Datenstrukturen und Algorithmen

Sommersemester 1998

Dr. Rudolf Fleischer

Universität Trier

und

Max-Planck-Institut
für Informatik
Saarbrücken

5. August 1998

Vorwort

Ich habe die Vorlesung Informatik II (Datenstrukturen und Algorithmen) im Sommersemester 1998 an der Universität Trier im Rahmen einer Vertretungsprofessur gelesen. Das vorliegende Skript ist eng angelehnt an das Skript einer gleichnamigen Vorlesung, die im Wintersemester 1997/98 an der Universität Saarbrücken zusammen mit Prof. Dr. Kurt Mehlhorn gelesen hatte.
Die Reihenfolge, in der der Stoff präsentiert wird, entspricht nicht immer genau dem Ablauf der Vorlesung. Außerdem enthält das Skript sicher auch noch einige kleinere Druckfehler, die mir bei der Überarbeitung entgangen sind.

Trier, den 5. August 1998

Dr. Rudolf Fleischer

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Lernziele

- Gute Algorithmen für wichtige Probleme.
- Paradigmen für gute Algorithmen.
- Korrektheitsbeweise.
- Analyse von Algorithmen.

Anhang A gibt eine Einführung in den Differenzenkalkül, eine allgemeine Methode zum Ausrechnen von Summen. Anhang B definiert einige grundlegende Begriffe aus der Wahrscheinlichkeitsrechnung, die wir in dieser Vorlesung brauchen.
Wenn in dieser Vorlesung Programme vorkommen, dann sind sie entweder in C++ geschrieben (teilweise unter Verwendung der Bibliothek LEDA, siehe Literaturliste) oder in C-ähnlichem Pseudocode.

1.2 Literatur

Es gibt in der Biliothek einen Semesterapparat zur Vorlesung. Die folgenden Bücher sind empfehlenswert.

Datenstrukturen und Algorithmen

[1] T.H. Cormen, C.E. Leiserson und R.L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, and London, England, 5. Auflage, 1991.

[2] K. Mehlhorn und S. Näher. *LEDA – A Platform of Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, England, 1998. In Vorbereitung. Einen Preview gibt es unter http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html.

[3] C. Meinel. *Effiziente Algorithmen.* Fachbuchverlag Leipzig, 1991.

[4] R.E. Tarjan. *Data Structures and Network Algorithms.* CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, Philadelphia, Pennsylvania, 1983.

[5] Skripte zur Vorlesung Datenstrukturen und Algorithmen an der Universität Saarbrücken.

http://www.mpi-sb.mpg.de/~rudolf/Vorlesung/DA98/lit.html.

Kombinatorik

[1] R.L. Graham, D.E. Knuth und O. Patashnik. *Concrete Mathematics – A Foundation for Computer Science.* Addison-Wesley, Reading, MA, 1989.

[2] A. Engel. *Wahrscheinlichkeitsrechnung und Statistik, Band 1.* Klett Studienbücher Mathematik. Ernst Klett Verlag, Stuttgart, 1973.

`C++` und LEDA

[1] S. Meyers. *Effektiv* `C++` *programmieren.* Addison-Wesley, Reading, MA, 1995.

[2] M.A. Ellis und B. Stroustrap. *The Annotated* `C++` *Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[3] B. Stroustrup. *The* `C++` *Programming Language.* AT&T Bell Telephone Laboratories, Inc., Murray Hill, NJ, 2. Auflage, 1993.

[4] K. Mehlhorn und S. Näher. LEDA : A platform for combinatorial and geometric computing. *Journal of the ACM*, 38(1):96–102, 1995.

[5] K. Mehlhorn, S. Näher und C. Uhrig. *The LEDA User Manual – Version 3.6.* 66123 Saarbrücken, 1998.

[6] T. Standish. Data Structures in Java, Addison-Wesley, Reading, MA, 1998.

1.3 Asymptotik

Wir arbeiten in dieser Vorlesung mit dem *RAM-Maschinenmodell* (RAM = Random Access Machine) :

- Ein Prozessor.
- Sequentiell.
- Direkte/indirekte Speicherzugriffe.

Im *Einheitskostenmaß* nehmen wir an, daß jeder Schritt der Maschine Kosten 1 verursacht, unabhängig von der Größe der Operanden. Demgegenüber verursacht im *logarithmischen Kostenmaß* die Manipulation von b-Bit-Zahlen Kosten b, aber dieses Kostenmaß spielt in dieser Vorlesung keine Rolle.

Beispiel 1.3.1 Die Zahlen $1, \ldots, n$ seien in beliebiger Reihenfolge in einem Feld $A[0, n-1]$ abgespeichert. Wir wollen die Zahl 1 finden. □

Das folgende einfache Programm löst unser Problem.

Deterministische Suche

begin

$i \leftarrow 0;$

while $A[i] \neq 1$ **do** $i \leftarrow i + 1;$ **od**

end

Im schlimmsten Fall finden wir die 1 erst im letzten Schleifendurchlauf. Wir haben bis dahin $(n-1)$-mal i inkrementiert und n-mal die Schleifenbedingung getestet. Zusammen mit der ersten Programmzeile macht das insgesamt Zeit $2n$. Wir sagen dann, das Programm hat *worst-case Laufzeit* $T = 2n$.
Wenn wir annehmen, daß die Zahlen zufällig in A verteilt sind, dann hat das Programm eine *mittlere Laufzeit* $\overline{T}$ von

$$\overline{T} = 1 + \frac{1}{n} \cdot \sum_{i=1}^{n} (2i - 1) = 1 + \frac{1}{n} \cdot (\frac{2}{2} n(n+1) - n) = n + 1 \, .$$

Wenn wir die erste Zeile des Programms durch

$i \leftarrow rand(0, n-1);$

ersetzen, wobei $rand(0, n-1)$ eine zufällige Zahl zwischen 0 und $n-1$ liefert, und dann anschließend i maximal n-mal modulo n inkrementiern, dann ist die Laufzeit des Programms bei einer festen Eingabe eine Zufallsvariable mit Erwartungswert

$$E[T] = n + 1$$

(die gleiche Rechnung wie bei der mittleren Laufzeit). Wir sagen dann, das Programm hat *erwartete Laufzeit* $n + 1$.

Wir haben in der Analyse oben jeden einzelnen Schritt des Programms gezählt. Das ist bei größeren Programmen sehr mühsam, und so genau wollen wir es eigentlich auch gar nicht wissen. Eigentlich interessiert uns nur das asymptotische Verhalten der Laufzeit für große n. Dazu reicht es, einfach die Anzahl der Schleifendurchläufe zu zählen (im worst-case n); die tatsächliche Laufzeit ist dann ein (kleines) konstantes Vielfaches dieser Zahl.
Wenn wir die Effizienz von Algorithmen untersuchen, interessieren wir uns nicht für solche konstante Faktoren, jedenfalls solange sie in einem vernünftigen Rahmen bleiben (einen Faktor von 10^{1000} sollten wir natürlich nicht einfach ignorieren). Uns interessiert mehr, ob die Laufzeifzeitfunktion linear, polynomiell oder exponentiell in der Eingabegröße wächst. Dies können wir formal durch die sogenannte *O-Notation* ausdrücken.

Definition 1.3.2 Sei $g : \mathbb{N} \mapsto \mathbb{R}_0^+$. Dann ist

$$\begin{aligned} O(g) &= \{f : \mathbb{N} \mapsto \mathbb{R}_0^+ \mid \exists c > 0\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}\,, \\ o(g) &= \{f : \mathbb{N} \mapsto \mathbb{R}_0^+ \mid \forall c > 0\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)\}\,. \end{aligned}$$

□

Für die Funktionen f in $O(g)$ ist also, bis auf einen konstanten Faktor, die Funktion g eine asymptotische obere Schranke. Für die Funktionen f in $o(g)$ gilt sogar, daß g asymptotisch deutlich schneller wächst.

Lemma 1.3.3

$$f \in o(g) \Longleftrightarrow \lim_{n \mapsto \infty} \frac{f(n)}{g(n)} = 0\,.$$

Beweis: Sei $f \in o(g)$. Wenn $\lim_{n \mapsto \infty} \frac{f(n)}{g(n)} = c > 0$ wäre, dann würde gelten :

$$\forall \epsilon\, \exists n_0\, \forall n \geq n_0 : \frac{f(n)}{g(n)} \in (c - \epsilon, c + \epsilon)\,.$$

Also mit $\epsilon = \frac{c}{2}$:

$$\forall n \geq n_0 : f(n) > \frac{c}{2} \cdot g(n)\,,$$

ein Widerspruch zur Definition von $o(g)$.

Wenn $f \notin o(g)$ ist, dann gibt es ein $c > 0$, so daß $f(n) \geq c \cdot g(n)$ für manche n gilt, die beliebig groß gewählt werden können. Dann kann aber nicht $\lim_{n \mapsto \infty} \frac{f(n)}{g(n)} = 0$ sein. □

Analog definieren wir asymptotische untere Schranken.

Definition 1.3.4 Sei $g : \mathbb{N} \mapsto \mathbb{R}_0^+$. Dann ist

$$\begin{aligned} \Omega(g) &= \{f : \mathbb{N} \mapsto \mathbb{R}_0^+ \mid \exists c > 0\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : c \cdot g(n) \leq f(n)\}\,, \\ \omega(g) &= \{f : \mathbb{N} \mapsto \mathbb{R}_0^+ \mid \forall c > 0\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : c \cdot g(n) < f(n)\}\,. \end{aligned}$$

□

Lemma 1.3.5

$$f \in \omega(g) \Longleftrightarrow \lim_{n \mapsto \infty} \frac{f(n)}{g(n)} = \infty\,.$$

Beweis: Analog zu Lemma 1.3.3. □

Zu gute Letzt brauchen wir noch eine Schreibweise dafür, daß sich zwei Funktionen asymptotisch nur um einen konstanten Faktor unterscheiden.

Definition 1.3.6 Sei $g : \mathbb{N} \mapsto \mathbb{R}_0^+$. Dann ist

$$\Theta(g) = \{f : \mathbb{N} \mapsto \mathbb{R}_0^+ \mid \exists c_1, c_2 > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}.$$

□

Das folgende Lemma folgt direkt aus den Definitionen oben.

Lemma 1.3.7 *Es gilt :*

$$f \in \Theta(g) \Longleftrightarrow f \in O(g) \cap \Omega(g)\,.$$

□

Obwohl $O(g)$ eigentlich eine Menge ist, schreiben wir normalerweise $f = O(g)$ statt $f \in \Omega(g)$. Der Beweis der folgenden Regeln ist elementar und sei zur Übung empfohlen.

Satz 1.3.8 *Es gilt :*

1. **Transitivität :** *Für* $op \in \{o, O, \omega, \Omega, \Theta\}$ *gilt :*
   Wenn $f(n) = op(g)$ *und* $g = op(h)$ *ist, dann ist* $f = op(h)$.
2. **Reflexivität :** *Für* $op \in \{O, \Omega, \Theta\}$ *gilt :*
$$f = op(f)\,.$$
3. **Symmetrie :**
$$f = \Theta(g) \Longleftrightarrow g = \Theta(f)\,.$$
4. **Vertauschte Symmetrie :**
$$\begin{aligned} f = O(g) &\iff g = \Omega(f)\,, \\ f = o(g) &\iff g = \omega(f)\,. \end{aligned}$$

□

Beispiel 1.3.9 Es ist

- $2n - 5 = \Theta(n)$.
- $\log^2 n = \omega(\log n)$.
- Für $0 < \alpha < \beta$, $0 < a < b$ und $0 < A < B$ haben wir folgende Hierarchie von zunehmend schneller wachsenden Funktionen :

$$\log^* n, \log\log n, \log^\alpha n, \log^\beta n, n, n^a \log^\alpha n, n^a \log^\beta n, n^b, A^n, A^n n^a, A^n n^b, B^n.$$

□

1.4 Amortisierte Analyse – Der Binärzähler

Wir wollen binär von 0 bis n zählen :

$$0 \to 1 \to 10 \to 11 \to 100 \to 101 \to \cdots$$

Dabei seien

$$c_i = 1 + \text{ Anzahl der endenden Einsen in } bin(i)$$

die Kosten (oder Zeit) des Übergangs von i nach $i+1$, wobei $bin(i)$ die Binärdarstellung von i ist. Die Gesamtkosten, um von 0 bis n zu zählen, sind dann

$$C_n = \sum_{i=0}^{n-1} c_i \, .$$

Im schlimmsten Fall kann $c_i = \log n$ sein. Also könnten wir C_n einfach durch $n \log n$ abschätzen. Allerdings beobachtet man, wenn man das ganze programmiert, daß die Laufzeit nicht wie $O(n \log n)$ wächst, sondern nur wie $O(n)$. Das liegt daran, daß bei diesem Problem von den n einzelnen Operationen nicht alle maximal teuer sein können (wir werden gleich sehen, daß sogar nur ein Bruchteil $\frac{1}{2^k}$ der Operationen Kosten k hat).
Also sollten wir die Kosten für eine Folge von n Operationen geschickter abschätzen als durch teuerste Operation mal Anzahl von Operationen. Dies nennt man *amortisierte Analyse*. Wenn jede Folge von n Operationen höchstens Zeit $T(n)$ braucht, dann kann man auch sagen, daß jede einzelne Operation *amortisiert* Zeit $\frac{T(n)}{n}$ braucht. Die amortisierte Zeit einer Operation ist also die durchschnittliche Zeit der Operation in einer langen Folge von Operationen.

Satz 1.4.1 *Es ist $C_n \le 2n$. Ein Zählschritt braucht also nur konstante amortisierte Zeit.*

Wir geben vier Beweise für diesen Satz. Der erste ist sehr technisch und erfordert mathematisches Geschick; der zweite und dritte sind im Prinzip sehr ähnlich, insbesondere aber "sauberer" in dem Sinne, daß wir nur sehr einfache Mathematik benötigen. Der vierte Beweis ist schließlich eine elegante Kurzfassung des zweiten bzw. dritten Beweises.

Beweis: [Version 1 : Die Ingenieursmethode (brute force)]
Es ist $c_i = k$ genau dann, wenn $bin(i)$ die Form

$$bin(i) = \underbrace{\ldots\ldots\ldots\ldots\ldots\ldots\ldots}_{\text{beliebig}} 0 \underbrace{1 \ldots\ldots 1}_{k-1 \text{ Einsen}}$$

hat. Es gibt $m_k \in \mathbb{N}$ solche i's, nämlich die Zahlen

$$\begin{aligned} \mathbf{0} \cdot 2^k &+ 2^{k-1} - 1, \\ \mathbf{1} \cdot 2^k &+ 2^{k-1} - 1, \\ \mathbf{2} \cdot 2^k &+ 2^{k-1} - 1, \\ &\vdots \\ (\boldsymbol{m_k} - \mathbf{1}) \cdot 2^k &+ 2^{k-1} - 1, \end{aligned}$$

wobei

$$\begin{array}{rcccl} (m_k - 1) \cdot 2^k + 2^{k-1} - 1 & < & n & \leq & m_k \cdot 2^k + 2^{k-1} - 1 \\ \Longleftrightarrow \qquad m_k - 1 & < & \dfrac{n - 2^{k-1} + 1}{2^k} & \leq & m_k \end{array}$$

ist. Unter Beachtung, daß $m_k \in \mathbb{N}$ ist, können wir damit m_k nach oben und unten abschätzen.

- Abschätzung nach oben :

$$\begin{array}{rrcl} & m_k - 1 & < & \dfrac{n - 2^{k-1} + 1}{2^k} = \dfrac{n}{2^k} \underbrace{-\dfrac{1}{2} + \dfrac{1}{2^k}}_{\leq 0 \text{ für } k \geq 1} \leq \dfrac{n}{2^k} \\ \Longleftrightarrow & m_k & < & \dfrac{n}{2^k} + 1 \\ \Longrightarrow & m_k & \leq & \left\lceil \dfrac{n}{2^k} \right\rceil . \end{array}$$

- Abschätzung nach unten :

$$\begin{array}{rrcl} & m_k & \geq & \dfrac{n - 2^{k-1} + 1}{2^k} = \dfrac{n}{2^k} \underbrace{-\dfrac{1}{2} + \dfrac{1}{2^k}}_{> -\frac{1}{2} \text{ für } k \geq 1} > \dfrac{n}{2^k} - \dfrac{1}{2} > \dfrac{n}{2^k} - 1 \\ \Longrightarrow & m_k & \geq & \left\lfloor \dfrac{n}{2^k} \right\rfloor . \end{array}$$

Es ist also $m_k \approx n/2^k$. Jetzt sind wir in der Lage, die Gesamtkosten C_n nach oben abzuschätzen :

$$\begin{array}{rcl} C_n & = & \sum_{i=0}^{n-1} c_i \\ & \leq & \sum_{k \geq 1} (k \cdot m_k) \\ & \approx & \sum_{k \geq 1} \left(k \cdot \dfrac{n}{2^k} \right) \\ & = & n \cdot \sum_{k \geq 1} \dfrac{k}{2^k} \\ & \stackrel{(A.4.5)}{=} & 2n . \end{array}$$

□

Beweis: [Version 2 : Die Bankier-Methode (credit-debit)]
Für eine Operation mit Kosten k müssen wir k DM bezahlen. Jede Operation bringt 2 DM mit. Die Operation bezahlen wir mit diesen 2 DM und, falls nötig, mit früher gespartem Geld. Wenn die Operation nur 1 DM kostet, dann können wir 1 DM sparen.
Wir realisieren das Konto dadurch, daß wir in jedem Bit, das den Wert 1 hat, 1 DM deponieren. Wir beginnen mit $i = 0$ und 0 DM auf dem Konto. Wenn $c_i = k$ ist, dann setzen wir die letzten $k - 1$ Bit (die den Wert 1 haben) auf 0 und das k-te Bit auf 1. Dabei werden $k - 2$ DM frei. Wir haben also insgesamt $2 + (k - 2)$ DM zur Verfügung, um die Operation zu bezahlen, was genau reicht. Also wird unser Konto nie negativ und die Gesamtkosten C_n sind höchstens so hoch wie die gesamten Einzahlungen, also $2n$. □

Beweis: [Version 3 : Die Physiker-Methode (Potentialfunktion)]
Wir interpretieren das Guthaben in der vorigen Beweisvariante als Energiepotential. Teure Operationen verringern die Anzahl der Einsen in der Binärdarstellung. Für $i \in \mathbb{N}_0$ definieren wir daher eine Potentialfunktion $pot : \mathbb{N}_0 \to \mathbb{N}_0$ und eine Amortisierungsfunktion $a : \mathbb{N}_0 \to \mathbb{N}_0$ wie folgt :

$$\begin{aligned} pot(i) &= \text{Anzahl der Einsen in } bin(i)\,, \\ a(i) &= c_i + pot(i+1) - pot(i)\,. \end{aligned}$$

a_i sind die *amortisierten Kosten* der i-ten Operation. Wenn $c_i = k \geq 1$ ist, dann haben $bin(i)$ bzw. $bin(i+1)$ die Form

$$\begin{aligned} i &: \underbrace{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots}_{\text{darin sind } x \text{ Einsen}} 0\,\underbrace{1\ldots\ldots1}_{k-1 \text{ Einsen}}\,, \\ i+1 &: \underbrace{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots}_{\text{darin sind ebenfalls } x \text{ Einsen}} 1\,\underbrace{0\ldots\ldots0}_{k-1 \text{ Nullen}}\,. \end{aligned}$$

Also ist für alle $i \in \mathbb{N}_0$

$$\begin{aligned} a(i) &= \underbrace{c_i}_{=k} + \underbrace{pot(i+1)}_{=x+1} - \underbrace{pot(i)}_{=x+k-1} \\ &= k + x + 1 - (x + k - 1) \\ &= 2\,. \end{aligned}$$

Damit ist

$$\begin{aligned} C_n &= \sum_{i=0}^{n-1} c_i \\ &= \sum_{i=0}^{n-1} (a_i + pot(i) - pot(i+1)) \\ &= \sum_{i=0}^{n-1} a_i + pot(0) - pot(n) \end{aligned}$$

$$\leq \quad 2n\,,$$

da $pot(0) = 0$ und $pot(n) \geq 0$ ist. □

Beweis: [Version 4 : Elegantes Potentialargument]
Den zweiten bzw. dritten Beweis können wir eleganter auch so formulieren :

$$C_n = \#\text{Operationen} + \#(\text{gelöschte Einsen}) \leq n + \#(\text{kreiierte Einsen}) = 2n\,.$$

□

1.5 Das Master Theorem für Rekursionsgleichungen

Bei der Laufzeitanalyse von Algorithmen spielen häufig Rekursionsgleichungen der folgenden Form eine Rolle :

$$T(n) = \begin{cases} 2 \cdot T(\frac{n}{2}) + c \cdot n & \text{, falls } n > 1\,, \\ c & \text{, falls } n = 1\,. \end{cases}$$

wobei $c > 0$ eine Konstante ist. Es ist nicht allzu schwierig, diese in eine geschlossene Form zu bringen.

Lemma 1.5.1 *Für $n = 2^k$ ist*

$$T(n) = c \cdot 2^k \cdot (k+1) = c \cdot n \cdot (1 + \log n)\,.$$

Beweis: Wir zeigen die Behauptung durch vollständige Induktion.

- $\boldsymbol{k = 0}$ **:** $\surd$
- $\boldsymbol{k > 0}$ **:**

$$\begin{aligned} T(2^k) &= 2 \cdot T(2^{k-1}) + c \cdot 2^k \\ &= 2 \cdot c \cdot 2^{k-1} \cdot k + c \cdot 2^k \\ &= c \cdot 2^k \cdot (k+1)\,. \end{aligned}$$

□

Allgemein wird eine Rekursionsgleichung allerdings etwas komplizierter aussehen :

$$T_f(n) \leq \begin{cases} D \cdot f(n) & \text{, falls } n \leq N\,, \\ f(n) + \sum_i T_f(n_i) & \text{, falls } n > N\,, \end{cases}$$

wobei $D \geq 0$ und $N \in \mathbb{N}$ Konstanten sind und es ferner zu jedem $n > N$ eine Folge $n_1, n_2, n_3, \ldots$ von natürlichen Zahlen gibt (die Größe der rekursiven Teilprobleme).

Unser Ziel ist es, eine asymptotische Abschätzung für T_f zu finden. Dazu werden wir uns von einem einfachen Spezialfall aus zu der obigen allgemeinen Form durchkämpfen.

Lemma 1.5.2 *Wenn die beiden folgenden Bedingungen erfüllt sind*

(1) $\exists b > 1\, \forall n > N\, \forall i : n_i \leq \frac{n}{b}$ *(echte Rekursion)*

(2) $\exists c > 0\, \forall n > N : \sum_i f(n_i) \leq c \cdot f(n)$ *(f ist "konvex")*

dann ist

$$T_f(n) = O\left(f(n) \cdot \sum_{j=0}^{\log n} c^j\right) .$$

Beweis: Sei $D' = \max(1, D)$. Dann genügt es zu zeigen, daß

$$T_f(n) \leq D' \cdot f(n) \cdot \sum_{j=0}^{\log_b n} c^j$$

ist. Für $1 \leq n \leq N$ ist

$$T_f(n) \leq D \cdot f(n) \leq D \cdot f(n) \cdot \sum_{j=0}^{\log_b n} c^j \leq D' \cdot f(n) \cdot \sum_{j=0}^{\log_b n} c^j .$$

Für $n > N$ ist

$$\begin{aligned}
T_f(n) &\leq f(n) + \sum_i T_f(n_i) \\
&\overset{(IV)}{\leq} f(n) + \sum_i \left(D' \cdot f(n_i) \cdot \sum_{j=0}^{\log_b n_i} c^j\right) \\
&\overset{(1)}{\leq} f(n) + D' \cdot \sum_i \left(f(n_i) \cdot \sum_{j=0}^{\log_b n - 1} c^j\right) \\
&\overset{(2)}{\leq} D' \cdot \left(f(n) + c \cdot f(n) \cdot \sum_{j=0}^{\log_b n - 1} c^j\right) \\
&= D' \cdot f(n) \cdot \sum_{j=0}^{\log_b n} c^j .
\end{aligned}$$

□

Bei der Anwendung dieses Lemmas können wir uns gewisse kleine "Ungenauigkeiten" erlauben. So ist es zum Beispiel möglich, in $T(n) = 2 \cdot T(\lceil \frac{n}{2} \rceil) + cn$ auch $b = 2$ zu wählen, obwohl $\lceil \frac{n}{2} \rceil > \frac{n}{2}$ ist für ungerade n.

Lemma 1.5.3 *Wir können (1) in Lemma 1.5.2 abschwächen zu*

(1′) $\exists b > 1\ \exists d \geq 0\ \forall n > N\ \forall i : n_i \leq \min(\frac{n}{b} + d, n-1)$.

Beweis: Seien b, d, n fest. Sei ferner

$$\begin{aligned} L_{b,d}(n) = \min\{k\ | &\text{Für alle Folgen } N_0, N_1, \ldots \text{ reeller Zahlen mit } N_0 = n \\ &\text{und } N_{k+1} \leq \min(\tfrac{N_k}{b} + d, N_k - 1) \text{ ist } N_k \leq 1\}\,. \end{aligned}$$

Offensichtlich ist $L_{b,d}(n) \leq n - 1$. Aber wir werden gleich eine deutlich bessere Schranke zeigen, nämlich daß $L_{b,d}(n) \approx \log_b n$ ist.
Da für $n_i \leq \min(\frac{n}{b} + d, n-1)$ gilt $L_{b,d}(n_i) \leq L_{b,d}(n) - 1$, können wir mit demselben Beweis wie in Lemma 1.5.2 zeigen, daß

$$T_f(n) = O\left(f(n) \cdot \sum_{j=0}^{L_{b,d}(n)} c^j\right)$$

ist. Wenn wir zeigen, daß es eine Konstante A gibt, so daß für alle $n \geq 1$ gilt $L_{b,d}(n) \leq A + \log_b n$, dann folgt

$$\sum_{j=0}^{L_{b,d}(n)} c^j \leq \sum_{j=0}^{A+\log_b n} c^j \leq \sum_{j=0}^{A} c^j \cdot \sum_{j=0}^{\log_b n} c^j = O\left(\sum_{j=0}^{\log_b n} c^j\right)\,.$$

Damit ist das Lemma gezeigt.

Es bleibt zu zeigen, daß $L_{b,d}(n) \leq A + \log_b n$ ist. Wir zeigen, daß

$$A = \frac{d+2}{1 - \frac{1}{b}}$$

ausreicht. Die längste mögliche Folge $N_0, N_1, \ldots$ in der Definition von $L_{b,d}(n)$ ist

$$\begin{aligned} N_0 &= n\,, \\ N_1 &= \frac{N_0}{b} + d\,, \\ &\vdots \\ N_k &= \frac{N_{k-1}}{b} + d\,, \\ N_{k+1} &= N_k - 1\,, \\ N_{k+2} &= N_{k+1} - 1\,, \\ &\vdots \end{aligned}$$

wobei k minimal ist mit $N_k - 1 \leq \frac{N_k}{b} + d$. Dies ist äquivalent zu

$$N_k \leq \frac{d+1}{1 - \frac{1}{b}}\,. \qquad (\star)$$

Wir müssen jetzt noch k abschätzen. Man sieht leicht durch Induktion :

$$\begin{aligned} N_i &= \frac{N_{i-1}}{b} + d \\ &= \frac{n}{b^i} + d \cdot \sum_{j=0}^{i-1} \frac{1}{b^j} \\ &= \frac{n}{b^i} + d \cdot \frac{1 - \frac{1}{b^i}}{1 - \frac{1}{b}} \\ &\leq \frac{n}{b^i} + \frac{d}{1 - \frac{1}{b}} \,. \end{aligned}$$

Insbesondere ist für $k = \lceil \log_b n \rceil$

$$N_k \leq \underbrace{\left(\frac{1}{b}\right)^{\lceil \log_b n \rceil} \cdot n}_{\leq 1} + \frac{d}{1 - \frac{1}{b}} \leq \frac{d+1}{1 - \frac{1}{b}} \,.$$

Wegen $(\star)$ ist also $k \leq \lceil \log_b n \rceil$.

Insgesamt folgt :

$$L_{b,d}(n) \leq k + N_k \leq \lceil \log_b n \rceil + \frac{d+1}{1 - \frac{1}{b}} \leq \log_b n + \frac{d+2}{1 - \frac{1}{b}} \,.$$

□

Wir können aber auch Bedingung (2) "ungenau" handhaben. Mit $f(n) = n^2$ und $n_1 = \cdots = n_4 = \lceil \frac{n}{2} \rceil$ etwa können wir auch $c = 1$ wählen, obwohl $\sum_i f(n_i) = 4 \cdot \left(\lceil \frac{n}{2} \rceil\right)^2 = 4 \cdot (\frac{n+1}{2})^2 > 1 \cdot n^2$ ist für ungerade n.

Lemma 1.5.4 *Wir können (2) in Lemma 1.5.3 abschwächen zu*

(2′) $\exists c > 0 \, \forall n > N : \sum_i f(n_i) \leq c \cdot f(n) + g(n)$,

wobei g eine Funktion ist mit

- $g = o(f)$ *(g ist "klein")*
- $\exists' > c \, \forall n > N : \sum_i g(n_i) \geq c' \cdot g(n)$ *(g ist "konkav")*

Beweis: Sei

$$h(n) = 2f(n) - \frac{2}{c' - c} \cdot g(n) \,.$$

Dann ist $h = \Theta(f)$ (weil $g = o(f)$ ist). Ferner ist

$$
\begin{aligned}
\sum_i h(n_i) &= \sum_i \left(2f(n_i) - \frac{2}{c'-c}\cdot g(n_i)\right) \\
&= \sum_i 2f(n_i) - \sum_i \left(\frac{2}{c'-c}\cdot g(n_i)\right) \\
&\overset{(2')}{\leq} 2c\cdot f(n) + 2g(n) - \frac{2c'}{c'-c}\cdot g(n) \\
&= c\cdot h(n)\,.
\end{aligned}
$$

Damit erfüllt h Bedingung (2) in Lemma 1.5.3. Also ist

$$
\begin{aligned}
T_h(n) &= O\left(h(n)\cdot \sum_{j=0}^{\log_b n} c^j\right) \\
&= O\left(f(n)\cdot \sum_{j=0}^{\log_b n} c^j\right)\,.
\end{aligned}
$$

Wenn wir noch zeigen können, daß aus $f = O(h)$ schon $T_f(n) = O(T_h(n))$ folgt, dann sind wir fertig. Um dies einzusehen, müssen wir uns zwei Dinge überlegen.

1. $T_f(n) = T_{f;N,D}(n)$ ist in der Tat unabhängig von N und D, weil es nämlich zu jedem N' ein D' gibt mit $T_{f;N,D} = \Theta(T_{f;N',D'})$.

   Wenn $N' > N$ ist, dann wählen wir $D' = \frac{f(n)+\sum_i T_{f;N,D}(n_i)}{f(n)}$. Dann gilt

$$
\begin{aligned}
T_{f;N',D'}(N') &\leq D'\cdot f(n) \\
&= f(n) + \sum_i T_{f;N,D}(n_i) \\
&= T_{f;N,D}(N')\,.
\end{aligned}
$$

   Für $n < N'$ ist

$$T_{f;N',D'}(n) \leq a\cdot T_{f;N,D}(n)$$

   für ein $a > 0$. Insgesamt ist also

$$T_{f;N',D'}(n) = O(T_{f;N,D}(n))\,.$$

2.

$$
\begin{aligned}
T_{af}(n) &\leq \begin{cases} D\cdot af(n) & \text{, falls } n \leq N\,, \\ af(n) + \sum_i T_{af}(n_i) & \text{, falls } n > N\,. \end{cases} \\
&= a\cdot T_f(n)\,.
\end{aligned}
$$

□

Korollar 1.5.5 *Unter den Voraussetzungen von Lemma 1.5.4 gilt :*

$$T_f(n) = \begin{cases} O(f(n)) & \text{, falls } c < 1 \text{ ,} \\ O(f(n) \cdot \log n) & \text{, falls } c = 1 \text{ ,} \\ O(f(n) \cdot n^{\log_b c}) & \text{, falls } c > 1 \text{ .} \end{cases}$$

Beweis:

$$\begin{aligned} \sum_{j=0}^{\log_b n} c^j &= \begin{cases} \frac{1}{1-c} & \text{, falls } c < 1 \text{ ,} \\ O(\log n) & \text{, falls } c = 1 \text{ .} \\ \frac{c^{\log_b n+1} - 1}{c-1} & \text{, falls } c > 1 \text{ .} \end{cases} \\ &= \begin{cases} O(1) & \text{, falls } c < 1 \text{ ,} \\ O(\log n) & \text{, falls } c = 1 \text{ ,} \\ O(n^{\log_b c}) & \text{, falls } c > 1 \text{ .} \end{cases} \end{aligned}$$

da $\dfrac{c^{\log_b n+1} - 1}{c-1} = O(c^{\log_b n}) = O(b^{\log_b c \cdot \log_b n}) = O(n^{\log_b c})$ ist. □

Daraus folgt nun unser Hauptsatz, das sogenannte *Master Theorem für Rekursionsgleichungen.*

Satz 1.5.6 (Master Theorem)
Sei $a \in \mathbb{N}$, $b > 1$, $k > 0$, $D > 0$, $N \geq 1$ *und*

$$T(n) \leq \begin{cases} D \cdot n^k & \text{, falls } n \leq N \text{ ,} \\ a \cdot T(\lceil \frac{n}{b} \rceil) + n^k & \text{, falls } n > N \text{ .} \end{cases}$$

Sei $\lambda = \log_b a$. *Dann ist*

$$T(n) = \begin{cases} O(n^k) & \text{, falls } \lambda < k \text{ ,} \\ O(n^k \cdot \log n) & \text{, falls } \lambda = k \text{ ,} \\ O(n^\lambda) & \text{, falls } \lambda > k \text{ .} \end{cases}$$

Beweis: Für $n > N$ ist $n_i = \lceil \frac{n}{b} \rceil \leq \frac{n}{b} + 1$ für $1 \leq i \leq a$. Wegen

$$\frac{n}{b} + 1 \leq n - 1 \Longleftrightarrow n \geq \frac{2b}{b-1}$$

gilt (1′), falls $n \geq 2 + \frac{2}{b-1}$ ist.

Es ist

$$\begin{aligned} \sum_i f(n_i) &= a \cdot \left(\left\lceil \frac{n}{b} \right\rceil \right)^k \\ &\leq a \cdot \left(\frac{n}{b} + 1 \right)^k \\ &= \frac{a}{b^k} \cdot n^k + O(n^{k-1}) \\ &= c \cdot f(n) + O(n^{k-1}) \end{aligned}$$

mit

$$c = \frac{a}{b^k} .$$

Wenn wir

$$g(n) = B \cdot n^{k-1}$$

für ein geeignetes $B > 0$ wählen, dann ist $g = o(f)$ und $\sum_i f(n_i) \leq c \cdot f(n) + g(n)$. Ferner ist

$$\begin{aligned} \sum_i g(n_i) &= a \cdot B \cdot \left(\left\lceil \frac{n}{b} \right\rceil \right)^{k-1} \\ &\geq a \cdot B \cdot \left(\frac{n}{b} \right)^{k-1} \\ &= B \cdot \frac{a}{b^{k-1}} \cdot n^{k-1} \\ &= c' \cdot g(n) \end{aligned}$$

mit

$$c' = \frac{a}{b^{k-1}} > c .$$

Also gilt auch $(2')$.

Wegen $c = \frac{a}{b^k}$ ist $\log_b a > k$ gdw. $c > 1$. Also ist nach Korollar 1.5.5

$$T(n) = \begin{cases} O(n^k) & \text{, falls } \lambda < k \text{ ,} \\ O(n^k \cdot \log n) & \text{, falls } \lambda = k \text{ ,} \\ O(n^\lambda) & \text{, falls } \lambda > k \text{ ,} \end{cases}$$

da $n^k \cdot n^{\log_b c} = n^k \cdot n^{\log_b a - k} = n^\lambda$ ist. □

Beispiel 1.5.7 Wir wenden das Mater Theorem an auf unser erstes einfaches Beispiel

$$T(n) \leq \begin{cases} 2 \cdot T(\lceil \frac{n}{2} \rceil) + c \cdot n & \text{, falls } n > 1 \\ c & \text{, falls } n = 1 \text{ .} \end{cases}$$

Diese Rekursionsformel "normieren" wir erst einmal, indem wir durch c dividieren :

$$T'(n) \leq \begin{cases} 2 \cdot T'(\lceil \frac{n}{2} \rceil) + n & \text{, falls } n > 1 \\ 1 & \text{, falls } n = 1 \text{ .} \end{cases}$$

Dann ist in der Notation von Satz 1.5.6 $k = 1$, $D = 1$, $b = 2$ und $a = 2$. Wegen $\lambda = \log_b a = 1 = k$ ist also $T'(n) = O(n \log n)$ (mittlerer Fall des Master Theorem). □

1.6 Multiplizieren beliebig langer Zahlen

Wir betrachten hier die Multiplikation zweier n-stelliger Binärzahlen a und b,

$$a = \sum_{i=0}^{n-1} a_i 2^i, \qquad b = \sum_{i=0}^{n-1} b_i 2^i \,,$$

und suchen

$$c = a \cdot b = \sum_{i=0}^{2n-1} c_i 2^i \,.$$

Grundoperationen seien Operationen des "Kleinen Einmaleins". Die Multiplikation nach der Schulmethode braucht mindestens n^2 Grundoperationen.

Das folgende effizientere Verfahren stammt von Karazuba und Offman (1962). Wir teilen a in

$$A_0 = \sum_{i=0}^{k-1} a_i 2^i \quad \text{und} \quad A_1 = \sum_{i=k}^{n-1} a_i 2^{i-k},$$

wobei $k = \lfloor \frac{n}{2} \rfloor$ ist, und verfahren mit b und B_0, B_1 analog. Es ist also

$$\begin{aligned} a &= A_1 \cdot 2^k + A_0 \quad \text{und} \\ b &= B_1 \cdot 2^k + B_0 \end{aligned}$$

und daher

$$\begin{aligned} a \cdot b &= (A_1 \cdot 2^k + A_0)(B_1 \cdot 2^k + B_0) \\ &= A_1 B_1 2^{2k} + (A_0 B_1 + A_1 B_0) \cdot 2^k + A_0 B_0 \\ &= A_1 B_1 2^{2k} + ((A_0 + A_1) \cdot (B_0 + B_1) - A_1 B_1 - A_0 B_0) \cdot 2^k + A_0 B_0 \,. \end{aligned}$$

An dieser Stelle führen wir nur noch drei Multiplikationen von Zahlen der Länge $\frac{n}{2}$ Bits aus statt vier Multiplikationen von Zahlen der Länge n Bits.

1.6.1 Der Algorithmus von Karazuba-Offman

Sei

$$\begin{aligned} P_1 &= A_1 \cdot B_1 \,, \\ P_2 &= A_0 \cdot B_0 \,, \\ S_1 &= A_0 + A_1 \,, \\ S_2 &= B_0 + B_1 \,, \\ P_3 &= S_1 \cdot S_2 \,, \\ c &= P_1 \cdot 2^{2k} + (P_3 - P_1 - P_2) \cdot 2^k + P_2 \,. \end{aligned}$$

Dann ist $c = a \cdot b$. Dies führt zu folgendem rekursiven Algorithmus.

Mult(a, b)
begin
  if Stellenzahl < 10
  then Schulmethode;
  else Karazuba-Offman;
  fi
end

Sei $M(n)$ die Anzahl der Grundoperationen bei n-stelliger Binärmultiplikation und $A(n)$ die Anzahl der Grundoperationen bei n-stelliger Binäraddition.

Lemma 1.6.1 $A(n) = n$.

Beweis: Schuladdition. □

Lemma 1.6.2 *$M(n)$ erfüllt die folgenden Ungleichungen :*

$$M(n) \le \begin{cases} c & \textit{für} \quad n \le 10\,, \\ an + 3M(\lfloor \frac{n}{2} \rfloor + 2) & \textit{für} \quad n > 10\,. \end{cases}$$

Dabei sind a und c Konstanten.

Beweis: Sei c die maximale Anzahl von Elementaroperationen bei der Multiplikation von Zahlen der Länge höchstens 10 Bit.
In jedem Fall ist $a \le 12$, da wir im rekursiven Schritt 6 Additionen brauchen und jede Addition auf Zahlen mit höchstens $2n$ Stellen arbeitet.
Ferner gibt es 3 Multiplikationen, deren Argumente höchstens $\lfloor \frac{n}{2} \rfloor + 2$ Stellen haben. □

Satz 1.6.3 *Es ist $T(n) = O(n^{\log 3})$.*

Beweis: Der dritte Fall des Master Theorem (Satz 1.5.6) mit $k = 1$, $N = 10$, $a = 3$ und $b = 2$ liefert $T(n) = O(n^{\log 3})$, da $k = 1 < \log_b a = \log 3 \approx 1.59$ ist. □

Frage: Warum gerade 10 ?
Oder : Ab welchem n lohnt sich ein rekursiver Schritt ?

Es ist

$$M_{\text{Schulmethode}}(n) = \beta n^2.$$

Also haben wir die folgenden rekursiven Kosten :

$$\text{Nur Schulmethode} \quad : \quad \beta n^2\,. \tag{1.1}$$

$$\text{Ein rekursiver Schritt, dann Schulmethode} \quad : \quad \alpha n + 3\beta \left(\frac{n}{2}\right)^2\,. \tag{1.2}$$

Ein einzelner rekursiver Schritt ist daher der Schulmethode überlegen, wenn

$$\alpha n + 3\beta\frac{n^2}{4} \leq \beta n^2 \iff \alpha n \leq \frac{1}{4}\beta n^2 \tag{1.3}$$

$$\iff n \geq 4\frac{\alpha}{\beta} \tag{1.4}$$

ist. Mit $\alpha \leq 6$ und $\beta \geq 2$ lohnt sich ein rekursiver Schritt also für $n \geq 4 \cdot \frac{6}{2} = 12$.

Bem.: Man kann sogar in Zeit $O(n \log n \log\log n)$ multiplizieren (Schönhage, Strassen : "Schnelle Multiplikation großer Zahlen", Computing 7, 1971, 281–292). $\square$

Kapitel 2

Sortieren

2.1 Splitsort

Sei S eine Menge von n paarweise verschiedenen Zahlen.
$\texttt{Select}(S, k)$ liefert die k-te Zahl aus der in aufsteigender Ordnung sortierten Menge S mit $1 \leq k \leq n$. Die entsprechend der Sortierung mittlere Zahl, also $k = \lceil \frac{n}{2} \rceil$, bezeichnet man als den *Median* von S.

Splitsort(S)

begin

$x \leftarrow \texttt{Select}(S, \lceil \frac{n}{2} \rceil)$;

$S_< \leftarrow \{y \in S \mid y < x\}$;

$S_> \leftarrow \{y \in S \mid y > x\}$;

return $\texttt{Splitsort}(S_<), x, \texttt{Splitsort}(S_>)$;

end

Lemma 2.1.1 *Falls* `Select` *bei Eingabegröße n höchstens d·n Vergleiche braucht (d eine Konstante), dann braucht* `Splitsort` *höchstens* $(d+1) \cdot n \cdot (1 + \log n)$ *Vergleiche.*

Beweis: Lemma 1.5.1 mit $c = d + 1$. □

Sei α eine feste Konstante (unabhängig von S), $\frac{1}{2} \leq \alpha < 1$. Die Funktion $\texttt{Goodsplitter}_\alpha(S)$ liefert ein $x \in S$ mit

$$\begin{aligned} |\{y \in S \mid y < x\}| &\leq \alpha \cdot n\,, \\ |\{y \in S \mid y > x\}| &\leq \alpha \cdot n\,. \end{aligned}$$

Oder äquivalent :

$$\begin{aligned} |\{y \in S \mid y < x\}| &\geq (1-\alpha) \cdot n\,, \\ |\{y \in S \mid y > x\}| &\geq (1-\alpha) \cdot n\,. \end{aligned}$$

`Goodsplitter`$_\alpha$ liefert also in folgendem Bild immer irgendein Element aus dem mittleren Intervall :

S (sortiert)

$(1-\alpha)\cdot n$ Elemente | | $(1-\alpha)\cdot n$ Elemente

2.1.1 Implementierung von `Select` durch `Goodsplitter`

Select(S, k)

begin

$x \leftarrow$ `Goodsplitter`$_\alpha(S)$;

$S_< \leftarrow \{y \in S \mid y < x\}$;

$S_> = \{y \in S \mid y > x\}$;

cases

$|S_<| \geq k$: **return** `Select`$(S_<, k)$;
$|S_<| = k-1$: **return** x;
$|S_<| < k-1$: **return** `Select`$(S_>, k-(|S_<|+1))$;

end

Lemma 2.1.2 *Falls* `Goodsplitter`$_\alpha$ *bei Eingabegröße* n *höchstens* $e\cdot n$ *Vergleiche braucht (e eine Konstante), dann braucht* `Select` *höchstens* $\frac{e+1}{1-\alpha}\cdot n$ *Vergleiche.*

Beweis: Sei $S(n)$ die Anzahl der Vergleiche von `Select` bei Eingabegröße n. Dann gilt:

$$S(n) \leq \underbrace{e\cdot n}_{\text{Vergleiche von Goodsplitter}_\alpha} + \underbrace{n}_{\textbf{cases}\text{-Vergleiche}} + \underbrace{S(\alpha\cdot n)}_{\text{rekursiver Aufruf}},$$

da `Goodsplitter`$_\alpha$ garantiert, daß $S_<$ und $S_>$ jeweils höchstens $\alpha\cdot n$ Elemente haben. Also ist

$$\begin{aligned} S(n) &\leq (e+1)\cdot n + S(\alpha\cdot n) \\ &\leq (e+1)\cdot n + (e+1)\cdot\alpha\cdot n + S(\alpha^2\cdot n) \\ &\leq (e+1)\cdot(n+\alpha n+\alpha^2 n+\alpha^3 n+\cdots) \\ &\leq \frac{e+1}{1-\alpha}\cdot n\,. \end{aligned}$$

□

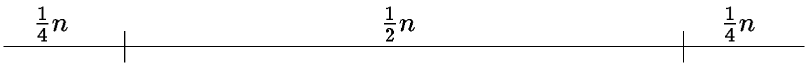


Abbildung 2.1: Jedes Element im mittleren Intervall ist ein $\frac{3}{4}$-`Goodsplitter`.

2.1.2 `Goodsplitter`, randomisiert

Goodsplitter$_\alpha(S)$

begin

  repeat

    Wähle $x \in S$ zufällig;

  until x ist ein α-`Goodsplitter`;

end

Lemma 2.1.3 *Der randomisierte* `Goodsplitter`$_\alpha$ *braucht im Mittel* $\frac{1}{2\alpha-1}$ *Versuche, um ein gutes Splitelement zu finden.*

Beweis: Sei $\beta = 2(1-\alpha)$. Sei I die Zufallsvariable, die die Anzahl der Münzwürfe zählt. Dann ist für $i \geq 0$

$$\begin{aligned} p(I=i) &= [2\cdot(1-\alpha)]^{i-1}\cdot(1-2\cdot(1-\alpha)) \\ &= \beta^{i-1}\cdot(1-\beta) \end{aligned}$$

und

$$\begin{aligned} E[I] &= \sum_{i\geq 1}(i\cdot p(I=i)) \\ &= \sum_{i\geq 1}(i\cdot\beta^{i-1}\cdot(1-\beta)) \\ &= (1-\beta)\cdot\sum_{i\geq 1}\frac{d}{d\beta}\,\beta^i \\ &= (1-\beta)\cdot\frac{d}{d\beta}\,\sum_{i\geq 1}\beta^i \\ &= (1-\beta)\cdot\frac{d}{d\beta}\,\frac{\beta}{1-\beta} \\ &= (1-\beta)\cdot\frac{(1-\beta)\cdot 1+\beta}{(1-\beta)^2} \\ &= (1-\beta)\cdot\frac{1}{(1-\beta)^2} \\ &= \frac{1}{1-\beta}\,. \end{aligned}$$

Wir hätten auch den Differenzenkalkül verwenden können :

$$\begin{aligned}
\sum_{i\geq 1}(i\cdot\beta^i) &= \sum_{x\geq 1}(\underbrace{x}_{f}\cdot\underbrace{\beta^x}_{\substack{\Delta g\\ \Rightarrow g=\frac{\beta^x}{\beta-1}}})\delta x\\
&= (x\cdot\frac{\beta^x}{\beta-1})\Big|_0^\infty - \frac{1}{\beta-1}\cdot\sum_{x\geq 0}\beta^{x+1}\delta x\\
&= \left(x\cdot\frac{\beta^x}{\beta-1}-\frac{1}{\beta-1}\cdot\beta\cdot\frac{\beta^x}{\beta-1}\right)\Big|_0^\infty\\
&= \frac{\beta}{(1-\beta)^2}\,.
\end{aligned}$$

□

Korollar 2.1.4 *Die randomisierte Version von* `Goodsplitter`$_\alpha$ *braucht eine erwartete Zahl von höchstens* $\frac{1}{2\alpha-1}\cdot n$ *Vergleichen.*

Beweis: Jede Iteration braucht n Vergleiche. Und es gibt eine erwartete Anzahl von $\frac{1}{1-\beta}=\frac{1}{2\alpha-1}$ Iterationen. □

Wenn wir nun alle vorangegangenen Lemmas zusammensetzen, erhalten wir den folgenden Satz.

Satz 2.1.5 *Splitsort mit dem randomisierten* `Goodsplitter`$_\alpha$ *braucht im Mittel höchstens* $\frac{5\alpha-2\alpha^2-1}{(1-\alpha)(2\alpha-1)}\cdot n\cdot(1+\log n)$ *Vergleiche.* □

2.1.3 `Goodsplitter`, deterministisch

Der folgende deterministische `Goodsplitter`-Algorithmus ist das Kernstück des Linearzeitalgorithmus zur Bestimmung des Medians von Blum, Floyd, Pratt, Rivest und Tarjan (1972).

Goodsplitter(S)

begin

Teile S in 5-elementige Teilmengen auf :

$$S=\bigcup_{i=1}^{\frac{n}{5}} S_i\,,$$

wobei die $S_i=\{s_i^{(1)},\ldots,s_i^{(5)}\}$ paarweise disjunkt sind;

Bestimme für jedes S_i den Median m_i;

Bestimme den Median M der Mediane m_i durch rekursiven Aufruf von `Select`;

end

Lemma 2.1.6 *M ist ein α-*`Goodsplitter` *mit* $\alpha=\frac{7}{10}$.

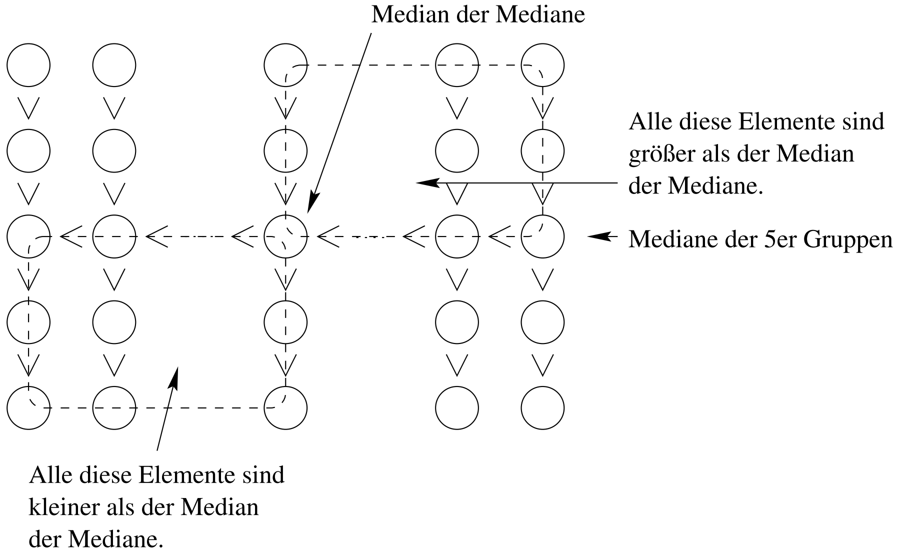


Abbildung 2.2: Der Median der Mediane.

Beweis: Sei o.B.d.A. $m_1 < m_2 < \cdots < m_{\frac{n}{5}}$ (durch Umnumerierung zu erreichen). Weiterhin sei $s_i^{(1)} > \cdots > s_i^{(5)}$ für $i \in \{1, \ldots, \frac{n}{5}\}$. Dann ist $m_i = s_i^{(3)}$ und $M = s_{\frac{n}{10}}^{(3)}$.
Wegen $s_i^{(j)} \leq M$ für $i \leq \frac{n}{10}$ und $j \in \{3, 4, 5\}$ ist dann

$$|\{s \in S \mid s \leq M\}| \geq \frac{3n}{10} = (1 - \alpha)n$$

und genauso

$$|\{s \in S \mid s \geq M\}| \geq \frac{3n}{10} = (1 - \alpha)n\,.$$

Dieses Argument ist in Abbildung 2.2 graphisch dargestellt. □

Sei $S(n)$ die Anzahl der Vergleiche von `Select` und $G(n)$ die Anzahl der Vergleiche von `Goodsplitter` bei Eingabegröße n. Dann ist

$$\begin{aligned} G(n) &\leq d \cdot \frac{n}{5} + S\left(\frac{n}{5}\right) \quad \text{und} \\ S(n) &\leq G(n) + n + S\left(\frac{7n}{10}\right)\,, \end{aligned}$$

wobei d die Anzahl der Vergleiche ist, um den Median von 5 Elementen zu bestimmen (in den Übungen werden wir sehen, daß $d = 6$ ist).

Lemma 2.1.7 *Es ist $S(n) \leq (10 + 2d) \cdot n$.*

Beweis: Durch Induktion über n :

- **$n \le 5$:** Es ist $s(n) \le d \le (10+2d)n$.
- **$n > 5$:** Es ist

$$
\begin{aligned}
S(n) &\le G(n) + n + S\left(\frac{7n}{10}\right) \\
&\le d \cdot \frac{n}{5} + S\left(\frac{n}{5}\right) + n + S\left(\frac{7n}{10}\right) \\
&= (1+\frac{d}{5}) \cdot n + S\left(\frac{n}{5}\right) + S\left(\frac{7n}{10}\right) \\
&\overset{(IV)}{\le} (1+\frac{d}{5}) \cdot n + (2+\frac{2}{5}d) \cdot n + (7+\frac{7}{5}d) \cdot n \\
&= 10n + 2dn\,.
\end{aligned}
$$

□

Insgesamt haben wir damit folgenden Satz.

Satz 2.1.8 *Splitsort mit dem deterministischen* `Goodsplitter` *braucht höchstens* $(11+2d) \cdot n \cdot (1+\log n)$ *Vergleiche.* □

Wenn wir dies mit der Laufzeitzeit des randomisierten $\texttt{Goodsplitter}_{\frac{7}{10}}$ vergleichen, so ergibt sich für diesen für $\alpha = \frac{7}{10}$ eine erwartete Laufzeit von ungefähr $13n(1+\log n)$ (Satz 2.1.5). Also bringt hier Randomisierung keine Verbesserung gegenüber dem deterministischen Algorithmus.

Wir sollten noch klären, warum wir gerade Gruppen von jeweils 5 Elementen gebildet haben. Wir betrachten dazu eine modifizierte Version des Algorithmus mit 3er-Gruppen. Dann erhalten wir für die Anzahl der Vergleiche die folgenden Rekursionsgleichungen :

$$
\begin{aligned}
G(n) &= f \cdot \frac{n}{3} + S\left(\frac{n}{3}\right) \quad \text{und} \\
S(n) &= G(n) + n + S\left(\frac{2n}{3}\right)\,,
\end{aligned}
$$

wobei f die Anzahl der Vergleiche zur Bestimmung des Medians von drei Elementen sei (offensichtlich ist $f = 3$).

Lemma 2.1.9 *Es ist* $S(n) \ge g \cdot n \cdot \log n$ *für eine Konstante* g.

Beweis: Durch vollständige Induktion über n. Wir betrachten nur den Induktionsschritt :

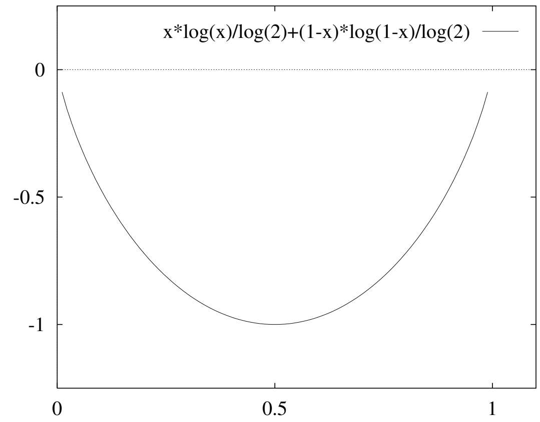


Abbildung 2.3: Graph der Funktion $x \log x + (1-x) \log (1-x)$.

$$\begin{aligned}
S(n) &= f \cdot \frac{n}{3} + S\left(\frac{n}{3}\right) + n + S\left(\frac{2n}{3}\right) \\
&\overset{(IV)}{\geq} (1 + \frac{f}{3}) \cdot n + g \cdot \frac{n}{3} \cdot \log \frac{n}{3} + g \cdot \frac{2n}{3} \cdot \log \frac{2n}{3} \\
&= g \cdot n \cdot \log n + n \cdot \left(1 + \frac{f}{3} + g \cdot \left(\underbrace{\frac{1}{3} \cdot \log \frac{1}{3} + \frac{2}{3} \cdot \log \frac{2}{3}}_{=:C \approx -0.92}\right)\right) \\
&= g \cdot n \cdot \log n + n \cdot \left(1 + \frac{f}{3} + g \cdot C\right) \\
&\geq g \cdot n \cdot \log n \,,
\end{aligned}$$

falls $1 + \frac{f}{3} + f \cdot C \geq 0$ ist. Dies ist äquivalent zu $g \leq \frac{1+\frac{f}{3}}{-C}$. Wenn $f = 3$ ist, bedeutet dies $g \leq 2.17$.
Wir benutzen außerdem, daß die Funktion $x \log x + (1-x) \log (1-x)$ auf dem Intervall $(0,1)$ den Wertebereich $(0,-1)$ besitzt (siehe Abb. 2.3). $\square$

2.2 Quicksort

Dieser randomisierte Sortieralgorithmus stammt von Hoare (1962).

Quicksort(S)

begin

Wähle ein $x \in S$ zufällig;

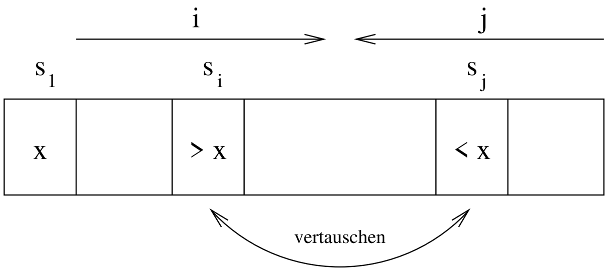
Teile S in $S_<$ und $S_>$;

return `Quicksort`($S_<$),x,`Quicksort`($S_>$);

end

Eine geschickte Implementierung :

```
Quicksort(S)
begin
    i ← rand(1, n);
    swap(S[1], S[i]);
    x ← S[1];
    i ← 2;
    j ← n;
    repeat
        while S[i] ≤ x und i < j
        do i + +; od
        while S[j] ≥ x und i < j)
        do j − −; od
        swap(S[i],S[j]);
    until i ≥ j;
end
```



Bem.: Bei unbalancierten Rekursionen sollte man zuerst die kleineren Teilprobleme bearbeiten; das hält die Tiefe des Kellers gering. □

Sei $T(n)$ die mittlere Anzahl von Vergleichen bei Eingabegröße n. Dann ist

$$\begin{aligned} T(0) &= 0\,, \\ T(1) &= 0 \quad \text{und} \\ T(n) &= \underbrace{\frac{1}{n}\cdot\sum_{i=1}^{n}}_{\text{Mittelwert}} \underbrace{((n-1)+T(i-1)+T(n-i))}_{\substack{\text{Aufwand, falls Teilungselement} \\ \text{Rang } i \text{ hat}}}\ . \end{aligned}$$

Diese Rekursiongleichung lösen wir wie folgt :

$$
\begin{array}{rrcl}
 & n \cdot T(n) & = & n \cdot (n-1) + 2 \cdot \sum_{i=1}^{n-1} T(i) \\
\Longrightarrow & (n-1) \cdot T(n-1) & = & (n-1) \cdot (n-2) + 2 \cdot \sum_{i=1}^{n-2} T(i) \\
\hline
\Longrightarrow & n \cdot T(n) - (n-1) \cdot T(n-1) & = & (n-1) \cdot 2 + 2 \cdot T(n-1) \\
 & \Longrightarrow \quad T(n) & = & \frac{2 \cdot (n-1)}{n} + \frac{n+1}{n} \cdot T(n-1)\,.
\end{array}
$$

Mit

$$
\begin{aligned}
a_0 &= 0\,, \\
a_n &= \frac{2\,(n-1)}{n} \quad \text{und} \\
b_n &= \frac{n+1}{n}
\end{aligned}
$$

für $n \geq 1$ haben wir

$$
\begin{aligned}
T(0) &= a_0 \quad \text{und} \\
T(n) &= a_n + b_n \cdot T(n-1) \\
&= a_n + b_n a_{n-1} + b_n b_{n-1} a_{n-2} + \ldots + b_n \ldots b_1 a_0\,.
\end{aligned}
$$

Satz 2.2.1 *Quicksort braucht im Mittel höchstens* $2(n+1)H_{n+1} \approx 2.89 \cdot (n+1) \cdot \log(n+1)$ *Vergleiche.*

Beweis:

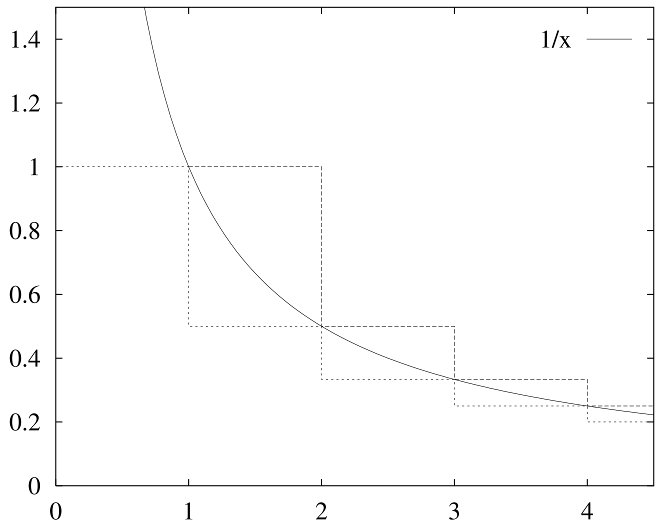
$$
\begin{aligned}
T(n) &= \sum_{k=1}^{n} (\underbrace{a_k}_{\frac{2(k-1)}{k}} \cdot \underbrace{b_{k+1} b_{k+2} \ldots b_n}_{\frac{k+2}{k+1}\frac{k+3}{k+2}\cdots\frac{n}{n-1}\frac{n+1}{n}}) \\
&= \sum_{k=1}^{n} \left(2 \cdot (n+1) \cdot \frac{k-1}{k(k+1)}\right) \\
&= 2 \cdot (n+1) \cdot \left(\underbrace{\sum_{k=1}^{n+1} \frac{1}{k+1}}_{=H_{n+1}-1} - \underbrace{\sum_{k=1}^{n} \frac{1}{k(k+1)}}_{=1-\frac{1}{n+1}} \right) \\
&\leq 2 \cdot (n+1) \cdot H_{n+1}\,.
\end{aligned}
$$

□

Hierbei ist H_n die n-te harmonische Zahl (siehe auch Abschnitt A.2.4) und es gilt :

$$\int_2^{n+1} \frac{\mathrm{d}x}{x} \leq \sum_{k=2}^{n} \frac{1}{k} \leq \int_1^n \frac{\mathrm{d}x}{x} .$$

Also ist $H_n \approx \ln n$.



2.3 Informationstheoretische untere Schranke

Wie gut sind eigentlich die Algorithmen des vorigen Abschnitts ? Oder anders gefragt : Wie schnell kann man überhaupt sortieren ?
Wir haben bisher nur *vergleichsorientierte Sortierverfahren* kennengelernt. Das sind Algorithmen, die als einzige Operationen auf den Elementen Vergleiche erlauben (wir dürfen also nicht zwei Elemente addieren oder multiplizieren).

Satz 2.3.1 (Informationstheoretische untere Schranke)
Jedes vergleichsorientierte deterministische Sortierverfahren braucht im schlechtesten Fall mindestens $\log(n!)$ *Vergleiche, um n Elemente zu sortieren.*

Beweis: Deterministische vergleichsorientierte Sortierverfahren lassen sich als Bäume interpretieren :

- Knoten sind mit Vergleichen "$a_i ? a_j$" markiert.
- Jeder Knoten hat zwei Kinder, die den beiden Ergebnissen des Vergleichs entsprechen.
- Blätter sind mit Permutationen von $(1, \ldots, n)$ markiert.

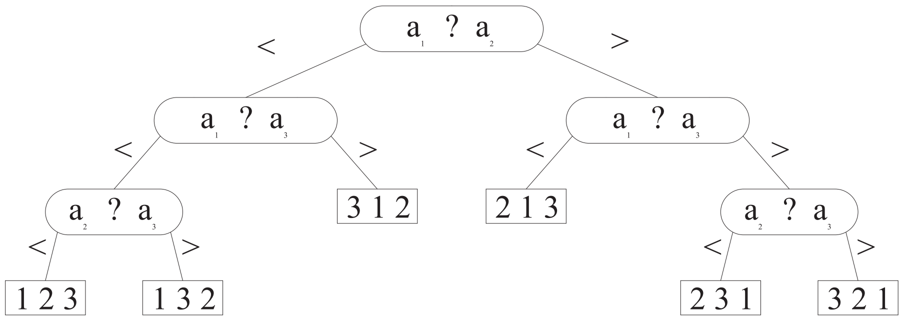


Abbildung 2.4: Sortier-Baum für drei Elemente a_1, a_2, a_3.

Für jede Permutation muß es mindestens ein Blatt geben, also insgesamt mindestens $n!$ Blätter. Ein binärer Baum der Höhe h hat aber höchstens 2^h Blätter. Also gilt für die Höhe h eines Sortierbaums $2^h \geq n!$ und damit $h \geq \log(n!)$. □

Bem.:

- Es ist $\log(n!) \approx \log(\frac{n}{e})^n = n \cdot \log \frac{n}{e} \approx n \log n - 1.44n$.
- Der Satz gilt auch mit $\log(n!) - 1$ und "im Mittel" statt "im schlechtesten Fall".
- Der Satz gilt auch für randomisierte Sortieralgorithmen. □

2.4 Sortieren ohne Vergleiche

Wenn wir mit den Elementen noch andere Operationen als nur Vergleiche machen dürfen, können wir manchmal die informationstheoretische untere Schranke durchbrechen. Allerdings brauchen wir dazu etwas mehr Struktur auf der Menge der zu sortierenden Elementea als nur eine lineare Ordnung (im folgenden etwa, daß wir ganze Zahlen sortieren, die nicht zu groß sind).

2.4.1 Countingsort

Gegeben : `array` $<$ `int` $>$ $A(0, n-1)$ mit $A[i] \in \{0 \ldots k-1\}$.

Aufgabe : Erzeuge sortierte Version von A in einem Feld B.

Countingsort(A, B)
begin
  array $<$ `int` $>$ $C(0, k-1)$;
  for (`int` $i = 0; i < k; i++$)
  do $C[i] \leftarrow 0$; **od**
  for (`int` $i = 0; i < n; i++$)
  do $C[A[i]]++$ **od**;

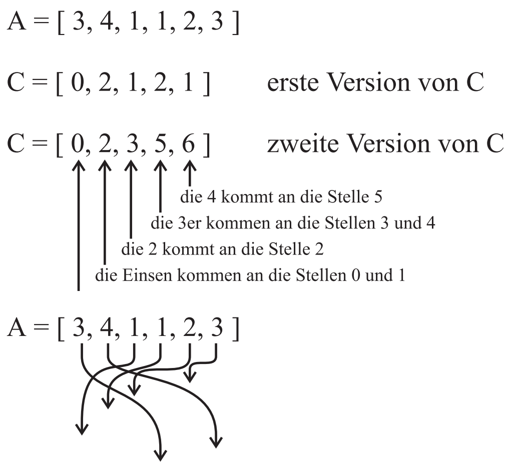


Abbildung 2.5: Ein Beispiel für `Countingsort`.

  // $C[j]$ ist die Anzahl der i mit $A[i] = j$

for (`int` $i = 1; i < k; i++$)

do $C[i] \leftarrow C[i] + C[i-1]$; **od**

  // $C[j]$ ist die Anzahl der i mit $A[i] \leq j$

for (`int` $i = n-1; i \geq 0; i--$)

do

  $B[C[A[i]]-1] \leftarrow A[i]$;

  $C[A[i]]--$;

od

end

Satz 2.4.1 *Die Laufzeit von* `Countingsort` *ist* $O(n+k)$. *Ferner ist* `Countingsort` stabil, *d.h. bei gleichen Schlüsseln ist die Reihenfolge in der Ausgabe die gleiche wie in der Eingabe.* □

2.4.2 Radixsort

Aufgabe : Sortiere n Zahlen im Bereich $\{0, \ldots, k^d - 1\}$.

Wir fassen die Zahlen als d-stellige Zahlen zur Basis k auf.

Radixsort

begin

for $i = 1$ **to** d

do

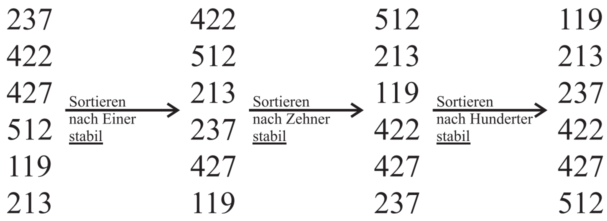


Abbildung 2.6: Ein Beispiel für `Radixsort`.

Sortiere nach i-ter Stelle (stabil);

od

end

Satz 2.4.2 *Die Laufzeit von* `Radixsort` *ist* $O(d(n+k))$.

Beweis: Wir sollten uns noch kurz von der Korrektheit von `Radixsort` überzeugen. Dazu machen wir Induktion nach d.

- $\boldsymbol{d = 1}$ **:** Einstellige Zahlen werden von *Countingsort* korrekt und stabil sortiert.
- $\boldsymbol{d > 1}$ **:** Wenn $(d-1)$-stellige Zahlen korrekt sortiert werden und wir stabil die d-te Stelle sortieren, werden insgesamt d-stellige Zahlen korrekt sortiert. □

Korollar 2.4.3 *Wir können n Zahlen aus* $\{0, \ldots, 2^n - 1\}$ *in Zeit* $O(\frac{n^2}{\log n})$ *sortieren.*

Beweis: Es ist $2^n = n^{\frac{n}{\log n}}$, also folgt die Behauptung, wenn wir `Radixsort` mit $k = n$ und $d = \frac{n}{\log n}$ benutzen. □

Man kann dies allerdings auch schneller in Zeit $O(n)$ machen (Kirkpatrick und Reisch, 1984).

Kapitel 3

Hashing

3.1 Dynamische Verwaltung von Mengen

Sei $U = \{0, \ldots, N-1\}$ unser Universum. Wir wollen eine Teilmenge $S \subseteq U$ unter den folgenden Operationen verwalten :

- `makeset`(S) : $S \leftarrow \emptyset$ Kreiert leere Menge S.
- `insert`(x, S) : $S \leftarrow S \cup \{x\}$ Fügt x in S ein.
- `delete`(x, S) : $S \leftarrow S \setminus \{x\}$ Löscht x aus S.
- `member`(x, S) : $x \in S$? Testet, ob x in S enthalten ist.

Die Größe von S bezeichnen wir im folgenden immer mit n.

3.1.1 Realisierungen

1. **Array :** Wir reservieren ein Feld der Größe N und speichern die Elemente von S ungeordnet in den ersten n Positionen ab. Wir brauchen dann auch noch eine zusätzliche Variable, die uns die Position der letzten benutzten Feldposition liefert (also jeweils n enthält).

   Lemma 3.1.1 *Wenn $|S| = n$ ist, dann braucht* `member` *Zeit $\Theta(n)$ (auch randomisiert), während die übrigen Operationen in Zeit $O(1)$ gehen (jedenfalls wenn man auf das zu löschende Element bereits einen Zeiger hat).*

   □

   Neben der teuren Zugehörigkeitsabfrage ist die statische Größe des Feldes ein weiterer Nachteil dieser Implementierung.

2. **Lineare Liste :** Wir speichern die Elemente von S ungeordnet in einer doppelt verketteten Liste ab.

   Lemma 3.1.2 *Wenn $|S| = n$ ist, dann braucht* `member` *Zeit $\Theta(n)$ (auch randomisiert), während die übrigen Operationen in Zeit $O(1)$ gehen (je-*

denfalls wenn man auf das zu löschende Element bereits einen Zeiger hat). □

Zumindest haben wir bei dieser Implementierung nicht mehr das Problem der statischen Größe.

3. **Bitvektor :** Wir reservieren einen Bitvektor der Länge N und kodieren die Menge S dadurch, daß wir das i-te Bit genau dann auf 1 setzen, wenn i in S ist.

   Lemma 3.1.3 *Wenn $|S| = n$ ist, dann braucht* `makeset` *Zeit $\Theta(n)$, während die übrigen Operationen in Zeit $O(1)$ gehen.* □

   Die teure Initialisierung (der ganze Bitvektor muß mit Nullen initialisiert werden) und die Tatsache, daß die Operation `union` (vereinige zwei Mengen) ebenfalls teuer ist, sind ungünstig. Bitvektoren sind allerdings die günstigste Implementierung (weil einfach), wenn $n \approx N$ ist.

   Dieses Verfahren ist ein Spezialfall der *direkten Adressierung* (wenn wir in der i-ten Feldposition noch andere Information abspeichern als bloß ein Bit für die Existenz des Elements).

3.1.2 Hashing mit Verkettung

Wir kommen nun zu einer Implementierung, die für $n \ll N$ effizient ist.

Definition 3.1.4 Eine *Hashfunktion* $h : U \to \{0, \ldots, m-1\}$ ist eine Abbildung von U in eine Hashtafel T der Größe m, die wir als Feld darstellen. In Tafelposition $T[i]$ speichern wir alle $x \in S$ mit $h(x) = i$, etwa als lineare Liste (das nennt man dann *Hashing mit Verkettung*). Für festes h bezeichnen wir diese Methode auch als *Standardhashing.*
Für $x \in U$ bezeichnen wir mit

$$C_x(S) = |\{y \in S \mid h(x) = h(y)\}|$$

die Größe der *Konfliktmenge* von x bzgl. S. □

In den Übungen werden wir noch eine andere Methode kennenlernen, die Elemente von S in T abzuspeichern, das sogenannte *offene Hashing.* Aber jetzt machen wir erst einmal ein paar Beispiele für Hashing mit Verkettung.

Beispiel 3.1.5

1. Sei $m = 1$ und $h(x) = 1$ für alle $x \in U$. Dann werden alle Elemente von S in einer linearen Liste in $T[1]$ abgespeichert.

2. Sei $m = N$ und $h(x) = x$ für alle $x \in U$. Diese Hashfunktion realisiert dann gerade das Prinzip der direkten Adressierung,

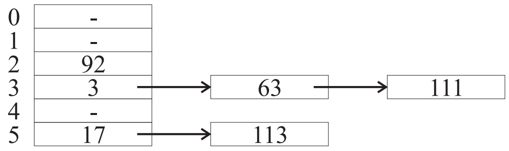


Abbildung 3.1: Hashing mit Verkettung für $N = 127$, $S = \{3, 17, 63, 92, 111, 113\}$, $m = 6$ und $h(x) = (x \bmod N) \bmod m$ (also $a = 1$).

3. Sei $h : x \mapsto ((ax) \bmod N) \bmod m$ für ein $a \in \{1, \ldots, N-1\}$. Solche Hashfunktionen sind wegen ihrer Einfachheit und Effizienz (wie wir später sehen werden) sehr beliebt. Abbildung 3.1 zeigt ein Beispiel für diese Art Hashfunktion. □

Lemma 3.1.6 *Wenn wir h in konstanter Zeit auswerten können, dann brauchen die Operationen* `insert`, `member` *und* `delete` *Zeit $O(1 + C_x(S))$.*

Beweis: Wir geben nur den Algorithmus für `member`.

member(x, S)
begin
  $y \leftarrow h(x)$;
  Suche x in der Liste $T[y]$;
end

□

Im schlechtesten Fall kann $C_x(S) = n$ sein, nämlich wenn h alle Elemente von S auf die gleiche Position abbildet. Wenn N groß ist, ist dies sogar unvermeidlich.

Lemma 3.1.7 *Sei h eine Hashfunktion. Wenn $N \geq (n-1) \cdot m + 1$ ist, dann gibt es ein $S \subseteq U$, $|S| = n$, so daß h alle Elemente von S auf die gleiche Tafelposition abbildet.*

Beweis: Es gibt ein $x \in U$ mit $|h^{-1}(x)| \geq \lceil \frac{N}{m} \rceil \geq n$. □

3.2 Mittlere Laufzeit von Hashing

Für die Analyse der mittleren Laufzeit von Hashing nehmen wir an, alle n-elementigen Teilmengen S von U seien gleichwahrscheinlich. Unser Wahrscheinlichkeitsraum ist somit

$$W = \{S \mid S \subseteq U \text{ und } |S| = n\}\,.$$

Die Wahrscheinlichkeit von S ist

$$ps = \frac{1}{|W|} = \frac{1}{\binom{N}{n}}\,.$$

Für festes $x \in U$ ist dann $C_x(S)$ eine Zufallsvariable, die S auf eine Zahl abbildet (nämlich die Anzahl der Elemente in S, die mit x kollidieren). Also können wir den Erwartungswert von C_x bilden :

$$\begin{aligned} E[C_x] &= \frac{1}{|W|} \cdot \sum_{S \in W} C_x(S) \\ &= \frac{1}{|W|} \cdot \sum_{S \in W} \sum_{\substack{y \in S, \\ h(y)=h(x)}} 1 \,. \end{aligned}$$

Hier wäre es nützlich, die beiden Summenzeichen vertauschen zu können, was aber nicht möglich ist, da das innere Summenzeichen vom äußeren abhängt. Mit einem Trick wird die Vertauschung dennoch möglich. Dazu definieren wir für $y \in U$ die folgende Indikatorfunktion :

$$i_y(S) = \begin{cases} 1 & \text{, falls } y \in S \,, \\ 0 & \text{, sonst.} \end{cases}$$

Nun können wir die “1” durch “$i_y(S)$” ersetzen und die Summe über U statt S laufen lassen. Dann sind die beiden Summen voneinander unabhängig und es folgt :

$$\begin{aligned} E[C_x] &= \frac{1}{|W|} \cdot \sum_{S \in W} \sum_{\substack{y \in U, \\ h(y)=h(x)}} i_y(S) \\ &= \frac{1}{|W|} \cdot \sum_{\substack{y \in U, \\ h(y)=h(x)}} \sum_{S \in W} i_y(S) \\ &\overset{\text{(Nebenrechnung)}}{=} \sum_{\substack{y \in U, \\ h(y)=h(x)}} \frac{n}{N} \\ &= \frac{n}{N} \cdot \left| h^{-1}(h(x)) \right| \,. \end{aligned}$$

Nebenrechnung :
Da i_y eine 0-1-wertige Zufallsvariable ist, ist

$$\frac{1}{|W|} \cdot \sum_{S \in W} i_y(S) = E[i_y] = p(i_y = 1) \,.$$

Dies ist aber gerade

$$p(i_y = 1) = \frac{|\{S \subseteq U \mid |S| = n \text{ und } y \in S\}|}{\binom{N}{n}} = \frac{\binom{N-1}{n-1}}{\binom{N}{n}} = \frac{n}{N} \,.$$

□

Definition 3.2.1 (Faire Hashfunktion)
Eine Hashfunktion h ist *fair*, wenn für alle $i \in \{0, \ldots, m-1\}$ gilt :

$$|\{y \in U \mid h(y) = i\}| \leq \left\lceil \frac{N}{m} \right\rceil .$$

□

Beispiel 3.2.2 Eine faire Hashfunktion ist zum Beispiel

$$h : x \mapsto x \bmod m .$$

□

Satz 3.2.3 *Wenn h fair ist, dann gilt für alle $x \in U$:*

$$E[C_x] \leq \frac{n}{m} + \frac{n}{N} .$$

Beweis: Nach der vorangegangenen Rechnung ist

$$E[C_x] \leq \frac{n}{N} \cdot \left\lceil \frac{N}{m} \right\rceil \leq \frac{n}{N} \cdot \left(\frac{N}{m} + 1 \right) = \frac{n}{m} + \frac{n}{N} .$$

□

Die Zahl $\frac{n}{m}$ in der Formel oben, also die Anzahl der abzuspeichernden Elemente geteilt durch die Größe der Tafel, nennt man den *Belegungsfaktor* (oft β genannt).

Korollar 3.2.4 *Die durchschnittliche Zeit pro Operation beträgt bei Hashing mit Verkettung $O(1 + \frac{n}{m}) = O(1 + \beta)$.*

Beweis: Für $N > m$ ist $E[C_x] \leq 2 \cdot \frac{n}{m}$ nach Satz 3.2.3. □

3.3 Universelles Hashing

Das Problem bei der Analyse der mittleren Laufzeit von Standardhashing ist, daß S normalerweise *nicht* gleichverteilt daherkommt. Dieses Problem versuchen wir dadurch zu lösen, daß wir umgekehrt die Hashfunktion zufällig aus einer sogenannten *universellen Klasse* von Hashfunktionen wählen.
Der (gleichverteilte) Wahrscheinlichkeitsraum W ist also bei universellem Hashing H und nicht $\{S \subseteq U \mid |S| = n\}$ wie bei Standardhashing. Bei Standardhashing gibt es schlechte S (bzgl. der vom Benutzer gewählten Hashfunktion), während bei universellem Hashing der Algorithmus die Hashfunktion bestimmt und die c-Universalität für *alle* S gilt. Griffig und einprägsam formuliert : *Bei* Standardhashing *kontrolliert der Anwender die Würfel (d.h. entscheidet, welches S drankommt), bei* universellem Hashing *kontrolliert der Algorithmus die Würfel.*

Definition 3.3.1 Sei $c \in \mathcal{R}$ und $U = \{0, \ldots, N-1\}$. Dann heißt eine Menge

$$H \subseteq \{h : U \to \{0, \ldots, m-1\}\}$$

von Hashfunktionen *c-universell*, wenn für alle $x, y \in U$, $x \neq y$, gilt :

$$|\{h \in H \mid h(x) = h(y)\}| \leq c \cdot \frac{|H|}{m}.$$

□

Wir zeigen nun, daß c-universelle Klassen H existieren und daß eine zufällige Hashfunktion aus einer solchen Klasse "gut" ist.
Sei $S \subseteq U$, $|S| = n$ und $x \in U$. Dann definieren wir für $h \in H$ die Indikatorfunktion

$$\delta_{x,S}(h) = |\{y \in S \mid y \neq x \text{ und } h(x) = h(y)\}|.$$

Für festes h ist $\delta_{x,S}(h) \approx C_x(S)$.

Lemma 3.3.2 *Sei N eine Primzahl. Für $a \in \{1, \ldots, N-1\}$ sei*

$$h_a : x \mapsto ((ax) \bmod N) \bmod m$$

eine Hashfunktion. Dann ist die Klasse

$$H = \{h_a \mid a \in \{1, \ldots, N-1\}\}$$

$(2 + \frac{m}{N-1})$*-universell.*

Beweis: Seien $x, y \in U$, $x \neq y$. Wieviele $a \in \{1, \ldots, N-1\}$ gibt es mit $h_a(x) = h_a(y)$?
Wenn $h_a(x) = h_a(y)$ ist, dann ist

$$((ax) \bmod N) \bmod m = ((ay) \bmod N) \bmod m$$

und daher gilt

$$m | \underbrace{(\underbrace{(ax) \bmod N}_{\in\{0,\ldots,N-1\}} - \underbrace{(ay) \bmod N}_{\in\{0,\ldots,N-1\}})}_{\in\{-(N-1),\ldots,N-1\}}.$$

Also ist

$$(ax) \bmod N - (ay) \bmod N = i \cdot m$$

$$\iff \quad a(x-y) \equiv i \cdot m \; (\bmod N) \qquad (\star)$$

für ein i mit $0 \leq |i| \leq \frac{N-1}{m}$.
Diese Gleichung hat wegen $x \neq y$ und N prim genau eine Lösung für a (denn wenn $az = w$ und $bz = w$ ist, dann ist $(a-b)z = 0$. Da N Primzahl ist, teilt dann N entweder $(a-b)$ oder z. Also ist $z = 0$ oder $a = b$).

Für jeden Wert von i gibt es also höchstens ein a mit $(\star)$. i kann aber höchstens $\frac{N-1}{m} \cdot 2 + 1$ Werte annehmen (die "+1" könnte man auch noch wegtricksen, denn i kann nicht 0 sein), also gibt es höchstens $\frac{N-1}{m} \cdot 2 + 1$ verschiedene a's mit $h_a(x) = h_a(y)$. Da $|H| = N - 1$ ist, ist die Klasse H also c-universell mit

$$c = \frac{2 \cdot \frac{N-1}{m} + 1}{\frac{N-1}{m}} = 2 + \frac{m}{N-1} .$$

□

Lemma 3.3.3 *Wenn H c-universell ist, dann ist*

$$E[\delta_{x,S}] \leq c \cdot \frac{n}{m} = c \cdot \beta .$$

Beweis: Sei

$$\delta_{x,y}(h) = \begin{cases} 1 & \text{, falls } x \neq y \text{ und } h(x) = h(y) , \\ 0 & \text{, sonst .} \end{cases}$$

für $x, y \in U$ und $h \in H$. Dann gilt für $S \subseteq U$:

$$\delta_{x,S}(h) = \sum_{y \in S} \delta_{x,y}(h) .$$

Es folgt :

$$\begin{aligned} E[\delta_{x,S}] &= E[\sum_{y \in S} \delta_{x,y}] \\ &= \sum_{y \in S} E[\delta_{x,y}] \\ &= \sum_{y \in S} p(\delta_{x,y} = 1) \\ &\leq \sum_{y \in S} (\frac{1}{|H|} \cdot (c \cdot \frac{|H|}{m})) \\ &= c \cdot \frac{n}{m} . \end{aligned}$$

□

Da selbst für die beste Hashfunktion $E[\delta_{x,S}]$ mindestens $1 \cdot \frac{n}{m}$ sein muß, sind c-universelle Hashfunktionen bis auf einen konstanten c Faktor optimal.

Satz 3.3.4 *Sei $S \subseteq U$, $|S| = n$ und H eine c-universelle Klasse von Hashfunktionen. Wenn wir $h \in H$ zufällig wählen, dann brauchen die Operationen* `insert`, `delete` *und* `member` *erwartete Zeit $O(1 + c \cdot \frac{n}{m})$.*

Beweis: Für $x \in U$ ist die erwartete Größe der Konfliktmenge höchstens $c \cdot \frac{n}{m}$.
□

Mit der Klasse von Hashfunktionen aus Lemma 3.3.2 sind die erwarteten Kosten also asymptotisch optimal $O(1 + \frac{n}{m})$.

3.4 Perfektes Hashing

Für jedes $S \subseteq U$ mit $|S| = n$ gibt es eine einfache Funktion

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

die injektiv in S ist, jedenfalls wenn $m \geq 4n$ ist. Ein solches h ist auch leicht zu finden.
Sei H c-universell und $S \subseteq U$ mit $|S| = n$. Für $0 \leq i < m$ und $h \in H$ definieren wir

$$\begin{aligned} B_{i,h} &= \{x \in S \mid h(x) = i\} \quad \text{und} \\ b_{i,h} &= |B_{i,h}| \,. \end{aligned}$$

Damit ist

$$b_{i,h} \cdot (b_{i,h} - 1) = |\{(x,y) \mid x, y \in S, x \neq y \text{ und } h(x) = h(y) = i\}|$$

und

$$\begin{aligned} \sum_{i=0}^{m-1} b_{i,h} \cdot (b_{i_h} - 1) &= |\{(x,y) \mid x, y \in S, x \neq y \text{ und } h(x) = h(y)\}| \\ &= \sum_{x \in S} \sum_{y \in S} \delta_{x,y}(h) \,. \end{aligned}$$

Summation über alle Hashfunktionen h ergibt

$$\begin{aligned} \frac{1}{|H|} \cdot \sum_{h \in H} \sum_{i=0}^{m-1} b_{i,h} \cdot (b_{i,h} - 1) &= \frac{1}{|H|} \cdot \sum_{h \in H} \sum_{x \in S} \sum_{y \in S} \delta_{x,y}(h) \\ &\overset{(3.3.3)}{\leq} \sum_{x \in S} c \cdot \frac{n}{m} \\ &= c \cdot \frac{n^2}{m} \,. \end{aligned}$$

Folgerungen:

(1) $\exists h: \; \sum_{i=0}^{m-1} (b_{i,h} \cdot (b_{i,h} - 1)) \;\leq\; c \cdot \frac{n^2}{m}$.

(2) Für die Hälfte der $h \in H$ gilt :

$$\sum_{i=0}^{m-1} (b_{i,h} \cdot (b_{i,h} - 1)) \leq 2c \cdot \frac{n^2}{m} \,.$$

(3) Wenn $m \leq cn^2$ ist, dann gilt :

$$\begin{aligned} & \exists h : \sum_{i=0}^{m-1} (b_{i,h} \cdot (b_{i,h} - 1)) \ < \ 1 \\ \Longrightarrow \quad & \exists h : \ b_{i,h} = 0 \text{ oder } b_{i,h} = 1, \text{ da die } b_{i,h} \in \mathbb{N}_0 \\ \Longrightarrow \quad & \exists h : \ h \text{ ist injektiv in } S \, . \end{aligned}$$

(4) Wenn $m > 2cn^2$ ist, dann sind mindestens 50% der $h \in H$ injektiv in S.

(5) Wenn $m = cn$ ist, dann gilt :

$$\begin{aligned} & \exists h : \sum_{i=0}^{m-1} (b_{i,h} \cdot (b_{i,h} - 1)) \ \leq \ n \\ \Longrightarrow \quad & \exists h : \sum_{i=0}^{m-1} b_{i,h}^2 \ \leq \ n + \sum_{i=0}^{m-1} b_{i,h} \ \leq \ 2n \, . \end{aligned}$$

(6) Wenn $m = 2cn$ ist, dann gilt

$$\sum_{i=0}^{m-1} b_{i,h}^2 \ \leq \ 2n$$

für mindestens 50% der $h \in H$. □

Aus Folgerungen (4) und (6) ergibt sich folgende Idee für ein 2-stufiges Hashing.

Perfektes Hashing

1. Sei die Tafelgröße $m = 2cn + 1$ und sei H eine c-universelle Klasse von Hashfunktionen von U nach $\{0, \ldots, m-1\}$. Wir wählen $h \in H$ zufällig, bis

$$\sum_{i=0}^{m-1} b_{i,h}^2 \leq 2n$$

ist. Dafür brauchen wir im Mittel 2 Versuche, also Zeit $O(n)$.

2. Sei $m_i = 2cb_{i,h}^2 + 1$ für $i = 0, \ldots, m-1$. Sei H_i eine c-universelle Klasse von Hashfunktionen von U nach $\{0, \ldots, m-1\}$. Wir wählen $h_i \in H_i$ zufällig, bis $h_i|_{B_{i,h}}$ injektiv ist. Dafür brauchen wir im Mittel 2 Versuche, also Zeit $O(b_{i,h}^2)$.

3. Sei $s_j = \sum_{i<j} m_i$. Dann benutzen wir insgesamt die Hashfunktion

$$x \mapsto s_{h(x)} + h_i(x) \, .$$

□

Die Gesamttafelgröße ist dabei

$$\begin{aligned}\sum_{i=0}^{2cn}(2cb_{i,h}^2+1) &= \sum_i 2cb_{i,h}^2+2cn+1\\ &\le 2c2n+2cn+1\\ &= (4c+2c)n+1\\ &= 6cn+1\,,\end{aligned}$$

also linear in n.

Satz 3.4.1 *Perfektes Hashing garantiert konstante Zugriffszeit. Eine perfekte Hashfunktion können wir in erwarteter linearer Zeit finden.*

Beweis: Wir brauchen Zeit

$$O\left(n+\sum_{i=0}^{m-1} b_{i,h}^2\right)=O(n)\,.$$

□

3.5 Dynamisches Hashing

Frage : Welche Tafelgröße ist ideal ?

- Zugriffszeit $\approx \beta = \frac{n}{m}$, also sollte m groß sein, um die Zugriffzeit zu minimieren.
- Platzbedarf $\approx n+m$, also sollte m klein sein, um den Platzbedarf zu minimieren.

Kompromiß : Wir wählen $m \approx n$.

Aber was ist, wenn wir n nicht kennen ?

Sei $i_0 \in \mathbb{N}$ eine kleine Konstante. Die Tafelgröße ist immer eine Zweierpotenz $m=2^i$ unter Einhaltung der folgenden Invariante :

> **Invariante :**
>
> Wenn $|S|=n$ ist, dann ist entweder $i=i_0$ oder $2^{i-1}\le n\le 2^{i+1}$.

Dann können wir wie folgt Elemente einfügen oder streichen :

- Einfügen in Tafel der Größe 2^i :
  - Füge ein wie gehabt.
  - Falls die Anzahl der Elemente jetzt gerade 2^{i+1} ist, dann gehe zu einer Tafel der Größe 2^{i+1} über.

- Streichen in Tafel der Größe 2^i :
  - Streiche wie gehabt.
  - Falls die Anzahl der Elemente jetzt gerade 2^{i-1} ist, dann gehe zu einer Tafel der Größe 2^{i-1} über.

Nach dem Übergang auf eine neue Tafel ist der Belegungsfaktor jeweils genau 1.

Satz 3.5.1 *Wenn die Kosten des Umspeicherns (in eine neue Tafel) linear in der Anzahl der umzuspeichernden Elemente sind, dann haben* `insert` *und* `delete` *amortisierte Laufzeit* $O(1)$.

Beweis: Wir benutzen die Bankier-Methode. Wir initialisieren das Konto mit $2 \cdot 2^{i_0}$ DM. Jede Operation bringt 2 DM mit, die wir sparen, wenn keine neue Tafel kreiiert wird.
Wir zeigen nun durch Induktion über die Anzahl der ausgeführten Operationen, daß wir immer mindestens $|(n - 2^i) \cdot 2|$ DM auf dem Konto haben.

- Zu Beginn ist $n = 0$ und wir haben $2 \cdot 2^{i_0}$ DM auf dem Konto.
- Induktionsschritt :

  Eine Operation, die nicht zum Umspeichern führt, zahlt 2 DM aufs Konto. Die Invariante gilt dann wieder.

  Ein Operation, die zum Umspeichern führt, zahlt 2 DM aufs Konto. Danach ist entweder $n = 2^{i+1}$ und es liegen mindestens $(2^{i+1} - 2^i) \cdot 2 = 2^{i+1}$ DM auf dem Konto, oder es ist $n = 2^{i-1}$ und es liegen mindestens $(2^i - 2^{i-1}) \cdot 2 = 2^{i-1}$ DM auf dem Konto. Der Kontostand reicht in jedem Fall zum Bezahlen des Umspeicherns. Nach dem Umspeichern ist $n = 2^i$ (für das neue i) und die Behauptung stimmt wieder. □

Beispiele für Anwendungen von Hashing sind Symboltafeln in Compilern oder die Kodierung/Verifizierung von Paßwörtern.

Bem.: Es gibt auch dynamisches perfektes Hashing (Dietzfelbinger, 1994). □

Kapitel 4

Vergleichsorientierte Suchverfahren

4.1 Suchbäume

Sei das *Universum* U linear geordnet (mit '$\leq$').

Definition 4.1.1 Ein *Suchbaum* ist ein Baum, in dem jeder Knoten v mit $d \geq 2$ Kindern mit $d-1$ Schlüsseln $k_1(v), ..., k_{d-1}(v) \in U$ markiert ist, wobei $k_1(v) \leq ... \leq k_{d-1}(v)$ ist. $deg(v) = d$ ist der *Grad* von v. Wir definieren auch immer $k_0(v) = -\infty$ und $k_d(v) = +\infty$.

Die *Tiefe* eines Suchbaums definieren wir wie folgt :
Die Wurzel hat Tiefe $depth(root) = 0$ und alle anderen Knoten v haben Tiefe $depth(v) = 1 + depth(Elter(v))$. Die *Tiefe von T* ist die maximale Tiefe eines Blattes, also $depth(T) = \max_v depth(v)$. □

4.1.1 Suchen

access(x)

begin

$v \leftarrow root$;

while v ist kein Blatt

do

bestimme $i, 1 \leq i \leq deg(v)$, so daß $k_{i-1}(v) \leq x \leq k_i(v)$ ist;

$v \leftarrow i$-tes Kind von v;

od

end

Die Laufzeit ist offensichtlich $O(depth(v))$, wenn die Suche in v endet.

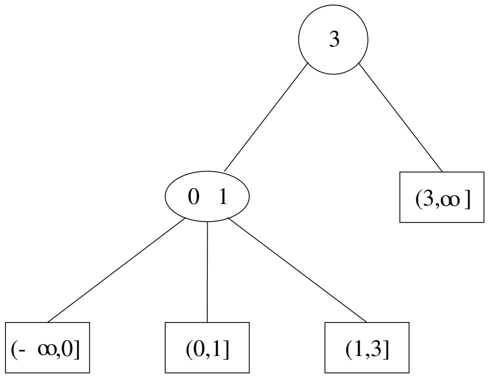


Abbildung 4.1: Beispiel eines Suchbaums. In den Blättern sind ihre *Ranges* vermerkt.

Definition 4.1.2 Für einen Knoten v bezeichnen wir mit

$$R(v) = \{x \in U \mid \text{Suche nach } x \text{ geht durch den Knoten } v\}$$

den *Bereich (Range)* von v. □

Lemma 4.1.3 *Sei U die Menge aller Werte, die in den Blättern des Baums enthalten sein können. Dann ist*

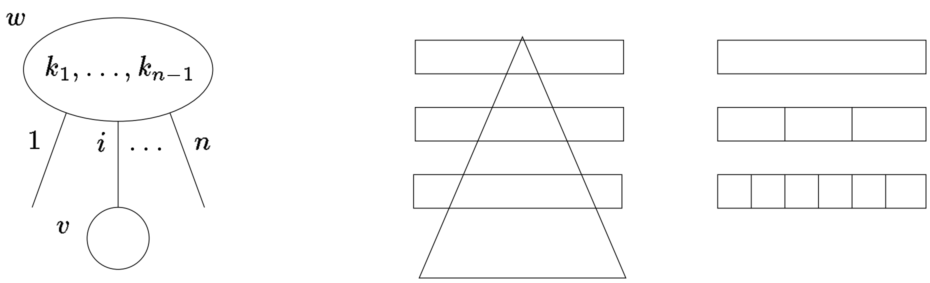
$$R(Wurzel) = U\ .$$

Sei w ein Knoten, der die Schlüssel $k_1, \ldots, k_{n-1} \in U$ mit $k_0 = -\infty < k_1 < \ldots < k_{n-1} < k_n = +\infty$ enthält, und sei v das i-te Kind von w. Dann ist

$$R(v) = R(w) \cap (k_{i-1}(w), k_i(w)]\ .$$

□

Informell ist der *Range* eines Knotens v eine Teilmenge von U, in der alle von v aus erreichbaren Blätter liegen. Von Ebene zu Ebene werden dabei die Unterteilungen von U immer feiner, wie die folgende Abbildung verdeutlicht :



Definition 4.1.4 T heißt *Suchbaum* für eine Menge $S \subseteq U$, wenn für jedes Blatt v von T gilt : $|R(v) \cap S| = 1$.
Ein Blatt heißt mit $y \in S$ *beschriftet*, falls $y \in R(v) \cap S$. □

4.1.2 Einfügen

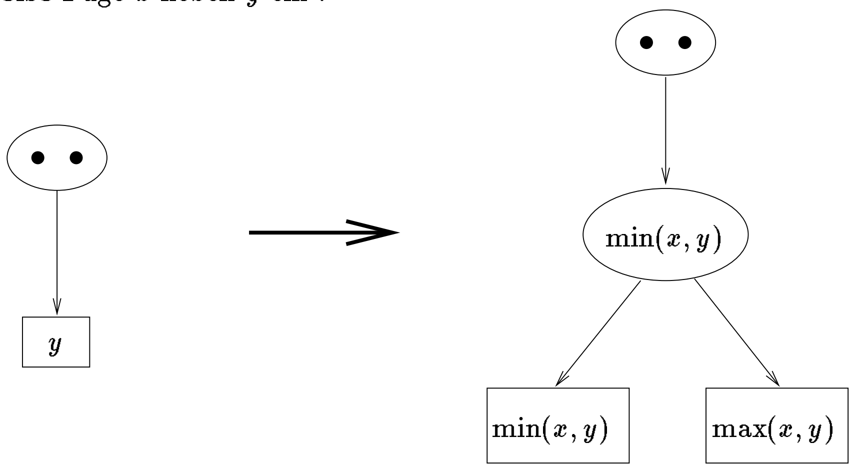
insert(x)

begin

Sei v das Blatt, in dem die Suche nach x endet, und sei y das eindeutige Element in S mit $\{y\} = R(v) \cap S$;

if $x = y$ **then** Tue nichts;

else Füge x neben y ein :



fi

end

4.1.3 Streichen

delete(x)

begin

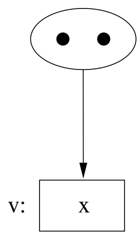
Suche nach x; sei v das Blatt, in dem Suche endet;

if v ist nicht mit x beschriftet

then Tue nichts;

else

Streiche das Blatt v, den Zeiger auf v und einen der benachbarten Schlüssel aus $Elter(v)$;



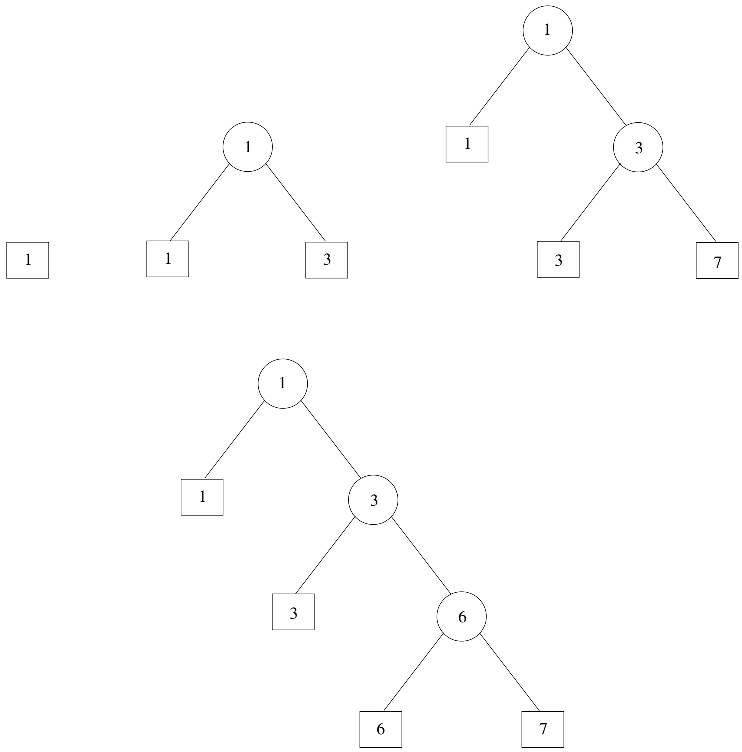


Abbildung 4.2: Einfügen von 1,3,7,6.

---

if $deg(Elter(v)) = 1$

then eliminiere $Elter(v)$;

fi

// stellt sicher, daß alle inneren Knoten mindestens

// Grad zwei haben

fi

end

Suchbäume können zu linearen Listen entarten (degenerieren), z.B. beim Einfügen von $1, 2, 3, \ldots$.
Antworten auf Entartung :

1. Vertrauen auf günstige Einfügungsreihenfolge.

2. Balancierte Bäume.

Zu 1. : Verwandtschaft Suchbäume - Quicksort.
$\Pi = \pi_1, \pi_2, \ldots, \pi_n$ Permutation der Zahlen $1, \ldots, n$.

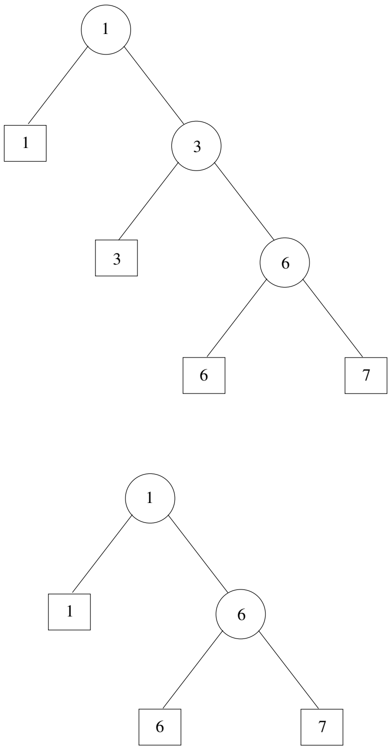


Abbildung 4.3: Streichen von 3.

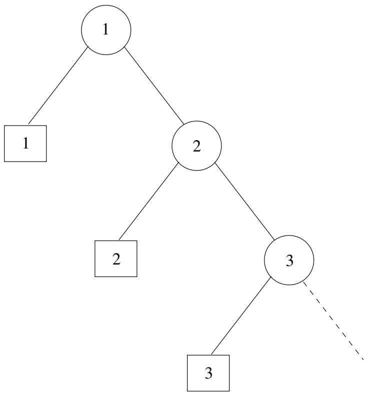


Abbildung 4.4: Entarteter Suchbaum nach Einfügen von $1, 2, 3, \ldots$.

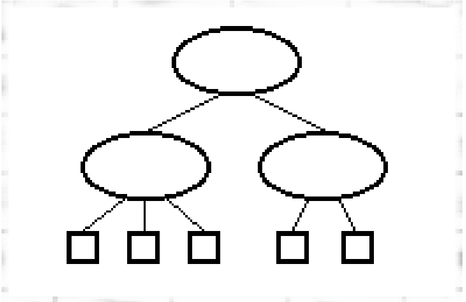
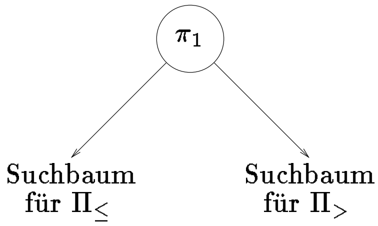


Abbildung 4.5: Ein $(2,4)$-Baum.



$\Pi_\leq$ = streiche alle Zahlen $> \pi_1$ aus Π,
$\Pi_>$ = streiche alle Zahlen $\leq \pi_1$ aus Π.

4.2 Balancierte Bäume

Es gibt verschiedene Ausprägungen von balancierten Bäumen : AVL-Bäume, Rotschwarz-Bäume, Randomisierte Suchbäume, $(2,4)$-Bäume, BB$[\alpha]$-Bäume, B-Bäume, Skiplists, ...

4.2.1 $(2,4)$-Bäume

Definition 4.2.1 Ein Baum heißt $(2,4)$-*Baum*, wenn gilt :

- Jeder innere Knoten hat mindestens zwei und höchstens vier Kinder.
- Alle Blätter haben gleiche Tiefe. □

Lemma 4.2.2 *Für die Höhe h eines $(2,4)$-Baums mit n Blättern gilt* $\log_4 n \leq h \leq \log_2 n$.

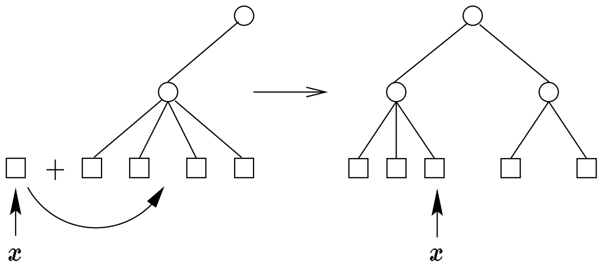
Beweis: Es ist

$$\begin{aligned} & 2^h \leq n \leq 4^h \\ \Longleftrightarrow \quad & \frac{\log n}{2} \leq h \leq \log n \,. \end{aligned}$$

□

Die Suchzeit in einem $(2,4)$-Baum ist also logarithmisch. Wir zeigen nun, daß auch `insert` und `delete` effizient realisierbar sind.

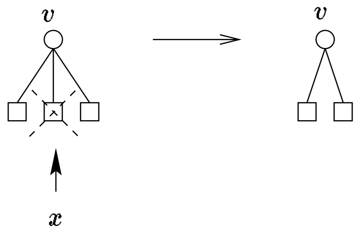
- **insert(x)** : Einfügen in einen Knoten mit weniger als 4 Kindern ist trivial. Bei genau 4 Kindern wenden wir folgende Operation an :



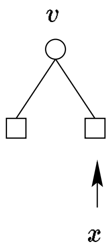
  Dies setzt sich gegebenenfalls rekursiv bis zur Wurzel fort; der Baum kann dabei um 1 höher werden.

- **delete(x)** :

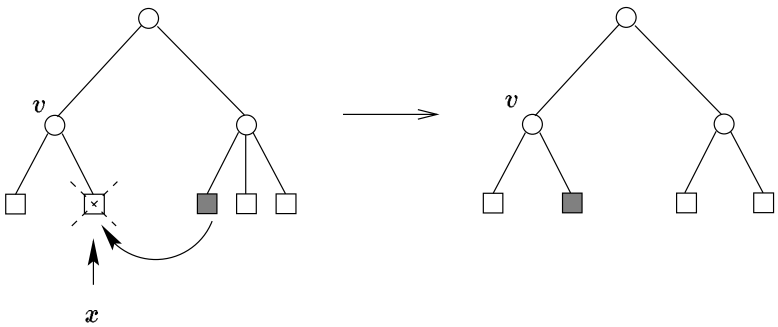
  - Wenn v mindestens 3 Kinder hat, dann löschen wir x :



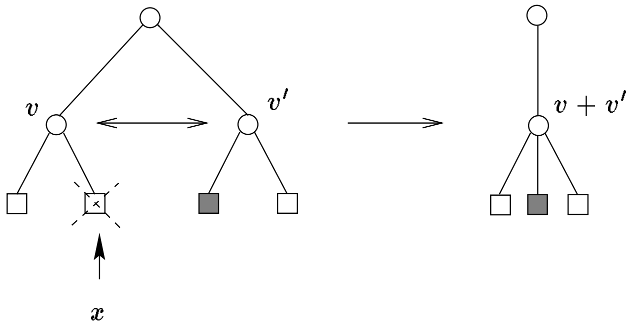
  - Wenn v genau 2 Kinder hat, dann :



    * Wenn der Nachbar von v mindestens 3 Kinder hat, dann stehlen wir eines der Kinder (in diesem Fall keine Rekursion).



* Wenn der Nachbar von v genau 2 Kinder hat, dann verschmelzen wir v mit seinem Nachbarn.



Dies setzt sich gegebenenfalls rekursiv bis zur Wurzel fort; der Baum kann um 1 niedriger werden.

Satz 4.2.3 *Wenn wir $S \subseteq U$ in einem $(2,4)$-Baum abspeichern, dann können wir die Operationen* `access`, `insert`, `delete`, `find_min` *und* `delete_min` *in Zeit $O(\log |S|)$ durchführen.*

Beweis:

- **access :** Beim Suchen laufen wir den Baum von der Wurzel aus abwärts bis zu einem Blatt hinunter. Die Suchzeit ist also abhängig von der Tiefe. Diese ist aber beschränkt durch $O(\log |S|)$.
- **insert/delete :**
  - Suchen nach dem Element dauert Zeit $O(\log |S|)$.
  - Die lokale Änderung (Einfügen/Löschen) geht in $O(1)$ Zeit.
  - Hochlaufen bis zur Wurzel kostet $O(\log |S|)$. Jeder Balancierungsschritt braucht Zeit $O(1)$. Damit braucht Rebalancieren insgesamt Zeit $O(\log |S|)$.
- **find_min :** Das Minimum befindet sich im linkesten Blatt. Es reicht also, das linke Rückgrat runterzulaufen (das *linke Rückgrat* ist der Weg von der Wurzel, der immer zum linken Kind geht; das *rechte Rückgrat* ist analog definiert). Also Zeit $O(\log |S|)$.
- **delete_min :** Ausführung von `find_min` und anschließend `delete` braucht Zeit $O(\log |S|)$.

$\square$

4.2.2 (a,b)-Bäume

Definition 4.2.4 Analog zur Definition von $(2,4)$-Bäumen definieren wir *(a,b)-Bäume*, und zwar für $b \geq 2a-1$.

- Jeder innere Knoten hat mindestens a und höchstens b Kinder.
- Alle Blätter haben gleiche Tiefe. $\square$

Bem.:

- Da $b \geq 2a - 1$ ist, werden immer nur Knoten mit mindestens $2a$ Kindern geteilt, so daß jeder neue Teilknoten anschließend mindestens a Kinder bekommen kann.
- $\log_b n \leq depth(T) \leq 1 + \log_a \frac{n}{2}$.

Beweis: Der rechte Teil der Ungleichung kommt daher, daß bei Bäumen mit Tiefe größer als 1 die Wurzel wenigstens zwei Kinder hat; eines dieser Kinder hat maximal $n/2$ Blätter und minimalen Verzweigungsgrad a. Beachte, daß alle Blätter von T gleiche Tiefe haben. $\square$

Frage : Was passiert bei einer Folge von mehreren Operationen ?

Satz 4.2.5 *Wenn wir in einem anfangs leeren* $(2,5)$*-Baum eine Folge von* n `insert` *und* `delete` *Operationen machen, dann gibt es höchstens* $2n$ *Rebalancierungen. Die amortisierten Kosten der Operationen sind also* $O(1)$.

Beweis: Für einen Knoten v definieren wir das Potential $Pot(v)$ in Abhängigkeit von seinem Grad als

$deg(v)$	1	2	3	4	5	6
$Pot(v)$	3	1	0	0	1	3

Die amortisierten Kosten c_{am} sind

$$c_{am} = c_{tats} + \Delta Pot\ .$$

Wenn wir zeigen können, daß $c_{am} \leq 2$ ist, dann folgt :

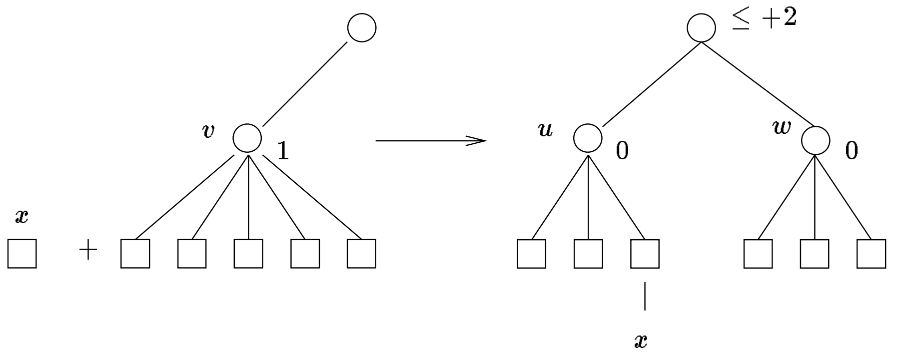
$$\begin{array}{rrcl} & 2n \geq \sum c_{am} & = & \sum c_{tats} + \underbrace{\sum \Delta Pot}_{Pot_{Ende} - Pot_{Anfang}} \\ \Longleftrightarrow & \sum c_{tats} & = & 2n - \underbrace{Pot_{Ende}}_{\geq 0} + \underbrace{Pot_{Anfang}}_{=0} \\ & & \leq & 2n\ . \end{array}$$

Beh.: Für jede einzelne Operation gilt :

$$\Delta Pot \leq 1 - (\#\text{Rebalancierungsoperationen})\ .$$

Beweis:

- **insert :** Sind beim Einfügen eines Knotens keine Rebalancierungen nötig, dann ändert sich das Potential maximal um eins. Den Fall, in dem Rebalancierungen notwendig sind, zeigt folgende Abbildung :



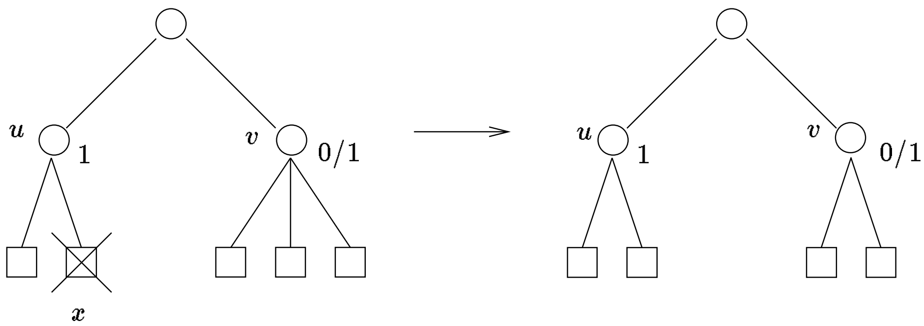
Der Knoten v hat vor dem Einfügen von x Potential eins, da er fünf Blätter hat. Nach der Rebalancierung haben die Knoten u und w jeweils Potential null. Hinzu kommt die Potentialänderung für den Vater von v. Diese beträgt maximal zwei, und zwar für den Fall, daß v vor der Rebalancierung fünf und nachher sechs Geschwister hat. Also gilt :

$$\Delta Pot \leq -1 + 2 \cdot 0 + 2 \leq 1 \; .$$

Falls sich die Rebalancierung rekursiv fortsetzt, kann die eine Rebalancierungsoperation eines Levels jeweils von der Potentialerniedrigung um -1 auf dem darunterliegenden Level ausgeglichen werden.

- **delete :**

    1. Wenn der zu löschende Knoten x mindestens zwei Geschwister hat, sind keine Rebalancierungen erforderlich; folglich steigt das Potential des Vaterknotens von x um maximal eins.
    2. Den Fall, in dem ein Kind gestohlen wird, zeigt folgende Abbildung :

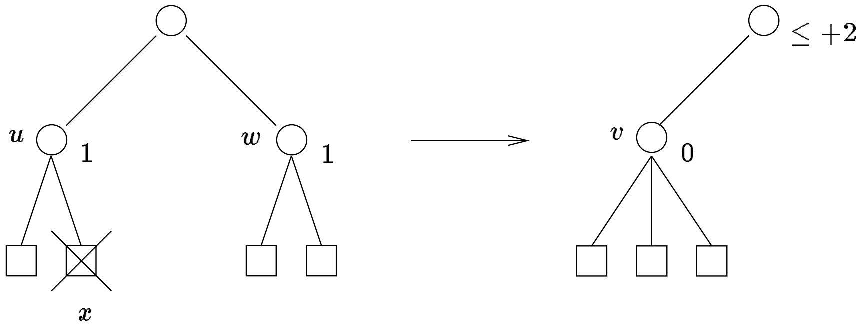


Knoten u hat Anfangspotential eins, da er zwei Blätter hat. Knoten v hat Anfangspotential null (falls der Knoten drei oder vier Blätter hat) bzw. eins (falls der Knoten fünf Blätter hat). Nach Stehlen eines Blatts von v hat dieser Knoten noch Potential eins (falls v jetzt zwei Blätter hat) bzw. Potential null (falls er noch drei oder vier Blätter hat). Knoten u hat nun Potential eins, da er jetzt zwei Blätter hat. Also gilt :

$$\Delta Pot \leq -1 + 1 + 1 \leq 1 \,.$$

In diesem Fall gibt es keine Rekursion.

3. Den Fall, in dem zwei benachbarte Knoten verschmolzen werden, zeigt folgende Abbildung :



Knoten u hat Ausgangspotential eins, da er nur ein Blatt hat. Knoten w hat Ausgangspotential eins, da er zwei Blätter hat. Nach dem Verschmelzen von u und w zu Knoten v hat dieser neu entstandene Knoten Potential null, da er drei Blätter hat. Hinzu kommt wieder die Potentialänderung für den Vaterknoten von u und w, die wiederum maximal zwei ist. Also gilt :

$$\Delta Pot \leq -1 - 1 + 0 + 2 \leq 0 \,.$$

Falls sich die Rebalancierung rekursiv fortsetzt, kann die eine Rebalancierungsoperation eines Levels jeweils von der Potentialerniedrigung um -2 auf dem darunterliegenden Level ausgeglichen werden.

Andere Betrachtungsweise : Wenn wir bei jeder Operation 2 DM mitbringen, können wir damit die konstanten Kosten der Operation und die Potentialänderung von maximal eins bezahlen. Bei Rebalancierungen kann sich ein `insert` beziehungsweise ein `delete` bis zur Wurzel fortsetzen. Allerdings steigt in einem solchen Fall das Potential des betroffenen Knotens um zwei. Damit können wir dann den zusätzlichen Aufwand „bezahlen“.

Sind bei einem `insert` oder `delete` insgesamt l solcher Rebalancierungsschritte erforderlich, so gilt

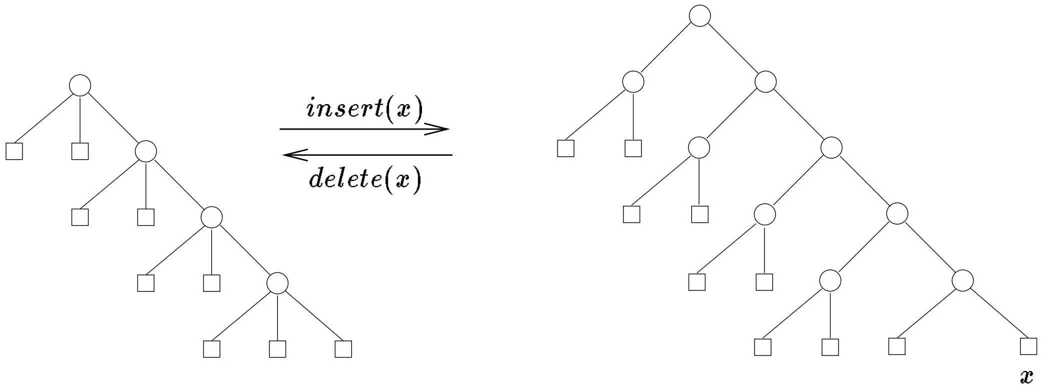
$$c_{am} \quad \leq \quad 2 + \sum_{i=1}^{l} (1 + (-1)) \quad \leq \quad 2 \,.$$

Die 2 steht dabei für das mitgebrachte Geld, die 1 für die konstanten Kosten einer Rebalancierung und die (-1) für den Wertgewinn durch die Zunahme des Potentials. □

Bem.: Der Satz gilt auch allgemein für (a, b)-Bäume mit $b \geq 2a$, allerdings ist der Beweis für $b = 2a$ etwas schwieriger. Für $b = 2a + 1$ kann man folgende Potentialfunktion verwenden :

$deg(v)$	$a-1$	a	$a+1$	$\dots$	$b-1$	b	$b+1$
$Pot(v)$	3	1	0	0	0	1	3

Für $b = 2a - 1$ gilt der Satz *nicht*, jedenfalls solange gemischte Folgen von `insert` und `delete` zugelassen sind, wie das folgende Beispiel zeigt (abwechselnd `delete`(x) und `insert`(x) hat jedesmal hohe Kosten) :



Der Satz gilt aber für Folgen von nur `insert` oder nur `delete` Operationen. □

4.2.3 Adaptives Sortieren

Eine Anwendung für $(2, 5)$-Bäume ergibt sich bei dem Sortieren einer Folge von Zahlen, die schon teilweise vorsortiert ist. Als Beispiel diene die Folge 1,3,7,5,8,10,12. Durch *Verschieben der 5 an die richtige Stelle* erhält man eine sortierte Folge.

Sei also $x_1, \dots, x_n$ eine Folge von n verschiedenen Zahlen.

$$F = |\{(i, j) | \, i < j \text{ und } x_i > x_j\}|$$

bezeichne die Anzahl der Inversionen in der Folge $x_1, \dots, x_n$. Im günstigsten Fall ist die Folge aufsteigend sortiert und es gibt 0 Inversionen. Im ungünstigsten Fall ist die Folge absteigend sortiert und es gibt $n - i$ Inversionen zwischen dem i-ten und den folgenden Elementen, insgesamt also $\sum_{i=1}^{n}(n-i) = n(n-1)/2$ Inversionen. Daher gilt für F :

$$0 \leq F \leq \frac{n(n-1)}{2} \,.$$

Wir sortieren nun die Folge $x_1, \dots, x_n$, indem wir die Elemente der Folge der Reihe nach in einen $(2, 5)$-Baum einfügen (dies nennt man auch *Insertionsort*).

Insertionsort

begin

1.

Füge x_1 in leeren Baum ein;

$\vdots$

$\boldsymbol{i}$.

Seien $x_1, \ldots, x_{i-1}$ bereits sortiert als Folge $y_1 < \cdots < y_{i-1}$.

Wir wollen nun x_i einfügen :

$$\underbrace{y_1 < \cdots y_{i-f_i-1}}_{i-1-f_i} < x_i < \underbrace{y_{i-f_i} < \cdots < y_{i-1}}_{f_i} .$$

x_i kommt an die $(i - f_i)$-te Stelle von links, wobei

$$f_i = |\{j \mid j < i \text{ und } x_i < x_j\}|$$

die Anzahl der Inversionen mit x_i als zweiter Komponente ist.

Für die f_i gilt : $F = \sum_{i=1}^{n} f_i$.

end

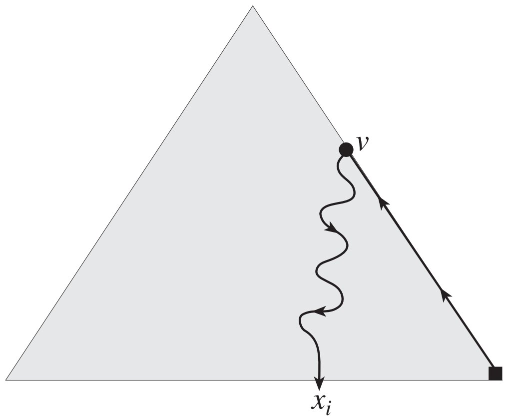


Abbildung 4.6: Einfügen von x_i.

Wenn x_i gar keine Inversionen mit $y_1, \ldots, y_{i-1}$ hat (also $f_i = 0$), dann wird x_i ganz rechts im Baum als neues Maximum eingefügt. Falls f_i klein ist, wird x_i ziemlich weit rechts eingefügt. Dies legt folgende Strategie nahe :
Wir beginnen die Suche nach x_i nicht in der Wurzel, sondern am rechtesten Blatt des Baumes. Wir laufen zuerst solange das rechte Rückgrat hoch, bis wir einen Knoten v finden, der auf dem Suchpfad von der Wurzel nach x_i liegt. Dann suchen wir von dort aus ganz normal nach unten.

Eine spezielle Suchmethode

begin

$v \leftarrow$ rechtestes Blatt im Baum;

while v ist nicht Wurzel und $x_i \leq key_{deg(v)-1}(v)$

do

$v \leftarrow Elter(v)$;

od;

Suche ganz normal von v aus nach unten;

Füge x_i ein und rebalanciere;

end

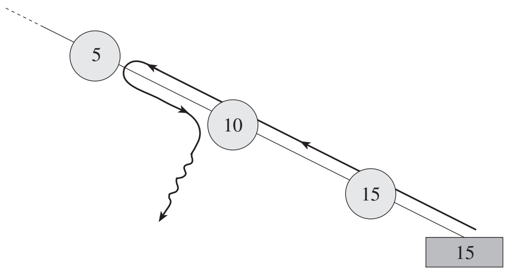


Abbildung 4.7: Beispiel eines Suchpfades für $x_i = 8$.

Satz 4.2.6 (Adaptives Sortieren)
Mit $(2,4)$-Bäumen können wir Folgen mit F Inversionen in Zeit $O(n\cdot(1+\log \frac{F}{n}))$ sortieren.

Beweis: Die Kosten für die Suche von x_i sind $O(h_i)$, wobei h_i die Höhe von v ist. Die amortisierten Kosten des Einfügens sind $O(1)$ (Satz 4.2.5).
Wir müssen also die Höhe h_i beschränken. Sei w das rechteste Kind von v und z das rechteste Kind von w.
Es gilt : $k_{deg(w)-1}(w) \geq x_i > k_{deg(v)-1}(v)$. Also wird x_i *nicht* in den Unterbaum mit der Wurzel z eingefügt. Folglich sind alle Blätter in dem Unterbaum mit Wurzel z größer als x_i. Dieser Unterbaum hat mindestens $2^{Höhe(z)} = 2^{h_i-2}$ Blätter. Also gibt es unter den $x_1, \ldots, x_{i-1}$ mindestens 2^{h_i-2} Elemente, die kleiner als x_i sind. Also gilt :

$$\begin{aligned} 2^{h_i-2} \leq f_i \quad \Longleftrightarrow \quad h_i &\leq 2 + \log f_i \\ \Longrightarrow \quad \sum_i h_i &\leq 2n + \sum_i \log f_i \\ &\leq 2n + \log \prod_i f_i \\ &\leq 2n + \log \left(\frac{F}{n}\right)^n = n \cdot (2 + \log \frac{F}{n}) \,. \end{aligned}$$

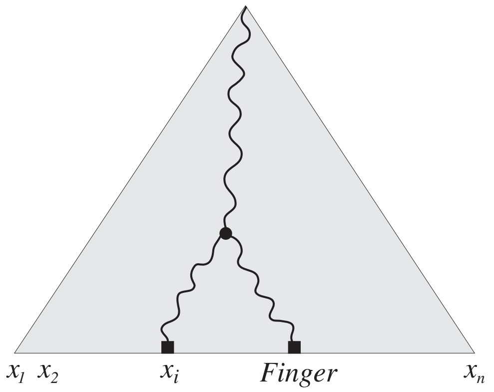


Abbildung 4.8: Fingersuche nach einem Element x_i aus $x_1, \ldots, x_n$.

Dabei haben wir benutzt :
Wenn $\sum f_i = F$ ist, dann ist $\prod_i f_i$ maximal, wenn $f_i = \frac{F}{n}$ ist für alle i. □

Für spezielle Werte von F ergibt sich :

- $F \approx n^2 \longrightarrow$ Laufzeit $O(n \log n)$.
- $F \approx n \longrightarrow$ Laufzeit $O(n)$.
- $F \approx n \log n \longrightarrow$ Laufzeit $O(n \log \log n)$.

4.3 Fingersuchbäume

Im letzten Abschnitt haben wir adaptives Sortieren dadurch beschleunigt, daß wir die Suche nach der Einfügestelle nicht in der Wurzel begonnen haben, sondern am rechtesten Blatt des Baumes. Dies läßt sich dahingehend verallgemeinern, daß wir die Suche nach einem Element x an einem beliebigen vorgegeben Blatt f im Suchbaum beginnen, einem sogenannten *Finger*. Und zwar suchen wir von f aus nach oben, bis wir einen gemeinsamen Vorfahren von x und f finden; dann suchen wir normal nach unten, bis wir x finden. Abb. 4.8 zeigt : Je länger der gemeinsame Pfad von der Wurzel des Baumes bis zum gemeinsamen Vorfahren ist, desto schneller geht die Suche.

In diesem Abschnitt führen wir folgende kurze Schreibweise ein : Es soll immer $\log(d) = \log(\max(1, d))$ bedeuten (für irgendwelche d).

Um die Suche wirklich effizient gestalten zu können, müssen wir den $(2,4)$-Baum noch ein wenig ergänzen. Und zwar verbinden wir die Knoten auf jedem Level durch eine doppelt verkettete Liste. Für jeden Knoten v sei dabei:

- ***LiGe***(v) **:** Linker Nachbar von v in dieser Liste.
- ***ReGe***(v) **:** Rechter Nachbar von v in dieser Liste.

Wir nenne einen solchen Baum *Fingersuchbaum*.

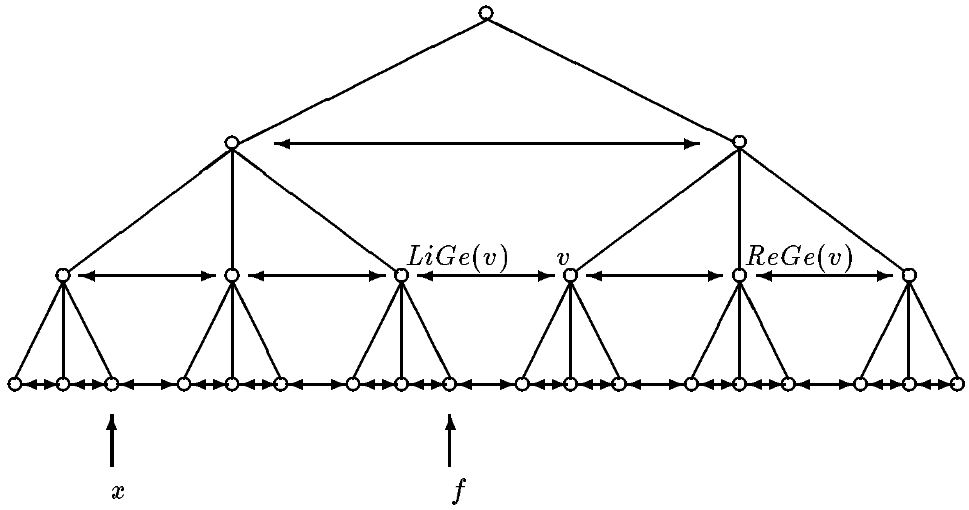


Abbildung 4.9: Ein Fingersuchbaum.

Satz 4.3.1 *In einem Fingersuchbaum können wir jedes Element x von einem beliebigen Finger f aus in Zeit $O(1 + \log d)$ suchen, wobei d die Anzahl der Blätter zwischen x und f ist.*

Beweis: Sei o.B.d.A. $x < f$. Wir benutzen folgenden Suchalgorithmus.

access(x)

begin

$v \leftarrow f$;

while $([(x \leq k_1(LiGe(v)))$ **or** $(x > k_{deg(v)-1}(v))]$ **and** $(v \notin$ linkes Rückgrat$))$

do

$v \leftarrow Elter(v)$;

od

// Wir wissen hier : $k_1(w) < x \leq k_{deg(v)-1}(v)$, wobei $w = LiGe(v)$.

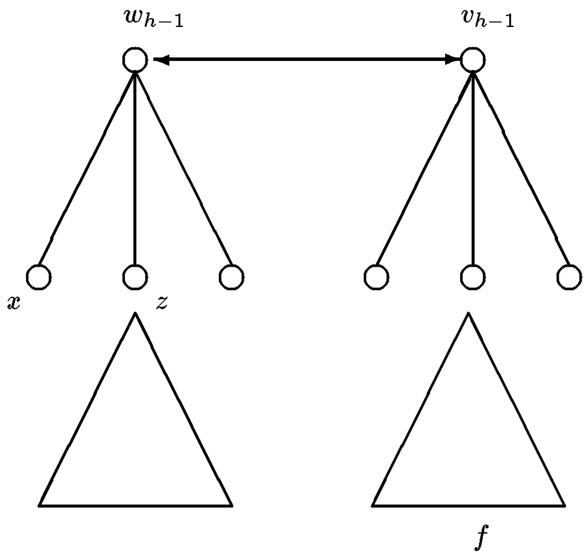
Fall 1 : $\exists 1 \leq i \leq deg(w) - 2 : k_i(w) < x \leq k_{i+1}(w)$:
Dann suche normal x unter w.

Fall 2 : $\exists \leq j \leq deg(v) - 2 : k_j(v) < x < k_{j+1}(v)$:
Dann suche normal x unter v.

Fall 3 : $k_{deg(w)-1}(w) < x \leq k_1(v)$:
$v \leftarrow$ linkestes Kind von v;
$w \leftarrow$ rechtestes Kind von w;
wiederhole;

end

Sei $v_1, v_2, \ldots, v_h$ die Folge der Knoten v bei der Aufwärtsuche und $w_1, w_2, \ldots, w_h$ die dazugehörige Folge von linken Nachbarn. Dann gilt auf Höhe $h-1$: $x \leq k_1(w_{h-1})$.



Das zweite Kind z von w_{h-1} existiert. Alle Blätter unter z liegen zwischen x und f. Daher ist $d \geq$ (#Blätter unter z) $= 2^{h-2} \iff h \leq 2 + \log d$. □

Bem.:

1. Minimum oder Maximum sind häufige Finger.
2. Statt $O(1+\log(d))$ geht auch $O(1+\min(\log(d), \log(n-d)))$:

   Mache aus der doppelt verketteten Liste einen doppelt verketteten Kreis und suche dann in beide Richtungen. □

4.4 Mergesort

Seien L_1 und L_2 sortierte Listen der Längen n_1 bzw. n_2 :

$$L_1 = (x_1 \leq x_2 \leq \ldots \leq x_{n_1}) \quad \text{und}$$

$$L_2 = (y_1 \leq y_2 \leq \ldots \leq y_{n_2})\,.$$

Merge(L_1, L_2) : Liefert sortierte Liste $L = (z_1 \leq z_2 \leq \ldots \leq z_{n_1+n_2})$, wobei $\{z_1, \ldots, z_{n_1+n_2}\} = \{x_1, \ldots, x_{n_1}\} \bigcup \{y_1, \ldots, y_{n_2}\}$ ist.

Mergesort(L)

// Generischer Algorithmus

begin

Sei $L = L_1 \cup L_2$;

`Mergesort`(L_1);

```
    Mergesort(L2);
    return Merge(L1, L2);
end
```

4.4.1 Drei Arten von Mergesort

1. **Scan-Merge :** Gehe von links nach rechts gleichzeitig über beide Listen und entferne jeweils das kleinere der beiden head-Elemente.

   Beispiel : $\left.\begin{array}{l} L_1 = 1-3-5-6 \\ L_2 = 2-4-8 \end{array}\right\} 1-2-3-4-5-6-8$

2. **Insertion-Merge :** Füge Elemente von L_2 nacheinander in L_1 ein.

3. **Finger-Merge :** Wie Insertion-Merge, aber starte Suche nach y_{i+1} bei Finger y_i.

Satz 4.4.1 *Sei $V(n_1, n_2)$ die Anzahl von Vergleichen, um Listen der Länge n_1 und n_2 zu mergen. Dann gilt :*

	$V(n_1,n_2)$	$V(n_1,1)$	$V(n,n)$	$V\left(n, \frac{n}{\log(n)}\right)$
`Scan`	n_1+n_2-1	n_1	n	$O(n)$
`Insertion`	$O(n_2\log(n_1+n_2))$	$O(\log n_1)$	$O(n\log n)$	$O(n)$
`Finger`	$O(n_2 + n_2\log\frac{n_1}{n_2})$ $= O(\log\binom{n_1+n_2}{n1})$	$O(\log n_1)$	$O(n)$	$O(n\cdot\frac{\log\log n}{\log n})$

Beweis: Wir müssen nur noch die Laufzeit von `Finger-Merge` zeigen. Sei dazu

$$\begin{array}{lcl} d_0 & = & \text{Anzahl der Elemente in } L_1 \text{ zwischen } -\infty \text{ und } y_1, \\ d_i & = & \text{Anzahl der Elemente in } L_1 \text{ zwischen } y_i \text{ und } y_{i+1} \text{ für } i \geq 1. \end{array}$$

Dann braucht `insert`(y_{i+1}) gerade Zeit $O(1+\log d_i)$. Also insgesamt:

$$\begin{aligned} T &= O\left(\sum_{i=0}^{n_2-1}(1+\log d_i)\right) \\ &= O\left(n_2 + \log\left(\prod_{i=0}^{n_2-1} d_i\right)\right) \\ &\leq O\left(n_2 + \log\left(\frac{n_1}{n_2}\right)^{n_2}\right) \quad , \quad \text{da } d_0+d_1+\cdots+d_{n_2-1} \leq n_1 \\ &= O\left(n_2 + n_2\log\left(\frac{n_1}{n_2}\right)\right) \\ &= O\left(\log\left(\frac{2n_1}{n_2}\right)^{n_2}\right) \\ &\leq O\left(\log\left(\frac{e\cdot n_1}{n_2}\right)^{n_2}\right) \end{aligned}$$

$$
\begin{aligned}
&\approx\ O\left(\log\left(\frac{n_1^{n_2}}{n_2!}\right)\right) \left(\qquad (\text{Sterling}: n_2! \approx \left(\frac{n_2}{e}\right)^{n_2}\right) \\
&\leq\ O\left(\log\left(\frac{(n_1+1)\cdots(n_1+n_2)}{n_2!}\right)\right) \\
&=\ O\left(\log\binom{n_1+n_2}{n_1}\right) .
\end{aligned}
$$

□

Bem.: Untere Schranke ist $\Omega\left(\log\binom{n_1+n_2}{n_1}\right)$.

Beweis: Wir benutzen wieder das Entscheidungsbaummodell. Ein Entscheidungsbaum für `Merge` hat mindestens $\binom{n_1+n_2}{n_1}$ Blätter, weil es so viele Möglichkeiten gibt, eine Liste der Länge $n_1 + n_2$ aus L_1 und L_2 zusammenzusetzen.

$L =$  $n_1 + n_2$

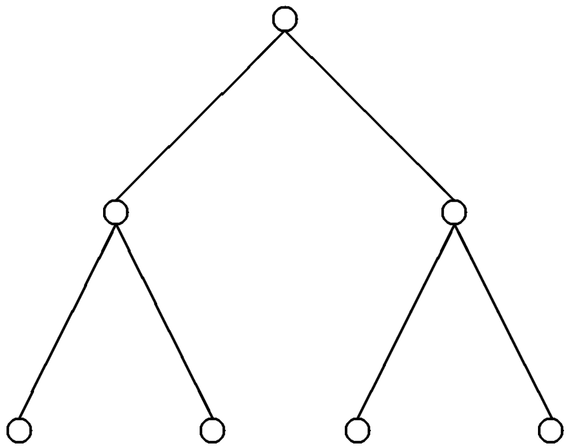
□

4.4.2 Split-Pattern für Mergesort

Wir müssen uns nun noch überlegen, wie wir die Liste L möglichst günstig in zwei kleinere Listen L_1 und L_2 aufspalten.

1. **Balancierter Baum :**

   Wir zeichnen die rekursiven Splits als Baum :



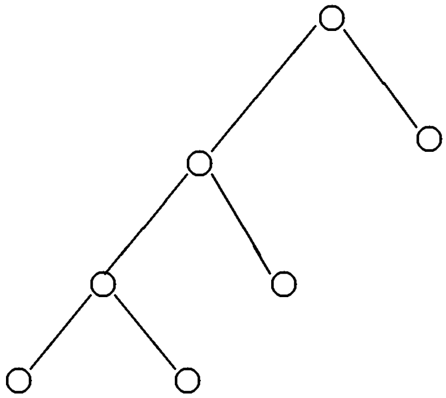
   Auf Level h gibt es 2^h Knoten mit jeweils $\frac{n}{2^h}$ Blättern darunter. Daher ist die Laufzeit von `Mergesort`

$$
\begin{aligned}
T &= \sum_{i=0}^{\log n} \left(2^h \cdot V\left(\frac{n}{2^{h+1}}, \frac{n}{2^{h+1}}\right)\right) \\
&= \sum_{i=0}^{\log n} \left(2^h \cdot \begin{cases} O(\frac{n}{2^{h+1}}) & \texttt{Scan, Finger} \\ O(\frac{n}{2^{h+1}} \cdot \log \frac{n}{2^{h+1}}) & \texttt{Insertion} \end{cases}\right)
\end{aligned}
$$

$$= \begin{cases} O(n \log n) & \texttt{Scan, Finger} \\ O(n \log^2 n) & \texttt{Insertion.} \end{cases}$$

2. **Unbalancierter Baum :**

   Wir spalten immer ein Element ab :



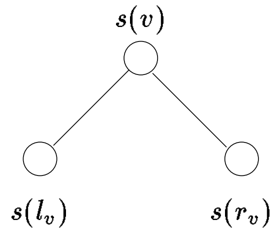
   In diesem Fall ist die Laufzeit

$$T = \sum_{i=1}^{n-1} V(i,1) = \begin{cases} O(\sum_{i=1}^{n} i) \quad = O(n^2) & \texttt{Scan} \\ O(\sum_{i=1}^{n} \log i) = O(n \log n) & \texttt{Insertion, Finger.} \end{cases}$$

In beiden Fällen ist `Finger-Merge` optimal. Der folgende Satz zeigt, daß dies kein Zufall ist.

Satz 4.4.2 `Mergesort` *mit* `Finger-Merge` *hat Laufzeit* $O(n \log n)$ *für alle Mischmuster.*

Beweis: Sei T ein Mischmuster, d.h. ein binärer Baum mit n Blättern. Für einen Knoten v sei $s(v)$ die Anzahl der Blätter in dem Unterbaum mit Wurzel v. l_v und r_v seien die beiden Kinder von v.



Der Mischschritt im Knoten v hat Kosten $O\left(\log \binom{s(v)}{s(l_v)}\right)$. Damit sind die Gesamtkosten

$$\begin{aligned} T &= \textstyle\sum_v O\left(\log \binom{s(v)}{s(l_v)}\right) \\ &= O\left(n + \textstyle\sum_v \log \frac{s(v)!}{s(l_v)! \cdot s(r_v)!}\right) \\ &= O(n + \textstyle\sum_v [\log(s(v)!) - \log(s(l_v)!) - \log(s(r_v)!)]) \,. \end{aligned}$$

In dieser Summe kommt der Beitrag $\log(s(w)!)$ eines Knoten w zweimal vor, und zwar einmal positiv für $v = w$ und einmal negativ für $v = Elter(w)$. Dies gilt für alle w außer der Wurzel und den Blättern. Die beiden Beträge für w kürzen sich dann weg. Übrig bleibt :

$$\begin{aligned} T &= O(n + \log(s(Wurzel)!) - n \cdot \log(s(Blatt)!)) \\ &= O(n + \log(n!) - n \cdot \log(1!)) \\ &= O(n \log n)\,. \end{aligned}$$

□

4.5 Verpackung von $(2,4)$-Bäumen als abstrakter Datentyp

- **Abstrakter Datentyp :**

  Spezifikation des Verhaltens unabhängig von der Angabe einer Implementierung.

  Eine Instanz vom Typ **sorted_sequence**$<$ **K**, **I** $>$ ist eine Folge von sogenannten *Items*. Jedes Item hat einen assoziierten Schlüssel in K und eine assoziierte Information in I. K muß linear geordnet sein; sei $<$ die Vergleichsoperation.

  Die Schlüssel in der Folge sind aufsteigend, d.h., wenn das Item $< k, i >$ vor dem Item $< k', i' >$ steht, dann ist $k < k'$.

- **Implementierung :**

  - Item = Zeiger auf ein Blatt eines $(2,4)$-Baumes.
  - `sorted_sequence` = $(2,4)$-Baum.

- **Operationen :**

  sorted_sequence$< K, I >$ S;

  Definiert S als `sorted_sequence` und initialisiert S mit leerer Folge.

  seq_item S.insert(K k, I i);

  Falls es kein Item in S gibt mit Schlüssel k, dann wird das Item $< k, i >$ zu S hinzugefügt. Falls es schon ein Item gibt, wird die Information des Items durch i ersetzt. In jedem Fall geben wir das Item zurück.

  K S.key(seq_item it);

  Liefert den Schlüssel des Items it.

  Vorbedingung : it ist ein Item in S (wird nicht überprüft, weil zu teuer !).

Kapitel 5

Priority Queues

Priority Queues (oder auch *Warteschlangen*) unterstützen die Operationen `insert`, `delete_min`, `find_min`, `delete` und `dec_prio`. Dabei erniedrigt die Prozedur `dec_prio` den Wert eines Items, der auch *Priorität* heißt.

Anwendungen von Priority Queues :

- Sortieren.
- Simulation (wichtigste Anwendung).
- Graphenalgorithmen (kürzeste Wege; kommt später).

5.1 Der Heap

Gegeben sei ein Feld H mit Elementtyp p und den Indizes $1, \ldots, n$, wobei n die Maximalgröße der Warteschlange ist. Die Variable `int` $size$ ist die Anzahl der in der Schlange enthaltenen Elemente. Das Feld repräsentiert dann *implizit* einen Baum mit den Knoten $1, \ldots, size$. Und zwar hat Knoten i Kinder $2i$ und $2i+1$ und Eltern $\lfloor \frac{i}{2} \rfloor$.
Der Baum ist *heap-geordnet*, wenn für jeden Knoten i, $2 \leq i \leq size$, gilt : $H[\frac{i}{2}] \leq H[i]$. Insbesondere ist $H[1]$ das Minimum. Ein heap-geordneter Baum heißt *Heap*. In einem Heap sind die Elemente auf jedem Pfad von der Wurzel zu einem Blatt aufsteigend sortiert.[1]

Einfügen in einen Heap ist einfach : Hänge das neue Element hinten an das Feld dran. Nun könnte für den Vaterknoten die Heapeigenschaft verletzt sein (wenn das neue Element nämlich kleiner ist). Dies läßt sich dadurch reparieren, daß man das Element mit seinem Vater vertauscht. Dies setzt sich eventuell rekursiv fort bis hoch zur Wurzel. Die Prozedur, die dieses Hochwandern ausführt, heißt `sift_up`. Sie realisiert gleichzeitig auch die Operation `dec_prio`.

insert(*P p*)
begin

[1] Dies gilt für *Min-Heaps*, wie wir sie gerade definiert haben. Wenn man fordert, daß jeder Knoten *größer* als seine Kinder ist, erhält man analog einen *Max-Heap*.

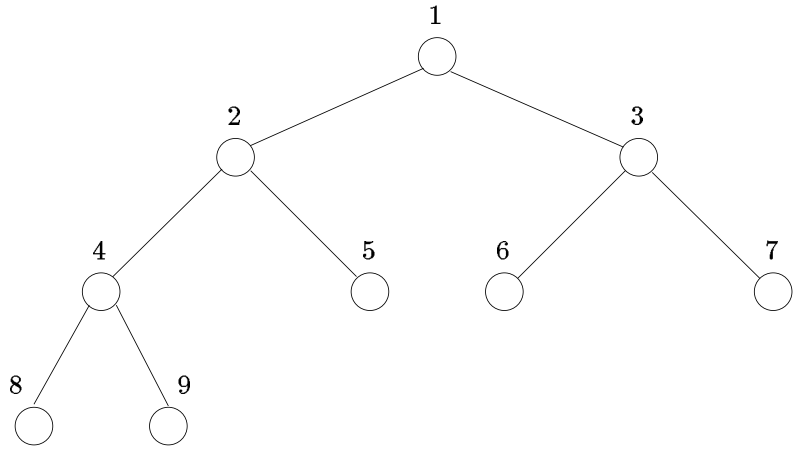


Abbildung 5.1: Ein impliziter Heap. Die Zahlen geben die Position eines Knoten im Array an. Knoten i hat Kinder $2i$ und $2i+1$ und Eltern $\lfloor \frac{i}{2} \rfloor$.

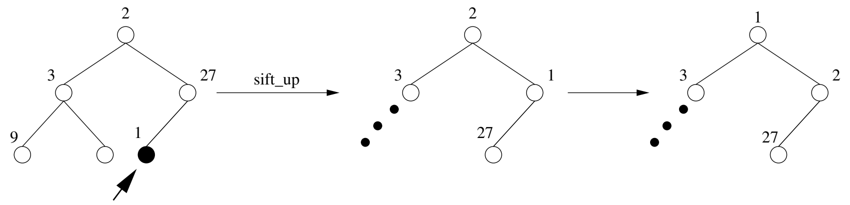


Abbildung 5.2: `insert`(1) in einem Heap.

    $H[++\,size] \leftarrow p$;
    `sift_up`($size$);
end

sift_up(int *k*)
// Lasse Element k den Heap hochlaufen, bis es paßt
begin
    while ($k > 1$ und $H[k] < H[k/2]$)
    do
        vertausche $H[k]$ und $H[k/2]$;
        $k \leftarrow k/2$;
    od
end

Löschen eines Elementes ist etwas schwieriger. Wir beginnen mit dem Löschen des kleinsten Elementes (das in der Wurzel steht). Dazu bringen wir das letzte Element des Feldes in die Wurzel, in der nun die Heap-Eigenschaft verletzt sein könnte. In diesem Fall vertauschen wir es mit dem kleineren seiner beiden Kinder. Dies kann sich wieder rekursiv fortsetzen. Die Prozedur, die dieses Runtersinken ausführt, heißt `sift_down`. Um ein beliebiges Element zu löschen, kann man nun zuerst seine Priorität auf $-\infty$ setzen, wodurch es zum Minimum wird, und es anschließend löschen.

delete_min()

begin

  $H[1] \leftarrow H[size--]$;

  `sift_down`(1);

end

sift_down(int k)

// Lasse Element k den Heap runtersinken, bis es paßt

begin

  if $(2 * k > size)$ /* keine Kinder */

  then

    return;

  else

    `int` $min \leftarrow 2 * k$;

    if ($min < size$ und $H[min+1] < H[min]$)

    then $min++$; **fi**

    if ($H[k] < H[min]$)

    then

      return;

    else

      vertausche $H[k]$ und $H[min]$;

      `sift_down`(min);

    fi

  fi

end

5.1.1 Implementierung von pq_item

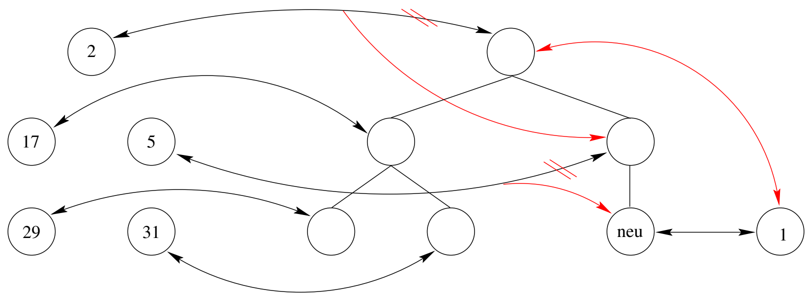
Es stellt sich die Frage, was wir als `pq_item` nehmen sollten.

Versuch 1 : `pq_item` = `int` ($\hat{=}$ Index im Feld H).

Das geht schief. Mit dieser Implementierung wäre *nicht* garantiert, daß das folgende Programm funktioniert :

`dec_prio` (it, p');
`prio` (it); // liefert nicht unbedingt p', da p' nach oben gewandert sein kann

Versuch 2 : Wir führen eine eigene Klasse `item_node` ein. Ein `item_node` enthält eine Priorität (Typ P), eine Information (Typ I) und einen Index in das Feld H. Die Elemente von H enthalten Zeiger auf `item_node`.
`pq_item` ist ein Zeiger auf `item_node` (**typedef** `item_node* pq_item`).



Das Feld H ist nun
$H[1 \ldots n]$: `array<pq_item>`;

Invarianten :

- $H[i] \rightarrow index = i$ für alle i, $1 \leq i \leq size$.
- $H[\frac{i}{2}] \rightarrow p \leq H[i] \rightarrow p$ für alle i, $2 \leq i \leq size$.

Die Prozedur für `sift_up` sieht dann so aus :

class `item_node`
begin
  P p;
  I i;
  `int` index;
end

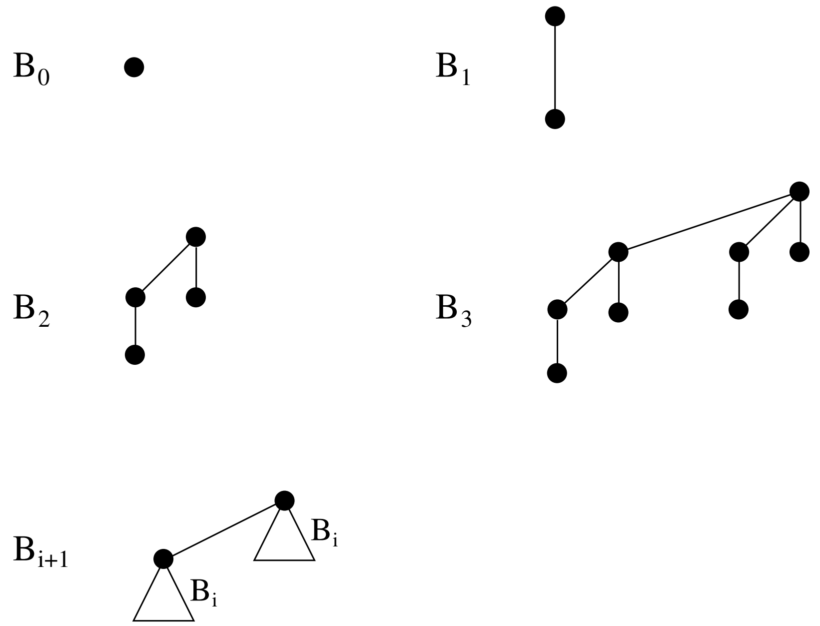
sift_up(i)
begin
  if ($i = 1$ or $H[i/2] \rightarrow p \leq H[i] \rightarrow p$)
  then return; **fi**

```
    // vertausche mit Elternknoten
    item_node* h ← H[i];
    H[i] ← H[i/2];
    H[i/2] ← h;
    H[i] → index ← i;
    H[i/2] → index ← i/2;

    // rekursiv für Elternknoten
    sift_up(i/2);
end
```

5.2 Die Binomial Queue

Wir definieren einen *Binomialbaum*[2] rekursiv :



Lemma 5.2.1 *Für einen Binomialbaum B_i gilt :*
B_i hat 2^i Knoten. Die Anzahl der Knoten in Tiefe t ist $\binom{i}{t}$. Insbesondere hat die Wurzel von B_i Grad i, und kein anderer Knoten von B_i hat einen so hohen Grad.

Beweis:

$$\begin{aligned} & \text{Anzahl der Knoten in Tiefe } t \text{ in einem } B_{i+1} \\ = \; & \text{Anzahl der Knoten in Tiefe } t \text{ in einem } B_i \\ & + \text{Anzahl der Knoten in Tiefe } t-1 \text{ in einem } B_i \\ = \; & \binom{i}{t} + \binom{i}{t-1} = \binom{i+1}{t} . \end{aligned}$$

[2] J. Vuillemin, 1972.

□

Wir sagen, B_i hat *Grad* i (das ist der Grad seiner Wurzel). Die *Priorität* $prio(B_i)$ von B_i ist definiert als die Priorität seines Wurzelelements.

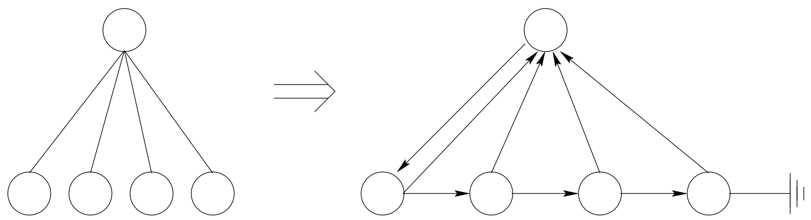
Definition 5.2.2 Ein Menge von heap-geordneten Binomialbäumen, von denen keine zwei den gleichen Grad haben, heißt *Binomial Queue*. □

Beispiel 5.2.3 Eine Binomial Queue für sieben Elemente besteht aus $B_2 + B_1 + B_0$. □

5.2.1 Darstellung im Speicher

Knoten haben folgende Komponenten :

- Verweis auf Vater.
- Verweis auf erstes Kind.
- Verweis auf rechtes Geschwister.
- Priorität.
- Grad.



Die Binomial Queue selbst wird dargestellt als

`array` <Zeiger auf Knoten> $R(30)$;

$$R[i] = \begin{cases} \texttt{nil} & \text{, falls es kein } B_i \text{ gibt} \\ \text{Zeiger auf Wurzel} & \text{, sonst} \end{cases}$$

5.2.2 Hilfsoperationen

Wir brauchen folgende Hilfsoperationen :

add_tree(node∗ T, int i)
// T ist Wurzel eines B_i und soll der PQ hinzugefügt werden
begin
  if $R[i] = \texttt{nil}$
  then
    $R[i] \leftarrow T$;
    return;

```
    fi
    T ← verschmelze(T, R[i]);
    R[i] ← nil;
    add_tree(T, i + 1);
end
```

```
verschmelze(node* T, node* R)
// T und R sind Zeiger auf Wurzeln von zwei Binomialbäumen
// gleichen Grades, die zu einem Binomialbaum von um 1 höheren
// Grad verschmolzen werden sollen
begin
    if prio(T) < prio(R)
    then
        Mache R zum ersten Kind von T;
    else
        Mache T zum ersten Kind von R;
    fi
end
```

5.2.3 Operationen des abstrakten Datentyps

```
insert(P p)
begin
    Erstelle neuen Binomialbaum T mit Grad 0, dessen Wurzel Priorität
    p hat;
    add_tree(T, 0);
end
```

```
delete_min()
begin
    Finde Wurzel mit minimaler Priorität und streiche sie;
    forall Kinder T der gestrichenen Wurzel
    do
        add_tree(T, deg(T));
    od
end
```

dec_prio(pq_item *it*, *P p*)
begin
  $it \rightarrow prio \leftarrow p$;
  `sift_up`(it);
end

Bem.:

- `insert` führt zu einem Aufruf von `add_tree`.
- `delete_min` führt zu höchstens maximaler Grad Aufrufen von `add_tree`.
- `add_tree` ist eine rekursive Prozedur :
  Jeder rekursive Aufruf (von `add_tree` durch sich selbst) verringert die Anzahl der Bäume in der Binomial Queue um eins (durch die Prozedur `verschmelze`). □

5.2.4 Amortisierte Analyse

Wir benutzen die folgende Potentialfunktion Φ :

$$\Phi = c \cdot (\text{Anzahl der Bäume in der Binomial Queue})\ ,$$

wobei c die Laufzeit eines Aufrufs von `add_tree` ist (*ohne* die Zeit für einen rekursiven Aufruf).

Lemma 5.2.4 *Die amortisierten Kosten eines Aufrufs von* `add_tree` *(inclusive der Kosten der rekursiven Aufrufe) sind* $O(1)$.

Beweis: Sei k die Anzahl der rekursiven Aufrufe. Dann sind die Gesamtkosten $(1+k)\cdot c$, nämlich jeweils c für die k rekursiven Aufrufe sowie für den äußersten Aufruf. Die Anzahl der Bäume verringert sich um k. Dies ergibt

$$c_{am} = \underbrace{\Phi_{nachher} - \Phi_{vorher}}_{-c\cdot k} + \underbrace{\text{tatsächliche Laufzeit}}_{(1+k)\cdot c} = c\ .$$

□

Wenn man noch die Potentialerhöhung um c für den neu einzufügenden Baum mitrechnen würde, dann wäre diese Analyse identisch mit dem Beweis der amortisierten Kosten eines Binärzählers. In der Tat kann man eine Binomial Queue als Binärzahl interpretieren (das Feld $R(30)$ entspricht einem 0/1-Bitvektor), wobei `add_tree`(T, i) der Addition einer 1 an Stelle i entspricht.

Satz 5.2.5 *Die Kosten der Grundoperationen in einer Binomial Queue sind :*

	Amortisierte Kosten	*Kosten im schlechtesten Fall*
`insert`	$O(1)$	$O(log n)$
`dec_prio`	$O(log n)$	$O(log n)$
`delete_min`	$O(log n)$	$O(log n)$
`delete`	$O(log n)$	$O(log n)$

Beweis: `insert` konstruiert einen Baum (Kosten $O(1)$ und Potentialerhöhung um c) und ruft dann `add_tree` auf. Also gilt für die amortisierten Kosten :

$$c_{am} = O(1) + c + c_{am}(\texttt{add\_tree}) = O(1)\,.$$

Die Kosten im schlechtesten Fall sind :

$$c_{wc} = O(1) + c_{wc}(\texttt{add\_tree}) = O(\log n)\,.$$

`delete_min` streicht einen Knoten und ruft für jedes Kind `add_tree` auf. Jedes Kind erhöht das Potential um c. Daher ist

$$c_{am} = (\#\ \text{Kinder}) \cdot (O(1) + c) + (\#\ \text{Kinder}) \cdot c_{am}(\texttt{add\_tree}) = O(\log n)\,.$$

Die Kosten im schlechtesten Fall sind (naiv gerechnet) :

$$c_{wc} = (\#\ \text{Kinder}) \cdot (O(1) + c_{wc}(\texttt{add\_tree})) = O(\log^2 n)\,.$$

Besseres Argument :
Vor `delete_min` gibt es höchstens $\log n$ Bäume. `delete_min` erhöht die Anzahl um höchstens $\log n$. Also können im schlechtesten Fall nicht für jedes Kind $\log n$ Verschmelzungen, sondern insgesamt nur höchstens $2 \log n$ Verschmelzungen von Bäumen stattfinden. □

Man kann aber auch `dec_prio` in amortisierter konstanter Zeit realisieren, und zwar mit den sogenannten *Fibonacci-Heaps.*

Satz 5.2.6 *In Fibonacci-Heaps gelten folgende amortisierte Laufzeitschranken :*

`insert`	:	$O(1)$
`dec_prio`	:	$O(1)$
`delete_min`	:	$O(\log n)$.

□

Kapitel 6

Union-Find, Partitionen

6.1 Definition

Wir möchten Partitionen einer Menge verwalten. Folgende Operationen sind erwünscht :

- Sind zwei Elemente im gleichen Block ?
- Vereinigung zweier Blöcke.
- Kreierung einer neuen Partition.

6.2 Anwendungsmöglichkeiten

1. In FORTRAN in den 50-er Jahren gab es *equivalent-statements* : Zwei Variablen konnten den gleichen Speicherplatz benutzen (da damals wenig Speicher zur Verfügung stand). Der Compiler mußte herausfinden, welchen Variablen er den gleichen Speicherplatz zuordnen durfte.

2. *Eine graphentheoretische Anwendung* : Man möchte die Zusammenhangskomponenten eines Graphen verwalten. Der Graph enthält am Anfang nur einzelne Knoten. Wenn eine Kante hinzugefügt wird, dann werden gegebenenfalls zwei Zusammenhangskomponenten vereinigt.

3. *Eine weitere graphentheoretische Anwendung* (kommt später nochmal ausführlich) : Die Kanten eines Graphen sind gewichtet. Man möchte die Teilmenge der Kanten finden, die alle Knoten verbindet (*aufspannender Baum*) und die geringsten Gesamtkosten hat.

   Algorithmus von Kruskal : Sortiere die Kanten nach Kosten und arbeite sie dann in aufsteigender Reihenfolge ab. Am Anfang ist jeder Knoten ein Block für sich. Jeder Block ist eine Zusammenhangskomponente im aktuellen Graph (aktueller aufspannender Baum). Wenn eine Kante zwei Knoten in verschiedenen Blöcken verbindet, dann wird sie zum aktuellen aufspannenden Baum hinzugefügt, andernfalls wird sie weggeworfen.

6.3 Realisierung

6.3.1 In LEDA

Eine Instanz P vom Datentyp `partition` ist eine Partition einer Menge von `partition_items`. Folgende **Operationen** sind realisiert :

- `partition` P : Kreiert neue Instanz P (Deklaration); die unterliegende Menge ist leer.
- `partition_item` P`.make_block(partition_item` it`)`: Fügt ein neues `partition_item` it zur unterliegenden Menge hinzu und macht es zu einem Block, der nur aus it besteht.
- `bool` P`.same_block(partition_item` it, `partition_item` it'`)` : Ist genau dann wahr, wenn it und it' im gleichen Block sind.
- P`.union_blocks(partition_item` it, `partition_item` it'`)` : Vereinigt die beiden Blöcke.
- `partition_item` P`.find(partition_item` it`)` : Gibt das kanonische Element des Blocks zurück, zu dem it gehört.

  Das *kanonische Element* ist ein ausgezeichnetes Element eines Blocks und wird bestimmt von der Implementierung bei der Bildung des Blocks.
- `int` P`.size(partition_item` it`)` : Gibt die Größe des Blocks zurück, zu dem it gehört.

6.3.2 Implementierung

- `partition` ist als Wald realisiert. Jeder Block ist ein Baum.

```
class node{
   node * parent;
   int size;
};
```

- `partition_item` ist ein Zeiger auf *node*.
- Die Operation `make_block` erzeugt einen neuen Knoten, setzt *size* auf 1 und gibt den Zeiger auf den Knoten zurück.
- Ein Block wird als Baum dargestellt. Jede Wurzel kennt die Größe ihres Baumes. Das kanonische Element eines Blocks ist die Wurzel des Blocks.
- **Naive** Version von `find` :

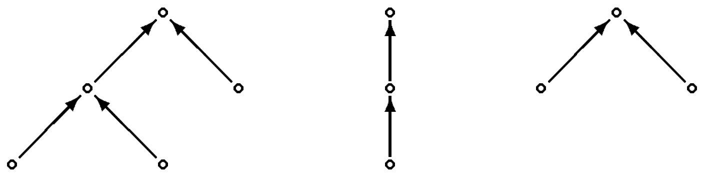


Abbildung 6.1: Eine Partition von 11 Elementen in 3 Blöcken.

partition_item find(partition_item *it*)
begin
  while $it \rightarrow parent \neq$ `nil`
  do
    $it \;\leftarrow\; it \rightarrow parent;$
  od
  return it;
end

- **Naive** Version von `union_blocks` :

union_blocks(partition_item *a*, partition_item *b*)
begin
  `partition_item` $x \leftarrow$ `find`(a);
  `partition_item` $y \leftarrow$ `find`(b);
  if $x = y$ **then return**; **fi**
  $x \rightarrow parent \;\leftarrow\; y;$
  $y \rightarrow size \;\leftarrow\; x \rightarrow size + y \rightarrow size;$
end

Diese beiden naiven Implementierungen sind schlecht, da im schlimmsten Fall nach mehreren `union` Operationen eine lineare Liste entstehen kann, und dann bei `find` eventuell die ganze Liste durchlaufen werden muß.

6.3.3 Bessere Varianten

1. **Weighted union rule :**

   Wir machen die leichtere Wurzel zum Kind der schwereren Wurzel. Dies stellt sicher, daß nur Bäume mit Tiefe $O(\lg n)$ entstehen können.

   union_blocks(partition_item *a*, partition_item *b*)
   begin
     if $(x \rightarrow size \leq y \rightarrow size)$
     then
       { wie gehabt }
     else
       { x und y vertauscht }

fi

end

2. **Path compression (Pfadkomprimierung) :**

   Wir laufen den Pfad zweimal von *it* bis zur Wurzel hoch. Beim zweiten Mal setzen wir den Vater aller Vorgänger von *it* auf die Wurzel. Damit ist später für alle Vorgänger it' von *it* die Laufzeit von **find**(it') $O(1)$.

Bei der Verwendung von Methode 1 und 2 zusammen ist die Laufzeit fast linear, nämlich $O((n+m)\cdot\alpha(m,n))$. Dabei ist $\alpha(m,n)$ das *Inverse der Ackermannfunktion* $a(n,m)$; diese Funktion wächst extrem langsam.

Satz 6.3.1 *Die Laufzeit für* n `union` *und* m `find` *Operationen in beliebiger Reihenfolge auf einer Partition von* n *Elementen ist :*

Naives `union`, *naives* `find` *:*	$O(n+m\cdot n)$
Gewichtetes `union`, *naives* `find` *:*	$O((n+m)\cdot\lg n)$
Naives `union`, `find` *mit Pfadkomprimierung :*	$O((n+m)\cdot\lg n)$
Gewichtetes `union`, `find` *mit Pfadkomprimierung :*	$O((n+m)\cdot\alpha(m,n))$
	$\leq O((n+m)\cdot\lg^* n)$.

□

Dabei ist $\lg^* n$ die minimale Anzahl der Anwendungen von lg auf n, damit ein Wert kleiner gleich 1 entsteht. $\lg^* n$ wächst ebenfalls extrem langsam und es ist $\alpha(m,n)\leq\lg^* n$.
Die erste Laufzeitschranke des Satzes ist klar. Wir werden in dieser Vorlesung nur die zweite und dritte Laufzeitschranke beweisen. Die vierte Laufzeitschranke geht im Prinzip analog zu dem Beweis der dritten Schranke, ist aber im Detail wesentlich komplizierter.

Satz 6.3.2 *Die Verwendung der gewichteten Vereinigungsregel garantiert, daß kein Baum höher als* $\log n$ *wird.*

Beweis: Seien $w(x)$ die Anzahl der Nachkommen von x und $h(x)$ die Höhe von x. Für jede Wurzel x gilt :

$$w(x)\geq 2^{h(x)}\,.$$

Daraus folgt sofort :

$$h(x)\leq\log w(x)\leq\log n\,.$$

Wir führen Induktion über die Anzahl der Vereinigungen. Vor der ersten Vereinigung gilt : $h(x)=0$ für alle Wurzeln x und $w(x)=1$.
Seien h' und w' die Höhe bzw. das Gewicht nach der Vereinigung. Dann ist

$$\begin{aligned} w'(x) &= w(x)+w(y) \quad \text{und}\\ h'(x) &= \max(h(x),1+h(y))\,. \end{aligned}$$

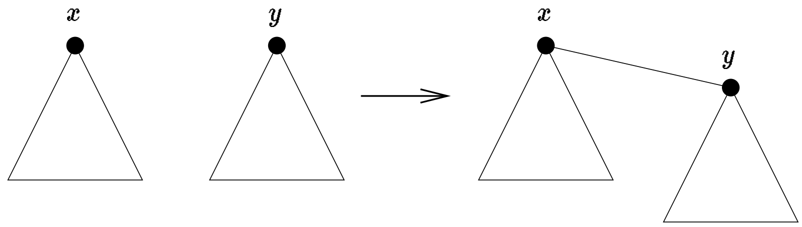


Abbildung 6.2: Vereinigung zweier Bäume mit $w(x) \geq w(y)$.

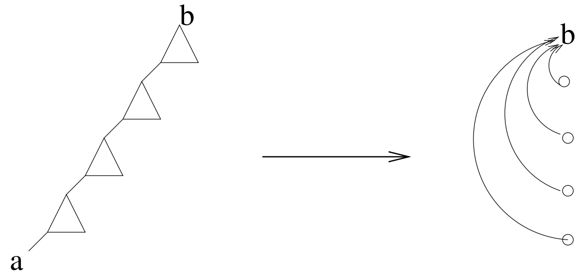


Abbildung 6.3: `partial_find`(a, b).

- **Fall 1 :** $h'(x) = h(x)$.
  Dann ist $w'(x) \geq w(x) \geq 2^{h(x)} = 2^{h'(x)}$.
- **Fall 2 :** $h'(x) = h(y) + 1$.
  Dann ist $w'(x) = w(x) + w(y) \geq 2w(y) \geq 2 \cdot 2^{h(y)} = 2^{h'(x)}$. □

Satz 6.3.3 *Pfadkomprimierung garantiert, daß die Kosten von m* `find` *und n* `union` *Operationen in beliebiger Reihenfolge beschränkt sind durch $O((n+m) \cdot \log n)$.*

Wir betrachten eine Folge von n `union` und m `find` Operationen. Sei T der Baum, der durch die `union` Operationen aufgebaut würde, wenn wir die m `find` Operationen nicht ausführen würden. Auf T führen wir dann anschließend `partial_find` Operationen aus. `partial_find` hat als Argumente zwei Knoten a und b, wobei b ein Vorfahr von a ist. Wir komprimieren dann den Pfad von a nach b. Die Kosten sind gleich der Anzahl der Kanten auf dem Pfad von a nach b. Offensichtlich kann jedes ursprüngliche `find` durch ein `partial_find` gleicher Kosten simuliert werden.
Also folgt Satz 6.3.3, wenn wir den folgenden Satz bewiesen haben. Beachte, daß zwar jedes ursprüngliche `find` einem `partial_find` entspricht, aber es gibt auch `partial_find` Operationen, die nicht als `find` in der ursprünglichen Datenstruktur interpretiert werden können.

Satz 6.3.4 *Sei T ein beliebiger Baum mit n Knoten. Dann sind die Kosten von m beliebigen* `partial_find` *Operationen $O((n+m) \cdot \log n)$.*

Beweis: Wir numerieren die Knoten von T mit den Zahlen 1 bis n, so daß die Zahlen entlang eines jeden Pfades aufsteigen.

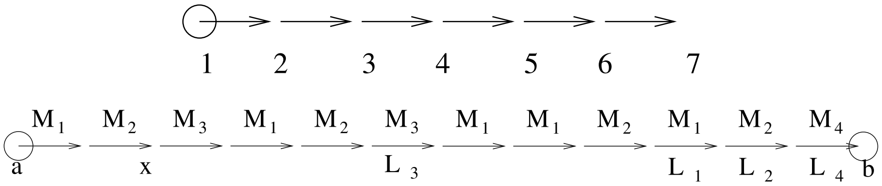


Abbildung 6.4: Ein Beispiel für die Mengen M_i und L_i.

Sei F die Menge aller Kanten, die insgesamt von den `partial_find` Operationen durchlaufen werden. Sei

$$M_i = \{(x,y) \in F | 2^i \leq \underbrace{y - x}_{= \text{Länge der Kante}} < 2^{i+1}\} ,$$

wobei $i = 0, 1, \ldots, \log n$ ist.
Sei

> $L_i = \{(x,y) \in M_i$ | Bei demjenigen `partial_find`, das die Kante (x,y) zerstört, ist (x,y) die letzte Kante (= die oberste) in M_i, die von diesem `partial_find` durchlaufen wird$\}$.

Die Gesamtkosten aller `partial_find` Operationen sind dann gerade $\sum_i |L_i| + \sum_i |M_i \setminus L_i|$. Die beiden folgenden Lemmas zeigen, daß diese Summe beschränkt ist durch $O((n+m) \cdot \log n)$. □

Lemma 6.3.5

$$\sum_i |L_i| \leq m \cdot (1 + \lceil log\ n \rceil) .$$

Beweis: Jedes `partial_find` kann höchstens eine Kante zu L_i beitragen. Also ist

$$\sum_i |L_i| \leq m \cdot (\text{ Anzahl der } L_i) = m(1 + \lceil lg\ n \rceil) .$$

□

Lemma 6.3.6

$$\sum_i |M_i \setminus L_i| \leq n \cdot (1 + \lceil lg\ n \rceil) .$$

Beweis: Wir zeigen, daß es für jedes x und jedes i höchstens eine Kante $(x,y) \in M_i \setminus L_i$ gibt.
Dazu betrachten wir ein beliebiges `partial_find`, das (x,y) enthält. Da $(x,y) \in M_i \setminus L_i$ ist, gibt es noch ein $(u,v) \in L_i$ im gleichen `partial_find`, aber später.

$$x \to y \to \cdots \to u \to v \to \cdots$$

Für jede Kante (x,z), die durch dieses oder ein späteres `partial_find` erzeugt wird, gilt $z \geq v$. Also ist :

$$z - x \geq v - x \geq v - u + y - x \geq 2^i + 2^i \; .$$

Damit ist aber $(x, z) \notin M_i$. $\square$

Kapitel 7

Minimale aufspannende Bäume

7.1 Graphen

Definition 7.1.1 Ein *Graph G* ist ein Paar (V, E), wobei

- V eine endliche Menge von *Knoten (vertices)* und
- E eine endliche Menge von Kanten (edges) ist. $\square$

Bem.: Darstellung von Graphen im Speicher kommt später. $\square$

7.1.1 Gerichteter Graph

- Für jede Kante haben wir zwei Funktionen :
  - $source : E \rightarrow V$
    Liefert den *Startknoten* einer Kante.
  - $target : E \rightarrow V$
    Liefert den *Endknoten* einer Kante.

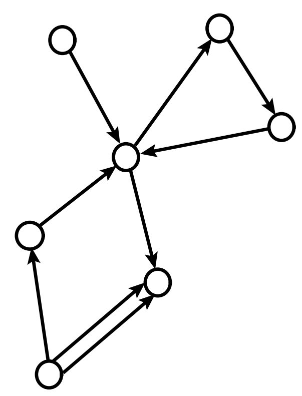


Abbildung 7.1: Ein gerichteter Graph.

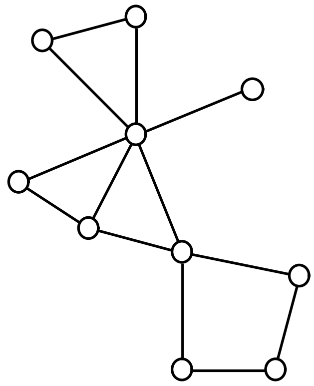


Abbildung 7.2: Ein ungerichteter Graph.

- **Schreibweise:** $e = (v, w)$ für eine Kante e mit $source(e) = v$ und $target(e) = w$.
- Ein *Pfad* p ist eine Folge von Kanten $[e_0, e_1, ..., e_k]$, wobei $target(e_i) = source(e_{i+1})$ für $0 \leq i < k$ ist. Wir definieren $source(p) = source(e_0)$ und $target(p) = target(e_k)$.
- **Schreibweise :** $x \underset{F}{\longrightarrow}^* y$,

  wenn F eine Menge von Kanten ist, x, y Knoten und es gibt einen Pfad von x nach y, auf dem nur Kanten in F benutzt werden.

7.1.2 Ungerichteter Graph

Bem.: In ungerichteten Graphen soll gelten :

$$\forall e : source(e) \neq target(e) ,$$

d.h., es gibt keine Selbstschleifen. □

- **Schreibweise :** $e = \{u, v\}$ für eine Kante e.
- Ein *Pfad* p ist eine alternierende Folge von Knoten und Kanten

  $[v_0, e_1, v_1, ..., e_k, v_k]$, wobei $e_i = \{v_{i-1}, v_i\}$ ist.
- **Schreibweise :** $x \underset{F}{\sim}^* y$,

  wenn F eine Menge von Kanten ist, x, y Knoten und es gibt einen Pfad von x nach y, auf dem nur Kanten in F benutzt werden.

7.1.3 Darstellung von gerichteten Graphen

Wir bilden für jeden Knoten und jede Kante einen Verbund (record, structure, ...).

- **Die Struktur zu einem Knoten enthält :**

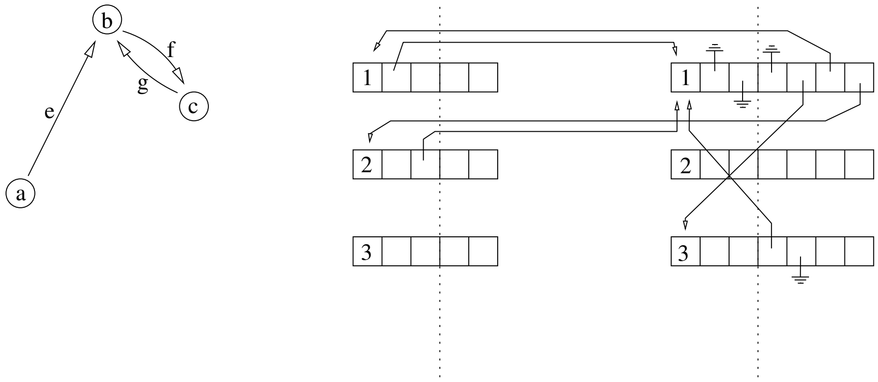


Abbildung 7.3: Interne Darstellung eines Graphen.

v1) Eine Nummer, die den Knoten eindeutig identifiziert.

v2) Einen Zeiger auf die Liste aller ausgehenden Kanten.

v3) Einen Zeiger auf die Liste aller einmündenden Kanten.

v4) Einen Zeiger für die Liste aller Knoten.

- **Die Struktur zu einer Kante enthält :**

e1) Eine Nummer, die die Kante eindeutig identifiziert.

e2) Einen Zeiger auf die Struktur für den Quellknoten.

e3) Einen Zeiger auf die Struktur für den Zielknoten.

e4) Einen Zeiger für die beiden Listen v2) und v3).

Diese Darstellung unterstützt :

1. Die Iteration über alle Knoten in Zeit $O(n)$.

2. Die Iteration über die von einem Knoten ausgehenden Kanten in Zeit $O(outdeg(v))$, wobei $outdeg(v)$ die Anzahl der ausgehenden Kanten von v ist.

3. Zugriff auf die Quelle und die Senke einer Kante in Zeit $O(1)$.

Die durch die Knoten und Kanten indizierten Felder können als Array oder als Hashtabelle dargestellt werden. Es ergeben sich dann folgende Kosten :

- Array : Zugriff in Zeit $O(1)$, Initialisierung in Zeit O(Länge des Intervalls der benutzten Nummern).

- Hashing : Zugriff in erwarteter Zeit $O(1)$, Initialisierung in Zeit $O(1)$.

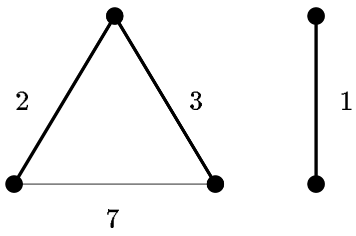


Abbildung 7.4: Die dicken Kanten bilden einen minimalen aufspannenden Wald.

7.2 Minimale aufspannende Bäume

7.2.1 Definitionen

Definition 7.2.1 Sei G ein ungerichteter Graph. Dann heißt $F \subseteq E$ *aufspannend*, wenn für alle Knoten x, y gilt : $x \underset{E}{\sim}^* y \Leftrightarrow x \underset{F}{\sim}^* y$. F heißt *aufspannender Wald*, wenn F auspannend ist und keine echte Teilmenge von F aufspannend ist. □

Sei $c : E \to I\!R$ eine *Kostenfunktion* (oder auch *Gewichtsfunktion*) auf den Kanten. Für $F \subseteq E$ definieren wir die *Kosten von F* als $c(F) = \sum_{e \in F} c(e)$. Wenn $e = (u, v)$ oder $e = \{u, v\}$ ist, dann schreiben wir auch $c(u, v)$ oder $c(\{u, v\})$ statt $c(e)$.

Definition 7.2.2 Ein *minimaler aufspannender Wald* ist ein aufspannender Wald von minimalem Gewicht. □

Notation : Für eine Menge F und ein Element e schreiben wir meist

- $F + e$ für $F \cup \{e\}$ und
- $F - e$ für $F \setminus \{e\}$. □

Lemma 7.2.3 (Zyklusregel)
Sei C ein Zyklus in G und $e \in C$ eine Kante, so daß $c(e) > c(e')$ für alle $e' \in C$, $e' \neq e$, ist. Dann gibt es keinen minimalen aufspannenden Wald, der e enthält.

Beweis: (Durch Widerspruch)
Sei F ein minimaler aufspannender Wald, der e enthält. Sei T der Baum in F, der e enthält. Dann besteht $T - e$ aus zwei Bäumen T_1 und T_2. Der Zyklus C muß mindestens eine Kante e', $e' \neq e$, enthalten, die T_1 und T_2 verbindet. Also ist $F - e + e'$ ein aufspannender Wald (siehe Abbildung 7.5). Ferner gilt

$$c(F - e + e') = c(F) - c(e) + c(e') < c(F)\,.$$

Das ist ein Widerspruch. □

Bem.: Ersetzt man im Lemma das '$>$' durch '$\geq$', dann gilt folgendes :

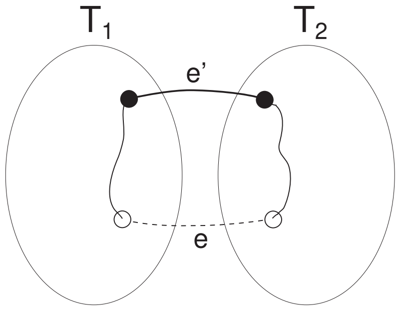


Abbildung 7.5: Skizze zur Zyklusregel.

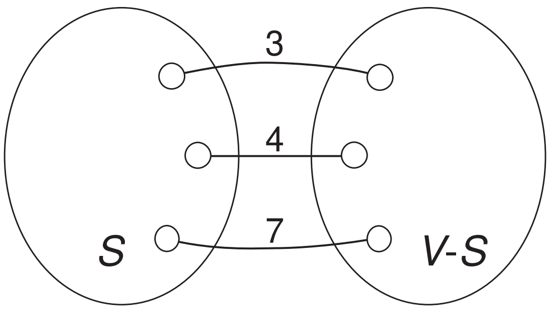


Abbildung 7.6: Beispielgraph zur Schnittregel. Die Kante mit Kosten 3 ist in jedem minimalen aufspannenden Wald enthalten.

Sei e eine Kante und sei C ein Kreis in G, so daß $c(e) \geq c(e')$ für alle $e' \in C$, $e' \neq e$, ist. Dann gibt es einen minimalen aufspannenden Wald, der e nicht enthält. □

Lemma 7.2.4 (Schnittregel)
Sei e eine Kante und S eine Teilmenge von V mit $|S \cap e| = 1$ (genau ein mit e verbundener Knoten liegt in S) und $c(e) < c(e')$ für jede Kante $e' \in E$ mit $e' \neq e$ und $|S \cap e'| = 1$. Dann ist e in jedem minimalen aufspannenden Wald enthalten.

Beweis: (Durch Widerspruch)
Sei $e = \{x, y\}$ und sei F ein minimaler aufspannender Wald, der e nicht enthält. Wegen $x \underset{E}{\sim}^* y$ gilt auch $x \underset{F}{\sim}^* y$. Sei p ein Pfad von x nach y in F. Dann muß es auf diesem Pfad eine Kante e' geben mit $|S \cap e'| = 1$. Sei $F' = F - e' + e$.
Dann ist F' aufspannend. Dies sieht man wie folgt : Seien u und v Knoten mit $u \underset{F}{\sim}^* v$. Sei q ein Pfad mit Kanten aus F, der u und v verbindet. Falls q e' nicht benutzt, dann gibt es q auch in F'. Andernfalls ist $q = q_1 \circ e' \circ q_2$, wobei q_1 ein Pfad von u zu einem Endpunkt von e' und q_2 ein Pfad zum anderen Endpunkt von e' ist. Wir zerlegen auch p an der Kante e', also $p = p_1 \circ e' \circ p_2$. Dabei nehmen wir o.B.d.A. an, daß p e' in der gleichen Richtung durchläuft wie q (Beachte : Der Graph ist ungerichtet !). Dann ist

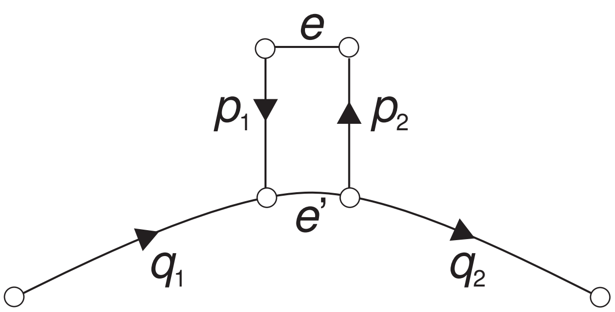


Abbildung 7.7: Skizze zum Beweis der Schnittregel.

$$q_1 \circ (p_1)^{rev} \circ e \circ (p_2)^{rev} \circ q_2$$

ein Pfad in F' von u nach v (siehe Abbildung 7.7).

Ferner ist $c(F') = c(F) - c(e') + c(e) < c(F)$, was ein Widerspruch ist. $\square$

7.2.2 Algorithmen

Wir betrachten nun zwei Algorithmen, um minimale aufspannende Wälder zu berechnen.

Algorithmus 1 (Kruskal)

begin
  Sortiere die Kanten aufsteigend nach Kosten,
    so daß $c(e_1) \le c(e_2) \le ... \le c(e_m)$ ist;
  $F \leftarrow \emptyset$;
  for $i \leftarrow 1$ **to** m
  do
    Sei $e_i = \{u, v\}$;
    if $(u \underset{F}{\sim}^* v)$ **then** Tue nichts;
    else $F \leftarrow F + e$;
    fi
  od
end

Lemma 7.2.5 *Der Algorithmus von Kruskal ist korrekt.*

Beweis: Seien $e_1, e_2, \ldots, e_m$ die Kanten sortiert nach Gewicht, d.h.

$$c(e_1) \le c(e_2) \le \ldots, \le c(e_m)\,.$$

Fall 1: Alle Kantenkosten sind paarweise verschieden.

Beh.: Der Algorithmus konstruiert einen aufspannenden Wald F.

Beweis: (indirekt)

Wir nehmen an, es gäbe Knoten x und y mit $x \underset{E}{\sim}^* y$ und $x \underset{F}{\not\sim}^* y$. Dann gibt es insbesondere auch Knoten x und y mit $\{x, y\} \in E$ und $x \underset{F}{\not\sim}^* y$. Aber wenn wir die Kante $\{x, y\}$ betrachten, dann würden wir sie zu F hinzunehmen. Dies ist ein Widerspruch. □

Der Algorithmus konstruiert einen *minimalen* aufspannenden Wald, da alle Kanten, die nicht nach F kommen, nach Lemma 7.2.3 zu keinem minimalen aufspannenden Wald gehören, und alle Kanten, die nach F kommen, nach Lemma 7.2.4 zu jedem minimalen aufspannenden Wald gehören.

Fall 2: Gleiche Kantenkosten sind erlaubt.

Wir verwenden folgendes Perturbationsargument :

Betrachte Kantenkosten c' mit $c'(e_i) = c(e_i) + i \cdot \varepsilon$, wobei $\varepsilon > 0$ ist. Dann gilt :

$$c'(e_1) < c'(e_2) < \ldots < c'(e_m) .$$

Sei F_0 der bzgl. c' konstruierte Wald. Das ist der *gleiche* Wald, der bzgl. c konstruiert wurde, denn nach dem Sortieren hängen die Entscheidungen des Algorithmus nicht mehr von den Gewichten ab. Nach Fall 1 ist F_0 optimal bzgl. c', d.h. $c'(F_0) \leq c'(F)$ für jeden aufspannenden Wald F. Es ist

$$c'(F_0) = c(F_0) + l_0 \cdot \varepsilon$$

für eine natürliche Zahl l_0. Analog ist

$$c'(F) = c(F) + l \cdot \varepsilon$$

für eine natürliche Zahl $l \geq l_0$. Also ist

$$c(F_0) \leq c(F) + (l - l_0) \cdot \varepsilon .$$

Da diese Ungleichung für jedes $\varepsilon > 0$ gilt, folgt

$$c(F_0) \leq c(F) .$$

□

Laufzeit: Seien $m = |E|, n = |V|$ und $n \leq m$. Dann brauchen wir Zeit :

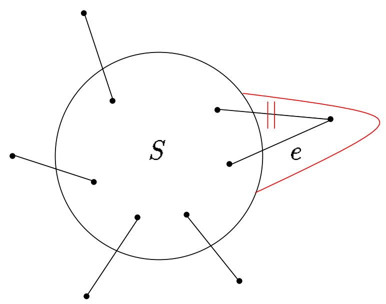
- Sortieren : $m \log m$.

- Union-Find :

  Wir kreieren n Blöcke und machen $n-1$ `union` und m `find` Operationen. Nach Satz 6.3.1 geht dies in Zeit $O(n + m \log n)$, wenn wir die gewichtete union-Regel verwenden.

Insgesamt ergibt sich damit der folgende Satz.

Satz 7.2.6 *Der Algorithmus von Kruskal läuft in Zeit $O(m \log m)$.* □



Algorithmus 2 (Prim)

// läßt den Baum wachsen, startend von einem Knoten s

begin

$S \leftarrow \emptyset$;

while $(S \neq V)$

do

Sei s ein beliebiger Knoten in $V \setminus S$;

$S \leftarrow S + s$;

while (es gibt Kante e mit $|S \cap e| = 1$)

do

Wähle unter den Kanten e mit $|S \cap e| = 1$ eine Kante von minimalem Gewicht und füge denjenigen Endpunkt von e, der nicht zu S gehört, zu S dazu;

od

od

end

Satz 7.2.7 *Der Algoriothmus von Prim ist korrekt.*

Beweis: Dies folgt direkt aus Lemma 7.2.4 (Schnittregel). Die innere Schleife des Algorithmus berechnet den Baum für eine Zusammenhangskomponente des Graphen, die äußere Schleife sorgt für die Bearbeitung aller Zusammenhangskomponenten. □

Implementierung (Pseudo-Code) :

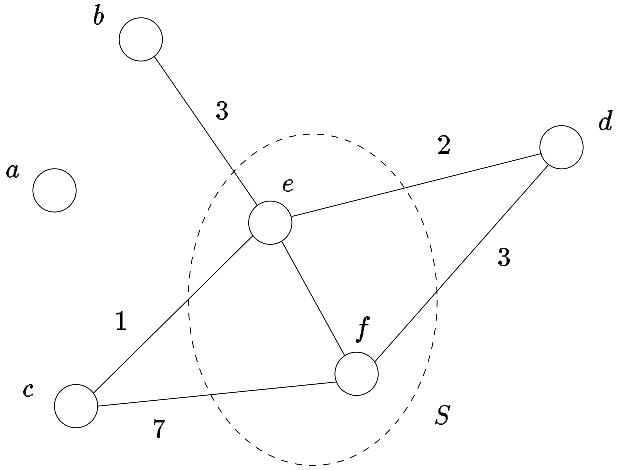
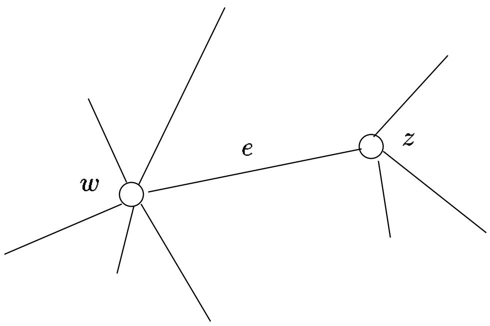


Abbildung 7.8: PQ enthält $(d, 2), (c, 1)$ und $(b, 3)$.

Wir benutzen eine Warteschlange PQ, die alle Knoten $w \notin S$ enthält, für die es eine Kante gibt, die w mit S verbindet. Die Priorität von w ist das kleinste Gewicht einer solchen Kante. Dann wählen wir in jeder Iteration den Knoten mit minimaler Priorität. Sei dies etwa w. Dann :



forall zu w inzidenten Kanten (w, z)

do

  if $(z \notin PQ)$

  then Füge $(z, c(e))$ zu PQ dazu;

  else Sei c' die momentane Priorität von z. Falls $c(e) < c'$, dann erniedrige die Priorität von z auf c'.

  fi

od

Implementierung (LEDA) :

```
node_array<bool>     in_S(G, false);
node_array<pq_item>  ITEM(G, nil);
```

```
pq_queue<NT, edge>  PQ;              // NT = Number-Type, z.B. int
list<edge>          F;
pq_item             it;
edge                e;
node                s, v, w;

forall_nodes (s, G) {
  if (in_S[s]) continue;
  in_S[s] = true;

  forall_adj (e, s) {
    w = G.opposite (e, s);           // anderer Endpunkt
    if (in_S[w]) continue;
    if (ITEM[w] == nil) {
      PQ.insert (c[e], e);
    } else {
      if (c[e] < PQ.prio (ITEM[w])) {
        PQ.decrease_p (ITEM[w], c[e]);
        PQ.change_inf (ITEM[w], e);
      }
    }
  }

  while (!PQ.empty()) {
    it = PQ.find_min ();
    e = PQ.inf (it);
    F.append (e);
    v = G.source (e);
    w = G.target (e);
    if (in_S[w]) LEDA_swap (v, w);
    in_S[w] = true;
    PQ.del_item (it);
    ITEM[w] = nil;
  }

  forall_adj (e, s) {
    w = G.opposite (e, s)            // anderer Endpunkt
    if (in_S[w]) continue;
    if (ITEM[w] == nil) {
      PQ.insert (c[e], e);
    } else {
      if (c[e] < PQ.prio (ITEM[w])) {
        PQ.decrease_p (ITEM[w], c[e]);
        PQ.change_inf (ITEM[w], e);
      }
    }
  }
```

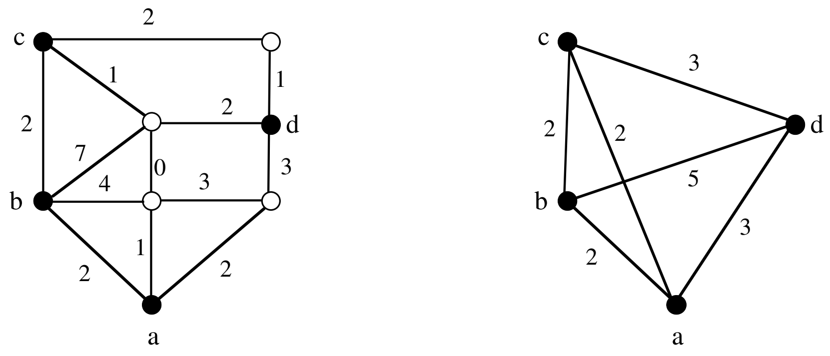


Abbildung 7.9: Links ein Graph G mit Terminalknoten $T = \{a, b, c, d\}$. Rechts der Graph G^* mit Knotenmenge T und Kantengewichten gleich kürzestem Abstand in G.

}

Invariante :

- `in_S[v] = true` $\iff$ $v \in S$.
- `ITEM[v] =` $\begin{cases} \texttt{nil} & \text{falls } v \in S\text{ .} \\ \texttt{nil} & \text{falls } v \notin S \text{ und es gibt keine Kante } \{v, w\} \text{ mit } w \in S. \\ \text{Das item in } PQ, \text{ das } (e, c(e)) \text{ enthält} & \text{falls } v \notin S \text{ und es gibt eine Kante } \{v, w\} \text{ mit } w \in S.\ e \text{ ist dann unter allen solchen Kanten eine Kante mit kleinstem Gewicht.} \end{cases}$

7.3 Steinerbäume

Sei G=(V,E) ein ungerichteter zusammenhängender Graph, $c : E \to \mathbb{R}_{\geq 0}$ eine Gewichtsfunktion und seien $T \subseteq V$ Terminale.

Gesucht : Eine Teilmenge $F \leq E$ minimaler Kosten, so daß

$$\forall s, t \in T : s \underset{F}{\sim}^{*} t\,.$$

Die Lösung dieses Problems ist ein sogenannter *Steinerbaum*. Leider ist das Steinerbaum-Problem NP-vollständig. Aber mit Hilfe von minimalen aufspannenden Bäumen kann man Lösungen berechnen, die höchstens einen Faktor von zwei über dem Optimum liegen.

Steiner-Approx

begin

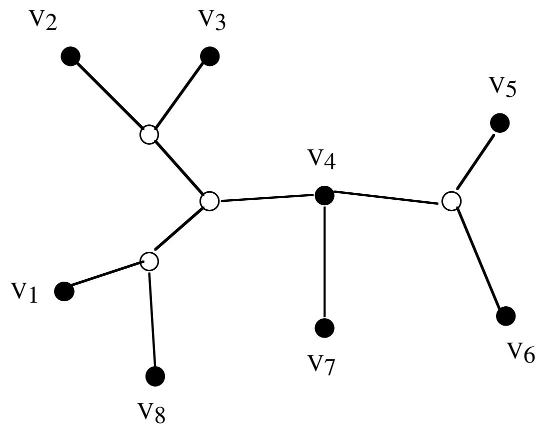
1. Sei $G^* = (T, E^*)$ der vollständige Graph mit Knotenmenge T. Jede Kante (u, v) in G^* entspricht irgendeinem kürzesten Weg $p_{u,v}$ von u nach v in G. Entsprechend definieren wir die Kosten der Kante (u, v) als
$$c^*(u, v) = c(p_{u,v}) .$$
2. Sei F^* ein minimaler aufspannender Baum in G^*.
3. Sei F' die Menge aller Kanten, die auf einem der Wege liegen, die den Kanten in $F*$ entsprechen, also
$$F' = \bigcup_{(u,v) \in F^*} p_{u,v} .$$
4. Sei F eine beliebige Teilmenge von F', die zyklenfrei ist und $u \underset{F}{\sim}^* v$ für alle $u, v \in T$ erfüllt.

end

Satz 7.3.1
Sei F die von `Steiner-Approx` *berechnete Kantenmenge und sei F_{opt} ein optimaler Steinerbaum. Dann ist $c(F) \leq 2 \cdot c(F_{opt})$.* `Steiner-Approx` *berechnet also eine 2-Approximation für das Steinerbaum-Problem.*

Beweis:

F_{opt}

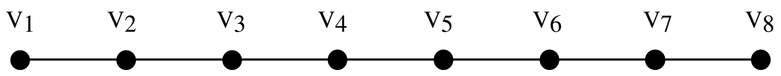


F_{opt} induziert in G die Pfade:

$$p_1 : v_1 \to^* v_2$$
$$p_2 : v_2 \to^* v_3$$
$$p_3 : v_3 \to^* v_4$$
$$p_4 : v_4 \to^* v_5$$

$$p_5 : v_5 \rightarrow^* v_6$$
$$p_6 : v_6 \rightarrow^* v_7$$
$$p_7 : v_7 \rightarrow^* v_8$$
$$p_8 : v_8 \rightarrow^* v_1$$

Die Gesamtkosten dieser Pfade sind höchstens $2 \cdot c(F_{opt})$.
Andererseits ist $c^*(v_i, v_{i+1}) \leq c(p_i)$, weil $c^*(v_i, v_{i+1})$ die Länge des kürzesten Weges von v_i nach v_{i+1} ist. Der folgende Baum in G^*



hat Kosten $c^*(v_1, v_2) + \ldots + c^*(v_7, v_8) \leq c(p_1) + \ldots c(p_8) \leq 2 \cdot c(F_{opt})$. Wir konstruieren aber den minimalen aufspannenden Baum F^* von G^*. Also ist

$$c(F) \leq c(F') \leq c^*(F^*) \leq 2 \cdot c(F_{opt}) \, .$$

$\square$

Kapitel 8

Tiefensuche

8.1 DFS

Wir möchten in einem gerichteten Graphen G alle Knoten mindestens einmal besuchen. Dazu machen wir *Tiefensuche (*`DFS` *= Depth First Search)*, d.h., wir starten in einem beliebigen Knoten und versuchen dann, möglichst lange einem Pfad neuer Knoten zu folgen; wenn wir einen bereits besuchten Knoten erreichen, machen wir Backtracking.
`DFS` benutzt die folgenden Knotenmarkierungen für Knoten v :

- $\mathit{dfsnum}[v]$: Zählt die Reihenfolge, in der die Knoten besucht werden; wird mit -1 initialisiert (bedeutet "nicht besucht").
- $\mathit{compnum}[v]$: Zählt die Reihenfolge, in der die Knoten abgeschlossen werden.
- $\mathit{dfscount}, \mathit{compcount}$: Die zugehörigen globalen Zählvariablen; werden mit 1 initialisiert.

`DFS` baut bei der Grapherkundung einen `DFS`-*Baum* T auf (genaugenommen ist T ein Wald mit je einem Baum für jede starke Zusammenhangskomponente von G). In diesem Baum sind genau diejenigen Kanten enthalten, deren Erforschung zu einem noch unbekannten Knoten geführt haben.

Definition 8.1.1 Sei (v, w) eine Kante in G. Wir nennen (v, w) eine

- *Baumkante*, falls $(v, w) \in T$.
- *Vorwärtskante*, falls $v \underset{T}{\longrightarrow}^* w$ und $(v, w) \notin T$.
- *Rückwärtskante*, falls $w \underset{T}{\longrightarrow}^+ v$.
- *Querkante*, falls weder $v \underset{T}{\longrightarrow}^* w$ noch $w \underset{T}{\longrightarrow}^* v$. □

Dadurch werden die Kanten von G in disjunkte Klassen von Vorwärts-, Rückwärts- und Querkanten eingeteilt.

DFS

```
dfs(v)
begin
    dfsnum[v] ← dfscount++;
    forall Kanten (v, w)
    do
        if dfsnum[w] = −1     // w noch unbesucht
        then
            dfs(w);     // Erkunde w
            T ← T + (v, w);     // (v, w) ist neue Baumkante
        fi
    od
    compnum[v] ← compcount++;     // w ist nun fertig
end

main()
begin
    dfscount ← 1;
    compcount ← 1;
    T ← ∅;
    forall Knoten v
    do
        if dfsnum[v] = −1
        then dfs(v); fi
    od
end
```

Satz 8.1.2
Auf einem Graphen $G = (V, E)$ hat `DFS` *Laufzeit* $O(|V| + |E|)$.

Beweis: Sei $|V| = n$ und $|E| = m$. Für jeden Knoten v wird genau einmal $\mathtt{dfs}(v)$ aufgerufen, also insgesamt n Aufrufe. Ein Aufruf kostet $O(outdeg(v))$. Also sind die Gesamtkosten

$$\begin{aligned} T &= O(n) + O\left(\sum_{v \in V} outdeg(v)\right) \\ &= O(n+m)\,. \end{aligned}$$

□

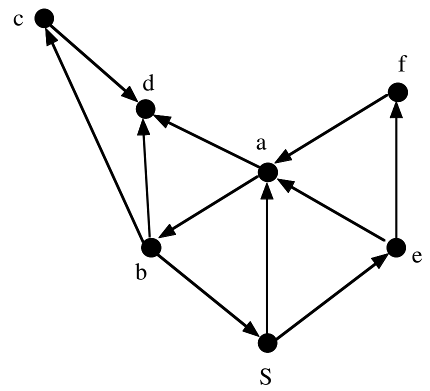


Abbildung 8.1: DFS, gestartet in S, besucht (zum Beispiel) die Knoten in der Reihenfolge $a, b, \ldots, f$.

Lemma 8.1.3 *Nach* DFS *gilt :*

(a) T ist ein Wald.

(b) Wenn $(v,w) \in T$ ist, dann ist

$$\begin{array}{rrcl} & \mathit{dfsnum}[v] & < & \mathit{dfsnum}[w] \\ \text{und} & \mathit{compnum}[v] & > & \mathit{compnum}[w]\,. \end{array}$$

(c) Wenn $(v,w_1),(v,w_2) \in T$ und $\mathit{dfsnum}[w_1] < \mathit{dfsnum}[w_2]$ ist, dann gilt :

$$\begin{array}{rrcl} & \max_{w \in T_1} \mathit{dfsnum}[w] & < & \min_{w \in T_2} \mathit{dfsnum}[w] \\ \text{und} & \max_{w \in T_1} \mathit{compnum}[w] & < & \min_{w \in T_2} \mathit{compnum}[w]\,. \end{array}$$

Dabei ist T_i der Unterbaum mit Wurzel w_i für $i = 1, 2$.

Beweis:

(a) Zu T kommen immer nur *neue* Knoten hinzu.

(b) $\mathit{dfs}(w)$ ist in $\mathit{dfs}(v)$ geschachtelt.

(c) w_2 wird erst erreicht, nachdem w_1 und T_1 fertig sind. □

Damit können wir die vier Typen von Kanten nach ihrer *dfsnum* und *compnum* klassifizieren.

Satz 8.1.4 *Sei $(v,w) \in E$. Dann gilt :*

(a) (v, w) ist Vorwärtskante oder Baumkante genau dann, wenn

$$\begin{array}{rrcl} & dfsnum[v] & \leq & dfsnum[w] \\ \textit{und} & compnum[v] & \geq & compnum[w]\,. \end{array}$$

Gleichheit gilt nur für Selbstschleifen.

(b) (v, w) ist Rückwärtskante genau dann, wenn

$$\begin{array}{rrcl} & dfsnum[v] & > & dfsnum[w] \\ \textit{und} & compnum[v] & < & compnum[w]\,. \end{array}$$

(c) (v, w) ist Querkante genau dann, wenn

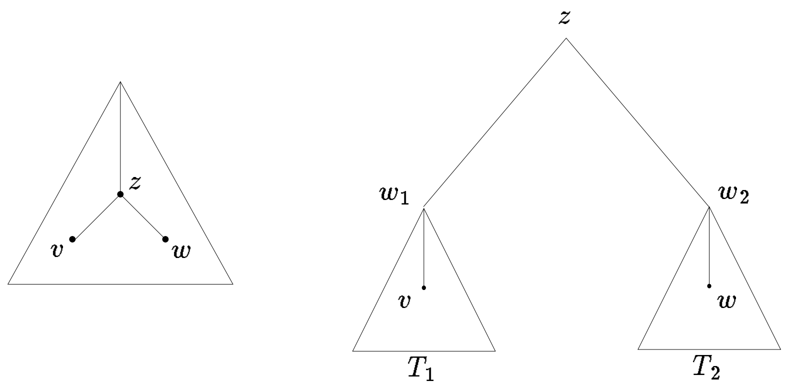
$$\begin{array}{rrcl} & dfsnum[v] & > & dfsnum[w] \\ \textit{und} & compnum[v] & > & compnum[w]\,. \end{array}$$

Beweis:

"$\Rightarrow$" : (a) Selbstschleife : Es gilt sogar Gleichheit.
Baumkante : Folgt aus Lemma 8.1.3(b).
Pfad $v \longrightarrow v_1 \longrightarrow v_2 \longrightarrow \ldots \longrightarrow w$: Induktion nach Pfadlänge.

(b) Umgekehrt wie Vorwärtskanten.

(c) Sei z der tiefste gemeinsame Vorfahr von v und w in T.



Da v später besucht wird als w, ist T_2 abgeschlossen, bevor w_1 erreicht wird. Die Behauptung folgt nun aus Lemma 8.1.3(c).

"$\Leftarrow$" : Wir müssen zeigen, daß der Fall $dfsnum[v] < dfsnum[w]$ und gleichzeitig $compnum[v] < compnum[w]$ unmöglich ist. Dann folgt die Behauptung aus dem Beweis für "$\Rightarrow$" zusammen mit der Beobachtung, daß jede Kante entweder Vorwärts-, Rückwärts- oder Querkante ist.

Sei $dfsnum[v] < dfsnum[w]$ für eine Kante (v, w). Dann wird v früher erreicht als w. Nun ist entweder $v \underset{T}{\longrightarrow}^* w$ und damit nach (a) $compnum[v] > compnum[w]$ oder w wird erst erreicht, nachdem v fertig ist. In diesem Fall ist aber auch $compnum[w] > compnum[v]$. Außerdem kann dies gar nicht sein, denn in v betrachten wir ja auch die Kante (v, w). □

8.2 Topologische Sortierung

Sei G ein gerichteter Graph ohne Selbstschleifen.

Definition 8.2.1 Eine Funktion $num : V \longrightarrow I\!R$ heißt *topologische Sortierung* von G, falls gilt

$$\forall (v, w) \in E : num(v) > num(w)\,.$$

□

Satz 8.2.2 *G besitzt eine topologische Sortierung genau dann, wenn G azyklisch ist.*

Beweis:

"⇒" : Sei num eine topologische Sortierung von G. Sei $v_1 \longrightarrow \cdots \longrightarrow v_k = v_1$ ein Zyklus. Dann ist $num(v_1) > num(v_2) > \cdots > num(v_k) = num(v_1)$. Widerspruch !

"⇐" : Wir geben zwei Beweise.

(1) Sei $i = 1$.
Wenn G azyklisch ist, dann exisitert ein Knoten v mit $outdeg(v) = 0$. Setze $num(v) = i$ und $i = i + 1$.
Entferne v aus G und wiederhole. Dann ist letztendlich für jeden Knoten v $num(v) > num(w)$ für alle vorher entfernten Knoten w und $num(v) < num(z)$ für alle noch nicht entfernten Knoten z. Da es von v aus nur Kanten zu den ersteren Knoten geben kann, ist num eine topologische Sortierung.

(2) Nach `DFS` ist $compnum[v]$ eine topologische Sortierung, denn es gibt in einem azyklischen Graphen keine Rückwärtskanten (und dann liefert Satz 8.1.4 die Behauptung). □

8.3 Starke Zusammenhangskomponenten

Definition 8.3.1 Ein gerichteter Graph G heißt *stark zusammenhängend*, wenn

$$\forall v, w \in V : \quad v \longrightarrow^* w \quad \text{und} \quad w \longrightarrow^* v\,.$$

Eine *starke Zusammenhangskomponente* ist ein maximaler Teilgraph von G, der stark zusammenhängend ist. □

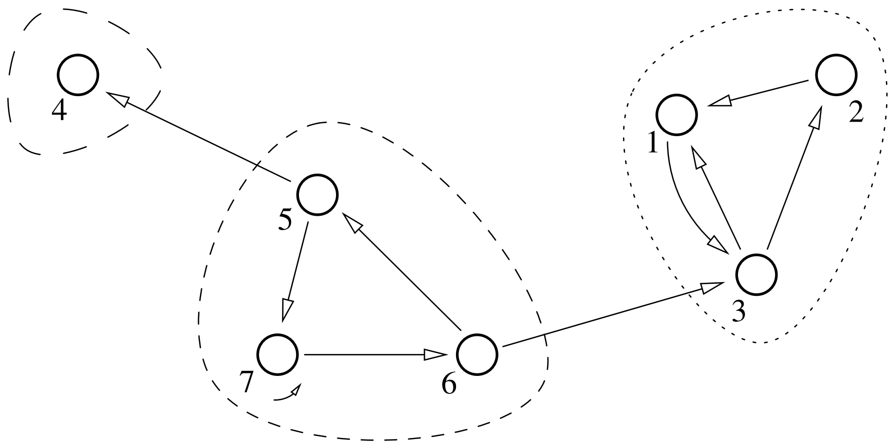


Abbildung 8.2: Beispiel für drei SZK's.

Zwei Knoten v und w eines gerichteten Graphen G gehören also zur gleichen SZK genau dann, wenn $v \longrightarrow^* w$ und $w \longrightarrow^* v$ gilt.
Der folgende Graph ist azyklisch :

- Knoten = SZK's von G.
- Kanten = $\{(C, C') \mid C$ und C' sind SZK's von G, $C \neq C'$ und es gibt eine Kante $(v, w) \in E$ mit $v \in C$ und $w \in C'\}$.

Das kann man sich so vorstellen, daß die Knoten jeder SZK zu einem einzigen Knoten zusammengezogen werden.

8.3.1 Algorithmen zur Berechnung der SZK's

Wir geben zwei Algorithmen.

Algorithmus 1 (der eleganteste, Sharir, 1985)

begin

1. Benutze `DFS`, um für jeden Knoten v *compnum*$[v]$ zu berechnen. Seien $v_1, \ldots, v_n$ die Knoten in aufsteigender Ordnung.
2. Drehe alle Kanten von G um (das erhält die SZK's). Nenne den resultierenden Graphen G'.
3. Markiere alle Knoten von G' als nicht betrachtet.
   while (Es gibt nicht betrachtete Knoten)
   do
   Sei v der nicht betrachtete Knoten mit der höchsten *compnum*. Markiere alle von v in G' erreichbaren Knoten, die noch nicht markiert sind. Sie bilden die v enthaltende SZK von G;
   od

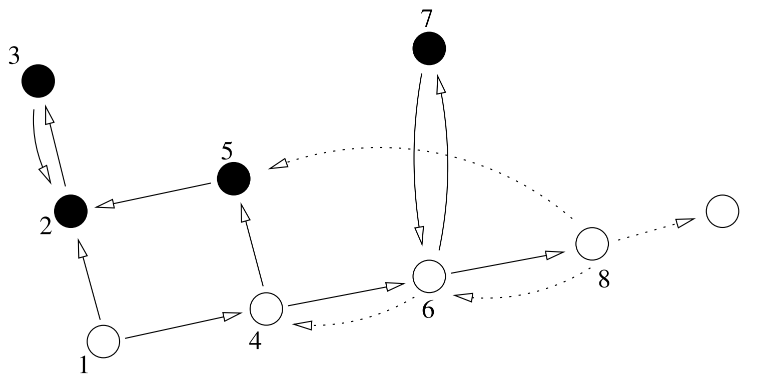


Abbildung 8.3: Berechnung der SZK's durch DFS. Die durchgezogenen Kanten sind schon abgearbeitet (wir sind also gerade im Knoten 8), die gestrichelten Kanten noch nicht. Die schwarzen Knoten sind bereits abgeschlossen.

end

Im Beispiel in Abb. 8.2 wird die Schleife für v_7, v_4 und v_3 durchgeführt. Die Korrektheit dieses Algorithmus werden wir in den Übungen zeigen.

8.3.2 Berechnung der SZK's durch DFS

Wir kommen nun zu einem zweiten Algorithmus zur Berechnung der SZK's. Dieser Algorithmus ist schneller als der Algorithmus von Sharir, weil er nur einmal den Graph durchlaufen muß. Er ist auch flexibler, weil man ihn leicht für andere Probleme modifizieren kann, etwa für die Berechnung der zweifachen Zusammenhangskomponenten eines Graphen.
Der Algorithmus macht im Prinzip einmal DFS. Wir benutzen folgende Bezeichnungen:

1. $G_c = (V, E_c)$ ist der jeweils aktuelle Teilgraph von G mit genau denjenigen Kanten $e \in E$, die DFS schon besucht hat.

2. Eine starke Zusammenhangskomponente C von G_c heißt *dauerhaft*, wenn der Aufruf von $dfs(v)$ für alle Knoten v von C schon abgeschlossen ist. Andernfalls heißt sie *vorläufig*.

3. Die *Wurzel* einer starken Zusammenhangskomponente C ist derjenige Knoten in C mit der kleinsten $dfsnum$.

Aus der Funktionsweise von DFS ergeben sich dann folgende Invarianten :

(I1) Es gibt keine Kante $(v, w) \in G$ mit : v liegt in einer dauerhaften Komponente und w liegt nicht in einer dauerhaften Komponente.

(I2) Die vorläufigen SZK's bilden einen Pfad, den wir P nennen.

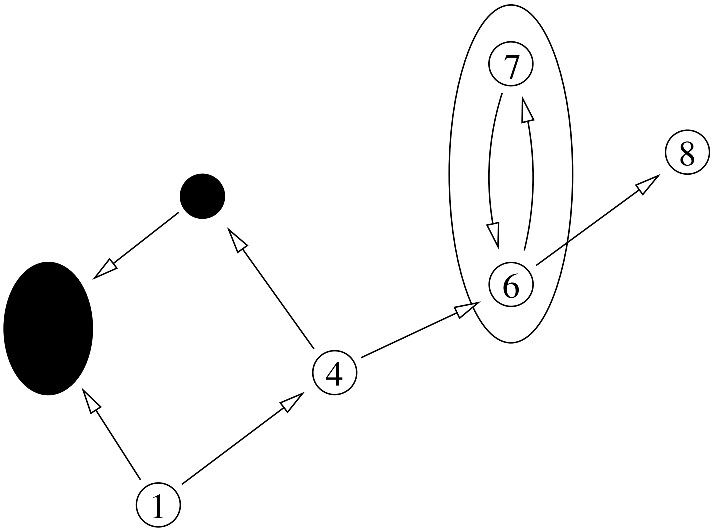


Abbildung 8.4: (Fortsetzung von Abb. 8.3) Die Knotenmengen $\{2,3\}$ und $\{5\}$ bilden jeweils dauerhafte SZK's. Die Mengen $\{1\}$, $\{4\}$, $\{6,7\}$ und $\{8\}$ bilden vorläufige SZK's.

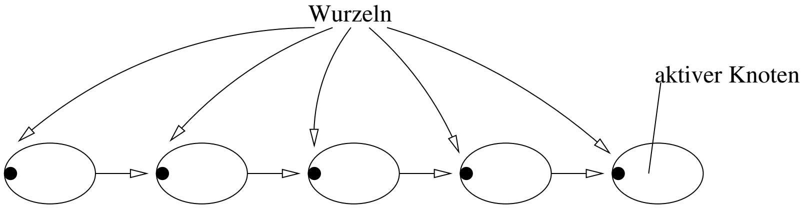


Abbildung 8.5: Der Pfad P der vorläufigen SZK's.

(I3) Wenn die SZK C vor der SZK C' auf P liegt und $v \in C$ und $w \in C'$ ist, dann ist $dfsnum[v] < dfsnum[w]$.

(I4) Für die Wurzeln v aller vorläufigen SZK's sind die Aufrufe $dfs(v)$ noch nicht abgeschlossen.

(I5) Der aktive Knoten (= oberster Knoten im Rekursionskeller) liegt in der letzten vorläufigen SZK auf P.

Dies hat zwei Konsequenzen :

1. Jede dauerhafte SZK von G_C ist auch eine SZK von G.

2. Wir können die vorläufigen SZK's mit Hilfe von zwei Kellern darstellen :

   - Der Keller aller Knoten, die in vorläufigen Komponenten liegen; aufsteigend geordnet nach $dfsnum$.
   - Der Keller aller Wurzeln von vorläufigen Komponenten; aufsteigend geordnet nach $dfsnum$.

Der zweite Keller ist offensichtlich eine Teilmenge des ersten Kellers. Im Beispiel von Abb. 8.3 enthält der erste Keller $(1, 4, 6, 7, 8)$, während der zweite Keller $(1, 4, 6, 8)$ enthält.
Die erste Eigenschaft sollten wir noch beweisen. Nehmen wir an, es gäbe eine dauerhafte SZK C von G_c, die keine SZK von G ist (C ist natürlich immer *Teilmenge* einer SZK von G). Dann gibt es Knoten $v \in C$ und $w \in G \setminus C$, so daß $v \underset{G}{\longrightarrow}^* w \underset{G}{\longrightarrow}^* v$ gilt. Sei etwa p ein Pfad in G von v nach w und q ein Pfad in G von w nach v. Der Kreis $p \circ q$ existiert nicht in G_c. Also muß auf $p \circ q$ mindestens ein Knoten liegen, für den `DFS` noch nicht abgeschlossen ist. Also enthält $p \circ q$ eine Kante (x, y) mit : x liegt in einer dauerhaften SZK, y liegt nicht in einer dauerhaften SZK. Dies ist ein Widerspruch zu (I1).

Wir können nun das Programm formulieren.

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

Ausgabe : Markierung der Knoten mit natürlichen Zahlen (*sccnum*) mit der Eigenschaft, daß $sccnum[v] = sccnum[w]$ genau dann gilt, wenn v und w in der gleichen SZK von G liegen.

Das Programm benutzt folgende Variablen und Datenstrukturen :

1. Keller :
   - *roots* : Die Wurzeln der vorläufigen SZK's.
   - *unfinished* : Die Knoten in den vorläufigen Komponenten.
2. Felder :
   - *dfsnum*, mit -1 initialisiert.
   - *sccnum*, mit -1 initialisiert.
3. Zähler :
   - *dfscount*, mit 0 initialisiert.
   - *sccnumcount*, mit 0 initialisiert.

scc_dfs(*node v*)

begin

$dfsnum[v] \leftarrow dfscount++;$

$unfinished.push(v);$

$roots.push(v);$

forall Kanten $e = (v, w)$

do

if $dfsnum[w] = -1$ // e ist Baumkante

```
        then scc_dfs(w);
        else          // e ist Vorwärts-, Rückwärts-, Querkante
           if sccnum[w] ≠ −1        //w in dauerhafter Komponente
           then Tue nichts;
           else                          //vereinige Komponenten
              while (dfsnum[roots.top()] > dfsnum[w])
              do roots.pop(); od
           fi
        fi
     od
     if v = roots.top()      // v ist Wurzel, Abschluß von scc_dfs(v)
     then                  // macht die oberste Komponente in P dauerhaft
        while dfsnum[unfinished.top()] ≥ dfsnum[v]
        do
           z ← unfinished.pop();
           sccnum[z] ← sccnumcount;
        od
        sccnumcount + +;
        roots.pop();
     fi
end


main()
begin
     dfscount = sccnumcount = 0;
     forall Knoten v ∈ V
     do
        dfsnum[v] = sccnum[v] = −1;
     od
     forall Knoten v ∈ V
     do
        if dfsnum[v] = −1
        then scc_dfs(v); fi
     od
end
```

Korrektheit:
Es genügt zu zeigen, daß (I1)–(I5) erhalten bleiben. Betrachten wir zunächst die Erforschung einer Kante $e = (v, w)$. Wir unterscheiden zwei Fälle :

1. e ist Baumkante :

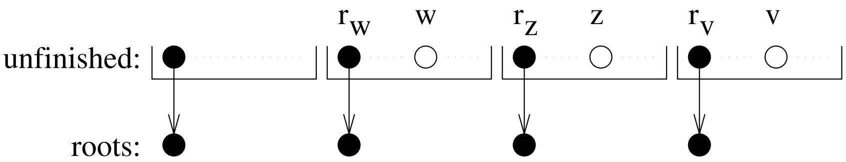
   Dann wird $\{w\}$ eine vorläufige SZK, und zwar die letzte auf dem Pfad P.

2. e ist keine Baumkante :

   Dann müssen wir zwei Fälle unterscheiden :

   (a) w liegt in einer dauerhaften Komponente :
   Aus (1) folgt, daß sich die SZK's von G_c nicht ändern.

   (b) w liegt in einer vorläufigen Komponente :
   Sei r_w die Wurzel der SZK, die w enthält, r_v die Wurzel der SZK, die v enthält, und sei z ein beliebiger Knoten in einer vorläufigen Komponente mit $dfsnum[r_w] \leq dfsnum[z]$. Sei r_z die Wurzel der Komponente, die z enthält.



   Es gilt in G_c :

   i. $v \longrightarrow^* r_v \longrightarrow^* v$,
   $w \longrightarrow^* r_w \longrightarrow^* w$ und
   $z \longrightarrow^* r_z \longrightarrow^* z$.

   ii. $dfsnum[r_w] \leq dfsnum[r_z]$ wegen (I3).
   Also $r_w \underset{T}{\longrightarrow}^* r_z$.

   iii. $dfsnum[r_z] \leq dfsnum[r_v]$, da nach (I5) r_v die Wurzel mit der größten $dfsnum$ ist. Also gilt nach (I3) : $r_z \underset{T}{\longrightarrow}^* r_v$.

   Insgesamt gilt also

   $$v \longrightarrow^* w \longrightarrow^* r_w \longrightarrow^* r_z \longrightarrow^* z \longrightarrow^* r_z \longrightarrow^* r_v \longrightarrow^* v$$

   Also ist es in Ordnung, alle SZK's, deren Wurzeln z $dfsnum[z] \geq dfsnum[r_w]$ erfüllen, mit der SZK mit Wurzel r_w zu vereinigen. Die Komponenten auf P vor der SZK mit Wurzel r_w bleiben unverändert.

Rückkehr aus dfs **:** Es passiert nur etwas, wenn wir eine Wurzel abschließen. Nach Definition einer Wurzel sind alle Knoten einer SZK Nachfolger ihrer Wurzel. Daher ist die Wurzel der letzte Knoten einer SZK, der abgeschlossen wird. Nach (I5) ist die Wurzel die oberste Wurzel in *roots* und die SZK besteht aus

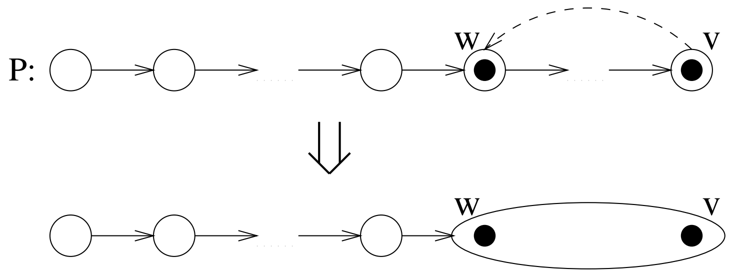


Abbildung 8.6: Vereinigen von Komponenten.

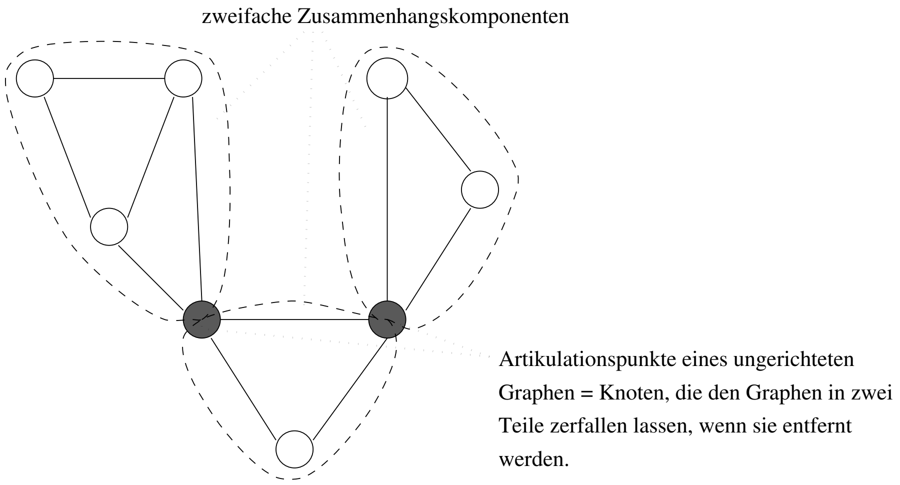


Abbildung 8.7: Zweifache Zusammenhangskomponenten eines Graphen.

allen Knoten in $unfinished$, deren $dfsnum$ größer oder gleich der $dfsnum$ der Wurzel ist. Diese Knoten müssen wir also aus $unfinished$ entfernen. □

Daraus erhalten wir den folgenden Satz.

Satz 8.3.2 *Wir können die SZK's eines Graphen $G = (V, E)$ in Zeit $O(|V| + |E|)$ berechnen.*

Beweis: Die Zeit außerhalb der **while**-Schleifen ist $O(n+m)$. Jede Ausführung eines Rumpfes einer der **while**-Schleifen entfernt einen Knoten aus dem Keller. Da jeder Knoten genau einmal in den Keller gelegt wird, kann er auch nur einmal entfernt werden. □

Weitere Anwednung von DFS : Berechnung von *zweifachen Zusammenhangskomponenten.*

Kapitel 9

Kürzeste Pfade

9.1 Ein generischer Algorithmus

Sei $G = (V, E)$ ein gerichteter Graph mit Kantenkosten $c : E \to R$ (die wir auch als Abstände interpretieren). Sei s ein ausgezeichneter Startknoten. Im *Single Source Shortest Path Problem (SSSP)* interessieren wir uns für den kürzesten Weg von s zu allen anderen Knoten v von G :

$$\begin{aligned} \mu(v) &= \text{Kosten des kürzesten Weges von } s \text{ nach } v \\ &= \inf\{c(p) \mid p \text{ Pfad von } s \text{ nach } v\} . \end{aligned}$$

Dabei definieren wir die Kosten eines Pfads $p = \{e_1, e_2, ..., e_k\}$ als die Summe seiner Kantenkosten :

$$c(p) = \sum_{i=1}^{k} c(e_i) .$$

Beachte, daß wir in der Definition von μ wirklich das Infimum bilden müssen und nicht etwa nur das Minimum (wenn der Graph Zyklen hat, gibt es unendlich viele Wege !). Ferner ist

$$\mu \in \Re \cup \{-\infty, \infty\} .$$

Lemma 9.1.1

1. *Es ist $\mu(v) < \infty$ genau dann, wenn v von s aus erreichbar ist.*
2. *Falls es einen Pfad von s nach v gibt, der einen negativen Zyklus enthält, dann ist $\mu(v) = -\infty$.*
3. *Falls v von s aus erreichbar ist und es keinen Pfad von s nach v gibt, der einen negativen Zyklus enthält, dann ist*

$$-\infty < \mu(v) < +\infty$$

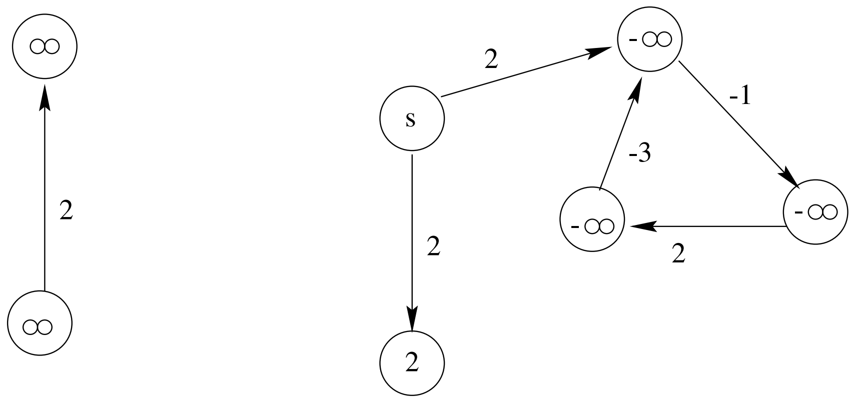


Abbildung 9.1: Kürzeste Entfernungen von s.

und es ist $\mu(v) = c(p)$ für einen einfachen Pfad p von s nach v.

4. *Falls $\mu(v) = -\infty$ ist, dann gibt es einen Pfad von s nach v, der einen negativen Zyklus enthält.*

Beweis:

1. Es existiert immer das Infimum über eine nichtleere Menge.

2. Sei $p = p_1 \circ p_2 \circ p_3$ ein Pfad von s nach v, der einen negativen Zyklus p_2 enthält. Dann ist

$$c(p_1 \circ \underbrace{p_2 \circ \ldots \circ p_2}_{k} \circ p_3) = c(p) + kc(p_2)$$

und daher $\mu(v) < c$ für jedes $c \in \Re$. Also ist $\mu(v) = -\infty$.

3. Sei p ein beliebiger Pfad von s nach v. Falls p einen Zyklus p_2 enthält, also $p = p_1 \circ p_2 \circ p_3$, dann ist

$$c(p_1 \circ p_3) = c(p) - c(p_2) \leq c(p)\,,$$

da p_2 kein negativer Zyklus ist.

Wir können also sukzessive Zyklen entfernen und erhalten schließlich einen einfachen Pfad p' von s nach v mit $c(p') \leq c(p)$. Also ist

$$\begin{aligned} \mu(v) &= \inf\{c(p') \mid p' \text{ ist einfacher Pfad von } s \text{ nach } v\} \\ &= \min\{c(p') \mid p' \text{ ist einfacher Pfad von } s \text{ nach } v\}\,. \end{aligned}$$

Es gilt Gleichheit, weil es nur endlich viele einfache Pfade von s nach v gibt.

4. Folgt aus (1) und (3). □

Lemma 9.1.2 *(Optimalitätskriterium)*
Sei $d : V \to \Re \cup \{-\infty, \infty\}$ eine Funktion mit

1. $d(v) \geq \mu(v)$ *für alle* $v \in V$.
2. $d(s) \leq 0$.
3. *Für alle Kanten* $e = (v, w)$ *gilt* $d(w) \leq d(v) + c(e)$ *(Dreiecksungleichung).*

Dann ist $d(v) = \mu(v)$ *für alle* $v \in V$.

Beweis: Annahme : Es gibt ein v mit $d(v) > \mu(v)$.
Dann ist $\mu(v) < \infty$. Wir unterscheiden nun zwei Fälle.

1. $\mu(v) > -\infty$:

   Dann ist $\mu(v) = c(p)$ für einen Pfad $p = [s = v_0, v_1, ..., v_k = v]$ von s nach v. Ferner ist $\mu(s) = 0$ (0 oder $-\infty$ sind die einzigen Werte, die $\mu(s)$ annehmen kann) und

   $$\mu(v_{i+1}) = \mu(v_i) + c(v_i, v_{i+1})$$

   (Präfixe von kürzesten Wege sind auch kürzeste Wege).

   Wegen $\mu(s) = 0$ und (1) und (2) folgt $d(s) = 0$. Sei $i > 0$ minimal, so daß $\mu(v_{i+1}) < d(v_{i+1})$ ist (i existiert wegen $\mu(v_k) < d(v_k)$). Dann gilt :

   $$\begin{aligned} d(v_{i+1}) &> \mu(v_{i+1}) = \mu(v_i) + c(v_i, v_{i+1}) \\ &= d(v_i) + c(v_i, v_{i+1})\,. \end{aligned}$$

   Dies ist aber ein Widerspruch zu (3).

2. $\mu(v) = -\infty$:

   Dann gibt es einen Pfad $p = p_1 \circ p_2 \circ p_3$ von s nach v, der einen negativen Zyklus p_2 enthält. Sei $p_2 = [e_1, e_2, ..., e_k]$, wobei $e_i = (v_i, v_{i+1})$ ist, und sei $v_1 = v_{k+1} = z$.

   Wegen $d(v) > -\infty$ ist auch $d(z) > -\infty$ (wegen (3)). Durch wiederholte Anwendung von (3) erhalten wir :

   $$\begin{aligned} d(z) &= d(v_{k+1}) \\ &\leq d(v_k) + c(e_k) \\ &\leq d(v_{k-1}) + c(e_{k-1]}) + c(e_k) \\ &\vdots \end{aligned}$$

$$\begin{aligned} &\leq d(v_1) + c(e_1) + \ldots + c(e_k) \\ &= d(z) + c(p_2) \\ &< d(z)\,, \end{aligned}$$

ein Widerspruch. □

Dieses Lemma erlaubt die Überprüfung von Fehlern in der Bestimmung der kürzesten Wege. Ferner läßt sich ein Algorithmus daraus herleiten.

Generischer Algorithmus

begin
  $d(s) \leftarrow 0$;
  $\pi(s) \leftarrow nil$;
  forall $v \in V - s$
  do
    $d(v) \leftarrow +\infty$;
    $\pi(v) \leftarrow nil$;
  od
  while es gibt Kante $e = (v, w)$ mit $d(w) > d(v) + c(e)$
  do
    Wähle eine beliebige solche Kante (v, w);
    $d(w) \leftarrow d(v) + c(e)$;
    $\pi(w) \leftarrow e$;
  od
end

Den Rumpf der **while**-Schleife bezeichnet man auch als *Relaxierung* der Kante e. Die willkürliche Wahl einer beliebigen Kante e zur Relaxierung macht den Algorithmus im allgemeinen ineffizient. Wir werden später sehen, wie man diese Wahl effizienter gestalten kann.

Beob.: Offensichtlich gilt :

- $d(v)$ ist die Länge eines Pfades von s nach v. Also gilt (1).
- Weil wir mit $d(s) = 0$ beginnen und d-Werte immer nur kleiner werden können, gilt (2).
- Der Algorithmus endet erst, wenn (3) gilt. □

Sei $P = \{e \mid e = \pi(v)$ für einen Knoten $v\}$ die Menge aller Kanten $\pi(v)$. Wir werden nun in einer Serie von Lemmas zeigen, daß die P-Kanten einen Baum mit Wurzel s bilden, wobei jeder Baumpfad einem kürzesten Weg entspricht.

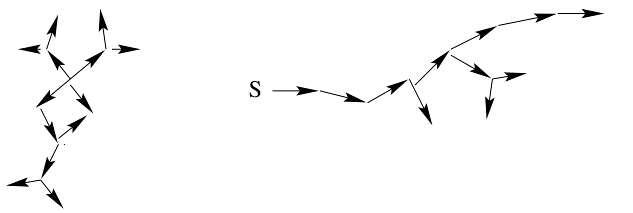


Abbildung 9.2: Der Graph, der von den P-Kanten aufgespannt wird.

Lemma 9.1.3

1. *Falls $d(v) < \infty$ ist, dann ist $d(v) = c(p)$ für einen Pfad p von s nach v. Ferner ist stets $d(s) \leq 0$.*

2. *Für $v \neq s$ gilt :*

$$\begin{array}{rcl} d(v) < \infty & \Longleftrightarrow & \pi(v) \neq nil \\ und \quad d(s) < 0 & \Longleftrightarrow & \pi(s) \neq nil \end{array}$$

3. *Für $v \neq s$ gilt : Falls es eine P-Kante gibt, die in v entspringt, dann gibt es auch eine P-Kante die in v endet.*

Beweis:

1. Folgt aus der vorangegangenen Beobachtung.

2. Eine P-Kante $e = (u, v)$ entsteht, wenn im Rumpf der **while**-Schleife $\pi(v) = e$ gesetzt wird. Die Schleife wird aber nur betreten, wenn $d(v) > d(u) + c(e)$ ist. Dann ist aber $d(u) < \infty$ und daher anschließend auch $d(v) \leq \infty$.

   $d(s)$ ist anfangs gleich 0 und wird später genau dann kleiner, wenn $\pi(s)$ gesetzt wird.

3. Eine P-Kante $e = (v, w)$ entsteht, wenn im Rumpf der **while**-Schleife $\pi(w) = e$ gesetzt wird. Die Schleife wird aber nur betreten, wenn $d(w) > d(v) + c(e)$ ist. Dann ist aber $d(v) < \infty$ und daher $\pi(v) \neq nil$ nach (2). □

Lemma 9.1.4 *Zu jedem Zeitpunkt der Ausführung gilt : Falls $\pi(y) = e = (x, y)$ ist, dann ist $d(y) \geq d(x) + c(e)$.*

Beweis: Zu dem Zeitpunkt, wo $\pi(y)$ auf e gesetzt wurde, galt :

$$d(y) = d(x) + c(e)\,.$$

$d(y)$ hat sich seitdem nicht mehr geändert (sonst hätte sich auch $\pi(y)$ geändert). $d(x)$ kann sich geändert haben, aber es kann nur kleiner geworden sein. □

Lemma 9.1.5 *P-Zyklen haben negative Kosten.*

Beweis: Sei $p = [e_1, e_2, ..., e_k]$ ein Zyklus von P-Kanten. Sei $e_i = (v_i, v_{i+1})$ und $v = v_1 = v_{k+1}$. Wir nehmen o.B.d.A. an, daß e_k als letztes zur P-Kante wurde. Gerade vor diesem Zeitpunkt galt :

$$\begin{array}{rcl} d(v_{k+1}) & > & d(v_k) + c(e_k) \quad \text{und} \\ d(v_{i+1}) & \geq & d(v_i) + c(e_i) \end{array}$$

für $1 \leq i \leq k-1$ nach Lemma 9.1.4. Also ist

$$\sum_{i=1}^{k} d(v_{i+1}) > \sum_{i=1}^{k} d(v_i) + \sum_{i=1}^{k} c(e_i) .$$

Wegen $d(v_1) = d(v_k + 1)$ folgt

$$\sum_{i=1}^{k} c(e_i) < 0 .$$

□

Lemma 9.1.6 *Wenn v auf einem P-Zyklus liegt oder auf einem P-Pfad, der von einem P-Zyklus aus erreichbar ist, dann ist $\mu(v) = -\infty$.*

Beweis: Wenn $\pi(v) \neq nil$ ist, dann ist nach Lemma 9.1.3(2) $d(v) < \infty$. Insbesondere gibt es dann nach Lemma 9.1.3(1) einen Pfad von s nach v in G. Wenn v auf einem P-Zyklus liegt, dann ist nach Lemma 9.1.5 $\mu(v) = -\infty$. Wenn v auf einem P-Pfad, abzweigend von einem P-Zyklus, liegt, dann ist analog ebenfalls $\mu(v) = -\infty$. □

Lemma 9.1.7 *Sei $-\infty < \mu(v) < \infty$. Wenn $d(v) = \mu(v)$ ist, dann gibt es einen P-Pfad von s nach v mit Kosten $\mu(v)$.*

Beweis: Wenn $d(v) < +\infty$ ist, dann ist nach Lemma 9.1.3(2) $\pi(v) \neq nil$. Nach Lemma 9.1.3(3) liegt dann entweder v auf einem P-Zyklus (oder auf einem P-Pfad, der auf einem P-Zyklus entspringt) oder es gibt einen P-Pfad von s nach v. Im ersteren Fall wäre aber $\mu(v) = -\infty$ nach Lemma 9.1.6. Da dies nicht der Fall ist, sei nun $(s = v_0, v_1, ..., v_k = v)$ ein P-Pfad von s nach v. Nach Lemma 9.1.4 ist

$$d(v) \geq d(v_{k-1}) + c(e_{k-1}) \geq \ldots \geq d(s) + \sum_{i=0}^{k-1} c(e_i) = \sum_{i=0}^{k-1} c(e_i) ,$$

wobei $e_i = (v_i, v_{i+1})$ ist.
Da andererseits $\mu(v)$ die Kosten eines minimalen Pfades sind und $d(v) = \mu(v)$ ist, muß $d(v) = \mu(v) = \sum_{i=0}^{k-1} c(e_i)$ sein. □

Wir klassifizieren nun die Knoten von G nach ihrer Erreichbarkeit von s aus in drei Klassen :

$$\begin{aligned} V^+ &= \{v \in V \mid \mu(v) = +\infty\}\,, \\ V^- &= \{v \in V \mid \mu(v) = -\infty\} \quad \text{und} \\ V^f &= V - \{V^+ \cup V^-\}\,. \end{aligned}$$

Dann zeigt die Serie der vorangegangenen Lemmas, daß die P-Kanten in der Tat kürzeste Wege in G darstellen.

Satz 9.1.8 *Wenn für alle $v \in V^f$ gilt $d(v) = \mu(v)$, dann induzieren die P-Kanten einen kürzeste-Wege-Baum auf V^f.* □

9.2 Effizientere Algorithmen

Der generische Algorithmus hat zwei große Probleme :

- Er terminiert nicht bei negativen Zyklen.
- Er kann auch bei nichtnegativen Kantenkosten exponentielle Laufzeit haben.

Wir definieren eine neue Menge U :

$$U \supseteq \{u \in V \mid d(u) < \infty \text{ und } \exists (u,v) \in E : d(u) + c(u,v) < d(v)\}\,,$$

wobei $d(u) + c(u,v) < d(v)$ eine verletzte Dreiecksungleichung ist. U ist eine *Kandidatenmenge zur Relaxierung.* Wir formulieren jetzt den generischen Algorithmus nochmal neu unter Verwendung der Menge U.

Generischer Algorithmus (nochmal)

begin
  forall Knoten v
  do
    $d(v) \leftarrow \infty$;
    $\pi(v) \leftarrow nil$;
  od
  $d(s) \leftarrow 0$;
  $U \leftarrow \{s\}$;
  while $U \neq \emptyset$
  do
    Wähle $u \in U$ beliebig;
    forall Kanten $(u,v) \in E$

```
        do
            if d(u) + c(u,v) < d(v)
            then
                d(v) ← d(u) + c(u,v);
                π(v) ← (u,v);
                U ← U + v;
            fi
        od
        U ← U − u;
    od
end
```

Das nächste Lemma zeigt, daß es im Prinzip immer eine gute Wahl für den Knoten $u \in U$ gibt; es gibt nämlich immer mindestens einen Knoten in U, der schon den korrekten d-Wert hat und der deshalb nach seinem Entfernen aus U nie wieder zu U hinzukommen kann.

Lemma 9.2.1 *(Der potentiellen optimalen Wahl)*

a) Wenn $v \in U$ ist mit $d(v) > \mu(v) > -\infty$, dann gibt es ein $u \in U$, so daß u auf einem kürzesten Weg von s nach v liegt und $d(u) = \mu(u)$ ist.

b) Wenn für $u \in U$ gilt $d(u) = \mu(u)$ und u wird aus U entfernt, dann kommt u später nie wieder nach U.

Beweis:

a) Sei $v \in U$ und $d(v) > \mu(v)$. Sei p ein kürzester Pfad von s nach v. Wegen $\mu(s) = 0 = d(s)$ gibt es einen ersten Knoten v_i auf p mit $d(v_i) > \mu(v_i)$. Es folgt :

$$\begin{aligned} d(v_i) &> \mu(v_i) = \mu(v_{i-1}) + c(v_{i-1}, v_i) \\ &= d(v_{i-1}) + c(v_{i-1}, v_i)\,. \end{aligned}$$

Dann ist aber $v_{i-1} \in U$ und $d(v_{i-1}) = \mu(v_{i-1})$.

b) Wenn ein Knoten v zu U hinzukommt, dann wird $d(v)$ erniedrigt. $d(u)$ kann aber nicht mehr weiter erniedrigt werden, also kann u nie wieder zu U hinzukommen.

□

Leider ist es im allgemeinen nicht möglich, diesen optimalen Knoten effizient zu bestimmen. Es gibt allerdings zwei wichtige Spezialfälle, in denen es doch möglich ist.

Lemma 9.2.2 *(Der algorithmischen optimalen Wahl)*

a) Wenn die Kantenkosten nicht-negativ sind, dann ist $d(u) = \mu(u)$ für das $u \in U$ mit dem kleinsten d-Wert.

b) Sei G azyklisch und $v_1, \ldots, v_k$ eine topologische Sortierung (mit $(v_i, v_j) \in E \Rightarrow i < j$). Dann ist $d(u) = \mu(u)$ für das $u \in U$ mit minimalem Index.

Beweis: Sei $u \in U$ so gewählt wie in a) bzw. b) und nehmen wir an, es sei $d(u) > \mu(u)$. Dann gibt es auf einem kürzesten Pfad $s \underset{G}{\rightarrow}^* u$ ein $v \in U$ mit $d(v) = \mu(v)$.

a) $d(v) = \mu(v) \leq \mu(u) < d(u)$: Widerspruch zur Wahl von u.

b) v hat kleineren Index als u : Widerspruch zur Wahl von u. □

9.2.1 Dijkstra's Algorithmus

Dijkstra's Algorithmus berechnet effizient kürzeste Wege in Graphen ohne negative Kantenkosten. Er basiert auf Lemma 9.2.2(a), d.h., er verwaltet die bereits gefundenen Knoten in einer Warteschlange und wählt in jedem Schritt den Knoten mit kleinsten Kosten aus. Wenn alle aus diesem Knoten ausgehenden Kanten relaxiert sind, ist er fertig und braucht nie wieder betrachtet zu werden.

Algorithmus von Dijkstra (1959)

```
begin
    forall Knoten v
    do
        d(v) ← ∞;
        π(v) ← nil;
    od
    d(s) ← 0;
    PQ  U;
    U.insert(s, 0);
    while (!U.empty())
    do
        u ← U.del_min();
        forall Kanten (u, v) ∈ E
        do
            c ← d(u) + c(u, v);
            if d(v) = ∞
            then
```

```
                U.insert(v, c);
                π(v) ← u;
                d(v) ← c;
            else
                if d(v) > c
                then
                    U.dec_prio(v, c);
                    π(v) ← u;
                    d(v) ← c;
                fi
            fi
        od
    od
end
```

Die Laufzeit von Dijkstra's Algorithmus ist abhängig von der Implementierung der Priority Queue. In der folgenden Tabelle sind die Laufzeiten mit einer Implementation der Priority Queue als Bitfeld, als Binominal Heap und als Fibonacci-Heap angegeben. Dabei ist zu beachten, daß der Algorithmus insgesamt maximal n `insert`, n `del_min` und m `dec_prio` Operationen macht.

	Zeit			
Implementierung von U	**insert**	**del_min**	**dec_prio**	**Summe**
U ist ein Bitfeld	$n \cdot 1$	$n \cdot n$	$m \cdot 1$	$n^2 + m$
U ist ein Binary Heap	$n \cdot \log n$	$n \cdot \log n$	$m \cdot \log n$	$(n + m) \cdot \log n$
U ist ein Fibonacci-Heap	$n \cdot 1$	$n \cdot \log n$	$m \cdot 1$	$n \cdot \log n + m$

Bei dichten Graphen sollte man also die Priority Queue als Bitfeld implementieren. Ansonsten haben Fibonacci-Heaps die beste Laufzeit.

9.2.2 Eine Anwendung

In modernen Autonavigations-Systemen möchte man auch kürzeste Strecken zwischen zwei Punkten einer Landkarte ausrechnen können. Leider ist hierfür selbst Dijkstra's Algorithmus zu langsam (Bordrechner haben nur kleine Speicher und sehr langsame CD-Laufwerke). Man hilft sich daher mit folgenden Tricks :

- Dijkstra findet kürzeste Wege durch Suche in einer langsam wachsenden Kreisscheibe um s. Wenn man gleichzeitig vom Zielpunkt t aus eine Suche nach s startet, kann man den Suchraum stark einschränken.

- Hilfreich ist auch, mit groben Schätzwerten für den Abstand zu beginnen (statt $+\infty$).

- Ein Preprocessing der kürzesten Wege zwischen allen Paaren von Orten würde zuviel Speicherplatz brauchen. Allerdings kann man in einem Preprocessing-Schritt die grobe Suchrichtung feststellen. Dazu teilt man die Karte in eine kleine Menge von Regionen auf, etwa 30 Regionen. Für jede Region berechnet man nun von jedem anderen Punkt s der Karte aus alle kürzesten Wege zu Zielpunkten innerhalb der Region. Wenn s genügend weit weg von der Region liegt, werden alle diese kürzesten Wege anfangs den gleichen Weg nehmen. Diese Richtung kann man nun in s abspeichern. Insgesamt braucht man also nur in jedem Punkt der Karte einen Vektor mit 30 Richtungen (für die 30 Regionen) abzuspeichern, hat also nur linearen Speicherbedarf.

  Die Suche nach einem kürzesten Weg folgt nun einfach solange wie möglich diesen Richtungsvektoren. Wenn man gleichzeitig von t aus s sucht, hat man bei weit entfernten Punkten s und t eine gute Chance, daß sich diese beiden Suchpfade irgendwo treffen. Dann hat man den kürzesten Weg gefunden, ohne eine einzige überflüssige Kante angeschaut zu haben.

Anhang A

Der Differenzenkalkül

In der ersten Übungsstunde lernen wir eine allgemeine Methode kennen wie man Summenformeln in eine geschlossene Form bringen kann, den sogenannten *Differenzenkalkül.* Wir beginnen mit einigen Beispielen, die zeigen, daß die ad hoc Berechnung von Summen im Einzelfall ziemlich knifflig sein kann.

A.1 Beispiele

Beispiel A.1.1

$$\sum_{i=0}^{n-1}(2i+1) = 1+3+5+...+(2n-1) = n^2 \; .$$

Dies sieht man durch folgenden Trick :

$$\begin{array}{ccccc} 1 & \cdot & \uparrow & \uparrow & \uparrow \\ 3 & \rightarrow & \nearrow & \uparrow & \uparrow \\ 5 & \rightarrow & \rightarrow & \nearrow & \uparrow \\ 7 & \rightarrow & \rightarrow & \rightarrow & \nearrow \end{array} \quad \Longrightarrow \quad 4^2 = 16 \; .$$

□

Beispiel A.1.2

$$\sum_{i=1}^{n} i = 1+2+3+...+n = \frac{n\cdot(n+1)}{2} \; .$$

Dies sieht man am einfachsten mit dem Trick von Gauß ein. Man erzählt, in der Schule habe Gaußens Lehrer die Schulklasse mit einer ihm aufwendig erscheinenden Aufgabe ruhigstellen wollen – der kleine Gauß jedoch sei schon nach zwei Minuten durch folgenden Trick fertig gewesen und weiter seinen (wahrscheinlich nicht unterrichtsbezogenen) Gedanken nachgehangen ... :

$$\begin{array}{ccccccccccc} & & 1 & + & 2 & + \ldots + & n & = & & \sum_{i=1}^n i \\ & & n & + & (n-1) & + \ldots + & 1 & = & & \sum_{i=1}^n i \\ \hline n\cdot(n+1) & = & (n+1) & + & (n+1) & + \ldots + & (n+1) & = & 2\cdot & \sum_{i=1}^n i \end{array}$$

□

Beispiel A.1.3

$$\sum_{i=1}^{n} i^2 = 1+4+9+...+n^2 = \frac{1}{6}\cdot n\cdot(n+1)\cdot(2n+1)\,.$$

Sei $S_n = \sum_{i=1}^n i^3$. Dann gilt :

$$\begin{aligned} S_n + (n+1)^3 &= S_{n+1} \\ &= \sum_{i=0}^{n}(i+1)^3 \\ &= \sum_{i=0}^{n}(i^3+3i^2+3i+1) \\ &= S_n + 3\cdot\sum_{i=1}^{n} i^2 + 3\cdot\sum_{i=1}^{n} i + (n+1) \\ \Longrightarrow \quad 3\cdot\sum_{i=1}^{n} i^2 &= (n+1)^3 - \frac{3}{2}\cdot n\cdot(n+1) - (n+1) \\ &= (n+1)\cdot((n+1)^2 - \frac{3}{2}\cdot n - 1) \\ &= (n+1)\cdot(n^2+\frac{1}{2}n) \\ \Longrightarrow \quad \sum_{i=1}^{n} i^2 &= \frac{1}{6}\cdot n\cdot(n+1)\cdot(2n+1)\,. \end{aligned}$$

□

Beispiel A.1.4

$$\sum_{i\geq 1}\frac{i}{2^i} = 2\,.$$

Dazu betrachten wir zunächst $\sum_{k=0}^{\infty}\frac{1}{2^k}$:

$$
\begin{aligned}
\frac{1}{2}\cdot\sum_{k=0}^{\infty}\frac{1}{2^k} &= \sum_{k=0}^{\infty}\frac{1}{2^k}-\frac{1}{2}\cdot\sum_{k=0}^{\infty}\frac{1}{2^k} \\
&= \sum_{k=0}^{\infty}\frac{1}{2^k}-\sum_{k=1}^{\infty}\frac{1}{2^k} \\
&= 1 \\
\Longrightarrow \quad &= \sum_{k=0}^{\infty}\frac{1}{2^k}=2\,.
\end{aligned}
$$

Nun zurück zum eigentlichen Problem :

$$
\begin{array}{rcl}
\displaystyle\sum_{i\geq 1}\frac{i}{2^i}=\frac{1}{2^1}+\frac{1}{2^2}+\frac{1}{2^3}+\frac{1}{2^4}+\ldots & \rightarrow & \displaystyle\frac{1}{2^1}\cdot\sum_{k=0}^{\infty}\frac{1}{2^k}=2^0 \\
\displaystyle\frac{1}{2^2}+\frac{1}{2^3}+\frac{1}{2^4}+\ldots & \rightarrow & \displaystyle\frac{1}{2^2}\cdot\sum_{k=0}^{\infty}\frac{1}{2^k}=\frac{1}{2^1} \\
\displaystyle\frac{1}{2^3}+\frac{1}{2^4}+\ldots & \rightarrow & \displaystyle\frac{1}{2^3}\cdot\sum_{k=0}^{\infty}\frac{1}{2^k}=\frac{1}{2^2} \\
\displaystyle\frac{1}{2^4}+\ldots & \rightarrow & \displaystyle\frac{1}{2^4}\cdot\sum_{k=0}^{\infty}\frac{1}{2^k}=\frac{1}{2^3} \\
\hline
\end{array}
$$

$$
\Longrightarrow \quad \sum_{i\geq 1}\frac{i}{2^i}=\sum_{i=0}^{\infty}\frac{1}{2^i}=2\,.
$$

□

Mehr zu diesem Themengebiet findet man in dem Buch von Hanson : *Table of Sequences and Functions*. Wir wenden uns aber nun dem Differenzenkalkül zu, einer allgemeinen Methode, um ohne raffinierte Tricks Summen auszurechnen. Diese Methode ist ausführlich erklärt in Kapitel 2.6 der *Concrete Mathematics* von Graham, Knuth und Patashnik.

A.2 Der Differenzenoperator

Die *Ableitung* einer Funktion f ist bekanntermaßen definiert durch

$$
f'(x)=\lim_{h\to 0}\frac{f(x+h)-f(x)}{h}\,.
$$

Der *Ableitungsoperator* D ordnet jeder Funktion f ihre Ableitung zu :

$$
Df(x)=f'(x)\,.
$$

Beispiel A.2.1 Es ist

$$Dx^m = m \cdot x^{m-1} .$$

□

Wir konstruieren nun als diskretes Analogon dazu den *Differenzenoperator* Δ :

$$\Delta f(x) = f(x+1) - f(x) .$$

Beispiel A.2.2 Es ist

$$\Delta x^2 = (x+1)^2 - x^2 = 2x + 1 .$$

□

Lemma A.2.3 *Der Operator Δ ist linear, d.h., für $\alpha, \beta \in I\!R$ und Funktionen f und g ist*

$$\Delta(\alpha f + \beta g) = \alpha \Delta f + \beta \Delta g .$$

□

A.2.1 Fallende Potenzen

Wie wir gerade in Beispiel A.2.2 gesehen haben, definiert der Differenzenoperator nicht so eine schöne Beziehung zwischen einfachen Potenzen von x wie der Ableitungsoperator (siehe Beispiel A.2.1). Dies können wir allerdings durch die Definition einer neuen, "alternativen" Potenz $x^{\underline{m}}$ (lies : *x hoch m fallend*) erreichen :

$$\begin{aligned} x^{\underline{0}} &= 1 \quad \text{und} \\ x^{\underline{m}} &= x \cdot (x-1) \cdot \ldots \cdot (x-m+1) \end{aligned}$$

für $x \in I\!R$ und $m \in I\!N$.

Beispiel A.2.4 Es ist

$$\begin{aligned} x^{\underline{1}} &= x , \\ x^{\underline{2}} &= x \cdot (x-1) , \\ x^{\underline{3}} &= x \cdot (x-1) \cdot (x-2) , \\ \ldots \end{aligned}$$

□

Lemma A.2.5 *Für $x \in I\!R$ und $m \in I\!N$ ist*

$$\Delta x^{\underline{m}} = m \cdot x^{\underline{m-1}} .$$

Beweis:

$$\begin{aligned}\Delta x^{\underline{m}} &= (x+1)^{\underline{m}} - x^{\underline{m}} \\ &= (x+1)\cdot x \cdots (x+1-m+1) - x\cdot(x-1)\cdots(x-m+1) \\ &= x\cdot(x-1)\cdots(x-m+2)\cdot((x+1)-(x-m+1)) \\ &= m\cdot x^{\underline{m-1}}\,.\end{aligned}$$

□

Darüberhinaus existiert ein Analogon zum Potenzgesetz.

Lemma A.2.6 *Für $x \in I\!R$ und $m, n \in \mathbb{N}_0$ ist*

$$x^{\underline{m+n}} = x^{\underline{m}} \cdot (x-m)^{\underline{n}}\,.$$

Beweis: Die Behauptung stimmt offensichtlich für $m = 0$ oder $n = 0$. Ansonsten gilt :

$$x^{\underline{m+n}} = \underbrace{x\cdot(x-1)\cdots(x-m+1)}_{x^{\underline{m}}}\cdot\underbrace{(x-m)\cdots(x-m-n+1)}_{(x-m)^{\underline{n}}}\,.$$

□

A.2.2 Diskrete Integrale

Das *unbestimmte Integral* von g ist bekanntermaßen definiert durch

$$\int g(x)\,dx = \text{ Klasse aller Funktionen mit Ableitung } g\,.$$

Zwischen Differentiation und Integration besteht nun folgender fundamentaler Zusammenhang.

Satz A.2.7 (Hauptsatz der Differential- und Integralrechnung) *Es gilt :*

$$g = Df \iff \int g(x)\,dx = f + C,$$

wobei C die Klasse aller konstanten Funktionen (also mit Ableitung 0) ist. □

Sei $a, b \in I\!R$. Wenn $g = Df$ ist, dann ist das *bestimmte Integral* von g an den Grenzen a und b definiert als

$$\int_a^b g(x)\,dx = f|_a^b = f(b) - f(a)\,.$$

Analog können wir ein *unbestimmtes diskretes Integral* (oder besser *unbestimmte Summe*) definieren.

$$\sum g(x)\,\delta x = \text{ Klasse aller Funktionen, deren Differenz } g \text{ ist.}$$

Satz A.2.8 *Es gilt :*

$$g = \Delta f \iff \sum g(x)\delta x = f + D\,,$$

wobei D die Klasse aller Funktionen h mit $\Delta h(x) = 0$ ist. □

Bem.: Die Klasse C ist eine *echte* Teilmenge der Klasse D, da zum Beispiel auch die Funktion $h(x) = a + b \cdot \sin 2\pi x$ in D ist. □

Sei $a, b \in \mathbb{Z}$. Wenn $g = \Delta f$ ist, dann definieren wir die *bestimmte Summe* von g an den Grenzen a und b durch

$$\sum_a^b g(x)\delta x = f|_a^b = f(b) - f(a)\,.$$

Bem.: Analog zum Integral ist

- $\sum_a^a g(x)\delta x = f(a) - f(a) = 0.$
- $\sum_a^b g(x)\delta x = f(b) - f(a) = -(f(a) - f(b)) = -\sum_b^a g(x)\delta x.$ □

Der folgende Satz zeigt nun, daß wir mit Hilfe der bestimmten Summe Summenformeln ausrechnen können. Dabei muß man aufpassen, daß man beim Übergang von der Summenformel zur bestimmten Summe nicht vergißt, die obere Grenze um 1 zu erhöhen.

Satz A.2.9 *Sei $a, b \in \mathbb{Z}$, $a < b$. Wenn $g = \Delta f$ ist, dann ist*

$$\boxed{\sum_a^b g(x)\delta x = \sum_{i=a}^{b-1} g(i)}$$

Beweis:

$$\begin{aligned}
\sum_a^b g(x)\delta x &= f(b) - f(a) \\
&= (f(b) - f(b-1)) + (f(b-1) - f(b-2)) + \cdots + (f(a+1) - f(a)) \\
&= \Delta f(b-1) + \Delta f(b-2) + \cdots + \Delta f(a) \\
&= g(b-1) + g(b-2) + \ldots + g(a) \\
&= \sum_{i=a}^{b-1} g(x)\,.
\end{aligned}$$

□

Beispiel A.2.10 Es ist

$$\Delta x^{\underline{2}} = 2 \cdot x^{\underline{1}}\,,$$

also

$$x^{\underline{1}} = \frac{1}{2} \cdot \Delta x^{\underline{2}} = \Delta \left(\frac{1}{2} x^{\underline{2}} \right) .$$

Daher ist

$$\sum_{i=0}^{n} i = \sum_{i=0}^{n} i^{\underline{1}} = \sum_{0}^{n+1} x^{\underline{1}} \delta x = \frac{1}{2} x^{\underline{2}} \Big|_{0}^{n+1} = \frac{1}{2} n \cdot (n+1) .$$

$\square$

Allgemein gilt :

Lemma A.2.11 *Für $m \in \mathbb{N}_0$ ist*

$$\sum_{i=0}^{n} i^{\underline{m}} = \frac{1}{m+1} \cdot (n+1)^{\underline{m+1}} .$$

Beweis: Es ist $\Delta x^{\underline{m+1}} = (m+1) \cdot x^{\underline{m}}$. $\square$

A.2.3 Negative Exponenten

Wie sollten wir fallende Potenzen mit negativen ganzzahligen Exponenten definieren ? Sinnvollerweise so, daß das Potenzgesetz weiterhin gilt, also

$$1 = x^{\underline{0}} = x^{\underline{-m+m}} = x^{\underline{-m}} \cdot (x+m)^{\underline{m}} .$$

Definition A.2.12 Für $m \in \mathbb{N}$ und $x \notin \{-1, \ldots, -m\}$ definieren wir

$$x^{\underline{-m}} = \frac{1}{(x+m)^{\underline{m}}} = \frac{1}{(x+1) \cdots (x+m)} .$$

$\square$

Satz A.2.13 *Für $m \in \mathbb{Z}$ gilt :*

$$\Delta x^{\underline{m}} = m \cdot x^{\underline{m-1}} .$$

Beweis: Für $m \in \mathbb{N}$ haben wir das früher bereits gezeigt (Lemma A.2.5).
Und für $m = 0$ ist $\Delta x^{\underline{0}} = 0 = 0 \cdot x^{\underline{-1}}$.
Sei nun also $m \in \mathbb{N}$. Dann ist

$$\begin{aligned}
\Delta x^{\underline{-m}} &= (x+1)^{\underline{-m}} - x^{\underline{-m}} \\
&= \frac{1}{(x+2) \cdots (x+m+1)} - \frac{1}{(x+1) \cdots (x+m)} \\
&= \frac{(x+1) - (x+m+1)}{(x+1) \cdots (x+m+1)} \\
&= -m \cdot x^{\underline{-(m+1)}} .
\end{aligned}$$

$\square$

Korollar A.2.14 *Für $m \in \mathbb{Z} \setminus \{-1\}$ und $a, b \in \mathbb{Z}$, $a < b$ ist*

$$\sum_{i=a}^{b} i^{\underline{m}} = \sum_{i=a}^{b} x^{\underline{m}} \delta x = \frac{1}{m+1} \cdot x^{\underline{m+1}} \Big|_a^{b+1} .$$

Beweis: Es ist $x^{\underline{m}} = \frac{1}{m+1} \cdot \Delta x^{\underline{m+1}}$. □

Beachte, daß für $m \in \mathbb{N}$ gilt : $0^{\underline{-m}} = \frac{1}{1 \cdots m} \neq 0$.

Beispiel A.2.15 Es ist

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \sum_{i=0}^{n-1} i^{\underline{-2}} = -1 \cdot x^{\underline{-1}} \Big|_0^n = -\left(\frac{1}{n+1} - \frac{1}{1}\right) = 1 - \frac{1}{n+1} .$$

Das können wir auch "zu Fuß" ausrechnen :

$$\begin{aligned}
\sum_{i=1}^{n} \frac{1}{i(i+1)} &= \frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \cdots + \frac{1}{n \cdot (n+1)} \\
&= \left(1 - \frac{1}{2}\right) + \left(\frac{1}{2} - \frac{1}{3}\right) + \left(\frac{1}{3} - \frac{1}{4}\right) + \cdots + \left(\frac{1}{n} - \frac{1}{n+1}\right) \\
&= 1 - \frac{1}{n+1} .
\end{aligned}$$

□

A.2.4 Die Harmonische Zahl

Was ist die Stammfunktion von $x^{\underline{-1}}$? Wir brauchen eine Funktion f mit $\Delta f = x^{\underline{-1}}$, d.h.

$$\begin{aligned}
f(x+1) - f(x) &= \frac{1}{x+1} \\
\Longrightarrow \quad f(x+1) &= \frac{1}{x+1} + f(x) \\
&= \frac{1}{x+1} + \frac{1}{x} + \cdots + \frac{1}{1} .
\end{aligned}$$

Definition A.2.16 Wir definieren für $x \in \mathbb{N}$ die *x-te Harmonische Zahl H_x* durch

$$H_x = \frac{1}{x} + \frac{1}{x-1} + \cdots + \frac{1}{1} .$$

□

Lemma A.2.17 *Für $x \in \mathbb{N}$ ist*

$$\Delta H_x = x^{\underline{-1}} .$$

Beweis: Es ist

$$\Delta H_x = H_{x+1} - H_x = \frac{1}{x+1} = x^{\underline{-1}} .$$

□

Damit ist die Harmonische Zahl das Analogon zum natürlichen Logarithmus, denn es ist $D \ln x = x^{-1}$.

Bem.: Für große x gilt

$$H_x \approx \ln x + 0.557 + \frac{1}{2x} .$$

□

Folgende Ungleichungskette kann man sich leicht klarmachen :

$$\ln(n+1) = \int_1^{n+1} \frac{1}{x} dx \leq H_x \leq 1 + \int_1^n \frac{1}{x} dx = 1 + \ln n .$$

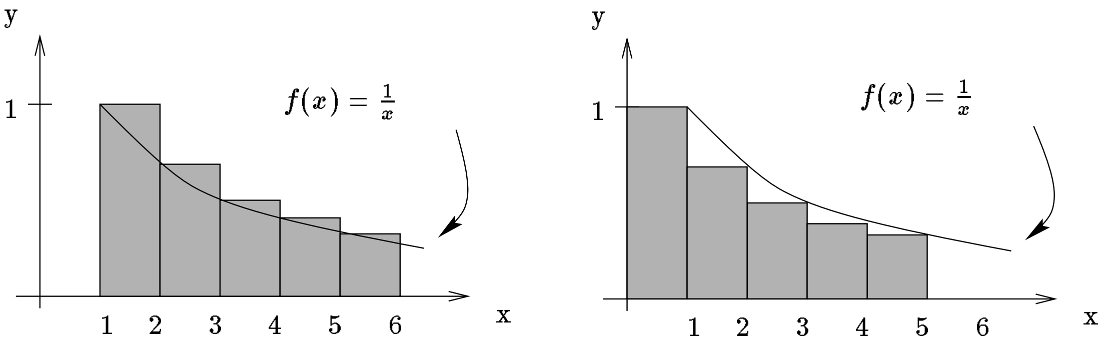


Abbildung A.1: Die harmonische Zahl H_5.

Die Summe der Flächeninhalte der schraffierten Rechtecke ergibt gerade H_x (im Bild : H_5). Der Inhalt der Fläche zwischen dem Funktionsgraphen und der x-Achse ist gerade der Wert des Integrals.

Insgesamt haben wir folgenden Satz bewiesen :

Satz A.2.18 *Es ist*

$$\sum_a^b x^{\underline{m}} = \begin{cases} \left. \frac{1}{m+1} \cdot x^{\underline{m+1}} \right|_a^b & \text{für } m \neq -1 , \\ H_x & \text{für } m = -1 . \end{cases}$$

□

A.2.5 Ein Analogon zur Exponentialfunktion

Wir suchen nun nach dem Analogon zur Exponentialfunktion ($De^x = e^x$), also eine Funktion f mit $\Delta f = f$:

$$\begin{aligned} & \Delta f(x) = f(x+1) - f(x) \overset{!}{=} f(x) \\ \iff \quad & f(x+1) = 2 \cdot f(x) \,. \end{aligned}$$

Dies gilt für die Funktion $f(x) = 2^x$.

Beispiel A.2.19 Es ist

$$\sum_{i=a}^{b} 2^i = \sum_{a}^{b+1} 2^x \delta x = 2^x \Big|_a^{b+1} = 2^{b+1} - 2^a \,.$$

□

Nun betrachten wir noch die allgemeine Exponentialfunktion c^x mit $c \neq 1$. Es ist

$$\Delta c^x = c^{x+1} - c^x = (c-1) \cdot c^x \,.$$

Daher ist

$$\sum_{a}^{b} c^x \delta x = \left. \frac{c^x}{c-1} \right|_a^b = \frac{c^b - c^a}{c-1} \,.$$

Dies ist die wohlbekannte Formel für die geometrische Reihe.

A.3 Partielle Summation

Erinnerung: *Partielle Integration*
Seien f und g zwei differenzierbare Funktionen, dann gilt :

$$\begin{aligned} (f \cdot g)' &= f \cdot g' + f' \cdot g \\ \Longrightarrow \quad \int (f \cdot g') dx &= f \cdot g - \int (f' \cdot g) dx \,. \end{aligned}$$

□

Analog gilt für die unbestimmte Summe :

$$\begin{aligned} \Delta(f \cdot g)(x) &= f(x+1)g(x+1) - f(x)g(x) \\ &= \underbrace{g(x+1)f(x+1) - f(x)g(x+1)}_{g(x+1)\Delta f(x)} + \underbrace{f(x)g(x+1) - f(x)g(x)}_{f(x)\Delta g(x)} \\ &= g(x+1)\Delta f(x) + f(x)\Delta g(x) \,. \end{aligned}$$

Dies entspricht fast der Formel für die partielle Integration, bis auf die unangenehme Tatsache, daß wir "$g(x+1)\Delta f(x)$" haben statt "$g(x)\Delta f(x)$". Diesen Schönheitsfehler "reparieren" wir durch die Einführung des sogenannten Shift-Operators.

Definition A.3.1 Der *Shift-Operator E* inkrementiert die Funktionsargumente, d.h.

$$Eg(x) = g(x+1) .$$

□

Damit können wir die obige Formel wie folgt umschreiben :

Produktformel für Δ-Operator :
$\Delta(f \cdot g) = f \cdot \Delta g + Eg \cdot \Delta f$

Oder äquivalent :

$$f \cdot \Delta g = \Delta(f \cdot g) - Eg \cdot \Delta f .$$

Summation auf beiden Seiten liefert dann den folgenden Satz.

Satz A.3.2 (Partielle Summation)
Es ist

$$\sum (f\Delta g)\delta x = f \cdot g - \sum (Eg \cdot \Delta f)\delta x .$$

□

A.4 Mehr Beispiele

Beispiel A.4.1 Im stetigen Fall ist

$$\int xe^x dx = xe^x - \int e^x dx = e^x \cdot (x-1) .$$

Analog ist im diskreten Fall

$$\begin{aligned}
\sum(\underbrace{x^{\underline{1}}}_{f}\cdot \underbrace{2^x}_{\substack{\Delta g \\ \Rightarrow g(x)=2^x}})\delta x &= x^{\underline{1}}\cdot 2^x - \sum(E2^x\cdot 1)\delta x \\
&= x^{\underline{1}}\cdot 2^x - \sum 2^{x+1}\delta x \\
&= x^{\underline{1}}\cdot 2^x - 2\cdot\sum 2^x\delta x \\
&= x^{\underline{1}}\cdot 2^x - 2\cdot 2^x \\
&= 2^x\cdot(x-2)\,.
\end{aligned}$$

□

Beispiel A.4.2 Es ist

$$\begin{aligned}
\sum(\underbrace{x^{\underline{2}}}_{f}\cdot \underbrace{2^x}_{\substack{\Delta g \\ \Rightarrow g(x)=2^x}})\delta x &= x^{\underline{2}}\cdot 2^x - \sum E2^x\cdot 2x^{\underline{1}}\delta x \\
&\overset{(A.4.1)}{=} x^{\underline{2}}\cdot 2^x - 4\cdot 2^x\cdot(x-2) \\
&= 2^x\cdot(x^{\underline{2}} - 4x + 8)\,.
\end{aligned}$$

□

Beispiel A.4.3 Im stetigen Fall ist

$$\int x\ln x\,dx = \frac{x^2}{2}\ln x - \int\frac{x^2}{2}\frac{1}{x}dx = \frac{x^2}{2}\ln x - \frac{x^2}{4}\,.$$

Analog ist im diskreten Fall (durch direkte Rechnung)

$$\begin{aligned}
\sum_{k=1}^{n-1} kH_k &= 1\cdot(\frac{1}{1}) + 2\cdot(\frac{1}{1}+\frac{1}{2}) + 3\cdot(\frac{1}{1}+\frac{1}{2}+\frac{1}{3}) + \cdots \\
&= \sum_{k=1}^{n-1}\left(\frac{1}{k}\cdot(k+(k+1)+\cdots+(n-1))\right) \\
&= \sum_{k=1}^{n-1}\left(\frac{1}{k}\cdot\left(\frac{n(n-1)}{2} - \frac{k(k-1)}{2}\right)\right) \\
&= \frac{n(n-1)}{2}\cdot H_{n-1} - \sum_{k=1}^{n-1}\frac{k-1}{2} \\
&= \frac{n(n-1)}{2}\cdot H_n - \frac{n-1}{2} - \frac{(n-2)(n-1)}{4} \\
&= \frac{n(n-1)}{2}\cdot H_n - \frac{n(n-1)}{4}\,.
\end{aligned}$$

Oder durch partielle Summation

$$
\begin{aligned}
\sum(\underbrace{x^{\underline{1}}}_{\substack{\Delta g \\ \Rightarrow g(x)=\frac{x^{\underline{2}}}{2}}} \cdot \underbrace{H_x}_{f})\delta x &= \frac{x^{\underline{2}}}{2} \cdot H_x - \sum(E(\frac{x^{\underline{2}}}{2}) \cdot x^{\underline{-1}})\delta x \\
&= \frac{x^{\underline{2}}}{2} \cdot H_x - \sum \frac{(x+1)^{\underline{2}} \cdot x^{\underline{-1}}}{2}\delta x \\
&= \frac{x^{\underline{2}}}{2} \cdot H_x - \sum \frac{x^{\underline{1}}}{2}\delta x \\
&= \frac{x^{\underline{2}}}{2} \cdot H_x - \frac{x^{\underline{2}}}{4}
\end{aligned}
$$

wegen

$$x^{\underline{1}} = x^{\underline{-1+2}} = x^{\underline{-1}} \cdot (x-(-1))^{\underline{2}}\,.$$

Daher ist

$$\sum_{k=1}^{n-1}(k \cdot H_k) = \frac{n^{\underline{2}}}{2} \cdot (H_n - \frac{1}{2})\,.$$

□

Beispiel A.4.4 Es ist

$$\Delta 2^{-(x-1)} = 2^{-x} - 2^{-(x-1)} = -2^{-x}$$

und daher

$$\sum_{i=0}^{n} \frac{1}{2^i} = \sum_{0}^{n+1} 2^{-x}\delta x = -2^{-(x-1)}\Big|_0^{n+1} = 2 - \frac{1}{2^n}\,.$$

□

Beispiel A.4.5 Es ist

$$
\begin{aligned}
\sum_{i=0}^{n} \frac{i}{2^i} &= \sum_{0}^{n+1}(\underbrace{x}_{f} \cdot \underbrace{2^{-x}}_{\Delta g})\delta x \\
&= (-x \cdot 2^{-(x-1)})\Big|_0^{n+1} + \sum_{0}^{n+1}(E2^{-(x-1)})\delta x \\
&= -\frac{n+1}{2^n} + \sum_{0}^{n+1} 2^{-x}\delta x
\end{aligned}
$$

$$
\begin{aligned}
&= -\frac{n+1}{2^n} - 2^{-(x-1)}\Big|_0^{n+1} \\
&= -\frac{n+1}{2^n} - \frac{1}{2^n} + 2 \\
&= 2 - \frac{n+2}{2^n}\,.
\end{aligned}
$$

□

Anhang B

Grundlagen der Wahrscheinlichkeitsrechnung

B.1 Definitionen

Definition B.1.1 (Grundbegriffe der Wahrscheinlichkeitsrechnung)

1. Sei W eine endliche Menge und $p : W \to [0,1]$ eine Funktion mit
$$\sum_{w \in W} p(w) = 1 \; .$$
Dann ist (W,p) ein *(endlicher) Wahrscheinlichkeitsraum.*
Für $p(w)$ schreiben wir auch p_w.

2. Unter einem *Ereignis* verstehen wir eine Teilmenge $V \subseteq W$.
Wir erweitern die Definition von p auf Ereignisse $V \subseteq W$ durch
$$p(V) = \sum_{w \in V} p_w \; .$$

3. Eine Funktion $X : W \to R$ heißt *Zufallsvariable.*

4. Für das Ereignis $X^{-1}(c) = \{w \in W \mid X(w) = c\}$ schreiben wir im folgenden auch $(X = c)$. Insbesondere ist
$$p(X = c) = \sum_{\substack{w \in W \\ X(w)=c}} p_w \; .$$
Analog schreiben wir $(X \leq c)$ für $\{w \in W \mid X(w) \leq c\}$.

5. Der *Erwartungswert* einer Zufallsvariablen X ist definiert als
$$E[X] = \sum_{w \in W} (p_w \cdot X(w)) \, .$$

$\square$

Lemma B.1.2 *Sei X eine Zufallsvariable. Dann ist*

$$E[X] = \sum_{c \in \mathbb{R}} (c \cdot p(X = c)) .$$

Beweis:

$$\begin{aligned} E[X] &= \sum_{w \in W} (p_w \cdot X(w)) \\ &= \sum_{c \in \mathbb{R}} \sum_{\substack{w \in W, \\ X(w)=c}} (p_w \cdot c) \\ &= \sum_{c \in \mathbb{R}} \left(c \cdot \sum_{\substack{w \in W, \\ X(w)=c}} p_w \right) \\ &= \sum_{c \in \mathbb{R}} (c \cdot p(X = c)) . \end{aligned}$$

□

Beispiel B.1.3 Wir verdeutlichen diese Definitionen an folgendem einfachen Beispiel :

"Wir werfen eine Münze solange, bis Kopf kommt".

Dann ist der Wahrscheinlichkeitsraum gegeben durch

$$\begin{aligned} W &= Z^* K = \{Z^i K \mid i \in \mathbb{N}_0\} , \\ p(Z^i K) &= \frac{1}{2^{i+1}} \text{ für } i \in \mathbb{N}_0 , \end{aligned}$$

wobei "Z" für das Ereignis "Zahl" und "K" für das Ereignis "Kopf" steht. Beachte, daß in der Tat

$$\sum_{w \in W} p_w = \sum_{i \geq 0} \frac{1}{2^{i+1}} = 1$$

ist. Die Zufallsvariable

$$X : W \mapsto \mathbb{R}, \quad X(Z^i K) = i + 1 \text{ für } i \in \mathbb{N}_0$$

zählt die Anzahl der Würfe, bis endlich zum ersten Mal Kopf erscheint. Ihr Erwartungswert ist

$$\begin{aligned} E[X] &= \sum_{i \geq 0} \frac{i+1}{2^{i+1}} \\ &= \sum_{i \geq 1} \frac{i}{2^i} \\ &\stackrel{(A.1.4)}{=} 2 . \end{aligned}$$

□

Lemma B.1.4 *(Linearität des Erwartungswerts)*
Seien X und Y Zufallsvariablen. Dann gilt

$$E[X+Y] = E[X] + E[Y]\,.$$

Beweis:

$$\begin{aligned} E[X+Y] &= \sum_{w \in W} (p_w \cdot (X+Y)(w)) \\ &= \sum_{w \in W} (p_w \cdot (X(w) + Y(w))) \\ &= E[X] + E[Y]\,. \end{aligned}$$

□

Wir interessieren uns oft für Zufallsvariablen, die nur nicht-negative ganze Zahlen als Werte annehmen.

Lemma B.1.5 *Sei X eine Zufallsvariable, die nur nicht-negative ganze Zahlen als Wert annimmt. Dann gilt*

$$E[X] = \sum_{i \in \mathbb{N}_0} (i \cdot p(X = i)) = \sum_{i \in \mathbb{N}_0} p(X > i)\,.$$

Beweis:

$$\begin{aligned} \sum_{i \in \mathbb{N}_0} p(X > i) &= \sum_{i \in \mathbb{N}_0} \sum_{j > i} p(X = j) \\ &= \sum_{j \in \mathbb{N}_0} \sum_{i < j} p(X = j) \\ &= \sum_{j \in \mathbb{N}_0} (j \cdot p(X = j))\,. \end{aligned}$$

□

Lemma B.1.6 (Markov-Ungleichung)
Sei X eine Zufallsvariable, die nur nicht-negative ganze Zahlen annimmt, mit Erwartungswert $E[X] = \mu$. Dann gilt für $t \geq 1$:

$$p(X \geq t \cdot \mu) \leq \frac{1}{t}\,.$$

Beweis:

$$\begin{aligned} E[X] &= \sum_{i \in \mathbb{N}_0} (i \cdot p(X = i)) \\ &\geq \sum_{i \geq t \cdot \mu} (i \cdot p(X = i)) \\ &\geq t \cdot \mu \cdot \sum_{i \geq t \cdot \mu} p(X = i) \\ &= t \cdot \mu \cdot p(X \geq t \cdot \mu)\,. \end{aligned}$$

□

Bem.: Eine 0-1-wertige Zufallsvariable heißt *Indikatorvariable*. Für eine Indikatorvariable X gilt

$$E[X] = 0 \cdot p(X = 0) + 1 \cdot p(X = 1) = p(X = 1)\,.$$

□