen“).

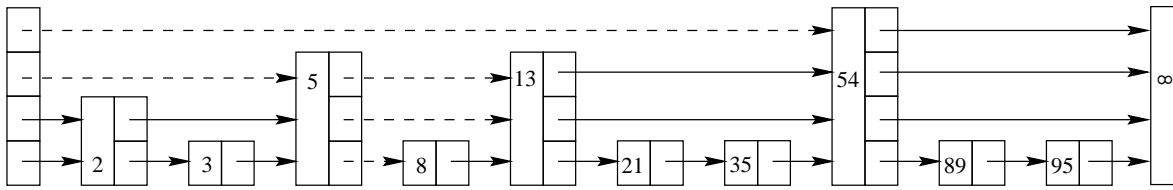


Abbildung 3.7.2: Eine Skipliste.

Sowohl bei der Suche nach dem Datum 8 als auch bei der Suche nach einem Datum im Bereich $(5, 8)$ werden die gestrichelten Zeiger angesprochen. Wir beginnen auf der höchsten Ebene im Anfangsdatum. Da der Zeiger auf 54 zeigt und $54 > 8$ ist, steigen wir eine Ebene hinab. Auf Ebene 2 finden wir erst 5 und dann das zu große Datum 13. Also steigen wir im Datum 5 eine Ebene ab und finden wieder einen Zeiger auf das zu große Datum 13. Daher gelangen wir auf Ebene 0. Dort finden wir das Datum 8, entweder das gesuchte Datum oder ein zu großes Datum. Ein zu großes Datum auf Ebene 0 beendet eine erfolglose Suche. Für die Prozedur CONCATENATE genügt es, für die vordere Skipliste die Zeiger auf das Ende und für die hintere Skipliste die Zeiger vom Anfangsdatum zu bestimmen und passend zu verbinden. Die SPLIT-Operation, z. B. in Abbildung 3.7.2 am Datum 8, ist ebenfalls einfach. Bei der Suche nach dem Datum 8 haben wir bereits alle Zeiger gefunden, die nun verändert werden müssen. Es sind die Zeiger über das Datum 8 hinaus (im Beispiel auf Ebene 3 der Zeiger auf 54, auf Ebene 2 und Ebene 1 die Zeiger auf 13) und die Zeiger vom Datum 8 weg (im Beispiel auf Ebene 0 der Zeiger auf 13). Diese Zeiger müssen im ersten abgetrennten Teil auf das Ende zeigen und im zweiten abgetrennten Teil vom neuen Anfangsdatum aus starten. Wenn in einem Teil nun Zeiger vom Anfang direkt auf das Ende zeigen, kann dort die Zahl der Ebenen verringert werden. Im Beispiel kann die Höhe des vorderen Teils um 1 sinken.

Wir kommen nun zu den Operationen INSERT und DELETE. Wenn ein Schlüssel bereits vorhanden ist, muss die gespeicherte Information bei einer Einfügung nur überschrieben werden. Ansonsten folgt die Einfügung auf eine erfolglose Suche. Nehmen wir an, dass im Beispiel 7 eingefügt werden soll. Für 7 wird die zugehörige Höhe h ausgewürfelt. Ist h größer als die momentan maximale Höhe, werden entsprechend viele neue Listen gebildet, die jeweils zwei Zeiger haben. Diese zeigen vom Anfang auf das neue Datum und von dort auf das Ende. Betrachten wir ansonsten Ebene l . Ein Datum mit Höhe h wird auf den Ebenen $0, \dots, h - 1$ dargestellt. Ist $h - 1 < l$, muss auf Ebene l nichts getan werden. Ist $h - 1 \geq l$, kennen wir von der erfolglosen Suche den Zeiger, der „durch das Datum 7“ geht, im Beispiel für $l = 1$ der Zeiger von 5 auf 13. Der Zeiger von 5 soll nun auf das neue Datum 7 zeigen und der Zeiger des neuen Datums auf dieser Ebene auf 13.

Beim Entfernen (im Beispiel von 13) suchen wir (erfolglos) nach dem Datum „13–“ (es ist kleiner als 13, aber größer als jedes existierende Datum, das kleiner als 13 ist). Dann finden wir das Datum 13 und alle Zeiger auf 13, im Beispiel auf Ebene 2 und Ebene 1 von 5 aus und auf Ebene 0 von 8 aus. Diese Zeiger müssen nun auf die Nachfolger von 13 zeigen, im Beispiel auf Ebene 2 auf 54, auf Ebene 1 ebenso und auf Ebene 0 auf 21. Dann kann das Datum entfernt werden. Gegebenenfalls sinkt die Höhe unserer Skipliste.

Die Verwaltung von Skiplisten ist also erstaunlich einfach. Die Rechenzeit aller Operationen ist proportional zur Rechenzeit für die Suche nach einem Datum zuzüglich der Höhe der Skipliste, wobei wir beim Einfügen die Höhe nach der Einfügung und beim Löschen die Höhe zu Beginn der Operation betrachten. Es genügt somit, die Rechenzeit für eine Suche und die Höhe randomisierter Skiplisten zu analysieren. Da Daten eine beliebige Höhe bekommen können, sind worst case Schranken nicht möglich. Wir können statt dessen Erwartungswerte berechnen und (allerdings nicht in dieser Vorlesung) nachweisen, dass große Abweichungen vom Erwartungswert extrem unwahrscheinlich sind. Bei unserer Analyse hilft es uns, dass Skiplisten kein Gedächtnis haben. Ob ein Datum früher einmal eingefügt wurde, hat auf das aktuelle Aussehen der Skipliste keinen Einfluss, wenn es inzwischen entfernt wurde. Wenn unter einem Schlüssel Informationen mehrfach eingefügt und entfernt wurden und schließlich eine Information eingefügt wurde, ist nur die letzte Einfügung von Interesse. Diese Beobachtung folgt unmittelbar aus der Beschreibung der Algorithmen für das Einfügen und Entfernen von Daten.

Wir beginnen unsere Analyse mit den Zufallsvariablen $H(n)$ für die Höhe einer Skipliste mit n Daten und $Z(n)$ für die Anzahl der Zeiger in der Skipliste.

Satz 3.7.1: i) $\text{Prob}(H(n) \geq h) \leq \min\{1, n/2^{h-1}\}$.

ii) $E(H(n)) \leq \lfloor \log n \rfloor + 3$.

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

iv) Der erwartete Platzbedarf für eine Skipliste auf n Daten beträgt $O(n)$.

Beweis:

i) Mit Prob wird probability (Wahrscheinlichkeit) abgekürzt. Die obere Schranke 1 für Wahrscheinlichkeiten ist trivial. Die Wahrscheinlichkeit, dass ein Datum eine Höhe von mindestens h bekommt, beträgt $(\frac{1}{2})^{h-1}$. Dass dies für mindestens eines der n Daten geschieht, kann durch $n \cdot (\frac{1}{2})^{h-1}$ abgeschätzt werden.

ii) Der Erwartungswert ist definiert durch

$$\begin{aligned}
 E(H(n)) &= \sum_{1 \leq h < \infty} h \cdot \text{Prob}(H(n) = h) \\
 &= \text{Prob}(H(n) = 1) + \text{Prob}(H(n) = 2) + \text{Prob}(H(n) = 3) + \dots \\
 &\quad + \text{Prob}(H(n) = 2) + \text{Prob}(H(n) = 3) + \dots \\
 &\quad + \text{Prob}(H(n) = 3) + \dots \\
 &= \text{Prob}(H(n) \geq 1) \\
 &\quad + \text{Prob}(H(n) \geq 2) \\
 &\quad + \text{Prob}(H(n) \geq 3) \\
 &\quad + \dots \\
 &= \sum_{1 \leq h < \infty} \text{Prob}(H(n) \geq h).
 \end{aligned}$$

Diese neue Formel für den Erwartungswert von Zufallsvariablen, die nur nichtnegative ganze Zahlen annehmen, ist sehr oft hilfreich. Die ersten $\lfloor \log n \rfloor + 2$ Summanden schätzen wir durch 1 ab, den $(\lfloor \log n \rfloor + 2 + i)$ -ten Summanden nach i) durch $(\frac{1}{2})^i$. Also kann die Summe durch $\lfloor \log n \rfloor + 3$ abgeschätzt werden.

iii) Der Erwartungswert ist linear. Das i -te Objekt hat eine durchschnittliche Höhe von 2, also durchschnittlich 2 Zeiger. Hinzu kommen $H(n)$ Zeiger für das Anfangsobjekt. Nun folgt das Resultat aus ii).

iv) folgt direkt aus iii).

□

Satz 3.7.2: Die erwartete Rechenzeit für jede der Operationen SEARCH, INSERT und DELETE beträgt $O(\log n)$.

Beweis: Die Vorbetrachtungen haben bereits gezeigt, dass wir uns auf die Analyse einer erfolglosen Suche beschränken können. Sie ist höchstens teurer als die zugehörige erfolgreiche Suche, da wir in jedem Fall bis auf Ebene 0 absteigen müssen. Wir betrachten wieder die Suche nach 7 in Abbildung 3.7.2.

Auf dem Suchweg werden Zeiger auf Daten, die größer als 7 sind, gefunden. Dies sind im Beispiel genau 4 Zeiger, im Allgemeinen ist dies genau die Anzahl der Ebenen. Deren erwartete Anzahl kennen wir aus Satz 3.7.1. Die Anzahl der restlichen Zeiger hängt vom Aussehen der Skipliste ab. Bei diesen Zeigern verläuft der Suchweg horizontal von links nach rechts, während er bei den zuvor betrachteten Zeigern vertikal absteigt.

Wie groß ist die erwartete Anzahl von horizontalen Schritten zwischen den Abstiegen? Aus dieser Sicht ist dies schwer zu analysieren, da wir nicht abschätzen können, wieviele Daten zwischen dem erreichten Datum und dem gesuchten Datum in der Skipliste enthalten sind. Statt dessen betrachten wir den Suchpfad rückwärts. Die Methode der Rückwärtsanalyse von Algorithmen ist noch nicht sehr alt, aber bereits sehr erfolgreich. Wir nehmen also an, dass wir in einem Datum auf Ebene l sind. Auf Suchpfaden werden Daten stets auf ihren höchsten Ebenen erreicht. Die Wahrscheinlichkeit, dass ein Datum, das auf Ebene l existiert, auch auf Ebene $l + 1$ existiert, beträgt nach Konstruktion genau $1/2$.

Auf dem Rückwärtspfad geht also jeder Schritt unabhängig von den vorherigen Schritten mit Wahrscheinlichkeit $1/2$ nach oben und mit Wahrscheinlichkeit $1/2$ nach links. Der Rückwärtspfad endet auf jeden Fall, wenn das Anfangsobjekt erreicht wird. Die erwartete Länge einer ununterbrochenen Phase von Schritten nach links ist also nicht größer als die erwartete Länge einer Phase ununterbrochener Münzwürfe, die alle Zahl zeigen. Diese erwartete Länge ist um 1 kleiner als die erwartete Zeit, bis zum ersten Mal Kopf erscheint und damit höchstens 1. Diese Abschätzung benutzen wir bis zum Aufstieg auf die Ebene $\lfloor \log n \rfloor$. Danach werden nur noch Daten erreicht, deren Höhe mindestens $\lfloor \log n \rfloor + 1$ ist. Die Wahrscheinlichkeit, dass ein Datum eine Höhe von mindestens $\lfloor \log n \rfloor + 1$ hat, ist durch $1/n$ beschränkt. Sei $X_i = 1$, falls das i -te Datum eine Höhe von mindestens $\lfloor \log n \rfloor + 1$ hat, und $X_i = 0$ sonst. Dann ist $E(X_i) \leq 1/n$ und $E(X_1 + \dots + X_n) \leq 1$. Dabei steht

$X_1 + \dots + X_n$ für die zufällige Anzahl von Daten, deren Höhe mindestens $\lceil \log n \rceil + 1$ beträgt. Insgesamt haben wir unseren Satz bewiesen. \square

Abschließend sollen die dynamischen Dateien kurz verglichen werden. Lineare Listen benötigen selbst im Durchschnitt lineare Suchzeiten. Hashingverfahren sind die einzigen, die im Durchschnitt auf $O(1)$ -Zeiten kommen, wenn die Datei eine einigermaßen stabile und vorhersagbare Größe hat. Ansonsten ist zeitweise ein kostenintensives Rehashing nötig. Operationen wie MIN, MAX, CONCATENATE und SPLIT werden nicht gut unterstützt. Das Durchschnittsverhalten der von uns betrachteten Implementierungen ist ein Durchschnitt über die zu betrachtenden Daten und nicht über Zufallszahlen.

Binäre Suchbäume sind wie die Hashingverfahren im worst case sehr schlecht. Wie bei allen Bäumen wird Platz für die Zeiger benötigt. Binäre Suchbäume sind sehr einfach zu implementieren, das gute Durchschnittsverhalten ist aber abhängig von den zu verarbeitenden Daten. 2-3-Bäume unterstützen alle betrachteten Operationen mit einer logarithmischen Laufzeit. Die Speicherplatzausnutzung in den Knoten ist nur zu 50% garantiert. B-Bäume sind Verallgemeinerungen von 2-3-Bäumen für die Verarbeitung externer Daten. AVL-Bäume nutzen wie alle binären Bäume den Platz in den Knoten voll aus. Sie garantieren eine logarithmische Tiefe und sind in vielerlei Hinsicht eine Alternative zu 2-3-Bäumen.

Skiplisten sind Vertreter eines neuen Trends. Wir müssen nicht auf die Zufälligkeit der zu verarbeitenden Daten vertrauen, können aber auf die Zufälligkeit von Zufallszahlen (auch wenn es nur Pseudozufallszahlen sind) vertrauen. Ein mit hoher Wahrscheinlichkeit (z. B. $1 - 2^{-100}$) garantiertes Verhalten ist in den meisten Fällen so gut wie garantiertes Verhalten im worst case. Skiplisten unterstützen alle Operationen und sind sehr einfach zu implementieren und zu verwalten. Randomisierte Datenstrukturen und Algorithmen stellen in vielen Fällen die bessere Alternative zu deterministischen Datenstrukturen und Algorithmen dar. Daher brauchen Studierende der Informatik verstärkt Stochastikkenntnisse.

4 Sortieren

4.1 Vorbemerkungen

Eine Datenstruktur ist ein Paket von Algorithmen, das eine vorgegebene Liste von Operationen auf Daten mit vorgegebener Struktur unterstützt. Datenstrukturen dienen als Hilfsmittel beim Entwurf von effizienten Algorithmen für komplexere Probleme. Dies gilt ebenfalls für Algorithmen für das Sortierproblem, dem am intensivsten untersuchten algorithmischen Problem. Neben Realisierungen der wichtigsten Datenstrukturen enthalten gängige Programmbibliotheken Realisierungen von Sortieralgorithmen. Die intensive Betrachtung von Sortieralgorithmen ist jedoch weiterhin eine sehr gute Einführung in das Gebiet Entwurf und Analyse effizienter Algorithmen. Wichtige Prinzipien werden an einem einfachen Beispiel exemplarisch vorgestellt.

Das allgemeine Sortierproblem besteht darin, eine Folge a_1, \dots, a_n von Elementen aus einer vollständig geordneten Menge so umzuordnen, dass die Elemente in aufsteigender Reihenfolge stehen. Im Gegensatz dazu stehen spezielle Sortierprobleme mit einer eingeschränkten Grundmenge wie z. B. der lexikographischen Ordnung auf Σ^k für ein endliches, geordnetes Alphabet Σ . Darüber hinaus sind wir nicht immer an einer vollständigen Sortierung interessiert. Wichtige eingeschränkte Sortierprobleme sind das Auswahlproblem (berechne das k -kleinste Element) und das Partitionsproblem (berechne die Menge der k kleinsten Elemente). Auch das in Kapitel 1 behandelte MAXMIN-Problem ist ein eingeschränktes Sortierproblem.

Historisch gesehen wurden Sortieralgorithmen auf besonders große Datenmengen angewendet. Daher spielte der benötigte Speicherplatz eine zentrale Rolle. Allgemein sprechen wir heute von einem externen Algorithmus, wenn die behandelte Datenmenge so groß ist, dass sie nicht einmal im internen Speicher Platz findet. Externe Algorithmen wurden erstmals für das Sortieren diskutiert. Heutzutage bilden sie einen eigenen Forschungszweig. Wir werden vor allem interne Sortieralgorithmen diskutieren und dabei untersuchen, welche Algorithmen in situ (am Platze) arbeiten, also außer dem Array für die Eingabe nur $O(\log n)$ zusätzlichen Platz benötigen.

Zwei weitere wünschenswerte Eigenschaften von Sortieralgorithmen sind Stabilität und Adaptivität. Ein Sortieralgorithmus heißt stabil, wenn Elemente mit gleichem Wert in der gegebenen Reihenfolge bleiben. Dies bedarf einer Erläuterung, denn gleiche Elemente können ja gar nicht unterschieden werden. Hierbei ist zu berücksichtigen, dass wir es mit komplexen Daten zu tun haben können, von denen nur ein Teil als Sortierkriterium dient. Wenn Daten bezüglich eines sekundären Merkmals bereits sortiert sind und nun bezüglich des primären Merkmals sortiert werden sollen, erhalten wir eine insgesamt sortierte Folge, ohne die sekundären Merkmale noch einmal zu betrachten, wenn der Sortieralgorithmus stabil ist. Adaptivität ist die Eigenschaft, auf Eingabefolgen, die bezüglich eines Vorsortiertheitsmaßes, „wenig unsortiert“ sind, besonders effizient zu sein. Dies ist eine wünschenswerte Eigenschaft, wenn Daten typischerweise vorstrukturiert sind.

Wir haben bereits in Kapitel 1 gesehen, dass konstante Faktoren für die Rechenzeit vom benutzten Rechnermodell abhängen. Sortieralgorithmen bieten die aus didaktischer Sicht

vorteilhafte Option, genauer vergleichbar zu sein. Dazu unterscheiden wir wesentliche Vergleiche, das sind Vergleiche zwischen zwei Elementen der Eingabe, unwesentliche Vergleiche wie Vergleiche zwischen Indizes, arithmetische Operationen, Zuweisungen und Vertauschungen. Wesentliche Vergleiche können besonders teuer sein. Die zu sortierenden Elemente können sehr komplex sein, z.B. binäre Bäume mit Inorder-Nummerierung, und ein wesentlicher Vergleich könnte darin bestehen festzustellen, welcher Baum bezüglich dieser Codierung lexikographisch kleiner ist. Daher sind wir an Sortieralgorithmen mit einer möglichst kleinen Anzahl an wesentlichen Vergleichen interessiert, wobei die Anzahl aller anderen Operationen von der gleichen Größenordnung wie die Anzahl der wesentlichen Vergleiche sein sollte.

In den Kapiteln 4.2–4.5 stellen wir die vier wichtigsten allgemeinen Sortieralgorithmen vor. In Kapitel 4.6 beweisen wir für das allgemeine Sortierproblem eine untere Schranke für die worst case und average case Anzahl an wesentlichen Vergleichen. Kapitel 4.7 ist den für die Anwendungen wichtigen speziellen Sortierproblemen gewidmet und Kapitel 4.8 mit dem Auswahlproblem einem der wichtigsten eingeschränkten Sortierprobleme. Exemplarisch wollen wir in Kapitel 4.9 untersuchen, wie wir einen Parallelrechner (ein Multiprozessorsystem) für das Sortierproblem nutzen können.

4.2 Sortieren mit binärer Suche

Wir können ein Experiment machen und Menschen dabei beobachten, wie sie einen Packen mit vielleicht 100 Karteikarten mit dreibuchstabigen Wörtern lexikographisch sortieren. Es gibt dabei ganz verschiedene Herangehensweisen, an denen wir Grundprinzipien von bekannten Sortieralgorithmen wiederfinden.

Einige gehen sequenziell vor und sortieren die ersten i Karten, bevor sie sich die anderen anschauen. Wenn sie die $(i + 1)$ -te Karte anschauen, suchen sie im sortierten Packen der ersten i Karten nach der richtigen Position für die $(i + 1)$ -te Karte. Dabei ist das Vorgehen nicht immer systematisch. Wenn die neue Karte das Wort UNI enthält und man sich erinnert, dass noch nicht viele Wörter mit U, V, W, X, Y oder Z da waren, wird man im hinteren Teil des Packens die Suche starten. Bei einem systematischen Vorgehen bietet sich an dieser Stelle die binäre Suche an. Wenn a_{i+1} an Position k eingeordnet werden soll, müssen die Daten an den Positionen k, \dots, i im Array eine Position nach rechts rücken. Dieser Sortieralgorithmus heißt Sortieren durch binäres Einfügen oder auch Insertionsort. Er arbeitet in situ und auch stabil (wenn a_{i+1} hinter den Elementen $a_j = a_{i+1}, j \leq i$ eingeordnet wird). Bezüglich der wesentlichen Vergleiche ist er nicht adaptiv, da die Anzahl der wesentlichen Vergleiche bei der binären Suche kaum vom Ergebnis abhängt.

Die Gesamtzahl der wesentlichen Vergleiche beträgt im worst case

$$\sum_{1 \leq i \leq n-1} \lceil \log(i+1) \rceil = \sum_{2 \leq i \leq n} \lceil \log i \rceil < \log(n!) + n.$$

Den Term $\log(n!)$ wollen wir genauer untersuchen. Nach der Stirling-Formel konvergiert der Quotient von $n!$ und $\sqrt{2\pi} n^{n+1/2} e^{-n}$ gegen 1. Daher folgt

$$\log(n!) \approx \log(\sqrt{2\pi} n^{n+1/2} e^{-n}) = n \log n - n \log e + O(\log n) \approx n \log n - 1,4427n.$$

Satz 4.2.1: Insertionsort kommt im worst case mit höchstens $n \log n - 0,4426n + O(\log n)$ wesentlichen Vergleichen aus.

Der Datentransport ist jedoch teuer, sogar im average case, wobei wir alle Permutationen der geordneten Folge als gleichwahrscheinliche Eingaben betrachten. Bei der average case Analyse nehmen wir o. B. d. A. an, dass die Eingabe eine zufällige Permutation der Folge $1, \dots, n$ ist.

Satz 4.2.2: Insertionsort benötigt im average case $\Theta(n^2)$ Operationen.

Beweis: Es ist offensichtlich, dass selbst im worst case $O(n^2)$ Operationen genügen. Wir zählen nun nur die Operationen, bei denen ein Datum nach rechts rückt.

Es gibt $(n-1)!$ Permutationen π mit $\pi(i) = j$. Also muss a_i mit Wahrscheinlichkeit $1/n$ am Ende an Position j stehen. Falls $j > i$ ist, muss a_i mindestens $j - i$ -mal nach rechts rücken. Die average case Zahl an Rechtsbewegungen, die a_i macht, beträgt also mindestens

$$\frac{1}{n}(0 + \dots + 0 + 1 + \dots + (n-i)) = (n-i)(n-i+1)/(2n).$$

Die average case Zahl an Rechtsbewegungen beträgt also mindestens (für die Rechnung vergleiche die Analyse von Algorithmus 1.3.1)

$$\frac{1}{2n} \sum_{1 \leq i \leq n} (n-i)(n-i+1) = \frac{1}{2n} \sum_{1 \leq i \leq n} (i-1)i = (n^2-1)/6.$$

□

Aus dieser Analyse haben wir etwas für den Entwurf weiterer Sortieralgorithmen gelernt. Wir können quadratische Laufzeit nicht verhindern, wenn Daten im Array in eine Richtung nur Schritte der Länge 1 machen.

4.3 Sortieren durch Mischen

Andere Personen werden in unserem Experiment den Stapel an Karteikarten in kleine Unterstapel aufteilen, die sie „direkt“ sortieren. Danach können zwei Unterstapel mit dem Reißverschlussverfahren verschmolzen werden, daher auch der Name Mergesort. Im Gegensatz zu Kartenspielen bedeutet „Mischen“ im Kontext des Sortierens, aus zwei sortierten Folgen $a_1 \leq \dots \leq a_k$ und $b_1 \leq \dots \leq b_m$ eine gemeinsame sortierte Folge $c_1 \leq \dots \leq c_{k+m}$ zu machen. Diese Operation bezeichnen wir mit

$$(c_1, \dots, c_{k+m}) := \text{Merge}(a_1, \dots, a_k; b_1, \dots, b_m).$$

Die Anzahl der wesentlichen Vergleiche ist beim Reißverschlussverfahren durch $k + m - 1$ beschränkt.

Bei einem systematischen Vorgehen sollten wir Teilfolgen der Länge 1 direkt behandeln, sie sind bereits sortiert. Aufgrund der linearen Rechenzeit des Reißverschlussverfahrens

sollten stets kürzeste Folgen gemischt werden. Dies kann an einem möglichst gut balancierten binären Baum, bei dem die n Blätter auf maximal zwei benachbarten Ebenen liegen, illustriert werden. Aus iterativer Sicht wird der Baum von den Blättern zur Wurzel bearbeitet, wobei jeder Knoten das Mischen der Folgen in den Kindern vorschreibt. Am Ende enthält die Wurzel das Ergebnis. Eleganter ist eine rekursive Beschreibung als Divide-and-Conquer Algorithmus, bei dem die Arbeit darin besteht, aus den Lösungen der Teilprobleme die Lösung des Problems zu berechnen.

Algorithmus 4.3.1: Mergesort(a_1, \dots, a_n)

- (1) Falls $n = 1$, STOP.
- (2) Falls $n > 1$,
 - $(b_1, \dots, b_{\lceil n/2 \rceil}) := \text{Mergesort}(a_1, \dots, a_{\lceil n/2 \rceil})$,
 - $(c_1, \dots, c_{\lfloor n/2 \rfloor}) := \text{Mergesort}(a_{\lceil n/2 \rceil+1}, \dots, a_n)$,
 - $(d_1, \dots, d_n) := \text{Merge}(b_1, \dots, b_{\lceil n/2 \rceil}; c_1, \dots, c_{\lfloor n/2 \rfloor})$.

Für die Anzahl der wesentlichen Vergleiche $V(n)$ (im worst case) gilt

$$V(1) = 0, \quad V(n) = V(\lceil n/2 \rceil) + V(\lfloor n/2 \rfloor) + n - 1.$$

Analog zur Analyse von Algorithmus 1.3.3 erhalten wir für $n = 2^k$ die explizite Lösung

$$V(n) = n \log n - n + 1.$$

Satz 4.3.2: Mergesort kommt für $n = 2^k$ mit $n \log n - n + 1$ wesentlichen Vergleichen aus, die Anzahl anderer Operationen liegt in der gleichen Größenordnung.

Damit ist Mergesort bezüglich der Rechenzeit viel effizienter als Insertionsort. Gravierender Nachteil ist, dass das Reißverschlussverfahren nicht in situ arbeitet. Wir haben vielleicht n_1 Elemente der a -Folge und n_2 Elemente der b -Folge bearbeitet, aber keinen freien Arrayblock der Länge $n_1 + n_2$ geschaffen. Es ist möglich, eine Variante von Mergesort auf zwei Arrays der Länge n zu realisieren. Besondere Bedeutung hat Mergesort beim externen Sortieren. Wir benötigen im Kernspeicher Platz für drei Datenblöcke. Für das Mischen zweier Folgen werden von diesen Folgen je ein Datenblock eingelesen und mit dem Reißverschlussverfahren gemischt. Im dritten Datenblock wird das Ergebnis gespeichert. Ist einer der beiden Eingabeblocks leer, wird der Folgeblock der entsprechenden Folge eingelesen. Ist der Ausgabeblock voll, wird er extern gespeichert.

Mergesort kann auf natürliche Weise stabil gestaltet werden. Darüber hinaus lässt sich Mergesort so anpassen, dass es sich adaptiv verhält. Bei einem einmaligen Durchlaufen der Eingabe werden die Stellen i mit $a_i > a_{i+1}$ erkannt. Die Datenblöcke zwischen diesen Stellen sind sortierte Teilfolgen. Der Sortieralgorithmus kann mit diesen sortierten Teilfolgen starten. Da mit höchstens n wesentlichen Vergleichen die Anzahl sortierter Teilfolgen halbiert werden kann, beträgt die Gesamtzahl wesentlicher Vergleiche höchstens $(\lceil \log k \rceil + 1)n$, wenn die Anzahl sortierter Blöcke k beträgt. Der Extrasummand n beschreibt die Vergleiche bei der Einteilung in sortierte Datenblöcke.

4.4 Quicksort

Für die meisten Probleme ist die beste Vorgehensweise von Rechnern eine andere als die beste Vorgehensweise von Menschen. Rechner können sich besser viele Informationen „merken“ als Menschen. Daher wird wohl niemand in unserem Experiment eine Vorgehensweise wählen, die das am häufigsten verwendete Sortiervverfahren, werbeträchtig Quicksort genannt, imitiert.

Quicksort ist ebenfalls ein Divide-and-Conquer Algorithmus, bei dem die Hauptarbeit allerdings darin besteht, das Problem zu zerlegen. Für ein Datum a_i (welches Datum wir wählen, diskutieren wir später) bestimmen wir eine Position j , so dass a_i in der sortierten Folge an Position j stehen kann. Gleichzeitig sorgen wir dafür, dass an den Positionen $1, \dots, j-1$ nur Daten $a_k \leq a_i$ und an den Positionen $j+1, \dots, n$ nur Daten $a_m \geq a_i$ stehen. Danach genügt es, Quicksort auf die Teilprobleme an den Positionen $1, \dots, j-1$ und $j+1, \dots, n$ anzuwenden. Probleme der Länge 0 oder 1 sind selbstverständlich bereits gelöst. Wir beschreiben nun, wie wir für das Datum a_i und die Positionen $1, \dots, n$ die Problemaufteilung mit $n-1$ wesentlichen Vergleichen vornehmen können.

1. Von a_1 beginnend wird gesucht, bis ein Datum $a_l \geq a_i$ gefunden wird. Dies ist spätestens für $l = i$ der Fall. Dabei wird durch Indexvergleiche sichergestellt, dass a_i nicht mit a_i verglichen wird.
2. Von a_n beginnend wird gesucht, bis ein Datum $a_r \leq a_i$ gefunden wird. Dies ist spätestens für $r = i$ der Fall. Wieder wird darauf geachtet, dass a_i nicht mit a_i verglichen wird.
3. Falls $l = r = i$, STOP. Ansonsten werden a_l und a_r vertauscht. Beginnend an den Positionen l , falls $r = i$, und $l+1$ sonst, sowie r , falls $l = i$, und $r-1$ sonst wird wie in (1) und (2) weitergesucht. Es ist zu beachten, dass das ausgewählte Datum an anderer Position stehen kann.

Das Verfahren führt $n-1$ wesentliche Vergleiche aus, da a_i mit jedem anderen Datum genau einmal verglichen wird. Die Zahl der Vertauschungen ist ebenfalls höchstens $n-1$, da jedes Datum mit Ausnahme von a_i nur einmal vertauscht wird.

Beispiel 4.4.1: $n = 13$, $i = 7$, $a_i = 53$.

15	47	33	87	98	17	53	76	83	2	53	27	44
15	47	33	44	98	17	53	76	83	2	53	27	87
15	47	33	44	27	17	53	76	83	2	53	98	87
15	47	33	44	27	17	53	76	83	2	53	98	87
15	47	33	44	27	17	53	53	83	2	76	98	87
15	47	33	44	27	17	53	2	83	53	76	98	87
15	47	33	44	27	17	53	2	53	83	76	98	87

Die Position 9 hat ihr endgültiges Datum 53 erhalten. Quicksort wird für die Bereiche $[1, 8]$ und $[10, 13]$ aufgerufen.

Wir haben bereits gesehen, dass die Zahl unwesentlicher Operationen von der gleichen Größenordnung wie die Zahl wesentlicher Vergleiche ist.

Wie sollten wir das „Zerlegungsdatum“ a_i wählen? Wir beginnen mit der naivsten Strategie.

Zerlegungsstrategie 1: Wähle als Zerlegungsdatum das erste Datum des betrachteten Bereichs.

Es ist möglich, dass wir im ersten Schritt das Problem in zwei Probleme der Größe 0 und $n - 1$ zerlegen. Wenn sich dieses Pech wiederholt, benötigen wir

$$(n - 1) + (n - 2) + \cdots + 2 + 1 = \binom{n}{2} = (n^2 - n)/2$$

wesentliche Vergleiche. Dabei wird jedes Datum mit jedem anderen verglichen. Da Quicksort kein Datenpaar zweimal vergleicht (nach der ersten Phase wird a_i nicht mehr verglichen), ist dies der schlechteste Fall und Quicksort führt im worst case $(n^2 - n)/2$ Vergleiche durch. Dies ist im Vergleich zu Insertionsort und Mergesort nicht gerade beeindruckend. Quicksort hat jedoch ein sehr gutes average case Verhalten, wenn wir annehmen, dass alle Permutationen von n verschiedenen Daten mit gleicher Wahrscheinlichkeit die Eingabe darstellen. Mit $V(n)$ bezeichnen wir in dieser Situation die average case Anzahl wesentlicher Vergleiche. Dann ist

$$V(0) = V(1) = 0.$$

Das Zerlegungsdatum landet mit Wahrscheinlichkeit $1/n$ an jeder Position j . Da alle Permutationen dieselbe Wahrscheinlichkeit haben, ist für das erste Datum jede Position gleich wahrscheinlich. Wir haben dann Teilprobleme der Größe $j - 1$ und $n - j$ zu lösen. Beim Vergleich mit a_1 wurden die Daten nur danach unterschieden, ob sie größer oder kleiner als a_1 sind. Wenn wir die $(j - 1)!$ möglichen Permutationen der $j - 1$ Daten, die kleiner als a_1 sind, betrachten (wobei die Menge der Positionen dieser Daten unverändert bleibt), dann erhalten wir am Ende der ersten Phase von Quicksort alle $(j - 1)!$ Permutationen dieser Daten auf den ersten $j - 1$ Positionen. Also sind in beiden Teilproblemen wieder alle Reihenfolgen gleich wahrscheinlich. Mit Wahrscheinlichkeit $1/n$ benötigen wir durchschnittlich $n - 1 + V(j - 1) + V(n - j)$ wesentliche Vergleiche, insgesamt folgt für $n \geq 2$

$$\begin{aligned} V(n) &= \sum_{1 \leq j \leq n} \frac{1}{n} (n - 1 + V(j - 1) + V(n - j)) \\ &= n - 1 + \frac{1}{n} \sum_{1 \leq j \leq n} (V(j - 1) + V(n - j)). \end{aligned}$$

In Kapitel 3.4 haben wir die Rekursionsgleichung

$$T(n) = n + \frac{1}{n} \sum_{1 \leq i \leq n} (T(i - 1) + T(n - i))$$

gelöst. Das Vorgehen können wir nachahmen. Die Änderungen sind minimal. Daher geben wir nur das Ergebnis an, wobei wieder $H(n)$ den Wert der harmonischen Reihe bezeichnet. Es ist

$$V(n) = 2(n+1)H(n) - 4n.$$

Es ist bekannt, dass $H(n) - \ln n$ gegen die so genannte eulersche Konstante $\gamma = 0,577\dots$ konvergiert. Also gilt

Satz 4.4.2: Quicksort mit der Zerlegungsstrategie 1 benötigt im worst case $(n^2 - n)/2$ wesentliche Vergleiche und im average case (bezogen auf die $n!$ möglichen Reihenfolgen von n verschiedenen Daten)

$$2(n+1)H(n) - 4n \approx 1,386n \log n - 2,846n$$

wesentliche Vergleiche.

Quicksort braucht in der ersten Phase mindestens $n - 1$ wesentliche Vergleiche und damit mindestens lineare Zeit. Wir erwarten also, dass die Anzahl wesentlicher Vergleiche eine konvexe Funktion ist. Dann ist es am besten, wenn das Zerlegungsdatum das mittlere Datum, auch Median genannt, ist, also an der Position $\lceil n/2 \rceil$ landet. Wenn wir mit $V'(n)$ die Anzahl wesentlicher Vergleiche bezeichnen, wenn als Zerlegungsdatum stets der Median gewählt wird, gilt $V'(0) = V'(1) = 0$ und für $n \geq 2$

$$V'(n) = n - 1 + V'(\lceil n/2 \rceil - 1) + V'(\lfloor n/2 \rfloor).$$

Eine ähnliche Rekursionsgleichung haben wir bei der Analyse von Algorithmus 1.3.3 gelöst, das Ergebnis ist

$$V'(n) = n \log n \pm O(n).$$

Hieran erkennen wir, wie groß (oder wie klein) der Raum für Verbesserungen ist. Allerdings ist es (siehe Kapitel 4.8) nicht so einfach, den Median zu berechnen. Eine Annäherung an dieses Ziel liefert

Zerlegungsstrategie 2: Wähle als Zerlegungsdatum den Median von $a_1, a_{\lceil n/2 \rceil}$ und a_n .

Es genügen stets die drei möglichen wesentlichen Vergleiche zwischen diesen drei Daten, um ihren Median zu berechnen. Dieser Median muss dann nur noch mit den $n - 3$ übrigen Daten verglichen werden. In der ersten Phase genügen also n wesentliche Vergleiche. Danach kann das Zerlegungsdatum auf jeder der Positionen $2, \dots, n - 1$ gelandet sein. Wir untersuchen den worst case, bei dem das Zerlegungsdatum stets auf Position 2 landet. Die Anzahl wesentlicher Vergleiche beträgt dann

$$n + (n - 2) + (n - 4) + \dots.$$

Für gerades n ist dieses

$$2 \left(\frac{n}{2} + \left(\frac{n}{2} - 1 \right) + \dots + 1 \right) = \frac{n}{2} \cdot \left(\frac{n}{2} + 1 \right) = \frac{1}{4}n^2 + \frac{1}{2}n$$

und wir haben gegenüber der ersten Zerlegungsstrategie ungefähr die Hälfte der wesentlichen Vergleiche eingespart. Interessanter ist der average case. Bei der Bestimmung des Medians von a_1, a_n und $a_{\lceil n/2 \rceil}$ vergleichen wir erst a_1 mit den anderen beiden Daten. Wenn a_1 der Median der drei Daten ist, genügen diese zwei wesentlichen Vergleiche. Dies ist bei Betrachtung aller Reihenfolgen der drei Daten mit Wahrscheinlichkeit $1/3$ der Fall. Ansonsten müssen die restlichen beiden Daten verglichen werden. Wir sparen also in der ersten Phase durchschnittlich $1/3$ wesentliche Vergleiche gegenüber dem worst case von n wesentlichen Vergleichen. Entscheidender ist, mit welcher Wahrscheinlichkeit das Zerlegungsdatum an Position j der sortierten Folge steht. Die drei betrachteten Daten nehmen in der sortierten Folge drei Positionen $i_1 < i_2 < i_3$ ein. Es gibt also $\binom{n}{3}$ Positionentripel, die alle gleich häufig auftreten. Damit der Median zur Position j gehört, muss $i_1 < j, i_2 = j$ und $i_3 > j$ sein. Also führen $(j-1)(n-j)$ Positionentripel dazu, dass wir das Problem in Blöcke der Länge $j-1$ und $n-j$ aufteilen. Für die average case Anzahl wesentlicher Vergleiche $W(n)$ gilt somit $W(0) = W(1) = 0, W(2) = 1$ und für $n \geq 3$

$$W(n) = n - \frac{1}{3} + \frac{(j-1)(n-j)}{\binom{n}{3}} \sum_{2 \leq j \leq n-1} (W(j-1) + W(n-j)).$$

Die Lösung dieser Rekursionsgleichung ist schwierig. Wir geben nur das Ergebnis an. Für $n \geq 6$ ist

$$W(n) = \frac{12}{7}(n+1)H(n-1) - \frac{477}{147}n + \frac{223}{147} + \frac{252}{147n} \approx 1,188 \log n - 2,255n.$$

Diese Variante von Quicksort stellt also eine echte Verbesserung dar und wird Clever Quicksort genannt.

Andererseits müssen wir bei Quicksort und Clever Quicksort darauf vertrauen, dass die von uns zu sortierenden Datenfolgen nicht schlecht strukturiert sind. Die den average case Analysen zugrundeliegenden Annahmen sind ja nicht unbedingt realistisch. Einen Ausweg aus diesem Dilemma kennen wir inzwischen: Randomisierung.

Zerlegungsstrategie 3: Wähle das Zerlegungsdatum zufällig unter den zu behandelnden Daten.

Zerlegungsstrategie 4: Wähle zufällig drei aus den zu behandelnden Daten und wähle als Zerlegungsdatum den Median dieser drei Daten.

Wir können nun die average case Analyse von Quicksort für die Zerlegungsstrategie 3 und von Clever Quicksort für die Zerlegungsstrategie 4 übernehmen und kommen zu denselben average case Rechenzeiten mit einem entscheidenden Unterschied. Der Durchschnitt bezieht sich nicht auf eine Wahrscheinlichkeitsverteilung auf dem Eingaberaum, sondern auf die Zufälligkeit der bei den Zerlegungsstrategien benutzten Zufallszahlen. Die Aussagen sind also average case Aussagen für jede Eingabe. Anders ausgedrückt: Es gibt keine worst case Eingaben mehr.

Abschließend noch einige Bemerkungen. Quicksort und Clever Quicksort benutzen außer dem Array für die Daten nur den Rekursionsstack. Bei geschickter Implementierung lässt sich der Extraplatz auf $O(\log n)$ beschränken (Übungsaufgabe). Dann arbeiten die Quicksort Varianten in situ. Sie sind nicht stabil, da z.B. a_1 und a_n vertauscht werden können

und a_n vorne bleibt, auch wenn $a_2 = \dots = a_n$ ist. Sie sind auch nicht adaptiv, da selbst im günstigsten Fall $n \log n - O(n)$ wesentliche Vergleiche anfallen. Manche Implementierungen der Quicksort-Varianten schalten für kleine Teilprobleme, etwa mit höchstens 10 Daten, auf Insertionsort um, da für diese kleinen Probleme dann der Verwaltungsaufwand sinkt.

4.5 Heapsort

Wenn wir zu unserem Experiment mit den Karteikarten zurückkehren, könnten wir versuchen, zunächst das lexikographisch kleinste Wort zu bestimmen, dann das zweitkleinste, usw. Sinnvollerweise sollten wir im Rechner dabei die bei der Berechnung des kleinsten Wortes gewonnenen Zusatzinformationen zur weiteren Verwendung abspeichern. Diese Idee führt schließlich zu Heapsort, einem Sortierverfahren, das Heaps (Haufen) als Datenstruktur verwendet.

Definition 4.5.1: Die Daten a_1, \dots, a_n in einem Array bilden einen (Min-)Heap, wenn für jede Position i die folgende Heapeigenschaft erfüllt ist. Das Datum a_{2i} , falls $2i \leq n$, und das Datum a_{2i+1} , falls $2i+1 \leq n$, sind nicht kleiner als a_i .

Diese Definition ist nicht sehr anschaulich. Daher werden Heaps zwar rechnerintern in Arrays verwaltet, zur Veranschaulichung aber als binäre Bäume dargestellt, wobei auf Ebene l die Arraypositionen $2^l, 2^l+1, \dots, 2^{l+1}-1$ von links nach rechts dargestellt werden. Dann sind alle Ebenen bis auf eventuell die letzte Ebene voll gefüllt, auf der letzten Ebene sind die Positionen von links aus gefüllt und die Kinder des Knotens für Position i im Array stellen die Positionen $2i$ und $2i+1$ im Array dar. Die Heapeigenschaft wird nun zu der Eigenschaft, dass die Daten an den Kindern eines Knotens nicht kleiner als das Datum am Elterknoten sind. In Abbildung 4.5.1 sind die Heappositionen für $n = 20$ dargestellt. Es ist klar, dass jeder Teilbaum eines Heaps selber ein Heap ist.

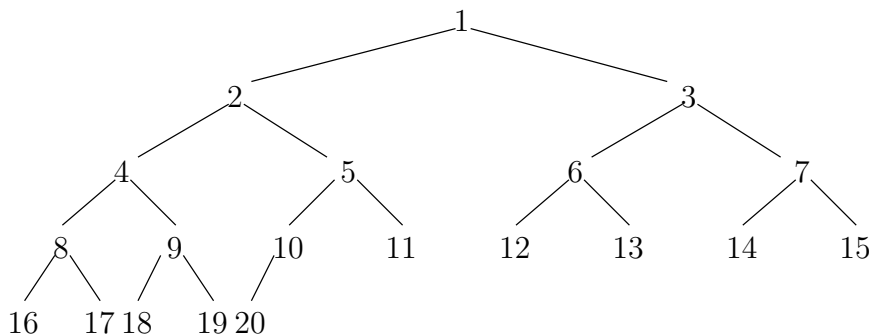


Abbildung 4.5.1: Die Heappositionen für $n = 20$.

In einem Heap können wir wie in einem Baum arbeiten, obwohl wir keine Zeiger haben. Die Kinder von Position i stehen an den Positionen $2i$ und $2i+1$ und der Elter an Position $\lfloor i/2 \rfloor$.

In der ersten Phase, der Heap Creation Phase, wird aus dem gegebenen Array ein Heap konstruiert. In der zweiten Phase, der Selection Phase, wird sortiert. Das Minimum ist bereits bekannt, es steht an der Wurzel. Es wird mit dem Datum an Position n vertauscht. Danach werden nur noch die Positionen $1, \dots, n-1$ betrachtet. Auf diesem Array wird die Heapeigenschaft wieder hergestellt. Das Datum an der Wurzel wird mit dem Datum an der letzten Position $n-1$ vertauscht, usw. Am Ende ist das Array absteigend sortiert. Zur Konstruktion eines Heaps und zur Reparatur eines Heaps nach dem Vertauschen von Daten ist die Prozedur $\text{reheap}(i, m)$ entscheidend. Sie betrachtet den Teilbaum mit Wurzel i und darin nur die Positionen $p \leq m$. Es wird angenommen, dass für alle Knoten des betrachteten Teilbaumes, mit Ausnahme der Wurzel, die Heapeigenschaft gesichert ist. Nach Anwendung der reheap -Prozedur soll die Heapeigenschaft überall in dem betrachteten Teilbaum gelten. Mit Hilfe der reheap -Prozedur lässt sich jede Heapsort-Variante folgendermaßen beschreiben.

Algorithmus 4.5.2: Heapsort-Rahmenprogramm

Input ist das Array a der Länge n .

(1) (Heap Creation Phase)

Für $i := \lfloor n/2 \rfloor, \dots, 1$: $\text{reheap}(i, n)$.

(2) (Selection Phase)

Für $m := n, \dots, 2$: vertausche $a(1)$ und $a(m)$, $\text{reheap}(1, m-1)$.

Die Korrektheit des Rahmenprogramms folgt leicht. Die Positionen $\lfloor n/2 \rfloor + 1, \dots, n$ haben keine Kinder, für sie ist die Heapeigenschaft trivialerweise stets erfüllt. Danach wird die Heapeigenschaft für alle inneren Knoten hergestellt, wobei beim Aufruf von $\text{reheap}(i, n)$ die Voraussetzungen für die Anwendung dieser Prozedur erfüllt sind. Es wurde bereits oben beschrieben, dass die zweite Phase die Daten eines Heaps sortiert.

Wir betrachten ein Array, in dem für alle Knoten bis auf die Wurzel die Heapeigenschaft gesichert ist. Wenn das Wurzeldatum nicht größer als eines der Kinder ist, dann haben wir bereits einen Heap. Ansonsten soll das Wurzeldatum „einsinken“. Das soll heißen, dass wir einen Teilpfad von der Wurzel zu einer Position i berechnen wollen, so dass wir einen Heap erhalten, wenn das Wurzeldatum an die Position i wechselt und alle anderen Daten auf diesem Teilpfad an die Position ihres Elters „aufsteigen“. Wie können wir entscheiden, ob dieser Teilpfad zum linken oder rechten Kind gehört? Damit die Heapeigenschaft an der Wurzel erfüllt wird, muss das kleinere der beiden Kinder „aufsteigen“. Somit ist der zu betrachtende Pfad ein Teilpfad des Pfades von der Wurzel zu einem Blatt, der stets das kleinere Kind wählt. Sind beide Kinder gleich groß, sind beide Richtungen erlaubt. Abbildung 4.5.2 zeigt symbolisch diesen Pfad von der Wurzel zu einem Blatt und Abbildung 4.5.3 mögliche Daten auf diesem Pfad.

Die Daten auf dem Pfad der „kleineren Kinder“ bilden bis auf das erste Element eine monoton wachsende Folge. Wenn wir im Beispiel die Daten 4, 11, 13, 28, 35, 46 im Heap eine Position aufsteigen lassen, haben wir an den neuen Positionen dieser Daten die Heapeigenschaft erfüllt, da ja das kleinere der Kinder aufgestiegen ist. Wenn das

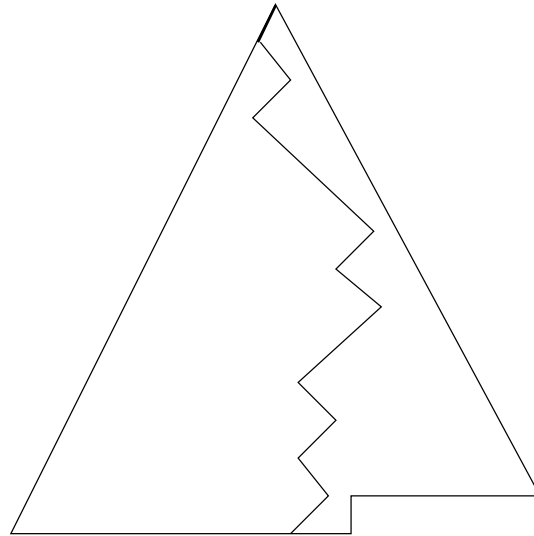


Abbildung 4.5.2: Ein Heap mit dem Pfad der „kleineren Kinder“.

57	4	11	13	28	35	46	58	64
----	---	----	----	----	----	----	----	----

Abbildung 4.5.3: Mögliche Daten auf dem Pfad der „kleineren Kinder“.

Wurzeldatum 57 an die Position, wo früher 46 stand, wechselt, ist dort die Heapeigenschaft erfüllt, da 57 kleiner als das kleinere der beiden Kinder dieser Position ist. An allen anderen Positionen haben sich die Daten und die Daten an den Kindern nicht verändert. Also ist reheap auf diese Weise erfolgreich.

Unser Ziel muss es also sein, die letzte Position p auf dem Pfad der kleineren Kinder zu berechnen, an der ein Datum kleiner als das Wurzeldatum steht. Wenn die Daten nicht verschieden sein müssen, können wir auch die letzte Position p , an der das Datum nicht größer als das Wurzeldatum ist, wählen. Wenn wir diese Position p kennen, können wir von da aus den Datenaustausch vornehmen. Am einfachsten ist dies, wenn wir die Positionen auf dem Weg der kleineren Kinder (maximal $\lfloor \log n \rfloor + 1$) extra abspeichern. Dann ist der Datenaustausch einfach ein zyklischer Shift auf diesen Positionen, in unserem Beispiel auf den Daten 57, 4, 11, 13, 28, 35, 46 mit dem Ergebnis 4, 11, 13, 28, 35, 46, 57.

Für die Berechnung von p müssen wir zwei Ziele verfolgen. Es muss der Pfad der kleineren Kinder verfolgt werden und es muss die richtige Position auf diesem Pfad berechnet werden. Wir werden dazu drei Strategien betrachten.

Strategie 1, die klassische Strategie, verfolgt beide Ziele simultan. Wenn die Position noch nicht erreicht ist, werden zwei wesentliche Vergleiche durchgeführt. Zunächst wird das kleinere der Kinder bestimmt und dann mit dem Wurzeldatum verglichen. Damit wird entschieden, ob die erreichte Position die richtige ist (kein Kind ist kleiner), und andernfalls, welches der beiden Kinder kleiner ist und damit auf dem Pfad der kleineren Kinder liegt. Es ist klar, dass ein Vergleich eingespart werden kann, wenn es nur noch ein Kind gibt, und dass die richtige Position sicher erreicht ist, wenn es keine Kinder

mehr gibt. In unserem Beispiel werden 7 Ebenen betrachtet und 14 wesentliche Vergleiche durchgeführt.

Die Strategien 2 und 3 trennen die beiden Aufgaben. Zunächst wird der gesamte Pfad der kleineren Kinder bestimmt und die Positionen werden abgespeichert. Für diese Phase ist die Anzahl der wesentlichen Vergleiche gleich der Länge des Pfades von der Wurzel zu einem Blatt, also entweder gleich der Baumtiefe oder um 1 kleiner. In unserem Beispiel sind es 8 wesentliche Vergleiche. Mit dem Array der Positionen des Pfades der kleineren Kinder können wir so arbeiten, als hätten wir das Array der Daten selber. Also können wir eine binäre Suche starten (Strategie 2). Im Beispiel ist die Arraylänge 8, wir vergleichen 57 mit 28, dann mit 46 und schließlich mit 58. Es werden also drei wesentliche Vergleiche durchgeführt. Gerade in der Selection Phase bringen wir ja zunächst ein tendenziell großes Datum vom letzten Blatt an die Wurzel und müssen erwarten, dass es tief einsinkt. Daher bildet die lineare Suche von unten (die bottom-up Strategie 3) eine sinnvolle Alternative. In unserem Beispiel braucht sie ebenfalls 3 wesentliche Vergleiche, so dass die Strategien 2 und 3 je 11 wesentliche Vergleiche brauchen. Wir können erwarten, dass die Strategie 2 ein besonders gutes worst case Verhalten hat. Im Vergleich zwischen Strategie 1 und Strategie 3 lässt sich feststellen, dass beide ungefähr gleich viele wesentliche Vergleiche brauchen, nämlich $(4/3)d$ Vergleiche bei Baumtiefe d , wenn die gesuchte Position auf Ebene $(2/3)d$ liegt. Bei Strategie 1 sind es je zwei Vergleiche für $(2/3)d$ Ebenen, bei Strategie 3 für die erste Phase d Vergleiche und dann bei der bottom-up Suche der zweiten Phase $(1/3)d$ Vergleiche.

Um die Strategien analysieren zu können, spielt sicherlich die Summe der Baumtiefen aller reheap Aufrufe eine wesentliche Rolle.

Lemma 4.5.3: Die Summe der Tiefen der Bäume, für die in der Heap Creation Phase die Prozedur reheap aufgerufen wird, ist kleiner als n .

Beweis: Wir wollen die Tiefen der Bäume mit Wurzel i , $1 \leq i \leq \lfloor n/2 \rfloor$, aufsummieren. Um die Anschaulichkeit zu erhöhen, beschreiben wir die folgenden Gleichungen informal.

$$\begin{aligned}
& \sum_{1 \leq i \leq \lfloor n/2 \rfloor} \text{Tiefe des Baumes mit Wurzel } i \\
&= \sum_{1 \leq d \leq \lfloor \log n \rfloor} d \cdot \text{Anzahl der Bäume, deren Tiefe genau } d \text{ ist} \quad (*) \\
&= \sum_{1 \leq d \leq \lfloor \log n \rfloor} \text{Anzahl der Bäume, deren Tiefe mind. } d \text{ beträgt} \quad (**)
\end{aligned}$$

Ein Baum der Tiefe d^* führt in $(*)$ einmal zum Summanden d^* und in $(**)$ d^* -mal, $1 \leq d \leq d^*$, zum Summanden 1. Dieser Rechenrick ist oft sehr hilfreich. Es werden $\lfloor n/2 \rfloor$ Bäume mit den Wurzeln $1, \dots, \lfloor n/2 \rfloor$ betrachtet. Nur die Knoten $1, \dots, \lfloor n/4 \rfloor$ sind Wurzeln von Bäumen, deren Tiefe mindestens 2 beträgt. Allgemein sind nur die Knoten $1, \dots, \lfloor n2^{-d} \rfloor$ Wurzeln von Bäumen, deren Tiefe mindestens d beträgt. Indem wir $\lfloor n2^{-d} \rfloor$ für $1 \leq d \leq \lfloor \log n \rfloor$ aufsummieren, zählen wir Bäume der Tiefe d genau d -mal. Es gilt

$$\sum_{1 \leq d \leq \lfloor \log n \rfloor} \lfloor n2^{-d} \rfloor < \sum_{1 \leq d < \infty} n2^{-d} = n.$$

□

Lemma 4.5.4: Die Summe der Tiefen der Bäume, für die in der Selection Phase die Prozedur `reheap` aufgerufen wird, ist kleiner als $n \log n$.

Beweis: Der Gesamtbaum hat Tiefe $\lfloor \log n \rfloor$ und es gibt $n - 1$ Aufrufe von `reheap`. □

Satz 4.5.5: Die worst case Zahl wesentlicher Vergleiche aller Heapsort-Varianten ist durch $2n \log n + 2n$ beschränkt. Die Zahl anderer Operationen ist nur um einen konstanten Faktor größer. Für den Aufbau eines Heaps genügen $2n$ wesentliche Vergleiche.

Beweis: Bei Baumtiefe d genügen stets $2d$ wesentliche Vergleiche. Damit folgt die Aussage aus den Lemmas 4.5.3 und 4.5.4. □

Für Strategie 1 ist die Abschätzung aus Satz 4.5.5 fast optimal. Für die anderen beiden Strategien können bessere Schranken bewiesen werden.

Satz 4.5.6: Die Heapsort-Variante, die mit Strategie 2 (Pfadsuche und binäre Suche) arbeitet, kommt im worst case mit $n \log n + n \log \log n + 3n$ wesentliche Vergleichen aus.

Beweis: Für die Heap Creation Phase bleiben wir bei der Abschätzung $2n$ für die Anzahl wesentlicher Vergleiche. In der Selection Phase genügen für jeden der $n - 1$ `reheap` Aufrufe $\lfloor \log n \rfloor$ wesentliche Vergleiche, um den Pfad der kleineren Kinder zu berechnen. Für die binäre Suche genügen jeweils $\lceil \log(\lfloor \log(n - 1) \rfloor) \rceil$ wesentliche Vergleiche, da die Heaps maximal $n - 1$ Daten enthalten. Dies lässt sich durch $\log \log n + 1$ abschätzen. □

Eine gute obere Schranke für die worst case Anzahl an wesentlichen Vergleichen ist für Strategie 3 schwieriger zu beweisen. Die bottom-up Suche kann manchmal bis zur Wurzel führen, aber eben nur manchmal.

Satz 4.5.7: Die Heapsort-Variante, die mit Strategie 3 (Pfadsuche und lineare bottom-up Suche) arbeitet, kommt im worst case mit $(3/2)n \log n + O(n)$ wesentlichen Vergleichen aus.

Beweis: Die Berechnung aller Pfade der kleineren Kinder kostet höchstens $n \log n + n$ wesentliche Vergleiche, die bottom-up Suchen in der Heap Creation Phase höchstens n wesentliche Vergleiche.

Nun genügt es zu zeigen, dass die bottom-up Suchen in der Selection Phase nicht mehr als $(1/2)n \log n + O(n)$ wesentliche Vergleiche verursachen. Wir beschränken uns auf den Fall $n = 2^k$, da sich die Argumente leicht auf allgemeines n übertragen lassen. Wir zeigen, dass die ersten $n/2$ bottom-up Suchen mit $(1/4)n \log n + O(n)$ Vergleichen auskommen. Danach haben wir einen Heap mit 2^{k-1} Daten und können analog argumentieren.

Es sei $d(m)$ die Ebene, in der das Datum an der Wurzel nach dem m -ten Aufruf von `reheap` landet. Dann ist $k - d(m)$ eine obere Schranke für die Zahl der für den m -ten Aufruf der bottom-up Suche benötigten wesentlichen Vergleiche.

Wir bezeichnen ein Datum als groß, wenn es zur größeren Hälfte der Daten gehört und sonst als klein. In einem Heap haben große Daten nur große Nachfolger. Also ist mindestens die Hälfte der Daten an den Blättern groß. Große Blätter können (beim Datenaustausch) nicht durch kleine Daten verdrängt werden. Daher sind während der ersten 2^{k-1} Runden mindestens 2^{k-2} der Vergleichsdaten, die an die Wurzel wechseln, groß.

Jede bottom-up Suche verursacht höchstens $\log n - 1$ Vergleiche. Wir schätzen die Kosten für 2^{k-2} Runden, darunter alle Runden mit kleinen Wurzeldaten, mit $(1/4)n \log n - (1/4)n$ ab. Es bleibt zu zeigen, dass die übrigen 2^{k-2} Runden nur Kosten $O(n)$ verursachen. Alle Wurzeldaten sind hierbei groß und daher nach 2^{k-1} Runden noch im Heap vorhanden. Ein Datum kann nur dann ein weiteres Mal Wurzeldatum werden, wenn es bei der letzten bottom-up Suche, bei dem es als Wurzeldatum beteiligt war, nur einen Vergleich verursacht hat und daher wieder in einem Blatt abgespeichert wurde. Dann steigt das frühere Blatt eine Position auf. Die Summe der zu diesen 2^{k-2} Runden gehörigen $d(m)$ -Werte ist also mindestens so groß wie die Summe der Tiefen von 2^{k-2} Knoten in einem binären Baum. Die Summe beträgt mindestens (für die Rechnung vergleiche Beispiel 1.6.4)

$$\sum_{1 \leq i \leq k-3} i2^i + k - 2 = (k-4)2^{k-2} + 2 + k - 2 \geq (k-4)2^{k-2}.$$

Also beträgt die zugehörige Summe aller $k - d(m)$ höchstens $4 \cdot 2^{k-2} = n$. Damit ist der Satz bewiesen. \square

(Schwierig zu beschreibende) Beispiele zeigen, dass die Konstante $3/2$ in Satz 4.5.7 nicht verkleinert werden kann. Average case Analysen für die Strategien 1 und 3 sind recht schwierig. Bei Strategie 2 ist dies anders, da selbst der best-case nur um $3n$ besser als der worst case sein kann. In jeder der zwei Phasen der höchstens $(3/2)n$ reheap Aufrufe unterscheiden sich ja worst case und best case um höchstens einen wesentlichen Vergleich. Die hier nicht durchgeführten average case Analysen der Strategien 1 und 3 zeigen die Überlegenheit von Strategie 3. Diese Strategie erfordert im average case nur $n \log n + O(n)$ wesentliche Vergleiche. (Experimente lassen vermuten, dass der lineare Term kleiner als $n/2$ ist.) Dagegen benötigt Strategie 1 im average case $2n \log n - O(n)$ wesentliche Vergleiche.

Aus der Beschreibung folgt, dass die Heapsort-Varianten in situ arbeiten, aber weder stabil noch adaptiv sind.

4.6 Eine untere Schranke für allgemeine Sortierverfahren

Unsere Sortieralgorithmen scheinen an eine Barriere zu stoßen: $n \log n$. Sind wir nur zu dumm, bessere Sortieralgorithmen zu entwerfen oder gibt es keine $o(n \log n)$ Sortierverfahren? Wir zeigen eine untere Schranke für allgemeine Sortierverfahren. Ein Sortierverfahren heißt ja allgemein, wenn es beliebige Folgen a_1, \dots, a_n von Elementen aus beliebigen geordneten Mengen sortieren kann. In dieser allgemeinen Situation ergeben nur wesentliche Vergleiche Informationen über die Ordnung der vorliegenden Daten.

Da es $n!$ Permutationen auf $\{1, \dots, n\}$ gibt, gibt es $n!$ so genannte Ordnungstypen auf Folgen der Länge n . Der Ordnungstyp (l_1, \dots, l_n) bedeutet, dass das Datum, das in der

Eingabe an Position l_j steht, in der Ausgabe an Position j steht. Mit der sortierten Folge ist (bei paarweise verschiedenen Daten) implizit auch der Ordnungstyp bekannt. Wir können also das allgemeine Sortierproblem mit der Berechnung des Ordnungstyps identifizieren.

Wir befreien nun allgemeine Sortierverfahren vom Verwaltungsoverhead und betrachten nur die wesentlichen Vergleiche. Die so befreiten Sortieralgorithmen lassen sich durch binäre Entscheidungsbäume gut darstellen, wobei wir nur Eingaben mit paarweise verschiedenen Daten betrachten.

Zu Beginn sind alle $n!$ Ordnungstypen möglich. Der erste Vergleich betrifft a_i und a_j . Die Menge der Ordnungstypen wird in zwei disjunkte Mengen von Ordnungstypen (falls $a_i < a_j$, steht i im Ordnungstyp vor j , sonst dahinter) zerlegt. Welches der zweite wesentliche Vergleich ist, hängt nur vom Ausgang des ersten wesentlichen Vergleichs (und vom Algorithmus) ab. In jedem Zweig des Entscheidungsbaumes darf der Sortieralgorithmus erst stoppen, wenn nur noch ein Ordnungstyp möglich ist. Für jedes allgemeine Sortierverfahren, das keine Vergleiche durchführt, deren Ergebnisse bereits aus früheren Vergleichen ableitbar sind, erhalten wir also einen binären Entscheidungsbaum mit $n!$ Blättern, für jeden Ordnungstyp ein Blatt. Sei d_π die Länge des Weges von der Wurzel zum Blatt π . Wenn die Eingabe vom Ordnungstyp π ist, benötigt der betrachtete Sortieralgorithmus genau d_π wesentliche Vergleiche. Die worst case Anzahl wesentlicher Vergleiche des Algorithmus beträgt also $\max\{d_\pi \mid \pi \text{ Ordnungstyp}\}$ und die average case Anzahl

$$\frac{1}{n!} \sum_{\pi \text{ Ordnungstyp}} d_\pi.$$

Beispiel 4.6.1: Wir betrachten für $n = 4$ die Quicksort-Variante, die als Vergleichsdatum immer das linkeste Datum im Array auswählt. Aus Kapitel 4.4 wissen wir, dass die worst case Anzahl wesentlicher Vergleiche $\binom{4}{2} = 6$ beträgt. Die average case Anzahl beträgt nach Satz 4.4.2

$$2(4 + 1) \left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \right) - 4 \cdot 4 = \frac{29}{6}.$$

Dies können wir nun auch am Entscheidungsbaum in Abbildung 4.6.1 ablesen. In ihm haben 12 Blätter Tiefe 4, 4 Blätter Tiefe 5 und 8 Blätter Tiefe 6. Es ist $\frac{1}{24}(12 \cdot 4 + 4 \cdot 5 + 8 \cdot 6)$ ebenfalls $\frac{29}{6}$. An jeden Knoten haben wir zwei Paare geschrieben. Das obere Paar (i, j) gibt an, dass a_i und a_j verglichen werden. Um den Verlauf von Quicksort besser verfolgen zu können, gibt das untere Paar (i', j') an, dass a_i momentan an Position i' und a_j an Position j' steht.

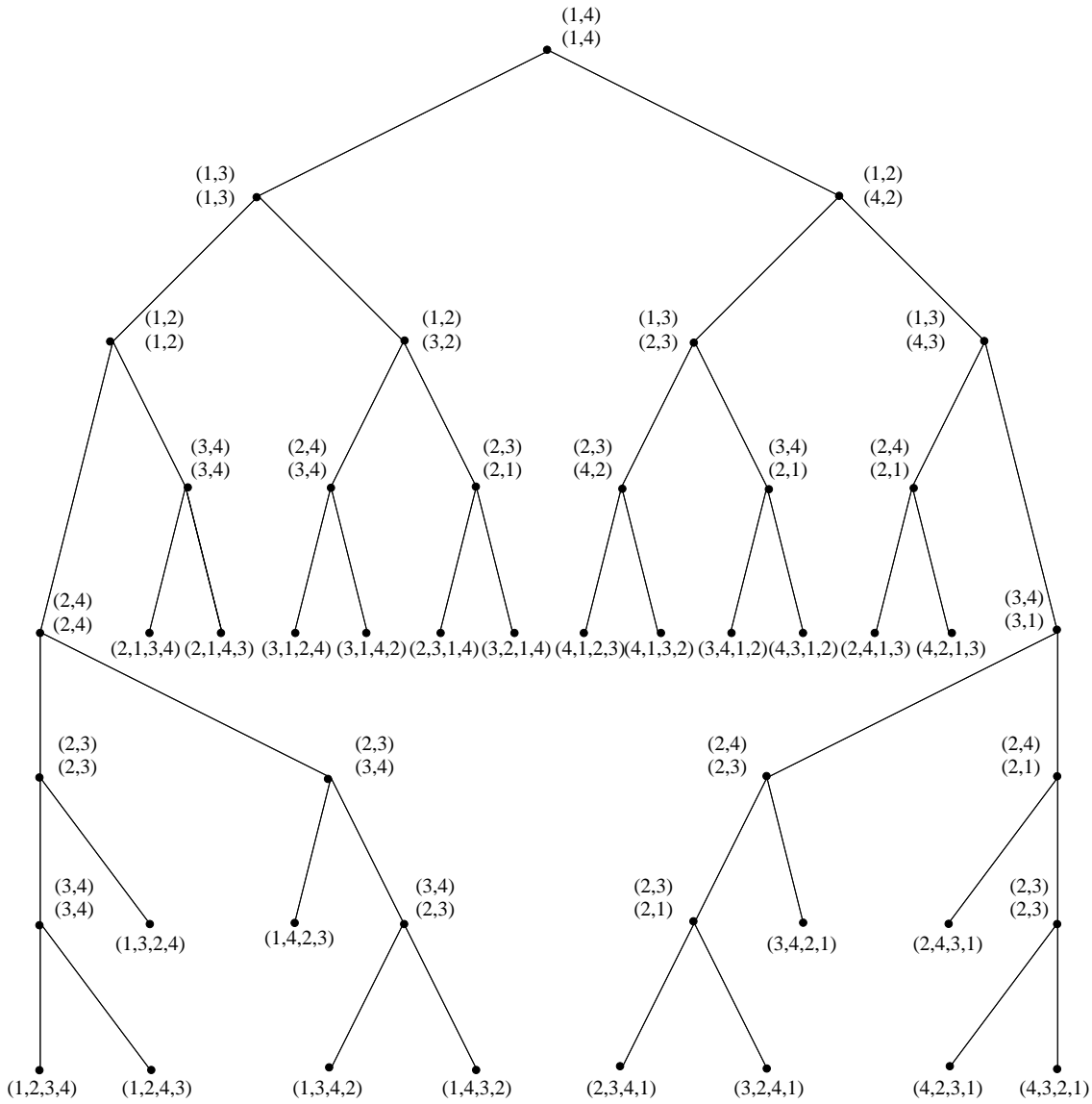


Abbildung 4.6.1: Der Entscheidungsbaum für Quicksort und $n = 4$.

Untere Schranken für allgemeine Sortiervverfahren erhalten wir also durch untere Schranken für die Tiefe und die durchschnittliche Tiefe von binären Bäumen mit $N = n!$ Blättern.

Lemma 4.6.2: Die Tiefe jedes binären Baumes mit N Blättern beträgt mindestens $\lceil \log N \rceil$.

Beweis: Der binäre Baum der Tiefe d mit der größten Zahl an Blättern ist ein vollständiger binärer Baum der Tiefe d und hat 2^d Blätter. Damit $2^d \geq N$ ist, muss $d \geq \lceil \log N \rceil$ sein. \square

Lemma 4.6.3: In einem Binärbaum mit N Blättern beträgt die durchschnittliche Tiefe der Blätter mindestens $\lceil \log N \rceil - 1$.

Beweis: Sei T ein binärer Baum mit N Blättern. Wir ersetzen diesen Baum durch einen anderen, dessen durchschnittliche Tiefe nicht größer ist und schätzen dann die durchschnittliche Tiefe des so erhaltenen Baumes ab. Sei d die Tiefe von T . Wenn ein Knoten auf Tiefe $d - 1$ nur ein Kind hat, können wir die durchschnittliche Tiefe senken, indem wir dieses Kind eliminieren.

Falls es ein Blatt in Tiefe $d' \leq d - 2$ gibt, können wir die durchschnittliche Tiefe verringern. Wir betrachten folgende drei Blätter: B auf Tiefe d' und zwei Geschwister auf Tiefe d . Die Summe der Tiefen dieser drei Blätter ist $d' + 2d$. Wir geben B zwei Kinder und erzeugen zwei Blätter auf Tiefe $d' + 1$, und wir streichen die beiden Geschwister und erzeugen ein Blatt auf Tiefe $d - 1$. Die drei Ersatzblätter haben als Summe ihrer Tiefen

$$2(d' + 1) + d - 1 = 2d' + d + 1 \leq d' + 2d - 1 < d' + 2d.$$

Diesen Prozess setzen wir fort, bis wir einen Baum T' erhalten, dessen durchschnittliche Tiefe nicht größer als die von T ist und der nur Blätter auf zwei aufeinanderfolgenden Ebenen $d - 1$ und d hat.

Wenn $d \leq \lceil \log N \rceil - 1$ ist, kann der Baum nach Lemma 4.6.2 nicht N Blätter haben. Für $d = \lceil \log N \rceil$ gibt es einen Baum mit N Blättern, für größere d wäre die durchschnittliche Tiefe größer. Also liegen alle Blätter auf den Ebenen $\lceil \log N \rceil - 1$ und $\lceil \log N \rceil$ und die durchschnittliche Tiefe beträgt mindestens $\lceil \log N \rceil - 1$. \square

Wir erhalten nun unser Hauptergebnis als einfaches Korollar, wobei wir uns an die Approximation von $\log(n!)$ aus Kapitel 4.2 erinnern.

Satz 4.6.4: Jedes allgemeine Sortiervverfahren benötigt im worst case mindestens $\lceil \log(n!) \rceil \approx n \log n - 1, 4427n$ Vergleiche und im average case mindestens $\lceil \log(n!) \rceil - 1$ Vergleiche.

Im Licht dieser unteren Schranken haben wir bereits sehr gute Sortieralgorithmen entworfen.

Was können wir aus diesem Kapitel, also dem Beweis unterer Schranken für den Entwurf guter Algorithmen lernen? Vergleiche sollten so ausgewählt werden, dass sie die Menge noch möglicher Ordnungstypen in zwei möglichst gleich große Mengen zerlegen. In diesem Sinn ist der erste Vergleich jedes Sortiervverfahrens gut. Der zweite Vergleich für Quicksort in Abbildung 4.6.1 ist in diesem Sinn nicht gut. Die jeweils noch möglichen 12 Ordnungstypen werden im Verhältnis 8:4 aufgeteilt.

Es folgt eine Übersicht über die bisher behandelten Sortieralgorithmen.

	Vergleiche worst case	Vergleiche average case	Sonstige Operationen	Extra- platz
Insertionsort	$n \log n - 0,443n$	kaum weniger	$\Theta(n^2)$	$O(1)$
Quicksort	$n^2/2$	$1,386n \log n - 2,846n$	$O(\# \text{Vergleiche})$	$O(\log n)$
Clever Quicksort	$n^2/4$	$1,188n \log n - 2,255n$	$O(\# \text{Vergleiche})$	$O(\log n)$
Heapsort klassisch	$2n \log n + O(n)$	$2n \log n \pm O(n)$	$O(n \log n)$	$O(1)$
Bottom-up Heapsort mit binärer Suche	$n \log n + n \log \log n + 3n$	kaum weniger	$O(n \log n)$	$O(1)$
Bottom-up Heapsort mit linearer Suche	$1,5n \log n + O(n)$	$n \log n + O(n)$	$O(n \log n)$	$O(\log n)$
Mergesort	$n \log n - n$	kaum weniger	$O(n \log n)$	$O(n)$
untere Schranke	$n \log n - 1,443n$	$n \log n - 1,443n$	$\Omega(n)$	$\Omega(1)$

4.7 Bucketsort

Wir haben in Kapitel 4.6 betont, dass die unteren Schranken nur für allgemeine Sortiervverfahren gelten. Gibt es nun Situationen, in denen spezielle Sortiervverfahren nutzen? Sehr häufig müssen Namen sortiert werden. Auf Überweisungsformularen sind dafür 27 Stellen vorgesehen. Es werden 29 Zeichen verwendet, die 26 Buchstaben, der Bindestrich, das Komma und das Leerzeichen. Auf diesen Zeichen gilt folgende Ordnung (\square = Leerzeichen)

$$\square < A < B < \dots < Z < - < , < .$$

Auf den Wörtern gilt die lexikographische Ordnung:

$$(b_k, \dots, b_1) < (c_k, \dots, c_1) :\Leftrightarrow \exists i : b_i < c_i \text{ und } b_j = c_j \text{ für } j > i.$$

Das Alphabet mit 29 „Buchstaben“ können wir mit $1, \dots, 29$ identifizieren und die Menge der Wörter mit $1, \dots, 29^{27} \approx 3 \cdot 10^{39}$.

Allgemein nehmen wir nun an, dass $a_1, \dots, a_n \in \{1, \dots, M\}$ zu sortieren sind.

Algorithmus 4.7.1: (1) Initialisiere ein Array der Länge M mit M leeren Listen $L(1), \dots, L(M)$, für die Zeiger auf das letzte Element verwaltet werden.

(2) Durchlaufe die Eingabe und hänge a_i an das Ende von $L(a_i)$.

(3) Hänge $L(1), \dots, L(M)$ aneinander.

Lemma 4.7.2: Zum Sortieren von n Daten aus $\{1, \dots, M\}$ genügen $O(n + M)$ Operationen.

Die Bezeichnung Bucketsort kommt daher, dass die Listen als Eimer aufgefasst werden und wir Daten mit Wert i in den Eimer mit Nummer i werfen. Auf die Reihenfolge der Daten mit demselben Wert kommt es ja auch eigentlich nicht an. Die Einfügung am Ende der Listen sichert jedoch, dass das Sortiervorgehen stabil ist, und das wird sich als wichtig erweisen.

In unserem Beispiel war $M = 29^{27}$, Algorithmus 4.7.1 lohnt sich also für realistische n nicht. Wir nutzen nun die Struktur unserer Daten besser aus.

Algorithmus 4.7.3: Verallgemeinerter Bucketsort für n Daten $a_i = (a_{i1}, \dots, a_{il}) \in \{1, \dots, M\}^l$ versehen mit der lexikographischen Ordnung.

(1) Führe l Runden von Algorithmus 4.7.1 aus, wobei die k -te Runde die Wörter a_i bezüglich a_{ik} sortiert.

Satz 4.7.4: Zum Sortieren von n Daten aus $\{1, \dots, M\}^l$ (bezüglich der lexikographischen Ordnung) genügen $O(l(n + M))$ Operationen.

Beweis: Wir benutzen Algorithmus 4.7.3. Die Zeitschranke folgt aus Lemma 4.7.2. Wir beweisen die Korrektheit. Sei $a_i < a_j$ und $a_{ik} < a_{jk}$, aber $a_{im} = a_{jm}$ für $m > k$. In der k -ten Runde wird a_i vor a_j einsortiert. Aufgrund der Stabilität von Algorithmus 4.7.1 bleibt diese Reihenfolge in den folgenden Runden erhalten. \square

In unserem Beispiel ist $l = 27$ und $M = 29$. Zusätzlich ist zu beachten, dass ein wesentlicher Vergleich zweier Wörter teurer ist als die in Algorithmus 4.7.3 durchgeführten Vergleiche von Buchstaben.

Da die Eingabe aus ln Buchstaben besteht, und normalerweise $n > M$ ist, ist die Rechenzeit von $O(l(n + M)) = O(ln)$ Operationen auf Buchstaben asymptotisch optimal.

Der verallgemeinerte Bucketsort aus Algorithmus 4.7.3 ist für unser Experiment mit den Karteikarten und dreibuchstabigen Wörtern vermutlich die beste Alternative.

4.8 Das Auswahlproblem

Es gibt Situationen, in denen wir die Daten nicht vollständig sortieren wollen, sondern nur das Datum mit Rang k bestimmen wollen. Dann sprechen wir vom Auswahlproblem, speziell für $k = \lceil n/2 \rceil$ vom Medianproblem.

Kann das Auswahlproblem mit $O(n)$ wesentlichen Vergleichen gelöst werden? Es gibt Verfahren, die im worst case mit $O(n)$ Vergleichen auskommen, allerdings sind diese bisher wegen der großen Konstanten nur von theoretischem Interesse.

Wir besprechen daher einen randomisierten Algorithmus, der ähnlich wie Quicksort im worst case quadratische Komplexität hat, aber bezüglich der average case Rechenzeit sehr effizient ist. Wir nehmen bei der Darstellung an, dass alle Daten paarweise verschieden sind.

Algorithmus 4.8.1: Quickselect $((a_1, \dots, a_n), k)$.

- (1) Wähle zufällig ein $i \in \{1, \dots, n\}$.
- (2) Führe mit a_i als Zerlegungsdatum die erste Phase von Quicksort aus. Es sei r die Position, die a_i schließlich einnimmt.
- (3) Falls $r = k$, stoppe mit Ausgabe a_i . Falls $r > k$, wende Quickselect auf die ersten $r - 1$ Daten im Array und den Rang k an. Falls $r < k$, wende Quickselect auf die hinteren $n - r$ Daten im Array und den Rang $k - r$ an.

Der Algorithmus ist offensichtlich korrekt. Für jedes $r \in \{1, \dots, n\}$ wird das Datum mit Rang r in Schritt (1) mit Wahrscheinlichkeit $1/n$ gewählt. Es entstehen also jeweils mit Wahrscheinlichkeit $1/n$ Probleme der Größe

$$n - 1, \dots, n - k + 1, 0, k, \dots, n - 1.$$

Sei $V(n)$ die durchschnittliche Zahl an Vergleichen für das Auswahlproblem und den worst case Rang. Dann gilt $V(1) = 0$ und

$$V(n) \leq n - 1 + \frac{1}{n} \left(\sum_{k \leq i \leq n-1} V(i) + \sum_{n-k+1 \leq i \leq n-1} V(i) \right) \quad (*)$$

Wir vermuten, dass V eine monoton wachsende Funktion ist. Dann nimmt die rechte Seite für $k = \lceil n/2 \rceil$ ihr Maximum an und es ist

$$V(n) \leq n - 1 + \frac{1}{n} \left(\sum_{\lceil n/2 \rceil \leq i \leq n-1} V(i) + \sum_{\lfloor n/2 \rfloor + 1 \leq i \leq n-1} V(i) \right).$$

Wir vermuten weiter, dass $V(n)$ linear wächst, auf der rechten Seite ist die durchschnittliche Problemgröße $3n/4$. Da

$$1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \dots = 4$$

ist, vermuten wir, dass $V(n) \leq 4n$ ist. Dies zeigen wir mit Induktion. Für $n = 1$ gilt die Vermutung. Da die Funktion $4n$ monoton wachsend ist, erhalten wir bei Abschätzung von $V(i)$ durch $4i$ den größten Wert auf der rechten Seite von $(*)$ für $k = \lceil n/2 \rceil$. Also folgt nach Induktionsvoraussetzung

$$\begin{aligned} V(n) &\leq n - 1 + \frac{1}{n} \left(\sum_{\lceil n/2 \rceil \leq i \leq n-1} 4i + \sum_{\lfloor n/2 \rfloor + 1 \leq i \leq n-1} 4i \right) \\ &= n - 1 + \frac{4}{n} \left(n(n-1) - \lceil n/2 \rceil (\lceil n/2 \rceil - 1)/2 - (\lfloor n/2 \rfloor + 1) \lfloor n/2 \rfloor / 2 \right) \\ &= n - 1 + \frac{4}{n} \left(n^2 - n - \lceil n/2 \rceil^2 / 2 - \lfloor n/2 \rfloor^2 / 2 + \lceil n/2 \rceil / 2 - \lfloor n/2 \rfloor / 2 \right) \end{aligned}$$

Es gilt

$$\lceil n/2 \rceil^2 + \lfloor n/2 \rfloor^2 \geq \left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)^2 = n^2/2.$$

Also ist

$$V(n) \leq n + \frac{4}{n} \cdot \frac{3}{4} n^2 = 4n.$$

Satz 4.8.2: Die erwartete Anzahl wesentlicher Vergleiche von Quickselect ist durch $4n$ nach oben beschränkt.

In der Analyse haben wir den worst case Rang $\lceil n/2 \rceil$ benutzt. Wenn wir den Median suchen, suchen wir auf der zweiten Stufe des Algorithmus häufig nicht ein Datum von mittlerem Rang in dem dann betrachteten Teilarray. Unsere Analyse ist also zu pessimistisch. Auf eine genauere Analyse wollen wir aber verzichten. Es kann gezeigt werden, dass die erwartete Anzahl wesentlicher Vergleiche $2(1 + \ln 2)n + o(n) \approx 3,39n$ ist.

4.9 Sortieren auf Parallelrechnern

Wir wollen für das Sortierproblem exemplarisch diskutieren, wie stark die Rechenzeit sinken kann, wenn wir einen Parallelrechner (Multiprozessorsystem) zur Verfügung haben, so dass Vergleiche verschiedener Daten gleichzeitig ausgeführt werden können. Da auf diese Weise die Anzahl von Vergleichen in einem Rechenschritt durch $n/2$ beschränkt ist, benötigen allgemeine Sortierverfahren in dieser Situation mindestens eine Zeit von $2 \log n - O(1)$. Die bisher bekannten Algorithmen, die eine worst case Zeit von $O(\log n)$ garantieren, haben eine so große Konstante in der Rechenzeit, dass der hier vorgestellte Algorithmus mit Rechenzeit $\Theta(\log^2 n)$ für praktisch relevante Werte von n schneller ist.

Eine Betrachtung der behandelten allgemeinen Sortierverfahren zeigt, dass nur Mergesort sich für die Parallelisierung anbietet. Die beiden Hälften können gleichzeitig sortiert werden. Allerdings müssen wir das Mischen zweier sortierter Folgen „parallelisieren“. Der folgende Algorithmus, den wir nur für Zweierpotenzen $n = 2^k$ beschreiben, heißt Batchmerge (BM).

Algorithmus 4.9.1: $\text{BM}(a_1, \dots, a_n; b_1, \dots, b_n)$, wobei $a_1 \leq \dots \leq a_n$ und $b_1 \leq \dots \leq b_n$ ist.

- (1) Falls $n = 1$, vergleiche a_1 und b_1 , setze $z_1 = \min(a_1, b_1)$ und $z_2 = \max(a_1, b_1)$, STOP.
- (2) Falls $n > 1$,

$$(v_1, \dots, v_n) := \text{BM}(a_1, a_3, \dots, a_{n-1}; b_1, b_3, \dots, b_{n-1})$$

$$(w_1, \dots, w_n) := \text{BM}(a_2, a_4, \dots, a_n; b_2, b_4, \dots, b_n).$$
 (Diese beiden BM-Aufrufe werden gleichzeitig bearbeitet.)
- (3) Vergleiche v_{i+1} und w_i , $1 \leq i \leq n-1$. Diese Vergleiche werden gleichzeitig ausgeführt. Setze $z_1 = v_1$, $z_{2i} = \min(v_{i+1}, w_i)$, $z_{2i+1} = \max(v_{i+1}, w_i)$, $1 \leq i \leq n-1$, $z_{2n} = w_n$.

Wegen der Auswahl der Daten in Schritt (2) wird Batchmerge auch Odd-Even-Merge genannt.

Beim ersten Lesen deutet nichts auf die Korrektheit von Batchmerge hin. Wir betrachten a_j . In der Ausgabe sind für a_j noch die Rangplätze $j, \dots, j + n$ möglich, das sind $n + 1$ Rangplätze. Wir zeigen nun, dass nach Schritt (2) nur noch 2 Rangplätze für v_{i+1} in Frage kommen, und zwar $2i$ und $2i + 1$, analog für w_i . Dabei sollte klar sein, dass v_1 das Minimum und w_n das Maximum aller Daten ist.

Wir illustrieren unseren Beweis zunächst an einem Beispiel. Sei $n = 128$. Wir betrachten das Element v_{85} , das o.B.d.A. aus der a -Folge stammt. Sei dies (1) a_{33} ((2) a_{101}). In diesem Fall enthält v_1, \dots, v_{84} die Elemente a_1, a_3, \dots, a_{31} , also 16 a -Elemente (a_1, a_3, \dots, a_{99} , also 50 a -Elemente) und somit 68 b -Elemente und zwar b_1, \dots, b_{135} (34 b -Elemente und zwar b_1, \dots, b_{67}). Also gilt

$$b_{135} \leq a_{33} \leq b_{137} \quad (b_{67} \leq a_{101} \leq b_{69}).$$

Wir kennen nun die Größenbeziehung von $a_{33}(a_{101})$ zu allen bis auf ein Datum. Unklar ist nur die Beziehung zu $b_{136}(b_{68})$. Kleiner als a_{33} sind mit Sicherheit 32 a -Elemente und 135 b -Elemente, zusammen 167 Elemente (100 a -Elemente und 67 b -Elemente, zusammen 167 Elemente). Also sind für v_{85} in *beiden* Fällen nur die Rangplätze 168 und 169 möglich. Das kann doch kein Zufall sein. Wir verallgemeinern unsere Überlegungen.

Das Datum v_{i+1} stammt aus einer der beiden Folgen, o.B.d.A. $v_{i+1} = a_{2j-1}$. Von allen a -Daten wissen wir, ob sie größer oder kleiner als v_{i+1} sind, $2j - 2$ Daten sind kleiner und $n - 2j + 1$ größer. In der v -Folge suchen wir links und rechts von v_{i+1} die nächstgelegenen b -Daten, dies sind für ein m die Daten b_{2m-1} und b_{2m+1} . Also sind mindestens $2m - 1$ b -Daten kleiner als v_{i+1} und $n - 2m$ größer. Nur von b_{2m} ist unbekannt, ob es kleiner oder größer als v_{i+1} ist. Falls b_{n-1} links von v_{i+1} steht, ist nur die Lage bezüglich b_n unbekannt. Falls b_1 rechts von v_{i+1} steht, ist v_{i+1} kleiner als alle b -Daten. Wir können m sogar berechnen. Links von v_{i+1} stehen die a -Daten a_1, \dots, a_{2j-3} , also $j - 1$ a -Daten. Daher stehen links von v_{i+1} noch $i - j + 1$ b -Daten, also $b_1, \dots, b_{2i-2j+1}$, d. h. $m = i - j + 1$. Wir wissen von $2j - 2$ a -Daten und $2i - 2j + 1$ b -Daten, dass sie kleiner als v_{i+1} sind, dies sind $2i - 1$ Daten, also ist der Rang von v_{i+1} mindestens $2i$. Da nur noch 2 Rangplätze möglich sind, ist der Rang höchstens $2i + 1$. Damit ist Batchmerge ein korrekter Mischalgorithmus. Es folgt die Beschreibung von Batchersort (BS).

Algorithmus 4.9.2: BS(a_1, \dots, a_n).

(1) Falls $n = 1$, STOP.

(2) Falls $n > 1$,

$$(b_1, \dots, b_{n/2}) := \text{BS}(a_1, \dots, a_{n/2})$$

$$(c_1, \dots, c_{n/2}) := \text{BS}(a_{n/2+1}, \dots, a_n).$$

(Diese beiden Aufrufe werden gleichzeitig bearbeitet.)

(3) $(d_1, \dots, d_n) := \text{BM}(b_1, \dots, b_{n/2}; c_1, \dots, c_{n/2})$.

Satz 4.9.3: Batchmerge mischt zwei Folgen der Länge $n = 2^k$ mit $n \log n + 1$ Vergleichen auf Parallelrechnern in Zeit $\log n + 1$. Batchersort sortiert eine Folge der Länge $n = 2^k$ mit $\frac{1}{4}n(\log n)(\log n - 1) + n - 1$ Vergleichen auf Parallelrechnern in Zeit $\frac{1}{2}(\log n)(\log n + 1)$.

Beweis: Sei $M(n)$ die Anzahl der Vergleiche und $PM(n)$ die Zeit auf Parallelrechnern, die BS für das Mischen zweier Folgen der Länge n braucht. Dann gilt $M(1) = PM(1) = 1$ und

$$\begin{aligned} M(n) &= 2M(n/2) + n - 1 \\ PM(n) &= PM(n/2) + 1. \end{aligned}$$

Hieraus folgt sofort die Behauptung für $PM(n)$. Die Behauptung für $M(n)$ folgt analog zur Analyse von Algorithmus 1.3.3.

Sei $S(n)$ die Anzahl der Vergleiche und $PS(n)$ die Zeit auf Parallelrechnern für das Sortieren einer Folge der Länge n . Dann gilt $S(1) = PS(1) = 0$ und

$$\begin{aligned} S(n) &= 2S(n/2) + M(n/2) = 2S(n/2) + \frac{n}{2} \log \frac{n}{2} + 1 \\ PS(n) &= PS(n/2) + PM(n/2) = PS(n/2) + \log n. \end{aligned}$$

Hieraus folgt leicht die Behauptung für $PS(n)$. Aus der Rekursionsgleichung für S folgt leicht, dass $S(n)$ durch $\frac{1}{2}n \log^2 n$ beschränkt ist. Das genaue Resultat kann mit einem Induktionsbeweis verifiziert werden. \square

Wir haben uns bisher keine Gedanken über die Architektur eines Parallelrechners gemacht, der Batchersort effizient bearbeiten kann. Die eigentlichen Vergleiche werden in Schritt (3) von Algorithmus 4.9.1 durchgeführt. Der Algorithmus enthält keine if-Abfragen. Daher kann er sogar hardwaremäßig implementiert werden.

Dazu benutzen wir die Darstellung als Sortiernetzwerk. Sortiernetzwerke der Eingabelänge n arbeiten auf n „Zeilen“. Prozessor P_i ist für die i -te Zeile verantwortlich. Zunächst beinhaltet er a_i . Eine vertikale Verbindung zwischen den Zeilen i und j ($i < j$) bedeutet, dass die Prozessoren P_i und P_j ihre Daten vergleichen, P_i das kleinere und P_j das größere der beiden Daten erhält. Am Ende muss auf Zeile i das Datum mit Rangplatz i stehen.

Abbildung 4.9.1 veranschaulicht Batchersort für $n = 16$.

Es fällt auf, dass eine Verbindung zwischen P_i und P_j nur nötig ist, wenn $|i - j|$ eine Zweierpotenz ist. Es sind b_1, \dots, b_8 und c_1, \dots, c_8 die sortierten Hälften. Zum Zeitpunkt T_1 sind die Folgen der Länge 4 sortiert, bis dahin sind 3 Zeittakte vergangen, da z. B. die nebeneinander gezeichneten Vergleiche (1, 3) und (2, 4) gleichzeitig erfolgen. Die Intervalle $[T_1, T_2]$ und $[T_2, T_3]$ laufen ebenso gleichzeitig in 2 Zeittakten ab, es folgt ein Zeittakt in $[T_3, T_4]$. Es folgen gleichzeitig in 2 Zeittakten die Intervalle $[T_4, T_5]$, $[T_5, T_6]$, $[T_7, T_8]$ und $[T_8, T_9]$. In einem Zeittakt werden die Intervalle $[T_6, T_7]$ und $[T_9, T_{10}]$ gleichzeitig bearbeitet. Das Intervall $[T_{10}, T_{11}]$ kostet einen Zeittakt. Die Rechenzeit bei Parallelverarbeitung beträgt also

$$10 = \frac{1}{2}(\log 16)(\log 16 + 1).$$

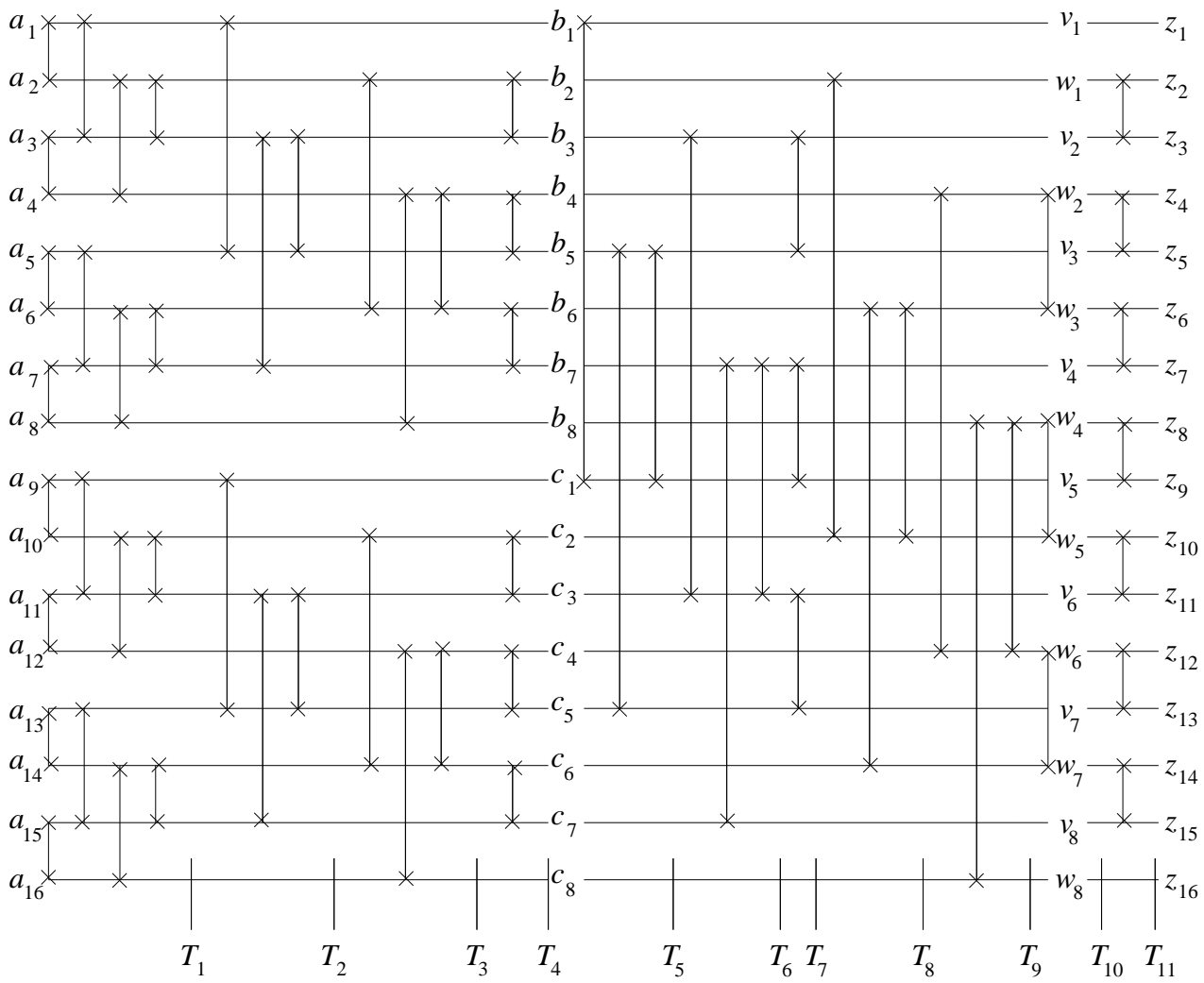


Abbildung 4.9.1: Batchersort für $n = 16$.

5 Entwurfsmethoden für Algorithmen

5.1 Vorbemerkungen

Natürlich gibt es keinen Algorithmus, um für ein gegebenes Problem einen effizienten Algorithmus zu entwerfen. Andererseits sind die zu lösenden Probleme nicht so verschieden, dass wir täglich eine geniale Idee produzieren müssen. Die guten Ideen sollten wir für wirklich neuartige Aufgaben sparen, die Alltagsprobleme sollten mit gängigen Methoden effizient gelöst werden. Dazu müssen wir jedoch die wichtigen Entwurfsmethoden für effiziente Algorithmen kennen und eingeübt haben. Entwurfsmethoden lassen sich nicht definieren oder auch nur exakt abgrenzen. Statt dessen beschreiben wir die Vorgehens- und Wirkungsweise der Methoden und wenden sie auf ausgewählte Beispielpunkte an. Je größer die Übung in der Anwendung, desto größer das Verständnis der Methode. Es soll noch darauf hingewiesen werden, dass die Abgrenzung zwischen den Methoden nicht immer eindeutig ist und manche Algorithmen Entwurfsmethoden vermischen.

5.2 Greedy Algorithmen

Diese Methode ist auf Optimierungsprobleme anwendbar, bei denen die möglichen Lösungen aus mehreren Teilen bestehen. Es wird eine Lösung stückweise konstruiert. Das nächste Lösungsteil wird dabei greedy (gierig) ausgewählt. Es wird das Teil gewählt, das den Wert der Teillösung am meisten erhöht, es wird aber nicht planvoll an die Zukunft gedacht. Es könnte ja sein, dass wir insgesamt am meisten erreichen, wenn wir nicht für den Augenblick optimieren. Ein Beispiel hierfür ist die Lebensplanung. Die Methode der greedy Algorithmen schreibt vor, heute keine Übungsaufgaben zu rechnen, sondern irgend etwas viel Schöneres zu tun. Das gilt auch morgen und übermorgen. Erfahrene Menschen behaupten, dass die greedy Lebensplanung nicht optimal ist.

Allgemein gibt es folgende Möglichkeiten bei der Anwendung von greedy Algorithmen:

- (1) Wir erhalten stets eine optimale Lösung.
- (2) Wir erhalten nicht immer eine optimale Lösung. Die errechnete Lösung weicht aber in ihrem Wert stets nur „wenig“ von einer optimalen Lösung ab.
- (3) Es kann vorkommen, dass wir sehr schlechte Lösungen erhalten.

Als erstes konkretes Beispiel diskutieren wir das Geldwechselproblem. Wenn wir im Supermarkt das Wechselgeld erhalten, möchten wir den Betrag sicherlich nicht in lauter 1-Cent-Münzen ausgezahlt bekommen. Im Normalfall möchten wir den Betrag mit möglichst wenigen Münzen (und Scheinen) erhalten. Die Kassiererin (in den meisten Fällen) oder der Kassierer hat also ein Optimierungsproblem zu lösen. Sie kennt die Wertigkeiten n_k, \dots, n_1 der Währung, wobei $n_k > n_{k-1} > \dots > n_1$ und $n_1 = 1$ ist, und den Betrag N , der zurückgezahlt werden muss. Ihre Aufgabe besteht darin, diesen Betrag durch möglichst wenige

Münzen zu realisieren. Formal sei a_i die gewählte Anzahl von n_i -Münzen. Dann soll

$$a_k + \cdots + a_1$$

unter den Nebenbedingungen

$$\begin{aligned} a_k n_k + \cdots + a_1 n_1 &= N \\ a_k, \dots, a_1 &\geq 0 \text{ und ganzzahlig} \end{aligned}$$

minimiert werden. Wir entwerfen einen greedy Algorithmus und betrachten die Münztypen gemäß ihrer Wichtigkeit, also ihrer Wertigkeit. Für jeden Münztyp werden möglichst viele Münzen gewählt. Der Restbetrag R wird mit $R := N$ initialisiert. Für $i = k, \dots, 1$ werden die a_i -Werte folgendermaßen gewählt:

$$a_i := \lfloor R/n_i \rfloor, R := R - a_i n_i.$$

Da $n_1=1$ ist, ist am Ende $R = 0$. Der Algorithmus liefert also eine zulässige Lösung, da die Nebenbedingungen erfüllt sind. Kassiererinnen beherrschen diesen greedy Algorithmus. Er braucht Zeit $O(k)$, aber wie gut ist die erzielte Lösung? Wir betrachten ein Münzsystem mit nur drei Münztypen, wobei $n_3 = 2n_2 + 1$ ist (z.B. $n_1 = 1, n_2 = 5, n_3 = 11$) und den Wechselbetrag $N = 3n_2$. Offensichtlich genügen drei Münzen vom Wert n_2 . Der greedy Algorithmus wählt eine Münze mit Wert n_3 . Der Restbetrag beträgt $R = 3n_2 - (2n_2 + 1) = n_2 - 1$ und dies führt dazu, dass noch $n_2 - 1$ Münzen vom Wert 1 gewählt werden. An Stelle von drei Münzen werden n_2 Münzen herausgegeben. Wir liegen also um einen beliebig großen Faktor, nämlich $n_2/3$, neben der optimalen Lösung. Dies widerspricht unserer Lebenserfahrung mit dem greedy Algorithmus. Sowohl für die alte DM-Währung wie auch für die neue Euro-Währung ist das Münzsystem so gewählt, dass der greedy Algorithmus stets optimale Lösungen liefert (Übungsaufgabe). Für das allgemeine Münzwechselproblem kann der greedy Algorithmus beliebig schlechte Lösungen liefern, für die Münzwechselprobleme der gängigen Währungen liefert der greedy Algorithmus jedoch sehr effizient optimale Lösungen.

Eines der wichtigsten Optimierungsprobleme ist das Traveling Salesman Problem (TSP). Gegeben sind n Orte und die Kosten $c(i, j)$, um von i nach j zu reisen. Gesucht ist die billigste Rundreise oder in der Sprache der Graphentheorie der billigste Hamiltonkreis. Hamiltonkreise berühren jeden Knoten genau einmal. Formal ist eine Permutation π auf $\{1, \dots, n\}$ gesucht, die

$$c(\pi(1), \pi(2)) + c(\pi(2), \pi(3)) + \cdots + c(\pi(n-1), \pi(n)) + c(\pi(n), \pi(1))$$

minimiert. Dabei kann o.B.d.A. $\pi(1) = 1$ gewählt werden. Es stehen also $(n-1)!$ Permutationen zur Wahl, so dass eine vollständige Suche exponentielle Kosten verursacht.

Wenn wir greedy eine Rundreise berechnen wollen, können wir folgendermaßen vorgehen. Wir starten an Knoten 1 und markieren die bereits erreichten Knoten. Vom zuletzt erreichten Knoten wählen wir eine billigste Kante zu einem noch nicht erreichten Knoten.

Wenn alle Knoten erreicht wurden, wählen wir die Kante zum Startknoten 1 zurück. Die Rechenzeit ist mit $O(n^2)$ linear in der Eingabelänge.

Sei $c(i, i+1) = 1$ für $1 \leq i \leq n-1$, $c(n, 1) = M$ für eine große Zahl M und $c(i, j) = 2$ sonst. Der greedy Algorithmus konstruiert die Rundreise $1, 2, \dots, n-1, n, 1$ mit Kosten $n + M - 1$, während optimale Lösungen nur Kosten $n + 3$ verursachen, z.B. die Rundreise $1, 2, \dots, n-2, n, n-1, 1$.

Als nächstes Problem betrachten wir das Rucksackproblem (Knapsack Problem KP). Gegeben sind ein Rucksack und n Objekte mit Gewichten $g_1, \dots, g_n \in \mathbb{N}$. Wir setzen uns eine Gewichtsschranke G und wollen den Rucksack optimal bepacken. Dazu schätzen wir Nutzenwerte v_1, \dots, v_n für die Objekte und können unsere Aufgabe formalisieren:

$$\begin{aligned} &\text{Maximiere } \lambda_1 v_1 + \dots + \lambda_n v_n \\ &\text{unter den Nebenbedingungen } \lambda_i \in \{0, 1\} \text{ und} \\ &\lambda_1 g_1 + \dots + \lambda_n g_n \leq G. \end{aligned}$$

Hierbei wird das i -te Objekt genau dann eingepackt, wenn $\lambda_i = 1$ ist.

Wenn wir gierig sind, welches Objekt packen wir dann zuerst ein? Es wäre sehr naiv, das Objekt mit dem größten Nutzen zu wählen. Das Gewicht sollte doch ebenfalls eine Rolle spielen. Es kommt auf den Nutzen pro Gewichtseinheit an, also auf die so genannte Effektivität $e_i := v_i/g_i$, $1 \leq i \leq n$. In einem Vorbereitungsschritt (Preprocessing) werden die Effektivitätswerte berechnet und absteigend sortiert. Nach Umnummerierung der Objekte können wir annehmen, dass $e_1 \geq \dots \geq e_n$ ist. Dann packen wir die Objekte der Reihe nach ein, wobei natürlich Objekte, mit denen die Gewichtsschranke überschritten würde, ausgelassen werden. Formal wird für $i = 1, \dots, n$ folgendes getan:

$$\begin{aligned} &\text{Falls } g_i \leq G, \quad \text{setze } \lambda_i := 1 \text{ und } G := G - g_i, \\ &\text{sonst} \quad \quad \quad \text{setze } \lambda_i := 0. \end{aligned}$$

Die Rechenzeit für das Preprocessing beträgt $O(n \log n)$ und für den eigentlichen greedy Algorithmus $O(n)$. Die berechnete Lösung ist offensichtlich zulässig. Diese kann aber beliebig schlecht sein, wie folgendes Beispiel zeigt: $n = 2$, $g_1 = 1$, $v_1 = 1$, $g_2 = G$, $v_2 = G - 1$. Optimal ist $\lambda_1 = 0$ und $\lambda_2 = 1$ mit Nutzen $G - 1$. Der greedy Algorithmus berechnet $\lambda_1 = 1$ und $\lambda_2 = 0$ mit Nutzen 1.

An dieser Stelle soll erwähnt werden, dass das TSP und das KP Probleme sind, für deren exakte Lösung keine Algorithmen mit polynomieller Rechenzeit bekannt sind. Es gibt sogar Ergebnisse, die zu der Hypothese führen, dass es solche Algorithmen auch gar nicht geben kann.

Im Hinblick auf Kapitel 5.5 wollen wir ein verallgemeinertes Rucksackproblem betrachten. Die Nebenbedingungen $\lambda_i \in \{0, 1\}$ werden durch $\lambda_i \in [0, 1]$ ersetzt. Da Nebenbedingungen abgeschwächt werden (relaxter gehandhabt werden), sprechen wir auch von einer Relaxation des Rucksackproblems. Daher ist klar, dass der Wert einer optimalen Lösung für das relaxierte Problem nicht kleiner als der Wert einer optimalen Lösung für das eigentliche Rucksackproblem ist. Interpretieren können wir den Wert λ_i folgendermaßen. Wir schneiden vom Objekt i einen λ_i -Anteil ab und packen diesen Anteil in den Rucksack.

Der Gewichtsbeitrag ist $\lambda_i g_i$ und es wird angenommen, dass der Nutzenbeitrag $\lambda_i v_i$ ist. Die Größe des Lösungsraumes hat wesentlich zugenommen, dennoch können wir dieses relaxierte Problem nach dem Preprocessing, also unter der Annahme, dass $e_1 \geq \dots \geq e_n$ ist, in linearer Zeit $O(n)$ optimal lösen:

Berechne das maximale i mit $g_1 + \dots + g_i \leq G$.

Setze $\lambda_1 := 1, \dots, \lambda_i := 1$.

Falls $i < n$, setze $\lambda_{i+1} := (G - g_1 - \dots - g_i)/g_{i+1}$.

Falls $i + 1 < n$, setze $\lambda_{i+2} := 0, \dots, \lambda_n := 0$.

Die Aussage über die Rechenzeit ist offensichtlich. Falls $i = n$ ist, passen alle Objekte in den Rucksack und es ist optimal, alle Objekte einzupacken. Der Algorithmus berechnet diese Lösung. Ansonsten wird die Gewichtsgrenze voll ausgenutzt. Nach dem Einpacken der ersten i Objekte ist noch Platz für ein Gewicht von $G^* := G - g_1 - \dots - g_i$. Mit $\lambda_{i+1} = G^*/g_{i+1}$ ist $\lambda_{i+1}g_{i+1} = G^*$ und der Anteil des $(i+1)$ -ten Objektes ist so groß gewählt, dass das Gesamtgewicht G beträgt. Es bleibt noch die Optimalität der berechneten Lösung zu beweisen.

Dazu betrachten wir ein vereinfachtes Rucksackproblem, bei dem alle Objekte das Gewicht 1 haben. Dann ist es offensichtlich optimal, die G Objekte mit dem größten Nutzen einzupacken. In dem relaxierten Problem erlauben wir die Zerlegung des i -ten Objektes in g_i Teile, die jeweils Gewicht 1 und Nutzen e_i haben. Der greedy Algorithmus benutzt implizit diese Zerlegung und packt dann G Objekte mit dem größten Nutzen ein. Auch mit der Freiheit, die Objekte weiter zu zerlegen, können wir den Nutzen der Rucksackbe-
packung nicht steigern.

Bisher haben wir greedy Algorithmen kennengelernt, die stets optimale Lösungen berechnen, und solche, die beliebig weit daneben liegen können. Wir betrachten jetzt ein Beispiel, bei dem greedy Algorithmen nicht immer optimale, aber niemals ganz schlechte Lösungen berechnen. Das Bin Packing Problem (BPP) gehört ebenfalls zu den Problemen, für die es vermutlich keine polynomiellen Algorithmen gibt. Es sind n Objekte mit den Gewichten $g_1, \dots, g_n \leq G$ in möglichst wenige Kisten zu packen, deren Tragfähigkeit durch G beschränkt ist.

Die first fit Strategie (FF) arbeitet gedanklich mit n zunächst leeren Kisten und packt das nächste Objekt in die Kiste mit der kleinsten Nummer, in die es passt. Am Ende werden natürlich die nicht benutzten Kisten ignoriert. FF lässt sich auf naive Weise in Zeit $O(n^2)$ implementieren. Die best fit Strategie (BF) strebt dagegen an, den Platz in den Kisten möglichst auszufüllen. Das i -te Objekt wird in die vollste Kiste gepackt, in die es noch hineinpasst. Diese Strategie lässt sich mit Hilfe von AVL-Bäumen in Zeit $O(n \log n)$ realisieren.

Im folgenden bezeichnen wir für eine Eingabe I mit $\text{OPT}(I)$ die minimal benötigte Anzahl von Kisten und mit $\text{FF}(I)$ und $\text{BF}(I)$ die Anzahl der von den beiden Strategien benutzten Kisten. Wir wollen unsere Strategien analysieren.

Beide Strategien haben die Eigenschaft, dass der Inhalt zweier benutzter Kisten ein Gesamtgewicht von mindestens $G + 1$ hat. Ansonsten wären die Objekte in eine gemeinsame

Kiste gepackt worden. Hieraus lassen sich die Aussagen

$$\text{FF}(I)/\text{OPT}(I) \leq 2$$

und

$$\text{BF}(I)/\text{OPT}(I) \leq 2$$

recht leicht ableiten. Es sei g das Gewicht der von FF oder BF am geringsten gepackten Kiste. Falls $g \geq G/2$ ist, sind alle Kisten mindestens halb voll und keine Strategie kommt mit weniger als der Hälfte der Kisten aus. Falls $g < G/2$ ist und nur eine Kiste benutzt wird, ist die Lösung optimal. Ansonsten beträgt die Gesamtbelastung der FF(I) Kisten (analog für BF(I))

$$\begin{aligned} g + (\text{FF}(I) - 1)(G - g) &= G + (\text{FF}(I) - 2)(G - g) \\ &\geq G + (\text{FF}(I) - 2) \cdot G/2 = \text{FF}(I) \cdot G/2 \end{aligned}$$

und wieder brauchen wir bei jeder Strategie mindestens $\text{FF}(I)/2$ Kisten, um das Gesamtgewicht zu verteilen. Also sind die von FF und BF berechneten Lösungen nicht beliebig schlecht. Mit einer viel aufwändigeren Analyse lassen sich $\text{FF}(I)/\text{OPT}(I)$ und $\text{BF}(I)/\text{OPT}(I)$ sogar durch 17/10 nach oben abschätzen. Andererseits gibt es Beispiele, für die

$$\text{FF}(I)/\text{OPT}(I) \geq 5/3$$

und

$$\text{BF}(I)/\text{OPT}(I) \geq 5/3$$

ist. Dies gilt nicht nur für „kleine Ausreißer“, wo 5 statt 3 Kisten benutzt werden, sondern auch für Eingaben mit beliebig großem Wert für $\text{OPT}(I)$.

Es sei m vorgegeben. Wir betrachten $n = 18m$ Objekte, von denen die ersten $6m$ Objekte jeweils das Gewicht 19, die folgenden $6m$ Objekte das Gewicht 43 und die letzten $6m$ Objekte das Gewicht 64 haben. Das Gewichtslimit pro Kiste sei 126. Offensichtlich ist $\text{OPT}(I) = 6m$, da je ein Objekt jedes Typs eine Kiste voll ausfüllen. Die Strategie FF und BF packen je 6 der ersten $6m$ Objekte in eine Kiste. Es werden also m Kisten benutzt. Dann passen keine weiteren Objekte mehr in diese Kisten hinein. Danach werden je 2 der folgenden $6m$ Objekte in eine Kiste gepackt. In keine dieser Kisten passt noch ein weiteres Objekt. Schließlich wird für jedes der $6m$ folgenden Objekte eine neue Kiste benötigt. Also gilt $\text{FF}(I) = \text{BF}(I) = 10m$.

Dieses Beispiel zeigt deutlich, dass sich ein Preprocessing lohnt. Die Objekte werden dabei so umnummeriert, dass $g_1 \geq \dots \geq g_n$ gilt. Es ist sicher besser, die großen Objekte zuerst einzupacken, da sie potenziell die größten Probleme verursachen. Die resultierenden Strategien heißen FFD und BFD ($D \hat{=}$ decreasing) und die folgenden Aussagen lassen sich beweisen:

$$\begin{aligned} \forall I : \text{FFD}(I) &\leq \frac{11}{9}\text{OPT}(I) + 4, \\ \forall I : \text{BFD}(I) &\leq \frac{11}{9}\text{OPT}(I) + 4 \\ \forall m \exists I : \text{OPT}(I) &\geq m \text{ und } \text{FFD}(I) = \text{BFD}(I) = \frac{11}{9}\text{OPT}(I). \end{aligned}$$

Im worst case haben FF und BF dasselbe Verhalten, ebenso FFD und BFD. Typischerweise sind aber die BF-Varianten den FF-Varianten überlegen.

Als letztes Beispiel betrachten wir den Algorithmus von Kruskal zur Berechnung minimaler Spannbäume (eigentlich: aufspannender Bäume). Dieser greedy Algorithmus liefert stets optimale Lösungen, aber der Beweis, dass dies so ist, ist nicht trivial. Darüber hinaus lernen wir eine Anwendung der Datenstrukturen für Partitionen kennen.

Ein Spannbaum auf einem ungerichteten zusammenhängenden Graphen $G = (V, E)$ ist ein Baum mit Knotenmenge V und einer Kantenmenge $E' \subseteq E$. Wenn wir eine Kostenfunktion $c : E \rightarrow \mathbb{R}^+$ haben, sind die Kosten eines Spannbaukes gleich der Summe der Kosten seiner Kanten. Schließlich ist ein minimaler Spannbaum ein Spannbaum mit minimalen Kosten. Dieses Problem hat viele Anwendungen, da es ein zentrales Problem ist, auf kostengünstigste Weise Objekte wie Städte oder Rechner zu verbinden. Wie wir es schon gewohnt sind, sortieren wir zunächst die Kanten nach steigenden Kosten. Danach wählen wir Kanten, bis wir einen Spannbaum erhalten haben. Eine Kante darf nicht gewählt werden, wenn ihre Endpunkte durch die bereits gewählten Kanten durch einen Weg verbunden sind. Die Kante würde ja einen Kreis produzieren und Kreise sind in Bäumen verboten. Da der zugrundeliegende Graph zusammenhängend ist, erzeugt dieser Algorithmus stets einen Spannbaum. Beim Beweis, dass dieser Spannbaum minimal ist, müssen wir vorsichtig sein. Schließlich hat ein ähnlicher Ansatz beim TSP nicht zu optimalen Rundreisen geführt. Kernstück des Korrektheitsbeweises ist das folgende Lemma.

Lemma 5.2.1: Sei V_1, \dots, V_k eine disjunkte Zerlegung der Knotenmenge V und sei T_i ein Spannbaum auf V_i , $1 \leq i \leq k$. Sei $e = (v, w)$ eine Kante zwischen zwei Mengen V_i und V_j mit $i \neq j$, die unter allen Kanten zwischen verschiedenen Mengen minimale Kosten hat. Unter allen Spannbäumen auf V , die die Kanten aus T_1, \dots, T_k enthalten, gibt es einen billigsten Spannbaum, der neben T_1, \dots, T_k auch e enthält.

Beweis: Sei T ein Spannbaum, der T_1, \dots, T_k , aber nicht e enthält. Wir beweisen das Lemma, indem wir einen Spannbaum T' konstruieren, der T_1, \dots, T_k und e enthält und nicht teurer als T ist.

Es entstehe T'' aus T durch Hinzufügung von e . Damit entsteht in T'' ein Kreis, auf dem e liegt. Da e die Komponenten V_i und V_j verbindet, muss auf dem Kreis eine weitere Kante $e' \neq e$ liegen, die zwei verschiedene Komponenten V_l und V_r , $l \neq r$, verbindet. Nach Voraussetzung ist $c(e') \geq c(e)$. Es entstehe T' aus T'' durch Streichung von e' . Dann gilt

$$c(T') = c(T) + c(e) - c(e') \leq c(T).$$

T' ist kreisfrei, da der einzige Kreis in T'' aufgebrochen wurde. T' spannt V auf, da die Kante e' durch einen Weg ersetzt wurde, der den Rest des Kreises enthält. \square

Mit diesem Lemma folgt die Korrektheit des Algorithmus von Kruskal. Zu jedem Zeitpunkt wird eine Kante hinzugefügt, die nach Lemma 5.2.1 für einen minimalen Spannbaum gewählt werden darf.

Wie aber implementieren wir den Test, ob eine Kante gewählt werden darf? Wir verwalten die Zusammenhangskomponenten des Graphen, der die bereits gewählten Kanten

enthält. Daher starten wir mit n einelementigen Mengen. Wenn wir die Kante $e = (v, w)$ wählen wollen, müssen wir testen, ob v und w in verschiedenen Mengen, die ja die Zusammenhangskomponenten darstellen, liegen. Dies ist der Test, ob $\text{FIND}(v) \neq \text{FIND}(w)$ ist. Wenn dieser Test positiv ausgeht, müssen die Mengen $\text{FIND}(v)$ und $\text{FIND}(w)$ vereinigt werden (eine UNION-Operation). Der Algorithmus kann abbrechen, wenn $n - 1$ Kanten ausgewählt worden sind. Daher ist es günstiger, die Kanten bezüglich ihrer Kosten in einem Min-Heap zu verwalten. Wenn von den m Kanten nur m^* betrachtet werden, genügen $O(m + m^* \log m)$ Operationen, um die m^* billigsten Kanten zu berechnen. Darüber hinaus werden dann $n - 1$ UNION-Operationen und $2m^*$ FIND-Operationen durchgeführt.

5.3 Dynamische Programmierung

Wir beschreiben die Methode der dynamischen Programmierung erst nach der Diskussion eines Beispielsproblems. Wir wollen binäre Suchbäume als statische Datenstruktur verwenden, in der n Daten unter den Schlüsseln $S_1 < \dots < S_n$ abgespeichert werden sollen. Im Gegensatz zu Kapitel 3 sind die Zugriffswahrscheinlichkeiten bekannt:

- mit Wahrscheinlichkeit $p_i > 0, 1 \leq i \leq n$, wird auf das i -te Datum zugegriffen und
- mit Wahrscheinlichkeit $q_j > 0, 0 \leq j \leq n$, wird nach einem Schlüssel S mit $S_j < S < S_{j+1}$ ($S < S_1$, falls $j = 0$, $S_n < S$, falls $j = n$) gesucht.

Das Ziel besteht darin, die erwartete (oder durchschnittliche) Suchzeit zu minimieren. Wenn der Suchpfad zu S_i genau l_i Knoten enthält (die Tiefe ist dann $l_i - 1$), beträgt die Suchdauer für S_i genau l_i (siehe auch Abbildung 5.3.1). Wenn der Suchpfad zum nil-Zeiger, der den Bereich (S_j, S_{j+1}) ($(-\infty, S_1)$, falls $j = 0$, und (S_n, ∞) , falls $j = n$) charakterisiert, genau m_j innere Knoten enthält, beträgt die Suchzeit für diesen Bereich m_j . Die erwartete Suchzeit des zugehörigen binären Suchbaumes T ist dann definiert durch

$$E(T) := \sum_{1 \leq i \leq n} p_i l_i + \sum_{0 \leq j \leq n} q_j m_j.$$

Um einen optimalen binären Suchbaum zu berechnen, ist folgende Erkenntnis wichtig. Jeder Teilbaum ist wieder ein binärer Suchbaum. Wie können wir die erwartete Suchzeit eines binären Suchbaumes T , dessen Wurzel S_k enthält und dessen Teilbäume T_1 und T_2 sind, über die erwarteten Suchzeiten von T_1 und T_2 ausdrücken? Die Wahrscheinlichkeit, dass wir den Baum T_1 erreichen, beträgt

$$P(T_1) := \sum_{1 \leq i \leq k-1} p_i + \sum_{0 \leq j \leq k-1} q_j,$$

analog ist

$$P(T_2) := \sum_{k+1 \leq i \leq n} p_i + \sum_{k \leq j \leq n} q_j.$$

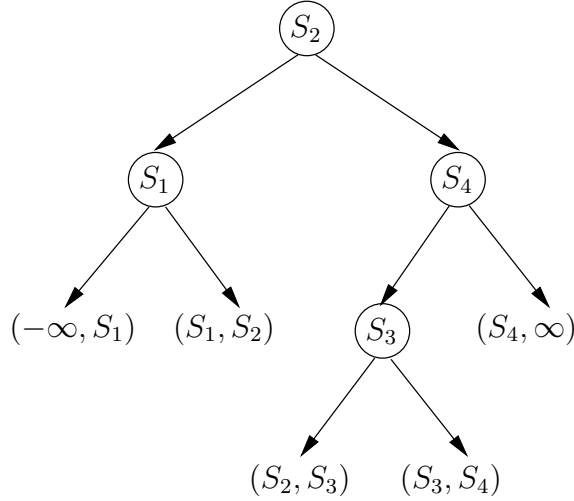


Abbildung 5.3.1: Ein binärer Suchbaum T für $n = 4$ mit $l_1 = 2, l_2 = 1, l_3 = 3, l_4 = 2, m_0 = 2, m_1 = 2, m_2 = 3, m_3 = 3$ und $m_4 = 2$.

Die Wahrscheinlichkeit, nach S_i in T_1 zu suchen, ist eine bedingte Wahrscheinlichkeit und damit der Quotient aus der Wahrscheinlichkeit, S_i zu suchen, und der Wahrscheinlichkeit, in T_1 zu suchen. Also betragen die bedingten Zugriffswahrscheinlichkeiten in T_1

$$p'_i := p_i/p(T_1), 1 \leq i \leq k-1, \text{ und } q'_j := q_j/p(T_1), 0 \leq j \leq k-1,$$

und in T_2

$$p''_i := p_i/p(T_2), k+1 \leq i \leq n, \text{ und } q''_j := q_j/q(T_2), k \leq j \leq n.$$

Wir können nun die erwartete Suchzeit folgendermaßen aufteilen. Mit Sicherheit wird der gesuchte Schlüssel S mit dem Schlüssel an der Wurzel S_k verglichen (dies ergibt einen Summanden von 1), mit Wahrscheinlichkeit $p(T_1)$ wird T_1 erreicht, wo die erwartete Suchzeit $E(T_1)$ (bezüglich der Wahrscheinlichkeiten p'_i und q'_j) beträgt, und mit Wahrscheinlichkeit $p(T_2)$ wird T_2 erreicht, wo die erwartete Suchzeit $E(T_2)$ beträgt. Also gilt

$$E(T) = 1 + p(T_1) \cdot E(T_1) + p(T_2) \cdot E(T_2).$$

Für unsere Rechnung ist es unangenehm, die neuen Wahrscheinlichkeiten wie p'_i auszurechnen. Es fällt jedoch auf, dass der Nenner, $p(T_1)$ oder $p(T_2)$, wegfällt, wenn wir vorab mit den Vorfaktoren, nämlich auch $p(T_1)$ und $p(T_2)$, multiplizieren. Daher definieren wir die relativen Kosten $C(T)$ von Bäumen T , wobei wir die Suchzeiten in den Teilbäumen mit den a-priori Wahrscheinlichkeiten (an Stelle der bedingten Wahrscheinlichkeiten) gewichten. In Abbildung 5.3.1 sei T^* der Teilbaum, dessen Wurzel den Schlüssel S_4 enthält. Dann ist

$$C(T^*) = 2p_3 + p_4 + 2q_2 + 2q_3 + q_4.$$

Da für den Gesamtbaum $C(T) = E(T)$ ist, können wir unsere Aufgabe dahingehend ändern, $C(T)$ zu minimieren. Sei

$$p(i, j) := p_i + \cdots + p_j + q_{i-1} + \cdots + q_j, 1 \leq i \leq j \leq n.$$

Dann gilt für einen Suchbaum T auf den Schlüsseln S_i, \dots, S_j mit den Teilbäumen T_1 und T_2

$$C(T) = p(i, j) + C(T_1) + C(T_2).$$

Wir können jetzt einen rekursiven Algorithmus entwerfen, wobei wir mit $C(i, j)$ die minimalen Kosten eines binären Suchbaumes mit den Schlüsseln S_i, \dots, S_j bezeichnen.

Da $1 \leq i \leq j \leq n$ ist, haben wir es mit $\binom{n}{2} + n \leq n^2$ Problemen zu tun, wobei wir eigentlich nur an $C(1, n)$ interessiert sind. Offensichtlich ist

$$C(i, i) = p(i, i)$$

und wir setzen $C(i, i-1) = 0$ für Teilbäume, die nur einen nil-Zeiger repräsentieren. Für das Problem $(1, n)$ haben wir n Möglichkeiten, den Schlüssel an der Wurzel zu wählen. Wenn wir S_k wählen, müssen wir $C(1, k-1)$ und $C(k+1, n)$ berechnen, um die optimalen Kosten aller binären Suchbäume mit Wurzel S_k zu erhalten. Schließlich wählen wir unter diesen Suchbäumen denjenigen mit den geringsten Kosten. Was können wir zur Rechenzeit $R(n)$ für Probleme mit n Schlüsseln sagen? Wir unterschätzen die Kosten, wenn wir $R(0) = R(1) = 0$ setzen und für die Auswahl der besten Lösungen nach Lösung aller Teilprobleme Kosten $n-1$ veranschlagen. Dann ergibt sich

$$R(n) = n - 1 + \sum_{1 \leq i \leq n} (R(i-1) + R(n-i))$$

und insbesondere $R(2) = 1$ und für $n \geq 3$

$$R(n) \geq 2R(n-1)$$

und damit

$$R(n) \geq 2^{n-1}.$$

Selbst diese grobe Abschätzung zeigt, dass die Rechenzeit des rekursiven Algorithmus exponentiell ist. Rekursion bietet stets die Gefahr der wiederholten Bearbeitung derselben Teilprobleme. Wir haben ja gesehen, dass es weniger als n^2 Teilprobleme gibt. Andererseits zeigt unsere Analyse, dass der Algorithmus mindestens 2^{n-2} Teilprobleme löst, also manche Teilprobleme exponentiell oft.

Die Methode der dynamischen Programmierung vermeidet dies, indem sie die Probleme iterativ nach wachsender Größe und damit nur einmal löst. Der Rechenzeitgewinn geht auf Kosten des Speicherplatzes, da wir uns die Lösungen für die Teilprobleme merken. In unserem Fall legen wir uns eine Tabelle $T(i, j)$, $1 \leq i \leq j \leq n$, für alle Teilprobleme an und wollen dort die optimalen Kosten $C(i, j)$ und den Schlüssel $S(i, j)$ in der Wurzel eines optimalen Suchbaumes abspeichern. Die optimalen Kosten $C(1, n)$ finden wir dann in $T(1, n)$. Außerdem wissen wir, dass wir einen optimalen Suchbaum mit einer Wurzel beginnen können, die $S(1, n)$ enthält. Falls $S(1, n) = S_k$ ist, finden wir einen optimalen Schlüssel für den linken Teilbaum in $S(1, k-1)$ und einen optimalen Schlüssel für den rechten Teilbaum in $S(k+1, n)$. Auf diese Weise können wir, nachdem alle Teilprobleme gelöst sind, einen optimalen binären Suchbaum in $O(n)$ Schritten berechnen. Da für Suchbäume auf S_i, \dots, S_j

$$C(T) = p(i, j) + C(T_1) + C(T_2)$$

ist, müssen Teilbäume optimaler binärer Suchbäume ebenfalls optimal sein. Daher gilt allgemein

$$C(i, j) = p(i, j) + \min\{C(i, k-1) + C(k+1, j) | i \leq k \leq j\}.$$

Diese Gleichung, mit der der Wert einer optimalen Lösung eines Problems als einfache Funktion der Werte optimaler Lösungen von kleineren Problemen ausgedrückt wird, heißt bellmansche Optimalitätsgleichung. Sie zu entdecken, ist das Kernstück bei der Entwicklung von Algorithmen, die der Methode der dynamischen Programmierung folgen. In unserem Fall lösen wir die Probleme nach wachsendem $j - i$. Dann sind bei der Betrachtung des (i, j) -Problems die Werte $C(i, k-1)$ und $C(k+1, j)$, $i \leq k \leq j$, bereits bekannt. Also kann $C(i, j)$ in Zeit $O(n)$ berechnet werden und ein k^* , für das die rechte Seite der bellmanschen Optimalitätsgleichung minimal ist, als $S(i, j)$ abgespeichert werden. Da wir weniger als n^2 Teilprobleme betrachten, beträgt die Gesamtrechnenzeit $O(n^3)$.

Satz 5.3.1: Optimale statische binäre Suchbäume zu gegebenen Zugriffswahrscheinlichkeiten $p_1, \dots, p_n, q_0, \dots, q_n$ können in Zeit $O(n^3)$ berechnet werden.

Die Methode der dynamischen Programmierung ist für Probleme vom so genannten Intervalltyp gut geeignet. Dabei ist die Grundmenge ein Intervall wie hier $[0, n]$ und alle Teilprobleme beziehen sich auf Teilintervalle $[i, j]$. Schließlich kann ein Problem effizient gelöst werden, wenn die Probleme für die echten Teilintervalle bereits gelöst sind. Wir wollen die Methode der dynamischen Programmierung auch auf andere Problemtypen anwenden und beginnen mit dem Rucksackproblem. Wie können wir ein Rucksackproblem sinnvoll einschränken? Wir können nur die ersten k der n Objekte betrachten, aber was nützt uns eine optimale Lösung des so eingeschränkten Rucksackproblems für die Lösung des Problems mit $k+1$ Objekten? Der entsprechende Trick besteht darin, die Anzahl der Objekte und die Gewichtsschranke zu variieren.

Es sei $R(k, g)$ für $k \in \{1, \dots, n\}$ und $g \in \{0, \dots, G\}$ das eingeschränkte Rucksackproblem mit k Objekten, deren Gewichte g_1, \dots, g_k und deren Nutzenwerte v_1, \dots, v_k betragen und bei dem das Gewichtslimit g beträgt. Wir legen eine Tabelle $T(k, g)$, $1 \leq k \leq n$, $0 \leq g \leq G$, an, wobei wir an der Position $T(k, g)$ mit $F(k, g)$ den optimalen Nutzen im Problem $R(k, g)$ und mit $D(k, g)$ eine optimale Entscheidung über das k -te Objekt abspeichern wollen. Es soll $D(k, g) = 1$ sein, wenn es in $R(k, g)$ optimal ist, Objekt k einzupacken, und $D(k, g) = 0$ sonst. Dann ist schließlich $F(n, G)$ der Wert einer optimalen Rucksackbepackung. Eine optimale Bepackung selber erhalten wir folgendermaßen. Falls $D(n, G) = 0$ ist, packen wir Objekt n nicht ein. Wir haben es dann mit dem Problem $R(n-1, G)$ zu tun und finden die Entscheidung über Objekt $n-1$ in $D(n-1, G)$. Falls $D(n, G) = 1$ ist, packen wir Objekt n ein. Damit ist das für die ersten $n-1$ Objekte erlaubte Gewichtslimit nur noch $G - g_n$ und wir haben es mit dem Problem $R(n-1, G - g_n)$ zu tun. Nach Berechnung der Tabelle können wir also eine optimale Rucksackbepackung in Zeit $O(n)$ berechnen. Auch für die Berechnung der Tabelle haben wir bereits die entscheidenden Ideen diskutiert. Wir legen zunächst sinnvolle Randwerte fest. Diese Randwerte dienen nur dazu, umständliche Fallunterscheidungen zu vermeiden. Es sei $F(k, g) = -\infty$, falls $g < 0$. In diesem Fall haben wir das Gewichtslimit überschritten und der Wert der

Bepackung ist $-\infty$. Außerdem ist $F(k, 0) = 0$ und $D(k, 0) = 0$, da wir bei Gewichts-limit 0 nichts einpacken dürfen. Schließlich ist für $g \geq 0$ auch $F(0, g) = 0$, da wir gar kein Objekt zum Einpacken haben. Wir kommen nun zum Normalfall mit $1 \leq k \leq n$ und $1 \leq g \leq G$. Das Objekt können wir einpacken oder nicht einpacken. In jedem Fall erhalten wir ein eingeschränktes Rucksackproblem, das wir optimal lösen sollten. Die eingeschränkten Probleme sind $R(k-1, g-g_k)$, wobei wir uns den Nutzen v_k des k -ten Objektes bereits gesichert haben, und $R(k-1, g)$. Also ist

$$F(k, g) = \max\{F(k-1, g-g_k) + v_k, F(k-1, g)\}.$$

Außerdem ist $D(k, g) = 1$, falls $F(k-1, g-g_k) + v_k \geq F(k-1, g)$ ist, und $D(k, g) = 0$ sonst. Wir können die Tabelle also zeilenweise füllen, wobei jeder neue Tabelleneintrag in Zeit $O(1)$ berechnet werden kann.

Satz 5.3.2: Das Rucksackproblem kann in Zeit $O(nG)$ gelöst werden.

Dies ist eine ungewöhnliche Rechenzeit. Ist sie polynomiell? Nein, denn die Rechenzeit muss auf die Länge (genauer die Bitlänge) der Eingabe bezogen werden. Wenn alle Zahlen in der Eingabe nicht größer als 2^n sind und $G = 2^n$ ist, dann ist die Länge der Eingabe $\Theta(n^2)$ und die Rechenzeitschranke von der Größenordnung $n2^n$ und damit exponentiell in der Eingabelänge. Wenn aber alle Zahlen in der Eingabe nicht größer als n^2 sind, liegt die Eingabelänge im Bereich $\Omega(n)$ und $O(n \log n)$ und die Rechenzeit $O(n^3)$ ist polynomiell. Für polynomiell in n kleine Gewichtswerte erhalten wir also einen polynomiellen Algorithmus.

Rechenzeiten, die exponentiell sein können, aber bei polynomiell kleinen Zahlen in der Eingabe polynomiell sind, heißen pseudopolynomiell.

Schließlich behandeln wir noch das klassische all-pairs-shortest-paths-(APSP-) Problem. Wir haben n Orte (mit $1, \dots, n$ bezeichnet) und eine Kostenmatrix $C = (c(i, j))$, wobei $c(i, j) \geq 0$ die Kosten der direkten Verbindung von i nach j angibt. Dabei ist $c(i, i) = 0$ und Werte $c(i, j) = \infty$ zeigen an, dass eine direkte Verbindung nicht existiert. Wir möchten nun die Distanzen $d(i, j)$ eines kostengünstigsten Weges von i nach j berechnen. Außerdem soll $N(i, j)$ den zweiten Knoten (Nachfolger) auf einem kostengünstigsten Weg von i nach j angeben. Auf diese Weise müssen nicht n^2 Pfade abgespeichert werden, die $\Theta(n^3)$ Knoten enthalten können, sondern nur $\Theta(n^2)$ Knotennummern. Hier bezeichnen wir mit $d_k(i, j)$ die Kosten eines optimalen Weges von i nach j , wenn alle Zwischenorte (außer dem Startpunkt i und dem Zielpunkt j) aus der Menge $\{1, \dots, k\}$ stammen müssen, und mit $N_k(i, j)$ den zugehörigen Nachfolger. Für $k = 0$ sind alle Zwischenorte verboten. Also ist

$$d_0(i, j) = c(i, j)$$

und

$$N_0(i, j) = j.$$

Da alle c -Werte nichtnegativ sind, können Kreise die Kosten nicht senken und wir können uns auf kreisfreie Wege beschränken. Wir nehmen nun an, dass alle Werte $d_k(i, j)$ und $N_k(i, j)$ bereits bekannt sind. Um $d_{k+1}(i, j)$ und $N_{k+1}(i, j)$ zu berechnen, müssen wir nur

entscheiden, ob der Ort $k+1$ auf einem kostengünstigsten Weg liegt oder nicht. Im ersten Fall zerfällt der Weg in den Weg von i nach $k+1$ und den Weg von $k+1$ nach j . Auf beiden Teilwegen kommt der Ort $k+1$ als Zwischenort nicht vor und wir können auf die bereits berechneten Werte zurückgreifen. Es folgt die bellmansche Optimalitätsgleichung

$$d_{k+1}(i, j) = \min\{d_k(i, j), d_k(i, k+1) + d_k(k+1, j)\}.$$

Falls $d_k(i, j) \leq d_k(i, k+1) + d_k(k+1, j)$ ist, können wir

$$N_{k+1}(i, j) = N_k(i, j)$$

setzen und ansonsten

$$N_{k+1}(i, j) = N_k(i, k+1).$$

Wir behandeln n^3 Tabelleneinträge $(d_k(i, j), N_k(i, j))$, $1 \leq i, j, k \leq n$, und jeder Eintrag lässt sich in Zeit $O(1)$ aus den Einträgen mit kleineren k -Werten berechnen.

Satz 5.3.3: Das APSP-Problem kann in Zeit $O(n^3)$ gelöst werden.

Ob wir die Methode der dynamischen Programmierung anwenden können, hängt also davon ab, ob wir das Problem so einschränken können, dass wir die Lösung eines Teilproblems effizient aus den Lösungen „kleinerer“ Probleme erhalten (bellmansche Optimalitätsgleichung).

5.4 Der Algorithmus von Dijkstra

Wie beim APSP-Problem ist eine Kostenmatrix $C = (c(i, j))$ mit $c(i, i) = 0$ vorgegeben. Beim single-source-shortest-paths-(SSSP)-Problem sind wir nur an den kürzesten Wegen von einem Ort s (dem Startort oder source) zu allen anderen Orten i interessiert. Die zu berechnende Information soll aus den Werten $d(i)$, der Kosten optimaler Wege von s nach i , und $V(i)$, den vorletzten Ort auf einem optimalen Weg von s nach i , bestehen. Wir können dann wieder optimale Wege von s nach i effizient rekonstruieren. Natürlich ist eine Lösung in Zeit $O(n^3)$ mit dem Algorithmus zur Lösung des APSP-Problems möglich. Es ist nicht ersichtlich, wie dieser Algorithmus wesentlich beschleunigt werden kann, wenn nur das SSSP-Problem gelöst werden soll. Der Algorithmus von Dijkstra wählt einen anderen Ansatz und ist dadurch für das SSSP-Problem wesentlich effizienter. Wir diskutieren diesen Algorithmus hier, da das behandelte Problem von grundlegender Bedeutung ist. Der Algorithmus von Dijkstra folgt allerdings keiner einzelnen Methode zum Entwurf effizienter Algorithmen. Wir werden aber feststellen, dass er Ideen von greedy Algorithmen ebenso adaptiert wie Ideen der dynamischen Programmierung.

Wir werden die $d(i)$ - und $V(i)$ -Werte in einer Reihenfolge i_1, \dots, i_n der Orte berechnen, so dass $d(i_1) \leq \dots \leq d(i_n)$ ist. Dies ist in gewisser Weise ein greedy Ansatz (erst den nächsten Ort, dann den zweitnächsten Ort, ...). Andererseits werden wie bei der dynamischen Programmierung eingeschränkte Probleme behandelt. Nach der Lösung des k -ten eingeschränkten Problems kennen wir die Menge $A(k)$ der k Orte mit der kleinsten Distanz von s (zur Vereinfachung verwenden wir bestimmte Artikel, obwohl mehrere

Orte dieselbe Distanz von s haben können) und für jeden Ort $i \notin A(k)$ kennen wir die Länge $d_k(i)$ eines kostengünstigsten Weges von s nach i , wobei der Weg nur über Orte aus $A(k)$ führen darf. Dabei sei $N_k(i)$ der vorletzte Knoten auf so einem Weg. Das Besondere ist, dass sich das Aussehen der eingeschränkten Probleme erst im Laufe des Algorithmus ergibt. Die Menge $A(k)$ ist vorab nicht bekannt.

Die Initialisierung für $k = 1$ ist einfach. Der Ort s liegt zu sich selbst am nächsten. Also setzen wir

- $A(1) := \{s\}, d(s) = 0, N(s) = \text{nil}$ (der Weg hat Länge 0),
- für $i \notin A(1) : d_1(i) = c(s, i), N_1(i) = s$.

Wir nehmen nun an, dass $A(k)$, $d(i)$ und $N(i)$ für $i \in A(k)$ und $d_k(i)$ und $N_k(i)$ für $i \notin A(k)$ bekannt sind. Welchen Ort fügen wir zu $A(k)$ hinzu, um $A(k+1)$ zu erhalten? Es sei j der Ort mit minimalen d_k -Wert, also insbesondere $j \notin A(k)$. Jeder Weg von s zu j startet in s und damit in $A(k)$ und endet in j und damit außerhalb von $A(k)$. Sei $(v_0 = s, v_1, \dots, v_l = j)$ ein kostengünstigster Weg von s zu j und sei (v_i, v_{i+1}) die erste Kante auf diesem Weg mit $v_i \in A(k)$ und $v_{i+1} \notin A(k)$. Dann betragen nach unserer Konstruktion die Kosten des Teilweges von v_0 zu v_{i+1} bereits mindestens $d_k(v_{i+1})$ und nach Wahl von j ist $d_k(v_{i+1}) \geq d_k(j)$. Damit gibt es keinen kostengünstigeren Weg von s nach j als den mit Kosten $d_k(j)$, der innerhalb von $A(k)$ zu $N_k(j)$ verläuft und dann direkt zu j . Analog können wir für alle anderen $j' \notin A(k)$ zeigen, dass $d(j') \geq \min\{d_k(m) | m \notin A(k)\} \geq d_k(j)$ ist. Damit können wir $A(k)$ um j zu $A(k+1)$ ergänzen:

- $A(k+1) = A(k) \cup \{j\}, d(j) = d_k(j), N(j) = N_k(j)$.

Nun müssen wir für $i \notin A(k+1)$ die Werte $d_{k+1}(i)$ und $N_{k+1}(i)$ berechnen. Wir haben j als neuen zulässigen Zwischenort zur Verfügung. Hier kehren wir zu den Ideen der dynamischen Programmierung zurück. Wir haben die Möglichkeit, den Zwischenort j zu ignorieren (in diesem Fall gehören zu den kostengünstigsten Wegen die Informationen $d_k(i)$ und $N_k(i)$) oder den Zwischenort j zu wählen. Dann ist es sicher optimal, einen kostengünstigsten Weg von s nach j zu wählen und danach nicht wieder zu einem Ort j' in $A(k)$ zurückzukehren (da $d(j') \leq d(j)$, hätten wir dann j als Zwischenort nicht gebraucht). Also endet der Weg mit der Kante (j, i) und die Kosten betragen $d(j) + c(j, i)$. Wir erhalten die bellmansche Optimalitätsgleichung

$$d_{k+1}(i) = \min\{d_k(i), d(j) + c(j, i)\}.$$

Falls $d_k(i) \leq d(j) + c(j, i)$ ist, setzen wir

$$N_{k+1}(i) = N_k(i)$$

und ansonsten

$$N_{k+1}(i) = j.$$

Schließlich ist $A(n) = \{1, \dots, n\}$ und wir haben die gewünschten Informationen beisammen. Es ist leicht zu sehen, dass beim Übergang von $A(k)$ zu $A(k+1)$ die Zeit $O(n)$ ausreicht. Folgendes ist durchzuführen:

- Berechnung eines Ortes j mit minimalem d_k -Wert.
- Berechnung von $A(k+1)$, $d(j)$ und $N(j)$ (hierfür reicht Zeit $O(1)$, da $A(k)$ nicht mehr gebraucht wird).
- Berechnung von $d_{k+1}(i)$ und $N_{k+1}(i)$, $i \notin A(k+1)$.

Satz 5.4.1: Der Algorithmus von Dijkstra löst das SSSP-Problem in Zeit $O(n^2)$.

Diese Rechenzeit ist linear in der Eingabelänge. In vielen Fällen gibt es nur wenige direkte Verbindungen und wir haben einen gerichteten Graphen durch Adjazenzlisten gegeben, wobei für jede Kante auch die Kosten gegeben sind. Dann lässt sich die Effizienz des Algorithmus von Dijkstra steigern. Zunächst bemerken wir, dass wir die d_k - und N_k -Werte nicht mehr brauchen, wenn wir die d_{k+1} - und N_{k+1} -Werte berechnet haben. Also können wir die alten Werte überschreiben. Wenn $A(k+1) = A(k) \cup \{j\}$ ist, gilt aber $d_{k+1}(i) = d_k(i)$ und $N_{k+1}(i) = N_k(i)$ für alle Orte i ohne Kante (j, i) . Also müssen wir nur die Adjazenzliste von j durchlaufen und die Parameter für die dort vermerkten Orte aktualisieren. Bei insgesamt m Kanten lässt sich damit die Berechnung aller d_k - und N_k -Werte in Zeit $O(m)$ durchführen. Die Aktualisierungen der A -Mengen und die Berechnung der d - und N -Werte kostet insgesamt Zeit $O(n)$. Dabei können wir eine doppelt verkettete Liste aller Orte außerhalb von $A(k)$ führen, wobei wir auf Ort j in der Liste direkt zugreifen können und daher Ort j in Zeit $O(1)$ aus der Liste entfernen können. Die Rechenzeit sinkt also auf $O(n+m)$ zuzüglich der Berechnung der Orte mit minimalem d_k -Wert. Dies kostet aber $n - k - 1$ Vergleiche, $1 \leq k \leq n - 1$, und damit bei dieser Vorgehensweise insgesamt Zeit $\Theta(n^2)$. Dies lässt sich durch den Einsatz geeigneter Datenstrukturen verbessern. Wir benötigen ein Array, um über den Ortsnamen i auf den aktuellen $d_k(i)$ -Wert zugreifen zu können. Die $d_k(i)$ -Werte werden in einem Min-Heap verwaltet, so dass der minimale Wert an der Wurzel steht und auch effizient entfernt werden kann. Wenn der $d_k(i)$ -Wert durch einen neuen $d_{k+1}(i)$ -Wert überschrieben wird, ist der neue Wert höchstens kleiner. Es ist nun einfach, diesen Wert im Heap aufsteigen zu lassen, bis die Heapeigenschaft an allen Stellen wieder hergestellt ist. Zur Initialisierung des Heaps genügen $O(n)$ Operationen und jede Aktualisierung eines Wertes ist in Zeit $O(\log n)$ möglich. Es kommt zu höchstens m Aktualisierungen. (Die Einzelheiten dieser Datenstrukturen werden in den Übungen behandelt.)

Satz 5.4.2: Der Algorithmus von Dijkstra kann das SSSP-Problem in Zeit $O(n + m \log n)$ lösen, wenn er auf gerichteten Graphen mit n Knoten und m Kanten, die durch Adjazenzlisten beschrieben sind, arbeitet.

Wenn wir nur an einem minimalen s - t -Weg interessiert sind, können wir den Algorithmus von Dijkstra abbrechen, wenn $d(t)$ und $N(t)$ berechnet sind.

5.5 Branch-and-Bound Algorithmen

Branch-and-Bound Algorithmen sind heuristische Optimierungsverfahren, die bei genügend langer Rechenzeit die Berechnung einer optimalen Lösung sichern, aber für die Rechenzeit keine gute Garantie liefern. Heuristisch bedeutet dabei, dass wir hoffen in vielen

Fällen mit einer kleinen Rechenzeit zum Ziel zu gelangen. Ein zusätzlicher Vorteil besteht darin, dass bei vorzeitigem Abbruch des Algorithmus für die bis dahin beste berechnete Lösung eine Schranke für den Abstand des Wertes dieser Lösung vom optimalen Wert geliefert wird. Hierbei unterscheiden wir Lösungen wie Touren beim TSP vom Wert der Lösung wie der Länge einer Tour.

Wir beschränken uns auf Optimierungsprobleme mit einer endlichen Anzahl möglicher Lösungen und nehmen an, dass der Wert einer Lösung effizient berechnet werden kann (wie z.B. beim TSP, KP oder BPP). Die folgende Beschreibung bezieht sich auf Maximierungsprobleme, da wir später als Beispiel das Rucksackproblem diskutieren wollen. Bei einer Übertragung auf Minimierungsprobleme müssen wir darauf achten, dass eine untere Schranke für den optimalen Wert eines Maximierungsproblems einer oberen Schranke für den optimalen Wert eines Minimierungsproblems „entspricht“.

Sei also nun ein Maximierungsproblem gegeben. Wir beschreiben die Module eines Branch-and-Bound Algorithmus und ihr Zusammenspiel.

Upper Bound Modul

Es soll effizient eine (möglichst gute) obere Schranke U für den Wert einer optimalen Lösung berechnet werden. Oft können dabei Relaxationen des gegebenen Problems betrachtet werden. Bei Optimierungsproblemen gibt es meistens Nebenbedingungen, die das Problem erst schwierig machen. Wenn wir diese Bedingungen abschwächen (das Problem „relaxter“ sehen), erhalten wir häufig ein effizient zu lösendes Optimierungsproblem. Da nun mehr Lösungen erlaubt sind, darunter alle Lösungen des gegebenen Problems, ist der Wert einer optimalen Lösung des relaxierten Problems (bei Maximierungsproblemen) eine obere Schranke für den Wert einer optimalen Lösung des eigentlichen Problems.

Lower Bound Modul

Es soll effizient eine (möglichst gute) untere Schranke L für den Wert einer optimalen Lösung berechnet werden. Greedy Algorithmen liefern zulässige Lösungen und der Wert jeder zulässigen Lösung ist (bei Maximierungsproblemen) eine untere Schranke für den Wert einer optimalen Lösung. Es hängt stark vom Problem ab, wie gut die durch greedy Algorithmen berechneten Lösungen sind.

In jedem Fall ist der Wert der berechneten Lösung maximal $U - L$ vom Wert einer optimalen Lösung entfernt. Wenn uns $U - L$ klein genug erscheint, können wir die Suche abbrechen und haben eine „genügend gute“ Lösung erhalten. Ist $U = L$, haben wir sogar eine optimale Lösung gefunden. Wir gehen nun davon aus, dass $U - L > 0$ ist und wir an einer garantiert optimalen Lösung interessiert sind. Natürlich kann die berechnete Lösung näher als $U - L$ am Optimum liegen und sogar optimal sein. Um eine optimale Lösung mit einer Garantie der Optimalität zu haben, müssen wir im Fall $U - L > 0$ noch arbeiten. Dazu wird das Problem zerlegt.

Branching Modul

Es werden Teilprobleme vom selben Typ erzeugt. Dabei ist ein Teilproblem P' von P ein Problem, bei dem die Menge zulässiger Lösungen eine echte Teilmenge der Menge zulässiger Lösungen von P ist und jede zulässige Lösung in P' denselben Wert wie in

P hat. Günstig ist es, wenn die Teilprobleme (in einem vagen Sinn) gleich „groß“ und gleich „schwer“ sind, so dass die Teilprobleme „kleiner“ und daher „leichter“ sind. Eine Zerlegung in möglichst wenige Teilprobleme ist vorzuziehen.

Alle unzerlegten Teilprobleme „überdecken“ das Gesamtproblem in dem Sinn, dass die Vereinigung der Menge zulässiger Lösungen die Menge zulässiger Lösungen im Gesamtproblem ergibt. Wenn wir also alle unzerlegten Teilprobleme lösen und die beste dieser Lösungen auswählen, erhalten wir eine optimale Lösung des Gesamtproblems. Dazu berechnen wir für die unzerlegten Probleme obere Schranken U_i und untere Schranken L_i . Da wir nun kleinere Probleme betrachten, sollte $U_i \leq U$ sein. Ansonsten können wir U_i durch U ersetzen. Es ist in jedem Fall U^* , das Maximum über alle U_i für unzerlegte Probleme, eine obere Schranke für das Gesamtproblem, da wir in keinem Teilproblem U^* übertreffen können. Damit wird U^* unsere neue obere Schranke U . Wir haben nun eine zulässige Lösung mit Wert L und für jedes Teilproblem eine Lösung mit Wert L_i . Daher können wir L durch das Maximum all dieser Werte ersetzen. Allgemein gilt

$$L_{alt} \leq L_{neu} \leq U_{neu} \leq U_{alt}$$

und die neue Maximaldifferenz zum Wert einer optimalen Lösung $U_{neu} - L_{neu}$ ist nicht größer als $U_{alt} - L_{alt}$. So fahren wir fort. Das Verfahren ist endlich, da irgendwann für jedes unzerlegte Teilproblem die Anzahl zulässiger Lösungen 1 wird und wir in diesem Fall den Wert dieser Lösung als untere und als obere Schranke für das Teilproblem angeben können. Teilprobleme mit $U_i = L_i$ werden nicht zerlegt, da sie gelöst sind.

Auf diese Weise erhalten wir eine Zerlegung des Problems, bis im schlechtesten Fall jedes unzerlegte Problem nur noch eine zulässige Lösung hat. In einem solchen Fall haben wir gegenüber einer erschöpfenden Suche (alle Lösungen ausprobieren) nichts gewonnen. Wieso hoffen wir, dass wir oft schnell sind? Nicht nur Teilprobleme mit $U_i = L_i$ müssen nicht zerlegt werden. Falls $U_i \leq L$ ist, kennen wir bereits eine zulässige Lösung, die von keiner zulässigen Lösung des i -ten Teilproblems übertroffen werden kann. Damit können wir das i -te Teilproblem als fertig bearbeitet einstufen. Das Ziel ist also, möglichst viele Teilprobleme möglichst früh als fertig bearbeitet einstufen zu können. Dies gelingt einerseits durch Zerlegungen, die die oberen Schranken möglichst stark senken und andererseits durch das Auffinden möglichst guter zulässiger Lösungen. In Abhängigkeit davon, welches der beiden Ziele vielversprechender ist, unterscheiden wir Suchstrategien.

Search Modul

Es wird unter den unzerlegten Teilproblemen eines ausgewählt, das im nächsten Schritt zerlegt werden soll (wobei dann für die neu entstehenden Teilprobleme obere und untere Schranken berechnet werden und danach die globalen Parameter U und L aktualisiert werden). Für die Wahl des zu zerlegenden Teilproblems gibt es grundsätzlich viele Strategien, die zwei wichtigsten sollen hier beschrieben werden. Wenn die unteren Schranken, also die Werte der berechneten zulässigen Lösungen, typischerweise gut sind, also recht nahe am Optimum liegen, dann ist es wichtig, die obere Schranke U zu senken. Vielleicht ist die beste berechnete zulässige Lösung bereits optimal und wir sind dabei, ihre Optimalität zu beweisen. In diesem Fall wählen wir ein unzerlegtes Problem mit $U_i = U$ zur

Zerlegung aus. Für dieses Problem gilt, dass dort optimistischerweise „am meisten zu holen“ ist. Andererseits werden wir ein solches Problem auf jeden Fall zerlegen müssen, um eine garantiert optimale Lösung zu erhalten (greedy Suche). Wenn allerdings die unteren Schranken typischerweise schlecht sind, sind wir erst einmal an der Berechnung einigermaßen guter Lösungen interessiert, um überhaupt Teilprobleme als fertig bearbeitet ansehen zu können. Bei der Tiefensuche verfolgen wir zunächst einen Zweig, bis das zugehörige Teilproblem gelöst ist. Von dem neu erzeugten Teilproblemen wählen wir jeweils eines, bei dem die obere Schranke am größten ist.

Learning Modul

Wir können unnötige Problemzerlegungen vermeiden, wenn wir aus den in einem Teilproblem bereits getroffenen Entscheidungen daraus folgende Entscheidungen ableiten. Da wir mit den erzwungenen Entscheidungen neues Wissen erwerben, wird hier von Lernen gesprochen.

Nun wollen wir die abstrakt beschriebenen Module für das Rucksackproblem mit Leben füllen.

Upper Bound Modul: In Kapitel 5.2 haben wir für die Lösung einer Relaxation des Rucksackproblems einen effizienten Algorithmus (Zeit $O(n)$ nach Sortierung der Effektivitätswerte) beschrieben. Der optimale Wert dieses relaxierten Problems kann als obere Schranke für das ursprüngliche Rucksackproblem verwendet werden.

Lower Bound Modul: In Kapitel 5.2 haben wir einen greedy Algorithmus für das Rucksackproblem vorgestellt. Er liefert eine zulässige Lösung, deren Wert eine untere Schranke für das Rucksackproblem liefert.

Branching Modul: Wir wählen ein Objekt i und zerlegen den Suchraum (oder Lösungsraum) in zwei disjunkte Bereiche. In einem Bereich ist $x_i = 0$ (Objekt i darf nicht gewählt werden und gehört zur Menge E der durch Exklusion verbotenen Objekte) und im anderen Bereich ist $x_i = 1$ (Objekt i muss gewählt werden und gehört zur Menge I der durch Inklusion erzwungenen Objekte). In beiden Teilproblemen streichen wir das Objekt i aus der Liste der zu betrachtenden Objekte. Bei der Inklusion muss das Gewichtslimit um g_i gesenkt werden, aber zum Nutzen jeder Lösung darf v_i addiert werden (vergleiche auch Kapitel 5.3). Die Wahl des Objektes i ist kritisch. Wir wollen die Lösung des relaxierten Problems in den Relaxationen beider Teilprobleme verbieten. Nur dies ermöglicht es, dass beide oberen Schranken der Teilprobleme kleiner als die obere Schranke des zu teilenden Problems sind, und nur so kann die aktuelle obere Schranke sinken. Dazu wählen wir das Objekt j mit $0 < \lambda_j < 1$ bei der Lösung des relaxierten Problems. Wenn es so ein Objekt nicht gibt, liefert die Lösung des relaxierten Problems eine zulässige Lösung und dieses Teilproblem wird nicht zerlegt. Indem wir das Objekt erzwingen oder verbieten, ist es nicht mehr möglich, das Objekt zu zerlegen.

Search Modul: Wir haben in Kapitel 5.2 gesehen, dass die greedy Lösung sehr schlecht sein kann, aber immerhin ist $U - L \leq v_{i+1}$, wenn Objekt $i+1$ bei der Lösung der relaxierten Problems zerlegt wird. Daher wählen wir die greedy Suchstrategie, die stets das unzerlegte Problem mit der größten oberen Schranke wählt.

Learning Modul: Hierauf kann beim Rucksackproblem verzichtet werden. Um aber einen

Eindruck zu geben, stellen wir eine sehr einfache Lernregel vor. Bei der Inklusion eines Objektes wird das Gewichtslimit gesenkt. Daraus lernen wir, dass alle Objekte, deren Gewicht bereits über dem neuen Gewichtslimit liegt, durch Exklusion verboten werden können.

Wir beenden diesen Abschnitt mit einem Beispiel, wobei wir die Teilprobleme durchnummerieren und P_1 das Ausgangsproblem ist. Allgemein ist

- I_k die Menge der in P_k durch Inklusion erzwungenen Objekte,
- E_k die Menge der in P_k durch Exklusion verbotenen Objekte,
- U_k die obere Schranke für P_k ,
- L_k die untere Schranke für P_k und
- M_k die Menge der vom greedy Algorithmus für P_k gewählten Objekte.

Wir verzichten auf den Einsatz des Lernmoduls.

Das Problem bezieht sich auf $n = 10$ Objekte und das Gewichtslimit 16. Wir beschreiben die Objekte durch Gewicht, Nutzen und Effektivität, wobei die Effektivitätswerte bereits absteigend sortiert sind.

Objekt	1	2	3	4	5	6	7	8	9	10
Gewicht	1	4	2	3	7	3	2	1	4	3
Nutzen	20	28	10	12	21	9	3	1	2	1
Effizienz	20	7	5	4	3	3	1,5	1	0,5	0,333...

$P_1 : I_1 = \emptyset, E_1 = \emptyset, U_1 = 88, L_1 = 83, M_1 = \{1, 2, 3, 4, 6, 7, 8\}.$

$P_2 : I_2 = \{5\}, E_2 = \emptyset, U_2 = 87, L_2 = 82, M_2 = \{1, 2, 3, 5, 7\}.$

$P_3 : I_3 = \emptyset, E_3 = \{5\}, U_3 = 83, L_3 = 83, M_3 = \{1, 2, 3, 4, 6, 7, 8\}.$

$U = 87, L = 83$, Zerlegung von P_2 .

$P_4 : I_4 = \{4, 5\}, E_4 = \emptyset, U_4 = 86, L_4 = 82, M_4 = \{1, 2, 4, 5, 8\}.$

$P_5 : I_5 = \{5\}, E_5 = \{4\}, U_5 = 85, L_5 = 82, M_5 = \{1, 2, 3, 5, 7\}.$

$U = 86, L = 83$, Zerlegung von P_4 .

$P_6 : I_6 = \{3, 4, 5\}, E_6 = \emptyset, U_6 = 84, L_6 = 72, M_6 = \{1, 3, 4, 5, 6\}.$

$P_7 : I_7 = \{4, 5\}, E_7 = \{3\}, U_7 = 84, L_7 = 82, M_7 = \{1, 2, 4, 5, 8\}.$

$U = 85, L = 83$, Zerlegung von P_5 .

$P_8 : I_8 = \{5, 6\}, E_8 = \{4\}, U_8 = 83, L_8 = 79, M_8 = \{1, 2, 5, 6, 8\}.$

$P_9 : I_9 = \{5\}, E_9 = \{4, 6\}, U_9 = 82, L_9 = 82, M_9 = \{1, 2, 3, 5, 7\}.$

P_8 und P_9 können schon gestrichen werden, da ihre obere Schranke die beste untere Schranke nicht übertrifft.

$U = 84, L = 83$, Zerlegung von P_6 .

$P_{10} : I_{10} = \{2, 3, 4, 5\}, E_{10} = \emptyset, U_{10} = 71, L_{10} = 71, M_{10} = \{2, 3, 4, 5\}.$

$P_{11} : I_{11} = \{3, 4, 5\}, E_{11} = \{2\}, U_{11} = 72, L_{11} = 72, M_{11} = \{1, 3, 4, 5, 6\}.$

$U = 84, L = 83$, Zerlegung von P_7 .

$P_{12} : I_{12} = \{4, 5, 6\}, E_{12} = \{3\}, U_{12} = 76, L_{12} = 65, M_{12} = \{1, 4, 5, 6, 7\}.$

$P_{13} : I_{13} = \{4, 5\}, E_{13} = \{3, 6\}, U_{13} = 82, L_{13} = 82, M_{13} = \{1, 2, 4, 5, 8\}.$

$U = 83, L = 83, \text{STOP.}$

Die Menge $M_3 = \{1, 2, 3, 4, 6, 7, 8\}$ ist optimal. Sie wurde sogar durch den greedy Algorithmus berechnet, die Arbeit war jedoch zum Nachweis der Optimalität nötig.

5.6 Eine allgemeine Analyse von divide-and-conquer Algorithmen

Wir haben schon einige divide-and-conquer Algorithmen kennengelernt. Dazu gehört die binäre Suche, bei der nur ein Teilproblem gelöst werden muss, aber auch ein Algorithmus für das Maxsummenproblem und Mergesort. Bei den beiden letztgenannten Problemen erhalten wir zwei Teilprobleme halber Größe und zusätzlich eine Rechenzeit von $\Theta(n)$. Dies führt zu einer Gesamt-Rechenzeit von $\Theta(n \log n)$. Batchersort ist ein divide-and-conquer Algorithmus, der für die Verbindung der Lösungen für die Teilprobleme mit Batchmerge einen weiteren divide-and-conquer Algorithmus aufruft. Wir wollen hier den Fall analysieren, dass wir Probleme der Größe n in a Probleme, die ungefähr Größe n/b haben, aufteilen und neben der Lösung der Teilprobleme eine Rechenzeit cn aufwenden. Es sei $R(n)$ die Gesamt-Rechenzeit und $R(1) = c$. Um uns nicht mit gerundeten Zahlen herum-schlagen zu müssen, betrachten wir nur Eingabegrößen $n = b^k$ für $k \in \mathbb{N}$, also $k = \log_b n$. Unsere Aufgabe besteht darin, die Rekursion

$$R(1) = c, R(n) = aR(n/b) + cn$$

für $n = b^k$ und $a, c > 0$ und $b > 1$ zu lösen.

Allgemein gilt

$$\begin{aligned} R(b^k) &= aR(b^{k-1}) + cb^k \\ &= a^2R(b^{k-2}) + acb^{k-1} + cb^k \\ &= a^3R(b^{k-3}) + a^2cb^{k-2} + acb^{k-1} + cb^k \\ &= a^kR(1) + a^{k-1}cb + a^{k-2}cb^2 + \dots + a^2cb^{k-2} + acb^{k-1} + cb^k. \end{aligned}$$

Diese Beziehung kann durch einen einfachen Induktionsbeweis verifiziert werden. Es folgt

$$R(b^k) = c \left(a^k + a^{k-1}b + a^{k-2}b^2 + \dots + a^2b^{k-2} + ab^{k-1} + b^k \right).$$

1. Fall: $a < b$.

$$\begin{aligned} R(b^k) &= cb^k \left(1 + \frac{a}{b} + \left(\frac{a}{b} \right)^2 + \dots + \left(\frac{a}{b} \right)^k \right) \\ &= cn \frac{1 - \left(\frac{a}{b} \right)^{k+1}}{1 - \frac{a}{b}} \leq cn \frac{1}{1 - \frac{a}{b}} \\ &= c \frac{b}{b-a} n = \Theta(n). \end{aligned}$$

2. Fall: $a = b$.

$$R(b^k) = c(k+1)b^k = cn(\log_b n + 1) = \Theta(n \log n).$$

3. Fall: $a > b$.

$$\begin{aligned} R(b^k) &= ca^k \left(1 + \frac{b}{a} + \left(\frac{b}{a}\right)^2 + \dots + \left(\frac{b}{a}\right)^k \right) \\ &= ca^k \frac{1 - \left(\frac{b}{a}\right)^{k+1}}{1 - \frac{b}{a}} \leq c \frac{a}{a-b} a^k. \end{aligned}$$

Es ist $a^k = a^{\log_b n} = n^{\log_b a}$.

Die letzte Gleichung lässt sich einfach verifizieren, indem auf beiden Seiten \log_b angewendet wird. Also ist

$$R(b^k) \leq c \frac{a}{a-b} n^{\log_b a} = \Theta(n^{\log_b a}).$$

Der Parameter c schlägt in allen Fällen als konstanter Faktor durch. Entscheidend ist das Verhältnis von a und b . Natürlich sollte a „relativ zu“ b möglichst klein sein. Wir können jetzt aber vorab entscheiden, ob es besser ist, 17 Teilprobleme der Größe $n/9$ zu bilden oder 24 Teilprobleme der Größe $n/12$. Was ist besser?

Nach diesen einfachen Rechnungen sollten wir auch keine Angst davor haben, geringfügig andere divide-and-conquer Rekursionsgleichungen zu lösen.

Im folgenden Abschnitt stellen wir einen divide-and-conquer Algorithmus vor, der das gegebene Problem der Größe N in 7 Teilprobleme der Größe $N/4$ zerlegt.

5.7 Matrixmultiplikation

Wir untersuchen die Multiplikation von zwei $n \times n$ -Matrizen X und Y . Sei $Z = XY$. Dann gilt nach Definition

$$z_{ij} = \sum_{1 \leq k \leq n} x_{ik} y_{kj}.$$

Zur Berechnung von z_{ij} genügen n Multiplikationen und $n - 1$ Additionen. Zwei Matrizen können also mit $2n^3 - n^2$ arithmetischen Operationen multipliziert werden. Wir wollen nun den divide-and-conquer Algorithmus von Strassen vorstellen. Die Multiplikation von 1×1 -Matrizen ist eine gewöhnliche Multiplikation. Für $n = 2^k$ werden die $n \times n$ -Matrizen X , Y und Z in jeweils vier $(n/2) \times (n/2)$ -Matrizen aufgeteilt.

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \quad Y = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \quad Z = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}$$

Für $n = 2$ ist nach Definition $Z_{12} = X_{11}Y_{12} + X_{12}Y_{22}$ und die Operationen sind arithmetische Operationen. Für $n > 2$ gilt diese Gleichung für Z_{12} weiterhin, nur beziehen sich die

Operationen nun auf $(n/2) \times (n/2)$ -Matrizen. Sei z.B. z_{ij} mit $1 \leq i \leq n/2 < j \leq n$ ein Element aus Z_{12} . Dann ist

$$z_{ij} = \sum_{1 \leq k \leq n/2} x_{ik}y_{kj} + \sum_{n/2 < k \leq n} x_{ik}y_{kj} = z_{ij}^1 + z_{ij}^2.$$

Der Summand z_{ij}^1 ist ein Element aus $X_{11}Y_{12}$ und z_{ij}^2 das entsprechende Element in $X_{12}Y_{22}$. Die Schulmethode zur Multiplikation von 2×2 -Matrizen führt 8 Multiplikationen und 4 Additionen aus. Additionen von $n \times n$ -Matrizen sind sehr billig. Sie kosten nur n^2 Operationen und sind daher (vermutlich) sehr viel billiger als Matrizenmultiplikationen. Es müsste sich also lohnen, in den Gleichungen für Z_{11} , Z_{12} , Z_{21} und Z_{22} die Zahl der (Matrizen-) Multiplikationen auch auf Kosten von zusätzlichen Additionen und Subtraktionen (von Matrizen) zu senken.

Satz 5.7.1: 2×2 -Matrizen lassen sich mit 12 Additionen, 6 Subtraktionen und 7 Multiplikationen multiplizieren.

Beweis: Der Beweis lässt sich leicht nachvollziehen, aber nur schwer motivieren. Mit 6 Additionen, 4 Subtraktionen und 7 Multiplikationen berechnen wir die folgenden Werte.

$$\begin{aligned} m_1 &= (x_{12} - x_{22}) \cdot (y_{21} + y_{22}), & m_2 &= (x_{11} + x_{22}) \cdot (y_{11} + y_{22}), \\ m_3 &= (x_{21} - x_{11}) \cdot (y_{11} + y_{12}), & m_4 &= (x_{11} + x_{12}) \cdot y_{22}, \\ m_5 &= x_{11} \cdot (y_{12} - y_{22}), & m_6 &= x_{22} \cdot (y_{21} - y_{11}), \\ m_7 &= (x_{21} + x_{22}) \cdot y_{11}. \end{aligned}$$

Daraus lässt sich nun das Ergebnis mit 6 Additionen und 2 Subtraktionen berechnen.

$$\begin{aligned} z_{11} &= m_1 + m_2 - m_4 + m_6, & z_{12} &= m_4 + m_5, \\ z_{21} &= m_6 + m_7, & z_{22} &= m_2 + m_3 + m_5 - m_7. \end{aligned}$$

□

Wir wenden diesen Satz auf $n \times n$ -Matrizen mit $n = 2^k$ an, die Multiplikationen der $(n/2) \times (n/2)$ -Matrizen werden rekursiv auf gleiche Weise ausgeführt. Mit $C(n^2)$ bezeichnen wir die Zahl arithmetischer Operationen für diesen Algorithmus zur Multiplikation von $n \times n$ -Matrizen. Dann gilt

$$C(1) = 1 \text{ und } C(n^2) = 7C((n/2)^2) + 18(n/2)^2.$$

Diese Rekursionsgleichung ist nicht von der gewünschten Form. Daher setzen wir $N = n^2$ und erhalten

$$C(1) = 1 \text{ und } C(N) = 7C(N/4) + 18N/4.$$

Nun hat $C(1)$ nicht die passende Größe. Wir betrachten daher folgende Rekursionsgleichung

$$D(1) = 9/2 \text{ und } D(N) = 7D(N/4) + (9/2)N.$$

Für $n = 2^k$, und somit $N = 4^k$, erhalten wir nach den Ergebnissen aus Kapitel 5.6

$$\begin{aligned} D(N) &= (9/2)7^k \left(1 - \left(\frac{4}{7} \right)^{k+1} \right) / \left(1 - \frac{4}{7} \right) \\ &= (9/2)7^k(7/3) - (9/2)4^{k+1}/3. \end{aligned}$$

Bei dieser Rekursion erhalten wir 7^k Probleme der Größe 1, für die wir jeweils Kosten $9/2$ statt 1 veranschlagen. Also ist

$$\begin{aligned} C(n^2) &= (9/2)7^k(7/3) - (9/2)4^{k+1}/3 - 7^k(7/2) \\ &= 7^{k+1} \left(\frac{3}{2} - \frac{1}{2} \right) - 6 \cdot 4^k \\ &= 7 \cdot 7^k - 6n^2 = 7n^{\log 7} - 6n^2. \end{aligned}$$

Die letzte Gleichung folgt, da $7^{\log n} = n^{\log 7}$ ist.

Satz 5.7.2: Mit dem Algorithmus von Strassen können zwei $n \times n$ -Matrizen für $n = 2^k$ mit $7n^{\log 7} - 6n^2$ arithmetischen Operationen berechnet werden.

Wann ist dies nun besser als die übliche so genannte Schulmethode?

$$\begin{aligned} 7n^{\log 7} - 6n^2 &< 2n^3 - n^2 &&\Leftrightarrow \\ 7n^{\log 7} &< 2n^3 + 5n^2 &&\Leftrightarrow \\ 7n^{\log 7-2} &< 2n + 5 \end{aligned}$$

Diese Ungleichung ist für $n = 512$ nicht erfüllt, wohl aber für $n = 1024$. Es wäre nun eine typische Verkürzung der Realität zu behaupten, dass die Strassen-Methode erst für $n > 512$ ($n \geq 1024$) sinnvoll anwendbar ist. Wenn wir bei der Strassen-Methode auf genügend kleine Matrizen stoßen, sollten wir natürlich auf die Schulmethode umsteigen. Sei nun SCHUL die Schulmethode und PUR die reine Strassen-Methode. Mit MIX bezeichnen wir folgende Methode. Auf der ersten Stufe benutzen wir die Strassen-Methode und in der Rekursion die bessere der Methoden SCHUL und MIX. Die Verwendung von MIX in der Rekursion an Stelle von PUR sichert, dass wir an der optimalen Stelle auf SCHUL umsteigen. Die folgende Tabelle zeigt, dass die Schulmethode durch MIX schon für $n = 16$ geschlagen wird.

n	2	4	8	16	32	64
SCHUL	12	112	960	7936	64512	520192
PUR	25	247	2017	15271	111505	798967
MIX	25	156	1072	7872	59712	436416

Tabelle 5.7.1: Die Anzahl von arithmetischen Operationen bei Algorithmen zur Matrixmultiplikation.

5.8 Die schnelle Fouriertransformation

Polynome bilden die vielleicht wichtigste Klasse von Funktionen. Die übliche Darstellung von Polynomen ist die so genannte Koeffizientendarstellung, bei der $(a_0, a_1, \dots, a_{n-1})$ das Polynom

$$f(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0$$

beschreibt. Nach dem Hauptsatz der Algebra sind Polynome höchstens $(n-1)$ -ten Grades durch die Angabe der Funktionswerte an n gegebenen Stellen eindeutig definiert. Mit $(z_1, b_1), \dots, (z_n, b_n)$, wobei $z_i \neq z_j$ für $i \neq j$ ist, können wir eindeutig das Polynom f höchstens $(n-1)$ -ten Grades darstellen, für das

$$f(z_i) = b_i, 1 \leq i \leq n,$$

ist. Beide Polynomdarstellungen haben ihre Vor- und Nachteile. In beiden Darstellungen lassen sich Polynome in Zeit $O(n)$ addieren (wobei allerdings die betrachteten Funktionsstellen z_1, \dots, z_n für beide Polynome dieselben sein müssen). Um ein Polynom zu differenzieren oder zu integrieren, ist die Koordinatendarstellung von Vorteil. Bei der Multiplikation zweier Polynome können sich die Grade addieren. Wenn wir je $2n-1$ Funktionsstellen von zwei Polynomen kennen, deren Grad durch $n-1$ beschränkt ist, erhalten wir $2n-1$ Funktionsstellen des Produktes durch $2n-1$ Multiplikationen. Ausgehend von den Koordinatendarstellungen (a_0, \dots, a_{n-1}) und (b_0, \dots, b_{n-1}) muss der so genannte Konvolutionsvektor (c_0, \dots, c_{2n-2}) berechnet werden, wobei

$$c_k = \sum_{\substack{0 \leq i, j \leq n-1 \\ i+j=k}} a_i b_j$$

ist. Die Berechnung aller c_k gemäß dieser Formel kostet $\Theta(n^2)$ arithmetische Operationen. Wenn wir effiziente Algorithmen hätten, um zwischen den beiden Darstellungen zu wechseln, könnten wir von der Koordinatendarstellung in die Funktionswertdarstellung wechseln, dort die Polynome in Zeit $O(n)$ multiplizieren und dann wieder in die Koordinatendarstellung wechseln. Wir untersuchen daher das Problem, für ein Polynom aus der Koordinatendarstellung (a_0, \dots, a_{n-1}) für frei zu wählende Stellen z_1, \dots, z_n die zugehörigen Funktionswerte zu berechnen. Da

$$f(z_i) = a_{n-1}z_i^{n-1} + \dots + a_1z_i + a_0$$

ist, kostet eine naive Realisierung $\Theta(n^2)$ Operationen. Die schnelle Fouriertransformation (FFT, fast Fourier transform) ermöglicht einen $O(n \log n)$ -Algorithmus. Die FFT ist aus vielen Anwendungen nicht mehr wegzudenken. In der Bildverarbeitung oder der Signaltheorie ermöglicht erst die FFT eine genügend effiziente Bearbeitung algorithmischer Fragestellungen. Die entscheidende Idee besteht in der geeigneten Auswahl der Stellen, an denen wir das Polynom auswerten wollen.

Definition 5.8.1: In einem kommutativen Ring R mit Einselement heißt $w \neq 1$ primitive n -te Einheitswurzel, wenn $w^n = 1$ ist und für alle $1 \leq k \leq n-1$ gilt

$$\sum_{0 \leq j \leq n-1} w^{jk} = 0.$$

Bevor wir nun zeigen, wie sich Polynome $(n-1)$ -ten Grades effizient an den Stellen w^0, \dots, w^{n-1} auswerten lassen, wenn w eine primitive n -te Einheitswurzel ist, diskutieren wir, ob es überhaupt derartige Zahlen gibt.

Bemerkung 5.8.2: In \mathbb{R} gibt es nur für $n = 2$ eine primitive Einheitswurzel, nämlich -1 .

Beweis: Aus $w \neq 1$ und $w^n = 1$ folgt in \mathbb{R} sofort $w = -1$ und n gerade. Für $n = 2$ ist die weitere Bedingung erfüllt, da $(-1)^0 + (-1)^1 = 0$ ist. Für $n > 2$ ist $k = 2$ zugelassen. Die Summe aller $(-1)^{2j}, 0 \leq j \leq n-1$, ist jedoch mit n von 0 verschieden. \square

Wieder dürfen wir nicht zu früh verzweifeln. Wir erweitern den Wertebereich der Polynome von \mathbb{R} auf \mathbb{C} .

Satz 5.8.3: Für $n > 1$ ist $e^{i2\pi/n}$ primitive n -te Einheitswurzel in \mathbb{C} . Dabei ist i die imaginäre Einheit.

Beweis: Nach Definition gilt

$$e^{ix} = \cos x + i \sin x.$$

Es ist also $e^{i2\pi/n} \neq 1$, da für $n > 2$ der Imaginärteil von 0 verschieden ist und für $n = 2$ der Wert -1 beträgt. Andererseits ist

$$(e^{i2\pi/n})^n = e^{i2\pi} = \cos(2\pi) + i \sin(2\pi) = 1.$$

Sei nun $1 \leq k \leq n-1$. Dann gilt (geometrische Reihe)

$$\begin{aligned} \sum_{0 \leq l \leq n-1} (e^{i2\pi/n})^{lk} &= \sum_{0 \leq l \leq n-1} (e^{i2\pi k/n})^l \\ &= \frac{1 - (e^{i2\pi k/n})^n}{1 - e^{i2\pi k/n}} = 0. \end{aligned}$$

Die letzte Gleichung gilt, da $e^{i2\pi k/n} = \cos(2\pi k/n) + i \sin(2\pi k/n)$ von 1 verschieden ist, während $e^{i2\pi k} = \cos(2\pi k) + i \sin(2\pi k) = 1$ ist. \square

Den formalen Beweis von Satz 5.8.3 können wir uns auch geometrisch veranschaulichen. Die Zahlen e^{ix} liegen auf dem Einheitskreis, denn

$$|e^{ix}| = |\cos x + i \sin x| = (\cos^2 x + \sin^2 x)^{1/2} = 1.$$

Insbesondere liegt $e^{i2\pi/n}$ auf dem Einheitskreis im Winkel $2\pi/n$. Bei der komplexen Multiplikation multiplizieren sich die Beträge. Alle Potenzen von $e^{i2\pi/n}$ liegen also auf dem Einheitskreis. Da $e^{ix}e^{iy} = e^{i(x+y)}$ (Additionstheoreme für sin und cos), addieren sich auch die Winkel. Für die n -te Potenz landen wir also auf dem Einheitskreis im Winkel $2\pi \stackrel{\Delta}{=} 0$, also auf der reellen Zahl 1. Die Zahlen $e^{i2\pi j/n}$ liegen also auf dem Einheitskreis in den Winkeln $2\pi j/n, 0 \leq j \leq n-1$. Wir können uns die Addition als Addition von Kräften vorstellen. Es wirken also auf den Nullpunkt n Kräfte gleicher Größe in die Richtungen $2\pi j/n, 0 \leq j \leq n-1$. Die Kräfte heben sich auf, die Summe ist also 0.

Wenn wir Polynome mit ganzzahligen Koeffizienten betrachten und uns Funktionswerte nur an ganzzahligen Stellen interessieren, ist die Verwendung komplexer Einheitswurzeln nicht ideal, da Rundungsfehler auftreten können, die bei Rechnungen in \mathbb{Z} vermieden werden können. Wir wählen m so groß, dass uns nur Zahlen interessieren, die dem Betrage

nach kleiner als $m/2$ sind. Dann können wir statt in \mathbb{Z} auch in \mathbb{Z}_m (also mod m) rechnen und erhalten dennoch exakte Resultate. In \mathbb{Z}_m für geeignete m gibt es nun wieder Einheitswurzeln.

Satz 5.8.4: Es seien n und w Zweierpotenzen und $m = w^{n/2} + 1$. Dann ist w primitive n -te Einheitswurzel in \mathbb{Z}_m .

Wir verzichten hier auf den Beweis von Satz 5.8.4 und stellen nur fest, dass es in den für uns wichtigen Situationen primitive n -te Einheitswurzeln gibt.

Definition 5.8.5: Sei R ein kommutativer Ring mit Einselement und primitiver n -ter Einheitswurzel w . Sei das Polynom f vom Grad $n-1$ durch seine Koeffizientendarstellung $a = (a_0, \dots, a_{n-1})$ gegeben. Die diskrete Fouriertransformierte $\text{DFT}_n(a)$ ist der Vektor $(f(w^0), \dots, f(w^{n-1}))$ aus Funktionswerten von f .

Da $f(w^j)$ die Summe aller $a_i w^{ij}$ mit $0 \leq i \leq n-1$ ist, lässt sich $\text{DFT}_n(a)$ als Matrix-Vektor-Produkt aus der $n \times n$ -Matrix W mit dem Eintrag w^{ij} an Position (i, j) , $0 \leq i, j \leq n-1$, und dem Vektor a beschreiben.

Da wir Polynome vom Grad d auch als Polynome vom Grad $d' > d$ auffassen können, setzen wir $n = 2^k$ voraus. Es gilt nun

$$\begin{aligned} f(w^j) &= a_0 + a_1 w^j + a_2 w^{2j} + \dots + a_{n-1} w^{(n-1)j} \\ &= [a_0 + a_2 w^{2j} + \dots + a_{n-2} w^{(n-2)j}] + w^j [a_1 + a_3 w^{2j} + \dots + a_{n-1} w^{(n-2)j}] \\ &= \sum_{0 \leq i \leq n/2-1} a_{2i} (w^2)^{ij} + w^j \sum_{0 \leq i \leq n/2-1} a_{2i+1} (w^2)^{ij}. \end{aligned}$$

Anstatt f an den Stellen w^0, \dots, w^{n-1} auszuwerten, genügt es, die Polynome f_1 und f_2 mit den Koeffizientendarstellungen $(a_0, a_2, \dots, a_{n-2})$ und $(a_1, a_3, \dots, a_{n-1})$ an den Stellen w^0, w^2, \dots, w^{n-2} auszuwerten. Da $w^n = 1$, gilt $w^0 = w^n$, $w^2 = w^{n+2}$, \dots , $w^{n-2} = w^{n+n-2}$. Wir müssen f_1 und f_2 also nur an den Stellen w^0, w^2, \dots, w^{n-2} auswerten. Damit haben wir es mit zwei Polynomen vom Grad $n/2 - 1$ und $n/2$ Funktionswerten zu tun. Außerdem ist, falls $n > 2$, w^2 eine primitive $(n/2)$ -te Einheitswurzel. Wäre $w^2 = 1$, würde nicht die Summe aller w^{2j} den Wert 0 haben. Offensichtlich ist $(w^2)^{n/2} = w^n = 1$, und für $1 \leq k \leq n/2 - 1$ ist

$$\sum_{0 \leq j \leq n/2-1} (w^2)^{jk} = w^0 + w^{2k} + \dots + w^{(n/2-1)2k}.$$

Es ist $1 \leq 2k \leq n-1$ und daher

$$w^0 + w^{2k} + \dots + w^{(n-1)2k} = 0$$

Andererseits ist

$$w^0 + w^{2k} + \dots + w^{(n/2-1)2k} + w^{(n/2)2k} + \dots + w^{(n-1)2k} =$$

$$w^0 + w^{2k} + \dots + w^{(n/2-1)2k} + w^{nk} (w^0 + w^{2k} + \dots + w^{(n/2-1)2k}) = 2(w^0 + \dots + w^{(n/2-1)2k}).$$

Da in den von uns betrachteten Ringen 2 ein multiplikatives Inverses hat, folgt

$$w^0 + w^{2k} + \dots + w^{(n/2-1)2k} = 0.$$

Damit können f_1 und f_2 rekursiv ausgewertet werden.

Wir berechnen nun vorab w^0, w^1, \dots, w^{n-1} mit $n-2$ Multiplikationen. Damit müssen in den rekursiven Aufrufen keine Potenzen von w mehr berechnet werden. Für $n=2$ müssen $f(1) = a_0 + a_1$ und $f(w) = a_0 + a_1 w$ berechnet werden. Es genügen 3 Operationen. Für $n > 2$ genügen nach den rekursiven Aufrufen eine Multiplikation und eine Addition, um $f(w^j)$ nach der oben entwickelten Formel zu berechnen. Für $j=0$ fällt die Multiplikation noch weg. Wenn wir mit $C(n)$ die Zahl der arithmetischen Operationen in R bezeichnen, gilt für $n=2^k$

$$C(2) = 3 \text{ und } C(n) = 2 \cdot C(n/2) + 2n - 1.$$

Wenn wir $C(1) = 0$ definieren, gilt die Rekursionsgleichung auch für $n=2$. Wir kennen die Rekursion

$$C_1(1) = 2 \text{ und } C_1(n) = 2 \cdot C_1(n/2) + 2n.$$

Ihre Lösung ist $C_1(n) = 2n \log n + 2n$. Wir erhalten dabei $nC_1(1) = 2n$ als Summand. Für

$$C_2(1) = 0 \text{ und } C_2(n) = 2 \cdot C_2(n/2) + 2n$$

lautet die Lösung also $C_2(n) = 2n \log n$. Nun müssen noch die Summanden „ -1 “ berücksichtigt werden. Sie treten bei jeder Problemteilung auf. Davon gibt es $1 + 2 + \dots + 2^{k-1} = n-1$ viele. Also ist

$$C(n) = 2n \log n - n + 1.$$

Schließlich müssen wir die $n-2$ Multiplikationen zur Berechnung der Potenzen von w addieren. Insgesamt folgt

Satz 5.8.6: Die schnelle Fouriertransformation lässt sich mit $2n \log n - 1$ Additionen und Multiplikationen in dem zugrunde liegenden Ring berechnen.

Wir wollen noch kurz diskutieren, wie wir aus $\text{DFT}_n(a)$ wieder a erhalten (dies ist die Transformation von der Darstellung eines Polynoms durch die Funktionswerte an sorgfältig ausgewählten Stellen in die Koordinatendarstellung). Wir erinnern daran, dass

$$\text{DFT}_n(a) = W \cdot a$$

ist. Nun lässt sich nachrechnen, dass $w^{-1} := w^{n-1}$ ebenfalls eine primitive n -te Einheitswurzel ist. Außerdem existiert in den von uns betrachteten Ringen n^{-1} . Damit lässt sich nachrechnen, dass die Matrix W invertierbar ist und W^{-1} den Eintrag $n^{-1}w^{-ij}$ an Position (i, j) , $0 \leq i, j \leq n-1$ hat. Somit ist

$$a = W^{-1} \cdot \text{DFT}_n(a).$$

Es sei W^* die Matrix mit dem Eintrag w^{-ij} an Position (i, j) . Dann lässt sich $a^* = W^* \cdot \text{DFT}_n(a)$ mit der schnellen Fouriertransformation berechnen. Um a zu erhalten, müssen die Einträge in a^* nur noch mit n^{-1} multipliziert werden.

Am Beispiel der schnellen Fouriertransformation für $n = 8$ wollen wir das Prinzip der Einsparungen noch einmal verdeutlichen.

$$\begin{aligned} & a_0 + a_1 w^l + a_2 w^{2l} + a_3 w^{3l} + a_4 w^{4l} + a_5 w^{5l} + a_6 w^{6l} + a_7 w^{7l} = \\ & [a_0 + a_2 w^{2l} + a_4 w^{4l} + a_6 w^{6l}] + w^l [a_1 + a_3 w^{2l} + a_5 w^{4l} + a_7 w^{6l}] = \\ & [a_0 + a_4 w^{4l}] + w^{2l} [a_2 + a_6 w^{4l}] + w^l ([a_1 + a_5 w^{4l}] + w^{2l} [a_3 + a_7 w^{4l}]). \end{aligned}$$

Innerhalb der $[\cdot]$ -Klammern kommen nur w^0 und w^{4l} vor. Dabei ist $w^0 = 1$ und w^{4l} kann nur die Werte 1 und w^4 annehmen. Für beide Möglichkeiten werden die $[\cdot]$ -Klammern ausgewertet. Die Multiplikationen mit w^{2l} müssen nur für $l \in \{0, 1, 2, 3\}$ durchgeführt werden. Nur die Multiplikation der (\cdot) -Klammer muss alle Faktoren 1, w , w^2 , w^3 , w^4 , w^5 , w^6 und w^7 berücksichtigen.

5.9 Nächste Nachbarn in der Ebene

In der algorithmischen Geometrie ist es ein grundlegendes Problem, für eine Punktmenge $S = \{p_1, \dots, p_n\} \subseteq \mathbb{R}^d$ zwei Punkte mit minimalem euklidischen Abstand zu berechnen. Natürlich kommen wir mit Zeit $O(n^2)$ aus, indem wir für alle Punktpaare den Abstand berechnen und dann das Minimum dieser Abstände bestimmen. Für $d = 1$, also n Punkte auf der Geraden, kommen wir mit folgendem einfachen Ansatz sogar mit $O(n \log n)$ Rechenschritten aus. Wir sortieren die Punkte, so dass $p_{i_1} \leq \dots \leq p_{i_n}$ ist. Dann brauchen wir nur noch die $n - 1$ Punktpaare $(p_{i_j}, p_{i_{j+1}})$, $1 \leq j \leq n - 1$, zu betrachten. In der Ebene können wir nicht sortieren. Um vom Fall der Geraden, also $d = 1$, etwas für den Fall der Ebene, also $d = 2$, zu lernen, beginnen wir mit einem divide-and-conquer Algorithmus für $d = 1$.

Auch hier nehmen wir an, dass die Punkte bereits sortiert worden sind. Sei also $p_1 \leq \dots \leq p_n$ und $M := p_{\lfloor n/2 \rfloor}$ der Median. Außerdem vereinbaren wir, dass für Teilprobleme mit nur einem Punkt der Minimalabstand $\delta = \infty$ ist. Nun lösen wir die Teilprobleme $S_1 = \{p_1, \dots, p_{\lfloor n/2 \rfloor}\}$ und $S_2 = \{p_{\lfloor n/2 \rfloor + 1}, \dots, p_n\}$ rekursiv, wobei wir selbstverständlich diese Folgen nicht mehr sortieren müssen. Es seien δ_1 und δ_2 die Minimalabstände der beiden Teilprobleme. Dann ist $\delta' := \min\{\delta_1, \delta_2\}$ eine obere Schranke für den Minimalabstand im Gesamtproblem. Damit $p_i \in S_1$ und $p_j \in S_2$ einen kleineren Abstand als δ' haben, muss $p_i \in [M - \delta', M]$ und $p_j \in [M, M + \delta']$ sein. Dafür kommt nur ein Punkt $p_1 \in S_1$ und ein Punkt $p_2 \in S_2$ in Frage. An der Nahtstelle zwischen S_1 und S_2 ist das Problem also „fast 0-dimensional“. Wir wollen nun für $d = 2$ genauso vorgehen und hoffen, dass das Problem an der Nahtstelle „fast 1-dimensional“ und daher einfach ist.

In einer Preprocessing Phase werden die Punkte bezüglich ihrer x -Koordinate sortiert. Es sei M der Median der x -Koordinaten. Wir zerlegen das Problem in ein Teilproblem S_1 mit $\lfloor n/2 \rfloor$ Punkten, deren x -Koordinate höchstens M ist, und ein Teilproblem S_2 mit $\lfloor n/2 \rfloor$

Punkten, deren x -Koordinate mindestens M ist. Die Teilprobleme werden rekursiv gelöst. Die Minimalabstände seien δ_1 und δ_2 . Dann sei $\delta' := \min\{\delta_1, \delta_2\}$. Nur Punkte $p \in S_1$ mit $x(p) \in (M - \delta', M]$ und $p' \in S_2$ mit $x(p') \in [M, M + \delta')$ können einen Abstand kleiner als δ' haben. Hierbei ist $x(p)$ die x -Koordinate von p .

Wir betrachten also nur noch die beiden vertikalen Streifen der Breite δ' , die links und rechts von der Vertikalen durch $(M, 0)$ liegen (siehe Abbildung 5.9.1).

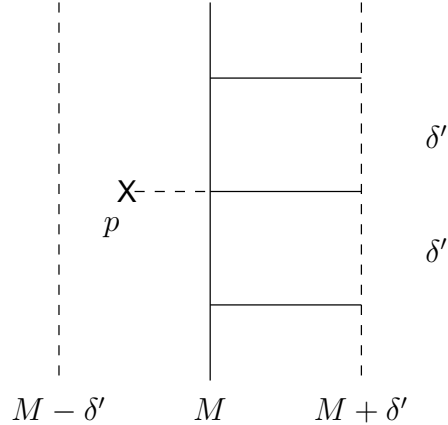


Abbildung 5.9.1: Ein Punkt p im linken δ' -Streifen und ein $\delta' \times 2\delta'$ -Rechteck im rechten δ' -Streifen, das alle Punkte p' rechts von der Vertikalen mit $d(p, p') < \delta'$ enthält.

Wenn wir jetzt noch die Punkte in den Teilproblemen nach den y -Koordinaten sortiert haben, können wir daraus in linearer Zeit die bezüglich der y -Koordinate sortierten Folgen der Punkte in den beiden Streifen herausfiltern. In Abbildung 5.9.1 ist ein Rechteck der Größe $\delta' \times 2\delta'$ eingezeichnet, das alle Punkte p' mit $x(p') \geq M$ und $d(p, p') < \delta'$ enthält. Da Punktepaaire aus S_2 einen Abstand von mindestens δ' haben, können in diesem Rechteck nicht mehr als sechs Punkte aus S_2 liegen. Wir können also hoffen, alle Punkte $p' \in S_2$, für die $d(p, p') < \delta'$ sein kann, schnell zu finden. Wir benötigen nicht nur die bezüglich der y -Koordinate sortierte Folge, sondern auch für die Teilprobleme die sortierten Teilfolgen. Daher wird die Sortierung bezüglich der y -Koordinate nicht vorab durchgeführt. Auf der untersten Ebene haben wir einelementige Mengen, die bezüglich der y -Koordinate sortiert sind. Wir führen jetzt implizit Mergesort durch. Immer wenn zwei Teilprobleme zu einem Gesamtproblem verbunden sind, werden die zugehörigen bezüglich der y -Koordinate sortierten Punktfolgen gemischt. Wir wissen, dass wir auf diese Weise für jedes betrachtete Teilproblem die Punkte bezüglich der y -Koordinate sortiert erhalten und dass der Gesamtaufwand zur Berechnung dieser Folgen $O(n \log n)$ beträgt.

Wir sind also in der Situation, dass wir δ' kennen und in linearer Zeit aus den bezüglich der y -Koordinate sortierten Punktmengen S_1 und S_2 die sortierte Teilfolge der Punkte in dem δ' -Streifen um die Vertikale durch $(M, 0)$ herausfiltern können. Unsere Aufgabe besteht darin festzustellen, ob es in diesen doppelt verketteten Listen L_1 und L_2 Punktepaaire (p, p') mit $p \in L_1, p' \in L_2$ und $d(p, p') < \delta'$ gibt, um gegebenenfalls den Minimalabstand dieser Punktepaaire zu berechnen. Wir starten an den Listenanfängen, also den Punkten

mit den jeweils kleinsten y -Koordinaten.

Zu jedem p in L_1 suchen wir in L_2 den frühesten Punkt p^* mit $y(p^*) \geq y(p) - \delta'$. Falls es einen solchen Punkt nicht gibt, können wir die Suche abbrechen. Ansonsten betrachten wir alle Punkte p^* in L_2 mit $y(p) - \delta' \leq y(p^*) \leq y(p) + \delta'$ und berechnen den Abstand zu p . Dabei speichern wir jeweils den gefundenen Minimalabstand. Nachdem wir p^* gefunden haben, müssen wir in L_2 maximal die nächsten fünf Punkte betrachten. Danach kehren wir zu p' zurück. Die Rechenzeit bei der Bearbeitung von p zerfällt also in die Zeit für die Suche nach p' und die Zeit $O(1)$ für die Betrachtung der Kandidaten p^* für einen kleineren Abstand als δ' zu p . Wenn wir p in L_1 durch seinen Nachfolger ersetzen, hat dieser höchstens eine größere y -Koordinate. Das zugehörige p' hat also auch höchstens eine größere y -Koordinate und wir müssen in L_2 nur „vorwärts“ suchen. Also ist die Gesamtzeit durch $O(n)$ beschränkt. Neben der Zeit von $O(n \log n)$ für das Sortieren, zerlegen wir das Problem in zwei fast gleich große Probleme, die rekursiv gelöst werden, und benötigen darüber hinaus eine Rechenzeit von $O(n)$. Damit gilt folgender Satz.

Satz 5.9.1: Nächste Nachbarn in der Ebene können in $O(n \log n)$ Rechenschritten berechnet werden.

5.10 Die Sweepline-Technik

Das Problem der Berechnung nächster Nachbarn gehörte bereits in die algorithmische Geometrie. Es konnte mit einem typischen divide-and-conquer Algorithmus effizient gelöst werden. Hier wollen wir eine Technik kennenlernen, die speziell auf Probleme der algorithmischen Geometrie zugeschnitten ist. Wir betrachten Probleme in der Ebene \mathbb{R}^2 . Der zugrunde liegende Raum ist also im Gegensatz zu den meisten anderen von uns betrachteten Problemen nicht diskret. Die Sweepline Technik ist eine Methode, Probleme zu diskretisieren. Wir stellen uns eine Vertikale vor, die die Ebene „überstreicht“. Ihr Schnittpunkt mit der x -Achse wandert also gedanklich von links aus dem Unendlichen kommend nach rechts ins Unendliche. Dabei hält die Gerade an Ereignispunkten an, dies sind Punkte, an denen „etwas passiert“. Es wird versucht, das Problem zu lösen, indem nur an den Ereignispunkten gerechnet wird. (Im \mathbb{R}^3 würde man eine Sweepebene verwenden.)

Diese Technik wollen wir an einem Beispiel, dem Rechteckmaßproblem, konkretisieren. Gegeben sind n achsenparallele Rechtecke R_1, \dots, R_n , die sich auch schneiden dürfen. Wir wollen die von allen Rechtecken zusammen überdeckte Fläche berechnen, also den Flächeninhalt von $R = R_1 \cup \dots \cup R_n$. Das Gebilde R (siehe Abbildung 5.10.1) kann recht kompliziert aussehen. Es kann viele Ausbuchtungen und Löcher haben.

Die erste Idee besteht in folgender Erkenntnis. Ereignispunkte sind die x -Koordinaten, an denen ein Rechteck beginnt (linke vertikale Seite) oder ein Rechteck endet (rechte vertikale Seite). Zwischen zwei benachbarten Ereignispunkten „ändert sich nichts“. Genauer: Zwischen zwei benachbarten Ereignispunkten ist die Länge des Durchschnitts der Sweepline mit R konstant. Diese Länge werden wir als Maßzahl bezeichnen. Da wir n Rechtecke haben, haben diese $2n$ vertikale Seiten. Die zugehörigen $2n$ x -Koordinaten werden in Zeit $O(n \log n)$ sortiert. Wir bezeichnen die sortierte Folge mit x_1, \dots, x_{2n} , wobei $x_i \leq x_{i+1}$ ist.

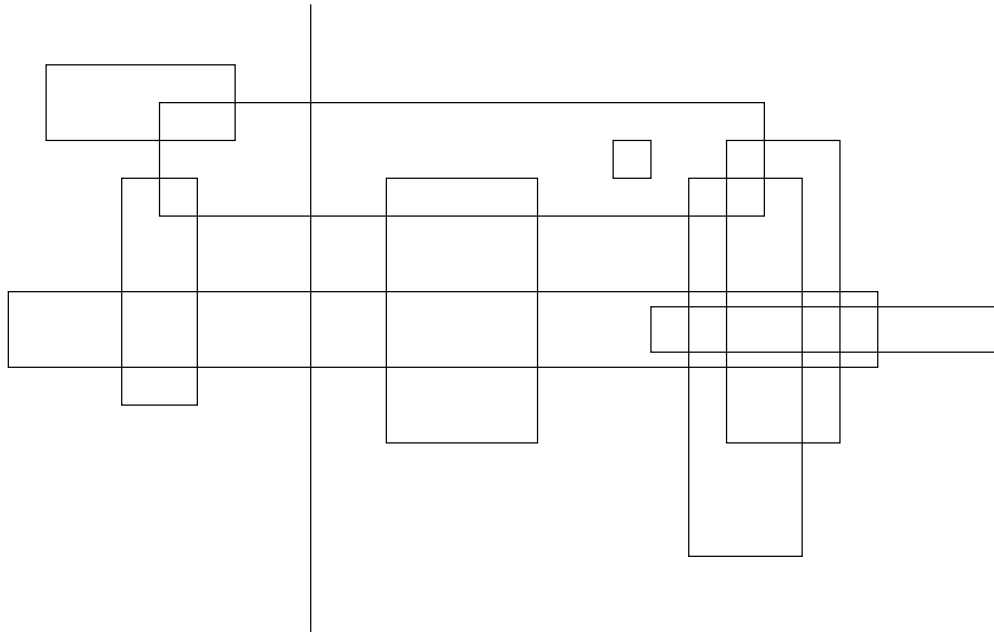


Abbildung 5.10.1: Eine Eingabe für das Rechteckmaßproblem und eine Sweepline.

Diese x -Werte bilden unsere Ereignispunkte. Wenn die Maßzahl im Intervall $[x_{i-1}, x_i]$ den Wert m_i hat, ist die gesamte überdeckte Fläche

$$M = \sum_{2 \leq i \leq 2n} m_i \cdot (x_i - x_{i-1}).$$

Unsere Aufgabe besteht also nur noch in der Berechnung der m_i -Werte. Im zugehörigen Ereignisintervall sind bestimmte Rechtecke aktiv (sie schneiden die Sweepline), die anderen Rechtecke sind passiv. An jedem Ereignispunkt wird ein passives Rechteck aktiv oder ein aktives Rechteck passiv. Der Durchschnitt eines aktiven Rechtecks mit der Sweepline lässt sich als Intervall auf der Sweepline beschreiben. Wir verwalten also aktive Intervalle und daher ist es naheliegend, Segmentbäume zu verwenden. Wir betrachten nun die y -Koordinaten, an denen Rechtecke beginnen oder enden, als zentrale Informationen. Diese $2n$ y -Koordinaten werden in Zeit $O(n \log n)$ sortiert zu $y_1 \leq \dots \leq y_{2n}$. Für jedes Rechteck R_j kennen wir die Indizes j_1 und j_2 , so dass R_j von y_{j_1} bis y_{j_2} reicht. Der Segmentbaum arbeitet mit $2n - 1$ Blättern, wobei das i -te Blatt das i -te Elementarintervall $[i, i + 1]$ repräsentiert. Wir wissen aus Kapitel 2.7, dass wir beliebige Intervalle $[i, j]$ in maximal $O(\log n)$ disjunkte Intervalle so zerlegen können, dass diese Teilintervalle im Segmentbaum repräsentiert sind und in Zeit $O(\log n)$ aufgesucht werden können. Um die Maßzahlen zu erhalten, müssen wir Zusatzinformationen im Segmentbaum dynamisch verwalten. Dazu wählen wir folgende Parameter für die Knoten v des Segmentbaumes, wobei v das Intervall $[i, j]$ repräsentiert:

- $c(v)$, die Anzahl der aktiven Intervalle, bei deren Zerlegung das an v repräsentierte Intervall entsteht,

- $m(v)$, die zu v gehörige Maßzahl, also die Länge des Durchschnitts des Intervalls $[y_i, y_j]$ mit der Vereinigung aller aktiven Intervalle, die nicht höher im Segmentbaum abgespeichert sind.

An der Wurzel r finden wir dann die für das betrachtete x -Intervall aktuelle Maßzahl.

Wie verwalten wir die m - und c -Parameter? Die Initialisierung ist einfach: Es ist $c(v) = 0$ und $m(v) = 0$ für alle Knoten v . Dann starten wir am ersten Ereignispunkt x_1 . Die Verwaltung der c -Werte ist ebenfalls einfach. Wenn ein Rechteck aktiv wird, wird es an $O(\log n)$ Stellen im Segmentbaum eingetragen, die zugehörigen c -Werte erhöhen sich also um 1. Wenn ein Rechteck passiv wird, wird an den zugehörigen Stellen der c -Wert um 1 kleiner.

Wenn v das Intervall $[i, j]$ repräsentiert, gilt für die m -Werte folgendes:

- Falls v ein Blatt ist, also $j = i + 1$ ist, ist $m(v) = 0$, falls $c(v) = 0$ ist, und $m(v) = y_j - y_i$, falls $c(v) > 0$ ist.
- Falls v innerer Knoten ist, also $j > i + 1$ ist, ist $m(v) = y_j - y_i$, falls $c(v) > 0$ ist (da das ganze Intervall überdeckt wird), und $m(v) = m(v_1) + m(v_2)$ für die Kinder v_1 und v_2 von v , falls $c(v) = 0$ ist. Im letzten Fall ist am Knoten v kein Rechteck eingetragen, die Kinder repräsentieren eine disjunkte Zerlegung von $[i, j]$ und daher addieren sich die Maßzahlen.

Wenn bei der Einfügung und Streichung eines Intervalls (oder Rechtecks) der Teilbaum mit Wurzel v des Segmentbaumes gar nicht erreicht wird, bleiben alle alten Informationen gültig. Wir betrachten also nur die $O(\log n)$ besuchten Knoten. Die Änderungen der m -Werte werden bottom-up durchgeführt (wir speichern die Suchpfade auf einem Stack und bearbeiten sie dann rückwärts), damit die Werte an den Kindern bereits aktuell sind. Die obige Beschreibung zeigt, dass für jeden Knoten eine Rechenzeit von $O(1)$ genügt. Also benötigen wir insgesamt Zeit $O(n \log n)$ für die beiden Sortierv Verfahren und darüber hinaus Zeit $O(\log n)$ für jeden der $2n$ Ereignispunkte.

Satz 5.10.1: Das Rechteckmaßproblem kann in Zeit $O(n \log n)$ gelöst werden.

5.11 Die Analyse von Spielbäumen

Spielbäume können endliche Spiele mit vollständiger Information zwischen zwei Personen, Alice und Bob, vollständig beschreiben. Sie beschreiben alle möglichen Spielverläufe. Zu jedem Zeitpunkt ist ein Spieler am Zug und hat die Auswahl zwischen endlich vielen Spielzügen. Dies wird durch entsprechend viele Kinder dargestellt. Somit beschreiben die Knoten der Spielbäume Spielsituationen. Blätter beschreiben die Spielsituationen, an denen das Spiel zu Ende ist und dann wird der Spielausgang, also wieviel Bob an Alice zahlen muss, am Blatt angegeben (bei negativen Beträgen, zahlt natürlich Alice an Bob).

Vollständige Information bedeutet hierbei, dass Alice und Bob den gesamten Spielbaum kennen. Dies impliziert, dass es keine Zufallsschritte gibt (Alice muss würfeln oder Bob

muss eine Karte vom Stapel nehmen). Außerdem sind Kartenspiele ausgeschlossen, bei denen Alice die Karten, die Bob hat, nicht kennt (und umgekehrt). Brettspiele wie Schach, Go, Dame, Mühle oder Halma gehören jedoch zu der beschriebenen Klasse. Diese Spiele sind aus spieltheoretischer Sicht trivial. Wir können Spielbäume bottom-up analysieren. An Blättern v ist das Spiel entschieden und der so genannte Spielwert $s(v)$ entspricht der Markierung des Blattes. Betrachten wir nun einen Knoten v , dessen Kinder v_1, \dots, v_k alle Blätter sind. Ist Alice am Zug, ist ihre optimale Strategie trivialerweise die, einen Zug zu wählen, der zu einem Kind mit dem größten $s(v_i)$ -Wert führt. Ist Bob am Zug, wählt er einen Zug, der zu einem Kind mit dem kleinsten $s(v_i)$ -Wert führt. In jedem Fall kennen wir den Wert eines Knotens, wenn wir den Wert der Kinder kennen:

Alice am Zug (Max-Knoten): $s(v) = \max\{s(v_1), \dots, s(v_k)\}$,

Bob am Zug (Min-Knoten): $s(v) = \min\{s(v_1), \dots, s(v_k)\}$.

Auf diese Weise können wir alle $s(v)$ berechnen und an der Wurzel steht der Wert des Gesamtspiels. Alice kann sich bei guter Spielweise die Auszahlung des Wertes sichern und Bob kann sich bei guter Spielweise dagegen sichern, mehr als den Wert zu zahlen (bei guter Spielweise wird der Zug zu dem Kind mit maximalem (Alice) bzw. minimalem Wert (Bob) gewählt).

Schach und Go sind also nur deswegen noch spannend, weil wir nicht in der Lage sind, den gesamten Spielbaum zu konstruieren und auszuwerten. Schachprogramme konstruieren einen Anfangsteil des Baumes und an den Blättern dieses Teilbaumes, an denen das Spiel noch nicht zu Ende ist, wird der Spielwert geschätzt. Der so gebildete Spielbaum wird dann ausgewertet und ein darin enthaltener optimaler Zug gewählt. Die Auswertung eines Spielbaumes ist mit der beschriebenen Vorgehensweise in linearer Zeit möglich. Also gibt es gar kein algorithmisches Problem? Ist lineare Zeit in jedem Fall auch nötig? Wir werden einen Algorithmus beschreiben, der in vielen Fällen ganze Teilbäume nicht betrachtet und dennoch den korrekten Spielwert berechnet. Diese Strategie heißt α - β -Pruning (α - β , um Alice und Bob zu beschreiben, und Pruning, da Teilbäume als uninteressant abgeschnitten werden).

Dazu vergeben wir auch vorläufige Spielwerte $s^*(v)$, wobei für Max-Knoten $s^*(v) \leq s(v)$ und für Min-Knoten $s^*(v) \geq s(v)$ gelten muss. Die Initialisierung $s^*(v) = -\infty$ für Max-Knoten und $s^*(v) = \infty$ für Min-Knoten ist daher sicherlich zulässig. Diese Initialisierungen werden erst vorgenommen, wenn die Knoten aufgesucht werden.

Wenn wir für einen Max-Knoten v den vorläufigen Spielwert $s^*(v)$ und eines seiner Kinder v_i den vorläufigen Spielwert $s^*(v_i)$ kennen, kann $s^*(v)$ durch $\max\{s^*(v), s^*(v_i)\}$ ersetzt werden, da an Max-Knoten $s(v) \geq s(v_i) \geq s^*(v_i)$ gilt. Für Min-Knoten kann $s^*(v)$ durch $\min\{s^*(v), s^*(v_i)\}$ ersetzt werden.

Der Rechenzeitgewinn entsteht in folgenden Situationen:

- Sei v Max-Knoten und der Min-Knoten v_i ein Kind von v . Falls $s^*(v_i) \leq s^*(v)$, muss der Teilbaum mit Wurzel v_i nicht weiter betrachtet werden.
- Sei v Min-Knoten und der Max-Knoten v_i ein Kind von v . Falls $s^*(v_i) \geq s^*(v)$, muss der Teilbaum mit Wurzel v_i nicht weiter betrachtet werden.

Wir zeigen die Korrektheit dieser Behauptungen. Im ersten Fall gilt wegen der Bedingungen für die s^* -Werte

$$s(v_i) \leq s^*(v_i) \leq s^*(v) \leq s(v).$$

Die korrekte Berechnung von $s(v_i)$ vermehrt unser Wissen nicht, denn wir kennen bereits die untere Schranke $s^*(v)$ für $s(v)$. Die zweite Behauptung folgt analog.

An einem kleinen Beispiel (siehe Abbildung 5.11.1) wollen wir die Vorgehensweise erläutern. Das wahre Einsparungspotenzial wird natürlich erst bei der Betrachtung großer Bäume deutlich.

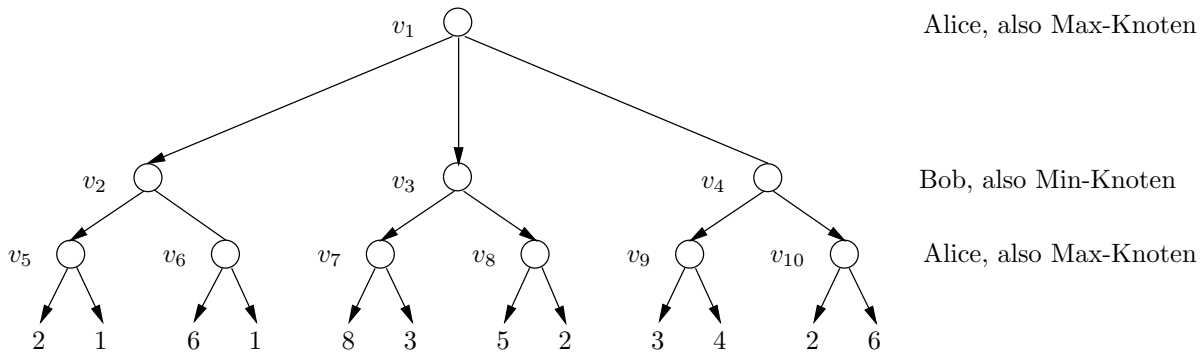


Abbildung 5.11.1: Ein Spielbaum.

Wir bearbeiten den Spielbaum mit einem DFS-Durchlauf. Als erstes wird v_5 ausgewertet. Es folgt

$$s(v_5) = 2, \text{ Zug 1 an } v_5 \text{ optimal, } s^*(v_2) = 2.$$

Für v_6 wird der linke Teilbaum aufgesucht, dessen Wert 6 bekannt ist. Also ist

$$s^*(v_6) = 6.$$

Da $s^*(v_6) \geq s^*(v_2)$, muss der Teilbaum mit Wurzel v_6 nicht weiter betrachtet werden. Dies impliziert

$$s(v_2) = 2, \text{ Zug 1 an } v_2 \text{ optimal, } s^*(v_1) = 2.$$

Als nächstes wird v_7 ausgewertet. Dies impliziert

$$s(v_7) = 8, \text{ Zug 1 an } v_7 \text{ optimal, } s^*(v_3) = 8.$$

Da $s^*(v_3) > s^*(v_1)$, kann der Wert von $s^*(v_1)$ auf Grund des neuen Wertes von $s^*(v_3)$ nicht verändert werden. Die Betrachtung des ersten Kindes von v_8 führt zu

$$s^*(v_8) = 5,$$

aber zu keiner Veränderung von $s^*(v_3)$.

Die Betrachtung des zweiten Kindes von v_8 führt zu

$$s(v_8) = 5, \text{ Zug 1 an } v_8 \text{ optimal, } s(v_3) = 5, \text{ Zug 2 an } v_3 \text{ optimal, } s^*(v_1) = 5.$$

Nach Auswertung von v_9 folgt

$$s(v_9) = 4, \text{ Zug 2 an } v_9 \text{ optimal, } s^*(v_4) = 4.$$

Da $s^*(v_4) < s^*(v_1)$, muss der Teilbaum mit Wurzel v_4 nicht weiter analysiert werden und es ist

$$s(v_1) = 5, \text{ Zug 2 an } v_1 \text{ optimal.}$$

Bei optimaler Spielweise wählt Alice Zug 2, dann Bob Zug 2 und schließlich Alice Zug 1. Dann zahlt Bob Alice 5 Euro. Vielleicht sollte Bob gar nicht spielen. Andererseits spielen wir auch Schach mit schwarzen Figuren, obwohl es vielleicht für Weiß eine Gewinnstrategie gibt.

Die Größe der Einsparungen hängt davon ab, in welcher Reihenfolge wir die Kinder betrachten. Ohne Vorwissen ist es günstig, die Reihenfolge zufällig zu wählen. Mit Vorwissen sollten die hoffnungsvollsten Züge zuerst betrachtet werden.

5.12 Randomisierte Suchheuristiken

Die von uns bisher entwickelten effizienten Algorithmen basieren auf einer Strukturanalyse des betrachteten Problems. Wir wollen uns nun noch der Situation zuwenden, in denen eine solche Strukturanalyse nicht möglich ist. Dies kann an mangelnden Ressourcen (Zeit, Geld, algorithmische Kenntnisse) liegen, aber auch daran, dass Informationen über das Problem fehlen (Black Box Optimierung). Das zweite Szenario tritt insbesondere bei der Optimierung technischer Systeme auf. Es gibt dann n freie Parameter, die eingestellt werden können. Das Verhalten des Systems wird durch die Parameterbelegungen beeinflusst, aber der direkte Zusammenhang, also die zu optimierende Funktion, ist nicht bekannt, da das System vielleicht zu komplex ist. Wir können Wissen über das System nur erlangen, indem wir Parameterbelegungen wählen, das Verhalten des Systems beobachten und daraus einzelne Funktionswerte gewinnen.

Dies können wir folgendermaßen modellieren. Die zu maximierende Funktion $f : S \rightarrow \mathbb{R}$ ist unbekannt, aber der Suchraum S ist vorgegeben. Wir haben uns hier auf diskrete Probleme spezialisiert, der Fall $S = \{0, 1\}^n$ und damit der Fall pseudoboolescher Funktionen ist dann der wichtigste Fall. Beim TSP ist jedoch der Suchraum die Menge aller Permutationen auf $\{1, \dots, n\}$. Wir können nun den ersten Suchpunkt $a_1 \in S$ wählen und erhalten den Wert $f(a_1)$. In der allgemeinen Situation kennen wir bereits die Suchpunkte a_1, \dots, a_{t-1} und ihre Funktionswerte (Fitnesswerte) $f(a_1), \dots, f(a_{t-1})$. In Abhängigkeit von diesem Wissen wählen wir den t -ten Punkt a_t und erhalten den Wert $f(a_t)$. Wenn wir die Suche abbrechen, können wir den besten ausgewerteten Suchpunkt a_i als Lösung verwenden. Natürlich kann in diesem Szenario globale Optimalität nicht gewährleistet werden. Die Frage ist, welche Suchstrategien in wichtigen Fällen ein gutes Ergebnis in erträglicher Zeit liefern. Suchstrategien beschreiben, wie für $(a_1, f(a_1), \dots, a_{t-1}, f(a_{t-1}))$ der nächste Suchpunkt a_t gewählt wird. Bei so schlechter Informationslage können wir nur von randomisierten Suchstrategien (oder Suchheuristiken) ein gutes Verhalten erhoffen.

An dieser Stelle unterscheiden wir allgemeine Suchheuristiken, die gar keine Kenntnisse über f voraussetzen, aber ein gutes Verhalten für viele typische Probleme haben sollen, und problemspezifische Suchheuristiken, bei denen einige Module auf die vorliegende Situation zugeschnitten sind.

Praktisch alle Suchheuristiken beginnen mit einer Informationsreduktion. Von den bereits ausgewerteten Suchpunkten werden nur einige abgespeichert, alle anderen werden „vergessen“. Oft wird nur ein Suchpunkt (der aktuelle Suchpunkt oder das aktuelle Individuum) für die Auswahl des nächsten Suchpunktes gespeichert (wobei im Hintergrund stets der beste bisher gesehene Suchpunkt gespeichert wird). Ansonsten ist es eine Menge von aktuellen Suchpunkten (die aktuelle Population) vorgegebener Größe. Die Initialisierung erfolgt mangels Vorwissen meistens rein zufällig. Es können aber auch aus anderen Betrachtungen bekannte vermutlich gute Suchpunkte berücksichtigt werden.

Bei der Suche ist es sicher sinnvoll, in der Nähe von guten Suchpunkten zu suchen, aber auch zu sichern, dass man aus lokalen Optima herauskommt.

Die lokale Suche arbeitet mit einem aktuellen Suchpunkt und erzeugt jeweils zufällig einen Nachbarn und wertet diesen aus. Es ist aber notwendig, einen sinnvollen Nachbarschaftsbegriff zu haben, in $\{0,1\}^n$ ist der Hammingabstand ein natürliches Distanzmaß und Nachbarn sind alle Punkte, die vom aktuellen Suchpunkt einen durch d beschränkten Hammingabstand haben. Der neue Suchpunkt verdrängt den aktuellen Suchpunkt, wenn sein f -Wert mindestens so groß wie der f -Wert des momentan aktuellen Suchpunktes ist. Auf diese Weise werden lokale Optima allerdings nicht verlassen. Dies kann jedoch durch eine Multistartstrategie ausgeglichen werden. Wenn über einen bestimmten Zeitraum keine (oder keine genügend große) Verbesserung des Fitnesswertes eingetreten ist, wird der aktuelle Suchpunkt durch einen zufälligen Suchpunkt ersetzt. Natürlich können die verschiedenen Suchläufe auch unabhängig und gleichzeitig erfolgen.

Wir wollen eine recht erfolgreiche lokale Suchstrategie, das k -Opting, für das TSP als Beispiel konkret ansprechen. Die aktuelle Rundreise wird an k zufällig gewählten Stellen aufgeschnitten und die k entstandenen Teilstücke werden in zufälliger Reihenfolge wieder zu einer Rundreise zusammengesetzt.

Auch der Metropolis-Algorithmus arbeitet mit nur einem aktuellen Suchpunkt und lokalen Umgebungen. Um lokale Optima zu verlassen, werden auch Verschlechterungen nicht ausgeschlossen. Ein in der Nachbarschaft des aktuellen Suchpunktes zufällig gewählter Suchpunkt kann den aktuellen Suchpunkt auch dann verdrängen, wenn sein f -Wert schlechter ist. Für einen Parameter T verdrängt der neue Suchpunkt x' den aktuellen Suchpunkt x mit Sicherheit, wenn $f(x') \geq f(x)$ ist, und ansonsten mit Wahrscheinlichkeit

$$e^{-(f(x)-f(x'))/T}.$$

Die Akzeptanzwahrscheinlichkeit wird also durch den Parameter T und die Fitnessdifferenz $f(x) - f(x')$ gesteuert.

Natürlich ist es wichtig, T „geeignet“ zu wählen. Intuitiv sind wir zu Beginn der Suche eher bereit, andere Gebiete des Suchraumes zu untersuchen, auch wenn dafür zuerst Verschlechterungen akzeptiert werden müssen. Am Ende hoffen wir in der Nähe eines sehr

guten Suchpunktes zu sein und wollen lokal optimieren. Dies legt es nahe, die so genannte Temperatur T in Analogie zur Herstellung besonders reiner Kristalle (Annealing) in geeigneter Weise abzusenken. Der Wert $T = \infty$ beschreibt eine zufällige Suche, bei der jeder neue Suchpunkt akzeptiert wird, während $T = 0$ der lokalen Suche entspricht. In dieser Situation haben wir die Entscheidung zu treffen, wie die Anfangstemperatur gewählt wird und wie sie abgesenkt wird (nur abhängig von der Zeit oder auch von der Entwicklung der f -Werte). Natürlich können diese Strategien ebenfalls die Option von Multistarts nutzen.

Eine andere Option, lokale Optima verlassen zu können, besteht darin, den nächsten Suchpunkt nicht nur in der Nähe zu suchen. Auf dem Raum $\{0,1\}^n$ hat sich folgender Veränderungsoperator (Mutationsoperator) bewährt. Für einen Wert p , der von n abhängen darf, wird x' aus x auf folgende Weise gebildet. Die Bits x'_i , $1 \leq i \leq n$, werden unabhängig voneinander erzeugt. Für jedes i ist

$$\text{Prob}(x'_i = x_i) = 1 - p$$

und somit

$$\text{Prob}(x'_i = 1 - x_i) = p.$$

Die einzelnen Bits werden mit Wahrscheinlichkeit p „geflippt“. Der Standardwert ist $p = 1/n$. Die Anzahl geflippter Bits ist binomialverteilt zu den Parametern n und p . Die erwartete Anzahl geflippter Bits beträgt np und ist somit im Fall $p = 1/n$ genau 1. Die Binomialverteilung für konstantes $\lambda = np$, hier $\lambda = 1$, kann gut durch die Poissonverteilung approximiert werden. Daher ist die Wahrscheinlichkeit, genau k Bits zu flippen, ungefähr $e^{-1}/k!$. Große Sprünge sind also möglich, aber wenig wahrscheinlich. Wenn der Hammingabstand zwischen x und x' genau d ist, beträgt die Wahrscheinlichkeit, x' aus x zu erzeugen, $p^d(1-p)^{n-d}$, und für $p = 1/n$ also $(1/n)^d(1 - (1/n))^{n-d} \geq (1/n)^d e^{-1}$. Dieser Suchalgorithmus wird (1+1)-evolutionärer Algorithmus genannt und ist in Kombination mit der Multistartstrategie recht erfolgreich.

Allgemein arbeiten evolutionäre Algorithmen jedoch mit Populationen, die aus mehr als einem Individuum bestehen. Die $(\mu + \lambda)$ -Strategie hat eine Populationsgröße von μ und erzeugt in einer Phase (Generation) λ neue Suchpunkte (Individuen, Kinder) durch Mutation. Die Auswahl, welche aktuellen Suchpunkte wieviele Kinder erzeugen, erfolgt in Abhängigkeit von den Fitnesswerten (dafür gibt es viele Strategien). Die neue Population wird durch Auswahl von μ aus den nun bestehenden $\mu + \lambda$ Individuen getroffen. Neben der Wahl der μ Individuen mit den größten f -Werten (survival of the fittest) gibt es auch Strategien, die Individuen mit größeren f -Werten nur bevorzugen.

Populationen sind nur sinnvoll, wenn sich ihre Individuen auch genügend unterscheiden, die Population also eine ausreichende Diversität hat. Andererseits hat die fitnessbasierte Auswahl die Tendenz, Suchpunkte mit hohem f -Wert und deren Kinder, die wieder in der Nähe liegen, zu bevorzugen. Dies führt zur Klumpenbildung (oft formal ungenau *vorzeitige Konvergenz* genannt) und hebt den Vorteil von größeren Populationen faktisch auf. Man hat dann nur den Nachteil der Verwaltung größerer Populationen, ohne den potenziellen Nutzen größerer Populationen zu haben. Die so genannten (μ, λ) -Strategien versuchen dem zu begegnen, in dem $\lambda > \mu$ Kinder erzeugt werden und bei der Auswahl der neuen Population nur die λ Kinder betrachtet werden.

Schließlich können Populationen dazu verwendet werden, neue Individuen in Abhängigkeit von mehreren Elternteilen, meistens genau zwei, zu erzeugen. Es seien also x und y zwei aktuelle Individuen, aus denen als Elternpaar (x, y) ein Kind z erzeugt werden soll. Dazu dienen so genannte Kreuzungsoperatoren. Wenn benachbarte Bitpositionen auch einen inhaltlichen Zusammenhang ausdrücken, sind Einpunkt-Kreuzungen (one point crossover) angemessen. Es wird die Schnittposition $i \in \{1, \dots, n-1\}$ zufällig gewählt und z durch

$$z_1 = x_1, \dots, z_i = x_i, z_{i+1} = y_{i+1}, \dots, z_n = y_n$$

definiert. Ansonsten sind gleichmäßige Kreuzungen (uniform crossover) die bessere Wahl. Falls $x_i = y_i$ ist, wird $z_i = x_i = y_i$ gesetzt und ansonsten erhält z_i jeweils mit Wahrscheinlichkeit $1/2$ den Wert 0 oder 1.

Neben den hier vorgestellten wesentlichen Strategien gibt es eine Unzahl von Alternativen. Es stellt sich daher die Frage, wie wir die verschiedenen randomisierten Suchheuristiken bewerten. Man kann nachweisen, dass keine dieser Heuristiken universell besser als die anderen ist. Dies entspricht ja auch gar nicht einem vernünftigen Ziel. Wir erhoffen ja eher, dass die verschiedenen randomisierten Suchheuristiken auf verschiedenen Funktionen gut sind. Wir können dann eine Intuition entwickeln, wann welche Heuristik angemessen ist oder auch verschiedene Heuristiken benutzen und das beste erzielte Ergebnis benutzen.

Lassen sich randomisierte Suchheuristiken überhaupt analysieren? Welcher Typ von Ergebnissen ist möglich? Wir können uns eine für uns interessante Funktionenklasse auswählen und zeigen, dass eine bestimmte randomisierte Suchheuristik mit einer Wahrscheinlichkeit von $1 - o(1)$ in $t(n)$ Schritten eine Lösung mit Güte $1 + \varepsilon(n)$ berechnet. Ein Beispiel dazu: Der (1+1)EA findet sogar das Optimum einer unimodalen Funktion (jeder nicht optimale Suchpunkt hat einen besseren Hamming-Nachbarn) mit b verschiedenen Funktionswerten in einer erwarteten Zeit von $O(nb)$, für affine Funktionen beträgt die erwartete Optimierungszeit $O(n \log n)$. Wir werden diese Ergebnisse hier nicht beweisen. Randomisierte Suchheuristiken lassen sich durch stochastische, meist sogar markoffsche Prozesse auf dem Suchraum beschreiben. Diese stochastischen Prozesse müssen analysiert werden. Die Aussagen helfen uns in folgender Weise. Wir erhalten bewiesene Aussagen für Funktionenklassen, die in Anwendungen vorkommen. Darüber hinaus können wir hoffen, dass die Aussagen für Funktionen, die sich von einer Funktion der betrachteten Klasse nur wenig unterscheiden, in ähnlicher Weise gelten. In jedem Fall ist die Analyse randomisierter Suchheuristiken ein spannendes Forschungsgebiet, das noch immer am Anfang steht und in dem in naher Zukunft viele Ergebnisse zu erwarten sind.