WS 2010/11

Effiziente Algorithmen und Datenstrukturen

Ernst W. Mayr

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2010WS/ea/

Wintersemester 2010/11

Kapitel 0 Organisatorisches

- Vorlesungen:
  - 4SWS
    Di 8:30–10:00 (MI 00.13.009A)
    Do 8:30–10:00 (MI HS2)
    Wahlpflichtvorlesung im Fachgebiet Algorithmen und Wissenschaftliches Rechnen (AWR), Bioinformatik
- Übung:
  - 2SWS Zentralübung: Di 14:15–15:45 (MI 00.08.038)
  - Übungsleitung: Tobias Lieber
- Umfang:
  - 4V+2ZÜ, 8 ECTS-Punkte (ModulNr.: IN2003)
- Sprechstunde:
  - nach Vereinbarung

- Vorkenntnisse:
  - Einführung in die Informatik
  - Grundlagen: Algorithmen und Datenstrukturen (GAD)
  - Einführung in die Theoretische Informatik (THEO)
  - Diskrete Strukturen, Diskrete Wahrscheinlichkeitstheorie (DS, DWT)
- Weiterführende Vorlesungen:
  - Effiziente Algorithmen und Datenstrukturen II
  - Randomisierte Algorithmen
  - Komplexitätstheorie
  - Approximationsalgorithmen
  - Internetalgorithmik
  - ...
- Webseite:

  http://wwwmayr.in.tum.de/lehre/2010WS/ea/

- Übungsleitung:
  - Tobias Lieber, MI 03.09.060 (lieber@in.tum.de)
    Sprechstunde: Mi 13:00–14:00
- Sekretariat:
  - Frau Lissner, MI 03.09.052 (lissner@in.tum.de)

- Übungsaufgaben und Klausur:
  - Ausgabe jeweils am Dienstag auf der Webseite der Vorlesung
  - Abgabe eine Woche später vor der Vorlesung
  - Besprechung in der Zentralübung
- Klausur:
  - Zwischenklausur (50% Gewicht), Termin: Fr 17.12.2010, 16:30–18:30 (MI HS1)
  - Endklausur (50% Gewicht), Termin: Fr 18.02.2011, 15:00–17:00 (MI HS1)
  - Wiederholungsklausur, Termin: tba
  - bei den Klausuren sind *keine* Hilfsmittel außer einem handbeschriebenen DIN-A4-Blatt zugelassen
  - Für das erfolgreiche Bestehen des Moduls sind erforderlich:
    1. Bestehen der zweigeteilten Klausur (mindestens 40% der Gesamtpunktzahl)
    2. Erreichen von mindestens 40% der Punkte bei den Hausaufgaben
  - vorauss. 12 Übungsblätter,
    das erste am 26. Oktober, das letzte am 2. Februar,
    jedes 40 Punkte

1. Vorlesungsinhalt

- Grundlagen
  - Maschinenmodelle
  - Komplexitätsmaße
- Höhere Datenstrukturen
  - Suchbäume
  - Hashing
  - Priority Queues
  - selbstorganisierende Datenstrukturen
  - Union/Find-Datenstrukturen
- Sortieren und Selektieren
- Minimale Spannbäume
- Kürzeste Wege
- Matchings in Graphen
- Netzwerkfluss

2. Literatur


Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman:
The design and analysis of computer algorithms,
Addison-Wesley Publishing Company: Reading (MA), 1974


Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:
Introduction to algorithms,
McGraw-Hill, 1990


Michael T. Goodrich, Roberto Tamassia:
Algorithm design: Foundations, analysis, and internet examples,
John Wiley & Sons, 2002


Volker Heun:
Grundlegende Algorithmen: Einführung in den Entwurf und die Analyse effizienter Algorithmen,
2. Auflage, Vieweg, 2003


Donald E. Knuth:
The art of computer programming. Vol. 1: Fundamental algorithms,
3. Auflage, Addison-Wesley Publishing Company: Reading (MA), 1997


Donald E. Knuth:
The art of computer programming. Vol. 3: Sorting and searching,
3. Auflage, Addison-Wesley Publishing Company: Reading (MA), 1997


Uwe Schöning:
Algorithmik,
Spektrum Akademischer Verlag, 2001


Christos H. Papadimitriou, Kenneth Steiglitz:
Combinatorial Optimization: Algorithms and Complexity,
Prentice Hall, 1982


Steven S. Skiena:
The Algorithm Design Manual,
Springer, 1998

Kapitel I Grundlagen

1. Ein einleitendes Beispiel

Berechnung von F_n, der n-ten Fibonacci-Zahl:

$$
\begin{aligned}
F_0 &= 0 \ , \\
F_1 &= 1 \ , \\
F_n &= F_{n-1} + F_{n-2} \text{ für alle } n \geq 2 \ .
\end{aligned}
$$

Also:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

1. **Methode:** Funktionales Programmieren

$$
\begin{aligned}
f(n) := \ & \textbf{if } n = 0 \textbf{ then } 0 \\
& \textbf{elif } n = 1 \textbf{ then } 1 \\
& \textbf{else } f(n-1) + f(n-2) \\
& \textbf{fi}
\end{aligned}
$$

Sei $T(n)$ der Zeitbedarf für die Berechnung von $f(n)$. Dann gilt:

$$
\begin{aligned}
T(0) &= T(1) = 1\,; \\
T(n) &= T(n-1) + T(n-2) \text{ für } n \geq 2\,.
\end{aligned}
$$

Damit gilt:

$$T(n) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

für geeignete Konstanten c_1 und c_2.

Einsetzen der Werte für $n = 0$ und $n = 1$ ergibt

$$c_1 = \frac{1}{2} + \frac{\sqrt{5}}{10}$$
$$c_2 = \frac{1}{2} - \frac{\sqrt{5}}{10}$$

und damit

$$T(n) = \left(\frac{1}{2} + \frac{\sqrt{5}}{10}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1}{2} - \frac{\sqrt{5}}{10}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n .$$

2. Methode: Dynamische Programmierung

array $F[0:n]$; $F[0] := 0$; $F[1] := 1$;
for $i := 2$ **to** n **do**
  $F[i] := F[i-1] + F[i-2]$

Der Zeitbedarf dieses Algorithmus ist offensichtlich

$$\mathcal{O}(n) \ .$$

3. Methode: Direkt (mit etwas mathematischer Arbeit)

$$F(n) := \text{round}\left(\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$
$$= \left\lfloor \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{1}{2} \right\rfloor .$$

Der Zeitbedarf hier ist

$$\mathcal{O}(\log n) .$$

2. Versuch einer Definition

Was sind „kombinatorische“ Algorithmen?

Eine mögliche Antwort:

Probleme, die, könnten wir alle Fälle aufzählen, trivial wären, aber eine sehr große Anzahl von Fällen haben.

Beispiele: Hamiltonscher Kreis (NP-vollständig), Eulerscher Kreis (P)

3. Ziel der Vorlesung

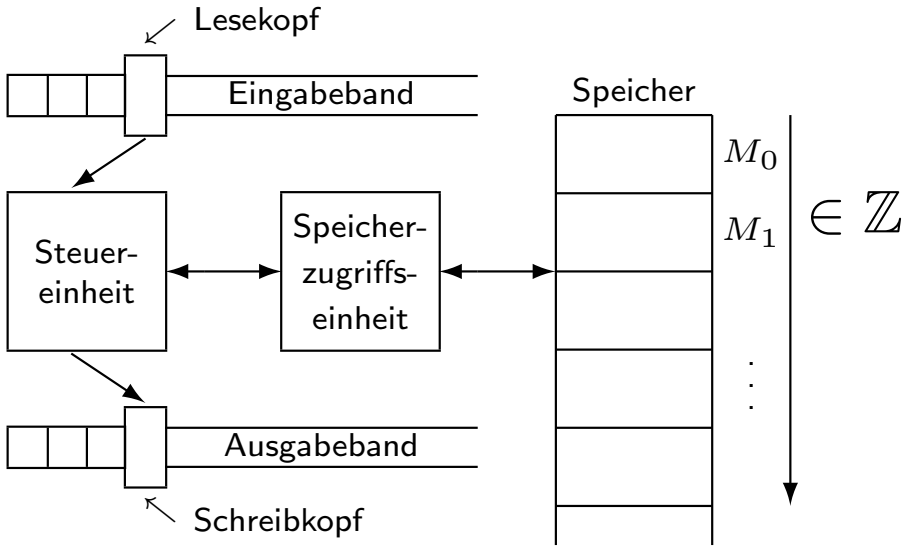
Der Zweck der Vorlesung ist das Studium fundamentaler Konzepte in der Algorithmentheorie. Es werden relevante Maschinenmodelle, grundlegende und höhere Datenstrukturen sowie der Entwurf und die Analyse sequentieller Algorithmen besprochen. Dabei wird eine Reihe verschiedener Analysemethoden (für entsprechend unterschiedliche Anforderungen) eingeführt.

Die betrachteten Problemklassen umfassen eine umfangreiche Auswahl der in der Praxis relevanten kombinatorischen Probleme, wobei die algorithmischen Ansätze sich in dieser Vorlesung jedoch praktisch auf deterministische, sequentielle, exakte Algorithmen beschränken.

Für weiterführende Konzepte und Ansätze (z.B. probabilistische, parallele, approximative Algorithmen) wird auf entsprechende Vorlesungen verwiesen.

4. Maschinenmodelle

- Turingmaschine (TM)
- Registermaschine (RAM)
- Boolesche Schaltkreise
- (Quantencomputer)
- (DNA-Computer)
- ...



Registermaschine

G. Bilardi, K. Ekanadham, P. Pattnaik:
On approximating the ideal random access machine by physical machines.
J.ACM **56**(5), Article 27, ACM, 2009

5. Komplexitätsmaße

Ein Problem ist formal eine Sprache $L \subseteq \Sigma^*$. $x \in \Sigma^*$ heißt eine Probleminstanz für L, wenn wir untersuchen wollen, ob $x \in L$.

Sei M eine (Turing- oder) Registermaschine.

- M entscheidet L, falls für alle $x \in \Sigma^*$ M nach endlicher Zeit hält mit

$$\begin{cases} \text{Antwort „ja“, falls } x \in L \\ \text{Antwort „nein“, falls } x \notin L \end{cases}$$

- M akzeptiert L, falls für alle $x \in \Sigma^*$ gilt

$$\begin{cases} \text{falls } x \in L : M \text{ hält mit Antwort “ja“} \\ \text{falls } x \notin L : M \text{ hält mit Antwort “nein“ oder hält nicht.} \end{cases}$$

- Berechnung von Funktionen:
  Seien Σ, Γ Alphabete. Eine TM (bzw. RAM) M berechnet eine (partielle) Funktion $f : \Sigma^* \to \Gamma^*$ gdw. für alle x im Definitionsbereich von f gilt:
  bei Eingabe x hält M nach endlich vielen Schritten, und zwar mit Ausgabe $f(x)$.

Wir berechnen die Komplexität eines Problems in Abhängigkeit von der Länge der Eingabe:
Eingaben $x \in \Sigma^n$ haben Länge n.

Insbesondere bei Funktionen oder Problemen, deren Eingabe als „als aus n Argumenten bestehend“ interpretiert werden kann, betrachten wir oft auch die uniforme Eingabelänge n.

Beispiel 1

Sollen n Schlüssel $\in \Sigma^*$ (vergleichsbasiert) sortiert werden, so nehmen wir als Eingabelänge gewöhnlich n, die Anzahl der Schlüssel, und nicht ihre Gesamtlänge.

Komplexitätsressourcen:

Man betrachtet u.a.

- Rechenzeit
- Speicherplatz
- Anzahl von Vergleichen
- Anzahl von Multiplikationen
- Schaltkreisgröße
- Programmgröße
- Schachtelungstiefe von Laufschleifen
- ...

Komplexität der Ressourceneinheiten:

Wir unterscheiden

- uniformes Kostenmodell: Die Kosten jeder Ressourceneinheit sind 1.
- logarithmisches Kostenmodell: Die Kosten eines Rechenschritts sind durch die Länge der Operanden bestimmt:

  1. Der Zeitbedarf eines Rechenschritts ist gleich der größten Länge eines Operanden des Rechenschritts.
  2. Der Platzbedarf einer Speicherzelle ist gleich der größten Länge eines darin gespeicherten Wertes.

Wir unterscheiden verschiedene Arten der Komplexität:

- worst-case Komplexität:

$$C_{\text{wc}}(n) := \max\{C(x);\ |x| = n\}$$

- durchschnittliche Komplexität (average complexity):

$$C_{\text{avg}}(n) := \frac{1}{|\Sigma^n|} \sum_{|x|=n} C(x)$$

allgemeiner: Wahrscheinlichkeitsmaß μ

$$C_{\text{avg}}(n) := \sum_{x \in \Sigma^n} \mu(x) \cdot C(x)$$

Wir unterscheiden verschiedene Arten der Komplexität:

- amortisierte Komplexität:
  durchschnittliche Kosten der Operationen in einer Folge von n Operationen
  worst-case über alle Folgen der Länge n von Operationen
- probabilistische oder randomisierte Komplexität:
  Algorithmus hat Zufallsbits zur Verfügung. Erwartete Laufzeit (über alle Zufallsfolgen) für feste Eingabe x, dann worst-case für alle $|x| = n$.

Beispiel 2

$r := 2$
for $i := 1$ **to** n **do** $r := r^2$ **od**
co `das Ergebnis ist` 2^{2^n} **oc**

- Zeitbedarf:
  - uniform: n Schritte
  - logarithmisch: $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$
- Platzbedarf:
  - uniform: $\mathcal{O}(1)$
  - logarithmisch: 2^n

6. Wachstumsverhalten von Funktionen

f, g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_+$.

- $g = \mathcal{O}(f)$ [auch: $g(n) = \mathcal{O}(f(n))$ oder $g \in \mathcal{O}(f)$] gdw.

$$(\exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \leq c \cdot f(n)]$$

- $g = \Omega(f)$ [auch: $g(n) = \Omega(f(n)$ oder $g \in \Omega(f)$] gdw.

$$(\exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \geq c \cdot f(n)]$$

- $g = \Theta(f)$ gdw. $g = \mathcal{O}(f)$ und $g = \Omega(f)$

f, g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_+$.

- $g = o(f)$ gdw.

$$(\forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \leq c \cdot f(n)]$$

- $g = \omega(f)$ gdw.

$$(\forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \geq c \cdot f(n)]$$

- $g = \Omega_\infty(f)$ gdw.

$$(\exists c > 0)[g(n) \geq c \cdot f(n) \text{ für unendlich viele } n]$$

- $g = \omega_\infty(f)$ gdw.

$$(\forall c > 0)[g(n) \geq c \cdot f(n) \text{ für unendlich viele } n]$$

Beispiel 3

- n^3 ist nicht $\mathcal{O}\left(\frac{n^3}{\log n}\right)$.
- $n^3 + n^2$ ist nicht $\omega(n^3)$.
- $100n^3$ ist nicht $\omega(n^3)$.

Bemerkung:
Die Groß-O-Notation wurde von D. E. Knuth in der Algorithmenanalyse eingeführt, siehe z.B.

Donald E. Knuth:
Big omicron and big omega and big theta.
SIGACT News **8**(2), pp. 18–24, ACM SIGACT, 1976

Sie wurde ursprünglich von Paul Bachmann (1837–1920) entwickelt und von Edmund Landau (1877–1938) in seinen Arbeiten verbreitet.

	Problemgröße n					
Wachs-tums-rate	10	20	30	40	50	60
n	.00001 Sekunden	.00002 Sekunden	.00003 Sekunden	.00004 Sekunden	.00005 Sekunden	.00006 Sekunden
n^2	.0001 Sekunden	.0004 Sekunden	.0009 Sekunden	.0016 Sekunden	.0025 Sekunden	.0036 Sekunden
n^3	.001 Sekunden	.008 Sekunden	.027 Sekunden	.064 Sekunden	.125 Sekunden	.216 Sekunden
n^5	.1 Sekunden	3.2 Sekunden	24.3 Sekunden	1.7 Minuten	5.2 Minuten	13.0 Minuten
2^n	.001 Sekunden	1.0 Sekunden	17.9 Minuten	12.7 Tage	35.7 Jahre	366 Jahrhdte
5^n	.059 Sekunden	58 Minuten	6.5 Jahre	3855 Jahrhdte	2×10^8 Jahrhdte	1.3×10^{13} Jahrhdte

Beispiel 4 (Mergesort)

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + cn \\
&= cn + 2T\left(\frac{n}{2}\right) \\
&= cn + 2\left(c\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= cn + cn + 4T\left(\frac{n}{4}\right) \\
&\approx cn \log_2 n \text{ (nur genau für Zweierpotenzen)}
\end{aligned}
$$

Methoden zur Lösung von Rekursionsgleichungen

1. Multiplikatorenmethode
2. Lineare homogene Rekursionsgleichungen können mit Hilfe des charakteristischen Polynoms gelöst werden
3. Umwandlung inhomogener Rekursionsgleichungen in homogene
4. Erzeugendenfunktionen
5. Transformation des Definitions- bzw. Wertebereichs
6. ...

Es gibt keinen vollständigen Satz von Methoden.

7.1 Multiplikatoren

Sei $f_1 = 1$, $f_n = 2f_{n-1} + n$ für $n \geq 2$.

$$\left.\begin{array}{rl} f_n = 2f_{n-1} + n & \mid \cdot 1 \\ f_{n-1} = 2f_{n-2} + n - 1 & \mid \cdot 2 \\ \vdots & \vdots \\ f_2 = 2f_1 + 2 & \mid \cdot 2^{n-2} \\ f_1 = 1 & \mid \cdot 2^{n-1} \end{array}\right\} \begin{array}{l} f_n = 2^{n-1} f_1 + \sum_{i=0}^{n-2} 2^i (n-i) \\ \quad = 2^{n+1} - n - 2 \end{array}$$

Durch Addieren aller Gleichungen erhalten wir:

$$f_n = 2^{n-1} + \sum_{i=0}^{n-2} 2^i(n-i) = \underbrace{2^{n-1}}_{(1)} + \underbrace{n\sum_{i=0}^{n-2} 2^i}_{(2)} - \underbrace{\sum_{i=0}^{n-2} i \cdot 2^i}_{(3)}$$

Term (2) (geometrische Reihe):

$$n\sum_{i=0}^{n-2} 2^i = n(2^{n-1} - 1)$$

Term (3) (mit der Substitution $n-2=k$ und x für 2):

$$\begin{aligned}\sum_{i=0}^{k} ix^i &= \sum_{i=1}^{k} ix^i = x\sum_{i=1}^{k} ix^{i-1} \\ &= x\sum_{i=1}^{k} \frac{dx^i}{dx} = x\frac{d}{dx}\sum_{i=1}^{k} x^i \\ &= x\frac{d}{dx}\left(\frac{x^{k+1}-1}{x-1}\right) \\ &= \frac{kx^{k+2}-x^{k+1}(k+1)+x}{(x-1)^2}\end{aligned}$$

Einsetzen von $k=n-2$ und $x=2$ ergibt:

$$\sum_{i=0}^{n-2} i2^i = 2^n(n-2)-2^{n-1}(n-1)+2 = 2^n n-2^{n+1}-2^{n-1}n+2^{n-1}+2$$

$(2) - (3)$:

$$n\sum_{i=0}^{n-2} 2^i - \sum_{i=0}^{n-2} i2^i = n \cdot (2^{n-1} - 1) - 2^n(n-2) + 2^{n-1}(n-1) - 2$$
$$= 2^{n-1}(2n-1) - 2^n(n-2) - n - 2$$

und schließlich $(1) + (2) - (3)$:

$$2^{n-1} + n\sum_{i=0}^{n-2} 2^i - \sum_{i=0}^{n-2} i2^i = 2^{n-1}(1 + 2n - 1) - 2^n(n-2) - n - 2$$
$$= 2^{n+1} - n - 2$$

7.2 Charakteristisches Polynom

Sei

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \text{ für } n \geq 2 \, .$$

Es handelt sich hier um eine lineare homogene Rekursionsgleichung zweiter Ordnung.

Ansatz: $f_n := a^n$ für ein unbekanntes a.
Dann muss gelten: $a^n - a^{n-1} - a^{n-2} = 0$. Da hier $a \neq 0$:

$$a^2 - a - 1 = 0; \text{ also } a_{1/2} = \frac{1 \pm \sqrt{5}}{2}$$

Falls $f_n = a_1^n$ und $f_n = a_2^n$ Lösungen der Rekursionsgleichung sind, dann auch $f_n = c_1 a_1^n + c_2 a_2^n$, für beliebige Konstanten c_1 und c_2.

$f_1 = 1$ und $f_0 = 0$ liefern zwei Gleichungen für c_1 und c_2, mit der Lösung:

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

Satz 5

Sei $p(x)$ das charakteristische Polynom zur (linearen homogenen) Rekursionsgleichung

$$p_0 f_n + p_1 f_{n-1} + \cdots + p_k f_{n-k} = 0 \qquad (1)$$

mit den konstanten Koeffizienten p_i. Seien r_i, $i = 1, \ldots, m$ die (i.a. komplexen) Wurzeln von $p(x)$, jeweils mit Vielfachheit m_i. Dann ist die allgemeine Lösung der Rekursionsgleichung (1) gegeben durch

$$f_n = \sum_{i=1}^{m} \left(r_i^n \sum_{j=0}^{m_i - 1} c_{ij} n^j \right),$$

mit Konstanten c_{ij}.

7.3 Erzeugendenfunktionen

Für lineare und nicht lineare Rekursionsgleichungen erhält man oft eine Lösung, indem man die f_n als Koeffizienten einer Potenzreihe betrachtet und eine geschlossene Form der dadurch definierten Funktion sucht.

Definition 6 (Erzeugendenfunktion)

Sei die Folge $(f_n)_{n\geq 0}$ gegeben. Die zugehörige

- (gewöhnliche) Erzeugendenfunktion ist

$$F(z) := \sum_{n=0}^{\infty} f_n z^n; \qquad z \in \mathbb{C}\,;$$

- exponentielle Erzeugendenfunktion ist

$$F(z) = \sum_{n\geq 0} \frac{\mathsf{f}_n}{n!} z^n; \qquad z \in \mathbb{C}\,.$$

Beispiel 7

1. Die Erzeugendenfunktion der Folge $(1, 0, 0, \ldots)$ ist

$$F(z) = 1\,.$$

2. Die Erzeugendenfunktion der konstanten Folge $(1, 1, 1, \ldots)$ ist

$$F(z) = \frac{1}{1-z}\,.$$

Falls $F(z) = \sum_{n>0} f_n z^n$, bezeichnet

$$[z^n]F(z)$$

den n-ten Koeffizienten f_n.

Sei $F(z) = \sum_{n \geq 0} f_n z^n$ und $G(z) = \sum_{n \geq 0} g_n z^n$.

ErzFkt.	n-tes Folgenglied	Anmerkungen:
cF	cf_n	
$F + G$	$f_n + g_n$	
$F \cdot G$	$h_n := \sum_{i=0}^{n} f_i g_{n-i}$ (Konvolution)	$\left(\sum_{i \geq 0} f_i z^i\right)\left(\sum_{i \geq 0} g_i z^i\right) = \sum_{i \geq 0} h_i z^i$ (mit $h_n = \sum_{i=0}^{n} f_i g_{n-i}$)
$z^k F$	if $n < k$ then 0 else f_{n-k} fi	
$\frac{F(z)}{1-z}$	$\sum_{i=0}^{n} f_i$	$\frac{1}{1-z} = \sum_{n \geq 0} z^n$
$z\frac{dF(z)}{dz}$	nf_n	
$\int_0^x F(t)dt$	if $n = 0$ then 0 else $\frac{f_{n-1}}{n}$ fi	$f_n z^n$ geht über auf $f_n \frac{z^{n+1}}{n+1}$
$F(cz)$	$c^n f_n$	$F(cz) = \sum_{n \geq 0} f_n c^n z^n$

Beispiel 8

$$F(z) := \sum_{n\geq 0} 2^n z^n = \frac{1}{1-2z}$$

$$G(z) := \sum_{n\geq 0} n z^n = \frac{z}{(1-z)^2}$$

$$\Rightarrow F(z)G(z) = \frac{z}{(1-z)^2(1-2z)} = \sum_{n\geq 0}\sum_{i=0}^{n}(n-i)2^i z^n$$

Partialbruchzerlegung:

$$\frac{z}{(1-z)^2(1-2z)} = \overbrace{\frac{-2}{(1-z)}}^{(1)} + \frac{-z}{(1-z)^2} + \overbrace{\frac{2}{1-2z}}^{(2)}$$
$$= \underbrace{\sum_{n\geq 0} 2^{n+1} z^n}_{(2)} - \sum_{n\geq 0} n z^n - \underbrace{2\sum_{n\geq 0} z^n}_{(1)} ;$$

Also:

$$\sum_{i=0}^{n} 2^i (n-i) = [z^n](FG)(z)$$
$$= [z^n]\left(\sum_{n\geq 0}(2^{n+1} - n - 2)z^n\right)$$
$$= 2^{n+1} - n - 2 \,.$$

7.4 Transformation des Definitions- bzw. Wertebereichs

Beispiel 9

$$f_0 = 1$$
$$f_1 = 2$$
$$f_n = f_{n-1} \cdot f_{n-2} \text{ für } n \geq 2 \, .$$

Setze

$$g_n := \log f_n \, .$$

Dann gilt

$$g_n = g_{n-1} + g_{n-2} \text{ für } n \geq 2$$
$$g_1 = \log 2 = 1, \; g_0 = 0 \text{ (für } \log = \log_2 \text{)}$$
$$g_n = F_n \text{ (n-te Fibonacci-Zahl)}$$
$$f_n = 2^{F_n}$$

Beispiel 10

$$f_1 = 1$$
$$f_n = 3f_{\frac{n}{2}} + n; \text{ für } n = 2^k \ ;$$

Setze

$$g_k := f_{2^k} \ .$$

Beispiel 10

Dann gilt:

$$g_0 = 1$$
$$g_k = 3g_{k-1} + 2^k,\ k \geq 1$$

Damit ergibt sich:

$$g_k = 3^{k+1} - 2^{k+1},\ \text{also}$$
$$f_n = 3 \cdot 3^k - 2 \cdot 2^k$$
$$= 3(2^{\log 3})^k - 2 \cdot 2^k$$
$$= 3(2^k)^{\log 3} - 2 \cdot 2^k$$
$$= 3n^{\log 3} - 2n\ .$$

Kapitel II Höhere Datenstrukturen

1. Grundlegende Operationen

Es sei U das Universum von Schlüsseln mit einer (totalen) Ordnung $\leq$. $S \subseteq U$ sei eine Teilmenge der Schlüssel. Gegeben seien eine Menge von Datensätzen $x_1, \cdots, x_n$, wobei jeder Datensatz x durch einen Schlüssel $k(x) \in S$ gekennzeichnet ist. Jeder Datensatz x besteht aus seinem Schlüssel $k(x)$ und seinem eigentlichen Wert $v(x)$.

IsElement(k, S):	ist $k \in S$, wenn ja, return $v(k)$
Insert(k, S):	$S := S \cup \{k\}$
Delete$(k; S)$:	$S := S \setminus \{k\}$
FindMin(S):	return $\min S$
FindMax(S):	return $\max S$
DeleteMin(S):	$S := S \setminus \min S$
ExtractMin(S):	return $\min S$, $S := S \setminus \min S$
DecreaseKey(k, Δ, S):	ersetze k durch $k - \Delta$
Union(S_1, S_2):	$S_1 := S_1 \cup S_2$
Find(k):	falls $k \in S$, so finde x mit $k = k(x)$
Merge(S_1, S_2):	$S_1 := S_1 \cup S_2$, falls $S_1 \cap S_2 = \emptyset$
Split(S_1, k, S_2):	$S_2 := \{k' \in S_1 \mid k' \geq k\}$ $S_1 = \{k' \in S_1 \mid k' < k\}$
Concatenate(S_1, S_2):	$S_1 := S_1 \cup S_2$; Vorauss.: *FindMax*$(S_1) \leq$ *FindMin*(S_2)

Datenstrukturklasse	mindestens angebotene Funktionen	realisiert in
Wörterbuch (Dictionary)	*IsElement()*, *Insert()*, *Delete()*	Hashtable, Suchbäume
Vorrangwarteschlange (Priority Queue)	*FindMin()*, *Insert()*, *Delete()*, *[IsElement()]*	balancierte, leftist Bäume, Binomial Queues
Mergeable heaps	*FindMin()*, *Insert()*, *Delete()*, *Merge()*	2-3-Bäume, Binomial Queues, Leftist-Bäume
Concatenable queues	*FindMin()*, *Insert()*, *Delete()*, *Concatenate()*	2-3-Bäume

Definition 11

- Bei einem externen Suchbaum werden die Schlüssel nur an den Blättern gespeichert, die inneren Knoten enthalten Verwaltungsinformationen.
- Bei internen Suchbäumen liegen die Schlüssel an den internen Knoten, die Blätter sind leere Knoten. Zeiger auf Blätter sind daher NIL-Pointer und werden gewöhnlich nicht angegeben.

2. (Balancierte) Suchbäume

Wir betrachten zunächst zwei Familien höhenbalancierter externer Suchbäume.

2.1 (a,b)-Bäume

Definition 12
Ein (a, b)-Baum ist ein externer Suchbaum mit folgenden Eigenschaften:

1. alle Blätter haben die gleiche Tiefe;
2. die Anzahl der Kinder eines jeden internen Knoten ist $\leq b$ und $\geq$ a (für die Wurzel: ≥ 2);
3. es gilt $b \geq 2a - 1$.
4. (a, b)-Bäume mit $b = 2a - 1$ heißen auch B-Bäume.

Bei jedem internen Knoten v mit $d = d(v)$ Kindern werden $d-1$ Schlüssel $k_1, \ldots, k_{d-1}$ gespeichert, so dass (mit $k_0 := -\infty$, $k_d = +\infty$) gilt:

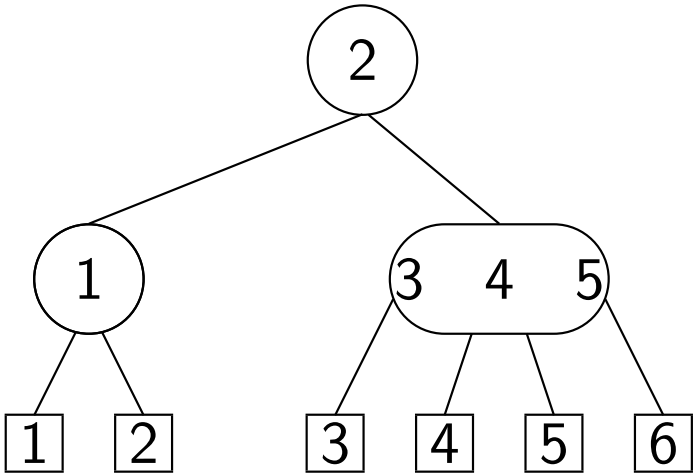
$$k_{i-1} < \text{ alle Schlüssel im } i\text{-ten Unterbaum von } v \leq k_i \ .$$

Also z.B.: $k_i :=$ maximaler Schlüssel im i-ten Unterbaum von v.

Weiter gilt (wie in jedem Suchbaum)

$$\max\{\text{Schlüssel im } i\text{-ten UB}\} \leq \min\{\text{Schlüssel im } i+1\text{-ten UB}\}\,.$$

Beispiel 13



$(2,4)$-Baum

Lemma 14

Sei T ein (a,b)-Baum mit n Blättern und der Höhe h (Höhe definiert als Anzahl der Kanten auf einem Pfad von der Wurzel zu einem Blatt). Dann gilt:

1. $2a^{h-1} \leq n \leq b^h$;
2. $\log_b n = \frac{\log n}{\log b} \leq h \leq 1 + \log_a \frac{n}{2}$.

Beweis:

1. n ist maximal für vollständigen Baum mit Verzweigungsgrad b und minimal für einen Baum, wo die Wurzel Grad 2 und alle anderen internen Knoten Grad a haben.
2. folgt durch Umformung aus (1).

□

Operationen:

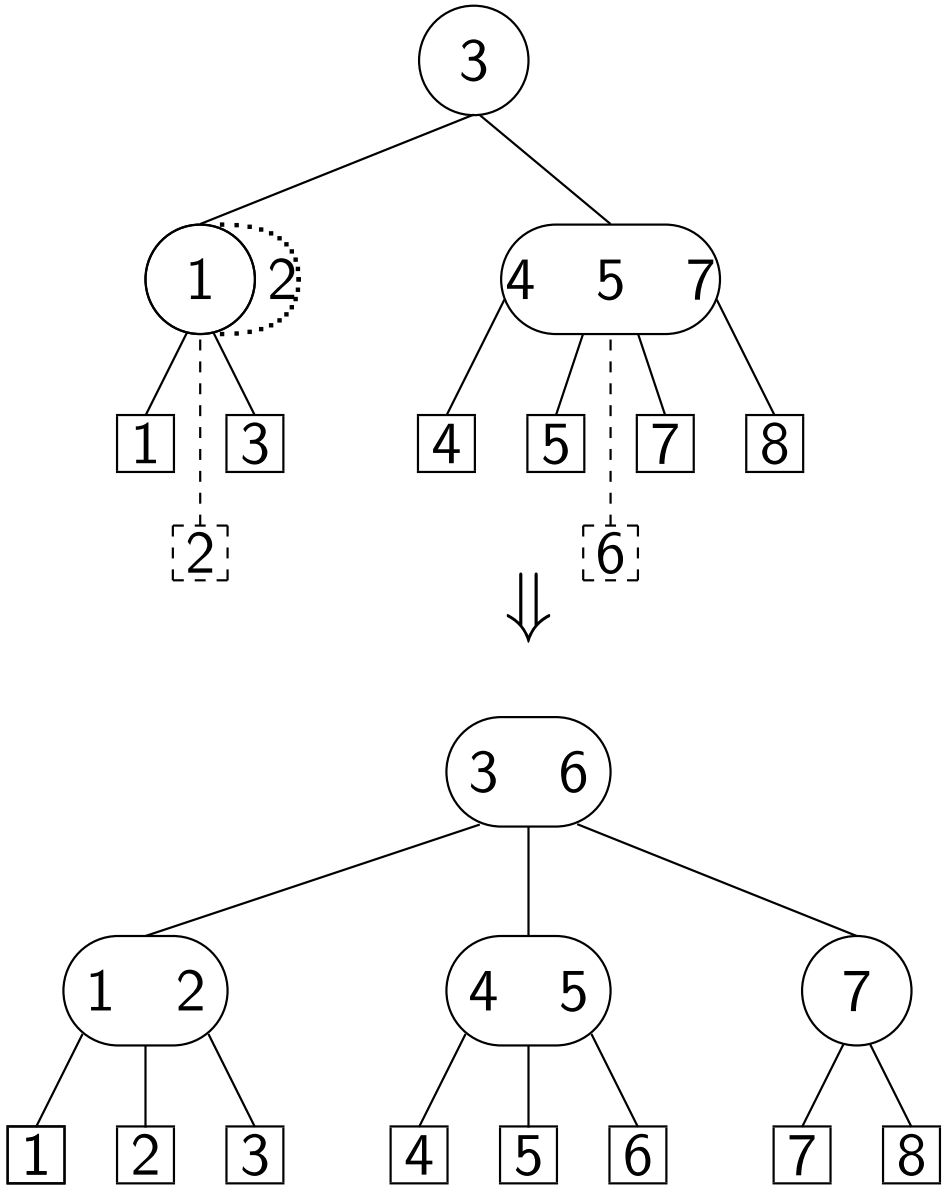
1. *IsElement*(k, S)

$v :=$ `Wurzel von` T`;`
while $v \neq$ `Blatt` **do**
  $i = \min\{s;\ 1 \leq s \leq$ (`#Kinder von` v) `und` $k \leq k_s\}$`;`
  $v :=$ i`-tes Kind von` v`;`
if $k = k(v)$ **then return** $v(k)$ **else return false fi** ;

Zeitbedarf: $\theta(h) = \theta(\log n)$

2. *Insert*(k, S)

```
Führe IsElement(k, S) aus; ⇝ Blatt w;
if k ≠ k(w) then
  co falls k < max S, enthält w den kleinsten
  Schlüssel ∈ S, der > k ist oc
  füge k if k < max S then links else rechts fi von w
  ein;
  v := Vater von w;
  if v hat nun mehr als b Kinder then
    Rebalancierung(v)
  fi
fi
```


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

Rebalancierung(v):

- spalte v in zwei Knoten v_1 und v_2, v_1 übernimmt die linke „Hälfte“ der Kinder von v, v_2 den Rest; da

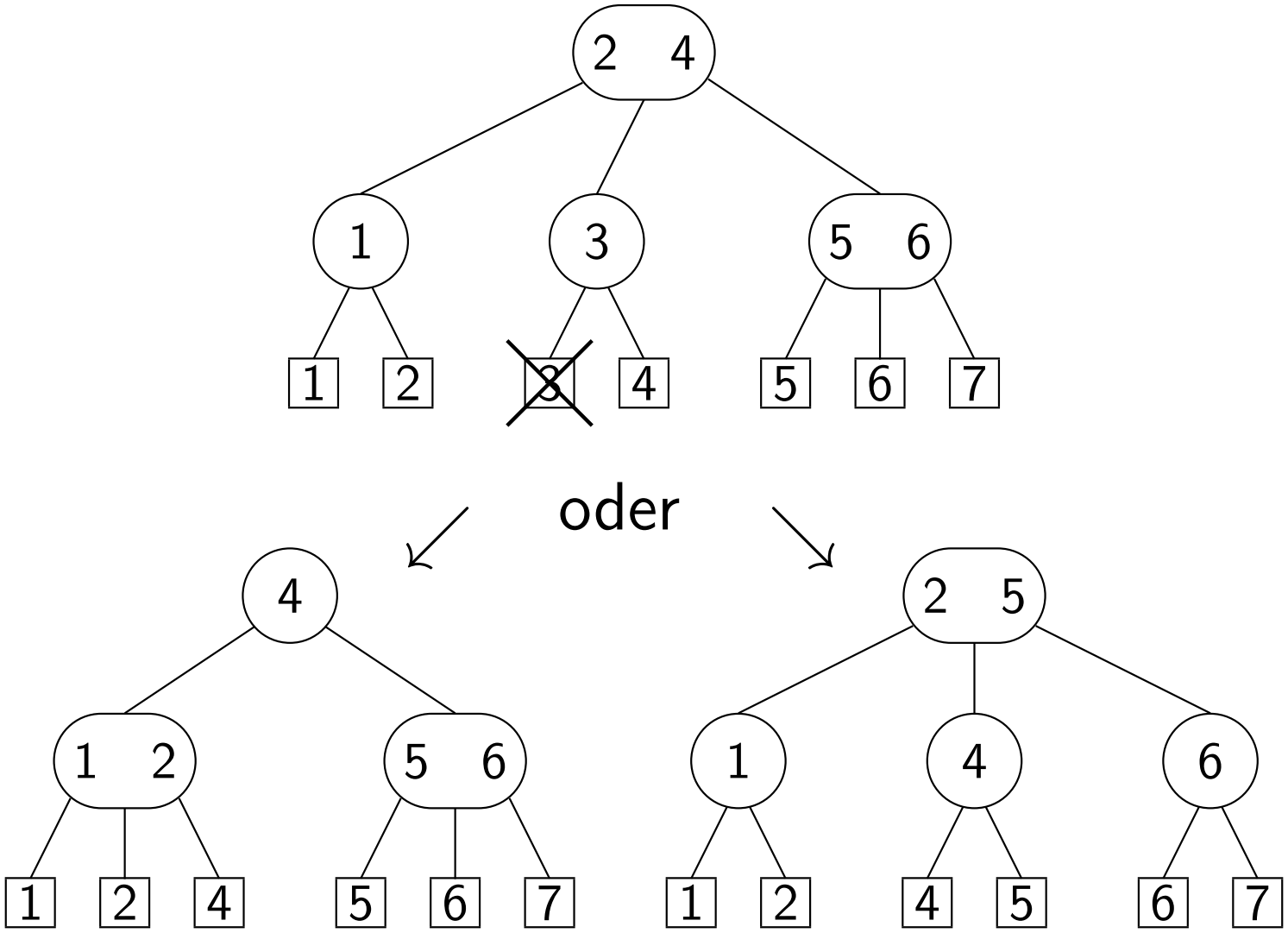
$$a \leq \lfloor \frac{b+1}{2} \rfloor \leq \lceil \frac{b+1}{2} \rceil \leq b \text{ und } b \geq 2a-1\,,$$

  erfüllen v_1 und v_2 die Gradbedingung;
- Vereinfachung: Falls ein unmittelbarer Nachbar v' von v Grad $< b$ hat, übernimmt v' das äußerste linke/rechte Kind von v.
- Falls nun der Vater u von v mehr als b Kinder hat, führe Rebalancierung(u) aus;
- Falls u die Wurzel des Baums ist, kann dadurch eine neue Wurzel geschaffen werden und die Höhe des (a,b)-Baums wachsen.

Kosten $\mathcal{O}(\log n)$

3. *Delete*(k, S):

Führe *IsElement*(k, S) aus. $\rightsquigarrow$ Blatt w. Sei $k(w) = k$;
$v :=$ Vater von w;
Lösche w;
if v hat nunmehr $< a$ Kinder **then**
  führe rekursiv aufsteigend Rebalancierung$'(v)$ durch
fi


2 4
1
3
5 6
1
2
3
4
5
6
7
oder
4
1 2
5 6
1
2
4
5
6
7
2 5
1
4
6
1
2
4
5
6
7

Rebalancierung$'(v)$:

- Falls ein unmittelbarer Nachbar v' von v Grad $> a$ hat, adoptiert v das nächste Kind von v';
- Ansonsten wird v mit einem unmittelbaren Nachbarn verschmolzen; die Gradbedingung für den dadurch entstehenden Knoten ist erfüllt; die Rebalancierung wird rekursiv/iterativ für den Vaterknoten fortgesetzt.

Zeitbedarf: $\mathcal{O}(\log n)$

Spezialfälle von (a, b)-Bäumen:

- $(2, 3)$-Bäume;
- $(2, 4)$-Bäume;
- $(a, 2a - 1)$-Bäume: B-Bäume

Bemerkungen:

1. zur Wahl von a:
   - Daten in RAM $\Rightarrow$ a klein, z.B. $a = 2$ oder $a = 3$.
   - Daten auf Platte $\Rightarrow$ a groß, z.B. $a = 100$.
2. Zur Wahl von b:
   $b \geq 2a$ liefert wesentlich bessere amortisierte Komplexität (ohne Beweis, siehe Mehlhorn).

Top-Down-Rebalancierung: $b \geq 2a$.

Bei der Restrukturierung nach der Top-Down-Strategie folgen wir wie gewohnt dem Pfad von der Wurzel zum gesuchten Blatt. Beim Einfügen stellen wir jetzt jedoch für jeden besuchten Knoten sicher, dass der Knoten weniger als b Kinder hat.
Wenn der gerade betrachtete Knoten ein b-Knoten ist, dann spalten wir ihn sofort auf. Weil der Vater kein b-Knoten ist (das haben wir ja bereits sichergestellt), pflanzt sich der Aufspaltungsprozess nicht nach oben hin fort. Insbesondere kann das neue Element ohne Probleme eingefügt werden, wenn die Suche das Blattniveau erreicht hat.
Damit diese Strategie möglich ist, muss b mindestens $2a$ sein (und nicht nur $2a - 1$), da sonst nach der Spaltung nicht genügend Elemente für die beiden Teile vorhanden sind.

Beim Löschen verfahren wir analog. Für jeden besuchten Knoten, außer der Wurzel des (a, b)-Baumes, stellen wir sicher, dass er mindestens $a + 1$ Kinder hat. Wenn der gerade betrachtete Knoten nur a Kinder hat, so versuchen wir zuerst, ein Element des rechten oder linken Nachbarknoten zu stehlen.
Haben beide Nachbarknoten nur jeweils a Kinder, so verschmelzen wir unseren Knoten mit einem der beiden Nachbarn. Ist der Vater nicht die Wurzel, so hatte er aber vorher mindestens $a + 1$ Kinder, und dieser Verschmelzungsprozess kann sich nicht nach oben fortsetzen. Andernfalls erniedrigt sich der Grad der Wurzel um 1, wenn die Wurzel Grad > 2 hat, oder die alte Wurzel wird gelöscht und der soeben verschmolzene Knoten wird zur neuen Wurzel.

Restrukturierung nach der Top-Down-Strategie sorgt nicht für eine bessere Laufzeit, sondern erleichtert die Synchronisation, wenn mehrere Prozesse gleichzeitig auf einen (a, b)-Baum zugreifen wollen.
Bei herkömmlichen (a, b)-Bäumen schreiten die Such- und die Restrukturierungsoperationen in entgegengesetzter Richtung fort, was dazu führt, dass sich oft Prozesse gegenseitig behindern (der eine will im Baum absteigen, der andere muss auf dem gleichen Pfad aufwärts restrukturieren).
Bei der Top-Down-Strategie gibt es nur Operationsfolgen, die den Baum hinabsteigen. Mehrere Prozesse können so in einer Art Pipeline gleichzeitig einen Pfad mit kurzem Abstand begehen.

2.2 Rot-Schwarz-Bäume

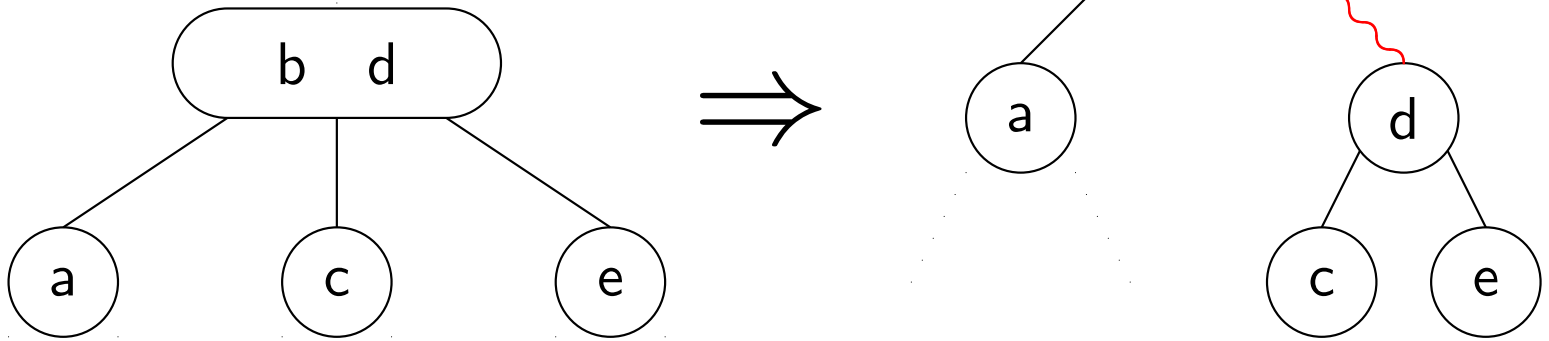
Definition 15

Rot-Schwarz-Bäume sind externe Binärbäume (jeder Knoten hat 0 oder 2 Kinder) mit roten und schwarzen Kanten, so dass gilt:

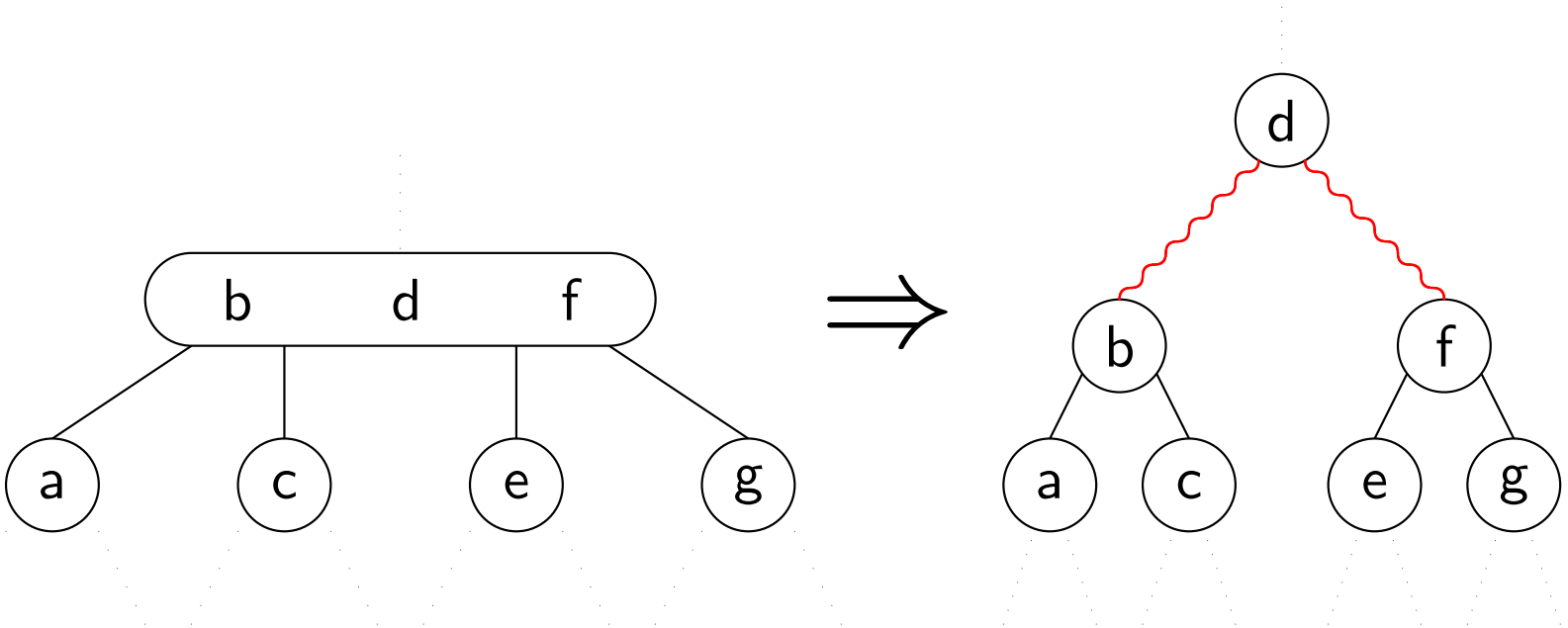
1. alle Blätter hängen an schwarzen Kanten (durchgezogene Linien)
2. alle Blätter haben die gleiche „Schwarztiefe“
3. kein Pfad von der Wurzel zu einem Blatt enthält (zwei oder mehr) aufeinanderfolgende rote Kanten (gewellte Linien).

Dabei ist die „Schwarztiefe“ eines Knoten die Anzahl der schwarzen Kanten auf dem Pfad von der Wurzel zu diesem Knoten.

Rot-Schwarz-Bäume können zur Implementierung von $(2,3)$- oder $(2,4)$-Bäumen dienen, mit dem Vorteil, dass alle internen Knoten Verzweigungsgrad $= 2$ haben!



Implementierung eines 4-Knotens:



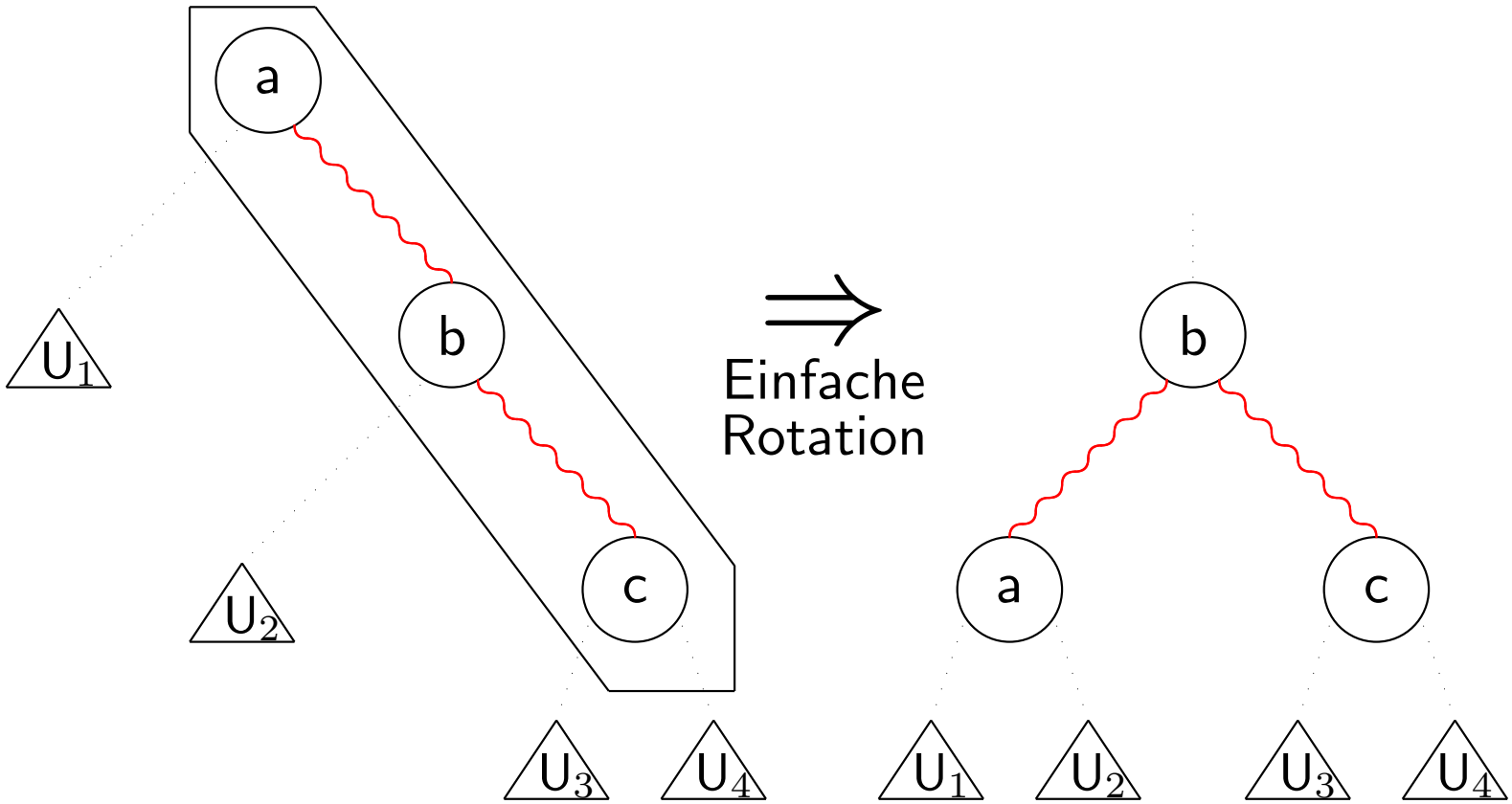
Operationen auf Rot-Schwarz implementierten (a, b)-Bäumen:

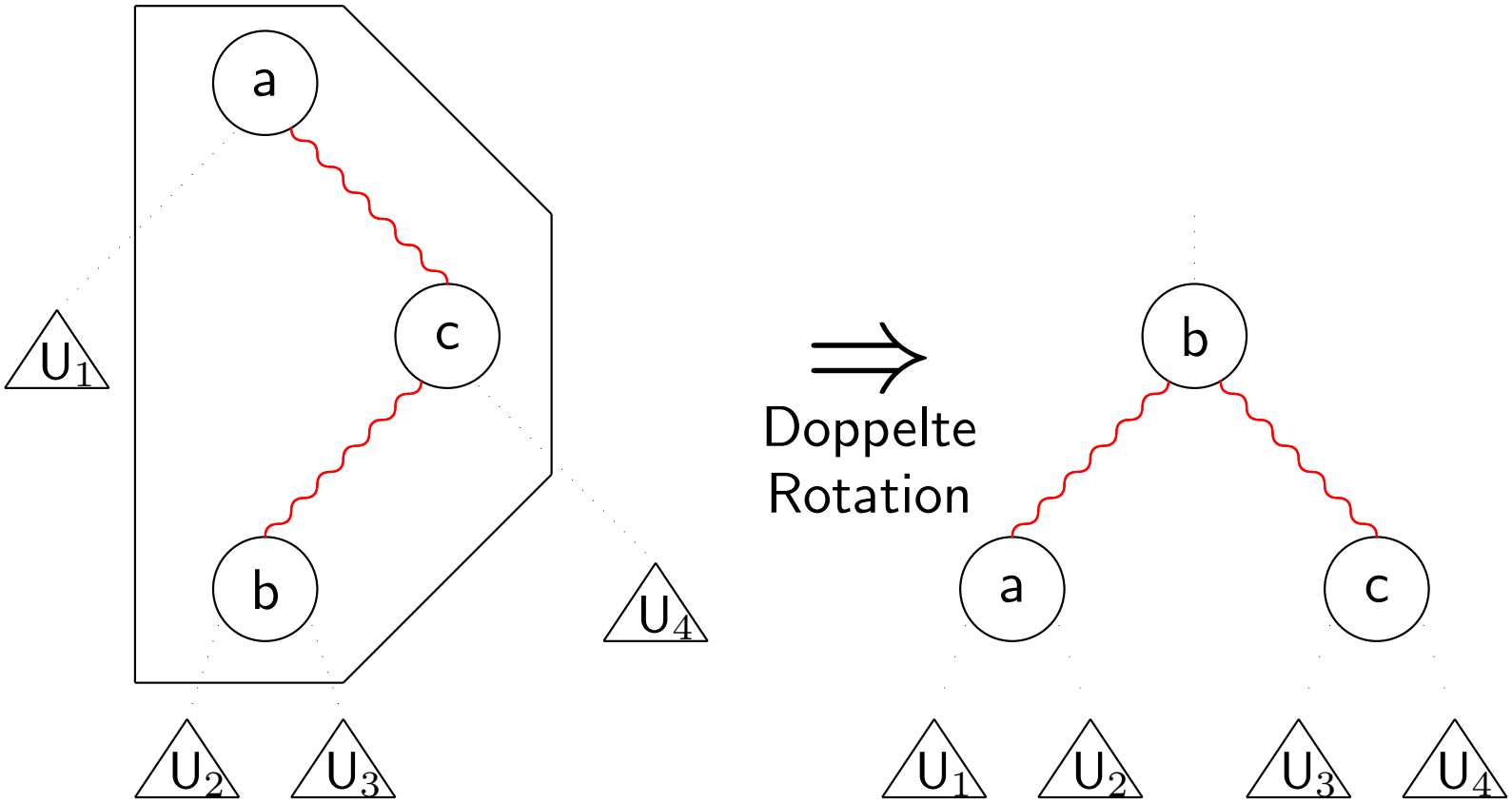
1. *IsElement*(k, T): vgl. (a, b)-Bäume

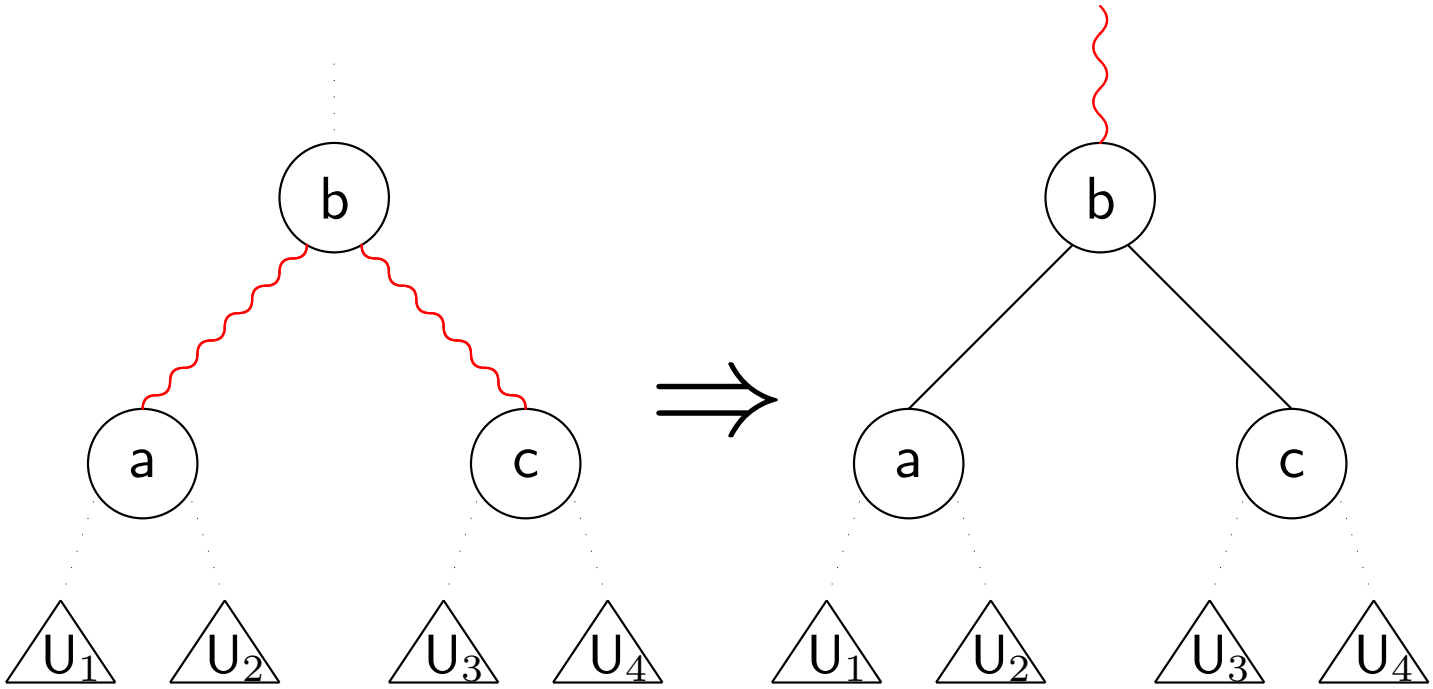
   $$\boxed{\text{Zeit } \mathcal{O}(\log n)}$$

2. *Insert*(k, T): Führe *IsElement*(k, T) aus $\rightsquigarrow$ Blatt w. Dort sei o.B.d.A. nicht das Element v gespeichert. Ersetze w durch neuen internen Knoten w' mit Kindern v, w, wobei v (bzw. w) ein neues Blatt mit Schlüssel k ist. w' erhält eine rote Eingangskante.

   Falls nun zwei aufeinanderfolgende rote Kanten an w' vorliegen, führe Rotationen zur Rebalancierung durch.


a
b
c
U_1
U_2
U_3
U_4
Einfache
Rotation
b
a
c
U_1
U_2
U_3
U_4

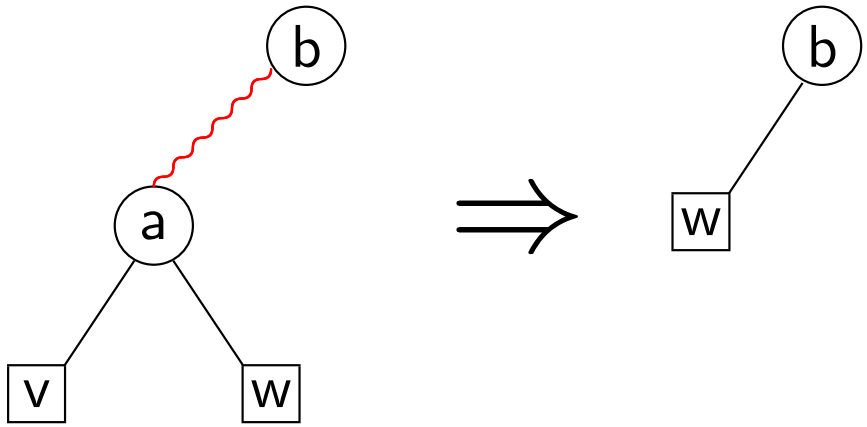

a
c
b
U_1
U_2
U_3
U_4
Doppelte
Rotation
b
a
c
U_1
U_2
U_3
U_4


b
a
c
U1
U2
U3
U4
b
a
c
U1
U2
U3
U4

3. *Delete*(k, T): Das Löschen des Blattes v macht es erforderlich, dass der Vater des Blattes durch den Bruder des Blattes ersetzt wird, damit der Binärbaumstruktur erhalten bleibt. Der Bruder von v ist eindeutig bestimmt.

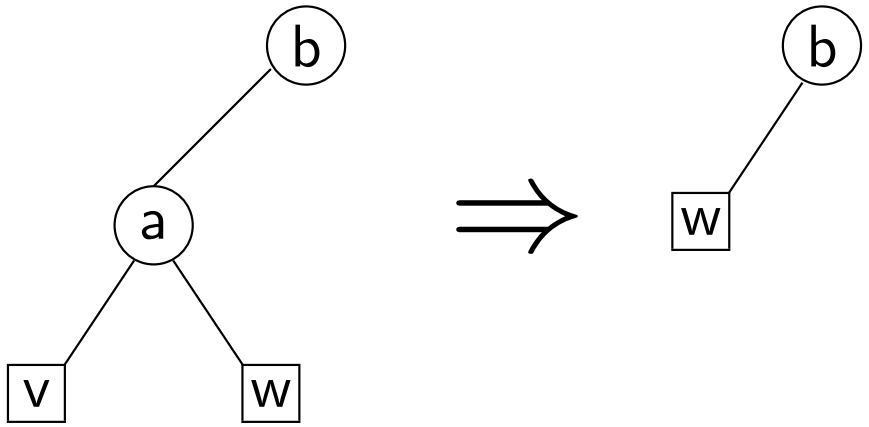
   Dabei treten zwei Fälle auf:

   1. Fall



   Dadurch ändert sich die Schwarztiefe von w nicht, denn die Eingangskante zu w wird (notwendig) zu schwarz umgefärbt. Auch kann es hierdurch nicht zu zwei aufeinanderfolgenden roten Kanten kommen.

2. Fall



Hierdurch verkleinert sich die Schwarztiefe von w um 1. Wir nennen einen Knoten, dessen Schwarztiefe um 1 zu gering ist, „zu hoch“. Falls die Eingangskante eines zu hohen Knoten rot ist, wird sie nach schwarz umgefärbt. Wir unterscheiden nun die folgenden Fälle:

1. Vater von w schwarz, Bruder von w rot: Der Baum wird so rotiert und umgefärbt, dass der Vater rot und der Bruder schwarz ist.
2. Bruder von w schwarz: Die Eingangskante des Bruders wird rot gefärbt und die Prozedur rekursiv mit dem Vater fortgesetzt, der nun (zusammen mit allen Knoten seines Unterbaums) zu hoch ist (Achtung: Falls Vater zuvor rot, Rebalancierung, da dann zwei aufeinander folgende rote Kanten).

Bemerkung: Falls der Bruder von w eine rote Ausgangskante hat, ist im zweiten Fall eine nicht-rekursive Vereinfachung möglich, indem die obersten Ebenen des Unterbaums, dessen Wurzel der Vater von w ist, geeignet umgeordnet werden.

Satz 16

Die Tiefe eines Rot-Schwarz-Baumes mit n Blättern ist $\mathcal{O}(\log n)$.

Beweis:

Hausaufgabe! □

Korollar 17

Die Wörterbuch-Operationen auf Rot-Schwarz-Bäumen benötigen Zeit $\mathcal{O}(\log n)$.

Beweis:

Hausaufgabe! □

Rot-Schwarz-Bäume wurden von R. Bayer unter der Bezeichnung symmetric binary B-trees erfunden und von Guibas und Sedgewick benannt und weiter erforscht.

R. Bayer:
Symmetric binary B-trees: Data structure and maintenance algorithms,
Acta Inf. **1**, pp. 290–306, 1972

Leo J. Guibas, Robert Sedgewick:
A dichromatic framework for balanced trees,
Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science, pp. 8–21. IEEE Computer Society, 1978

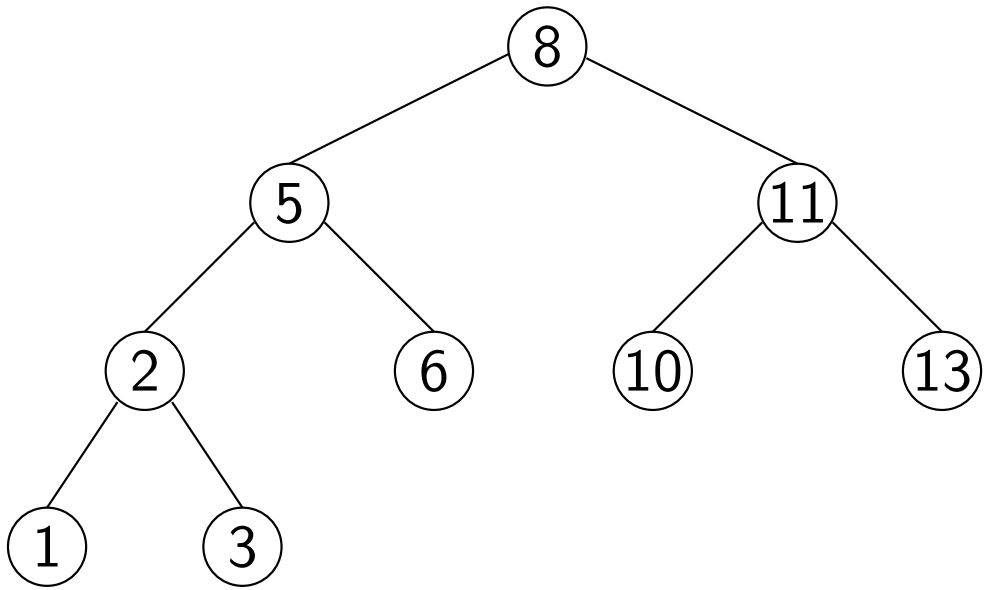
3. Binäre Suchbäume

3.1 Natürliche binäre Suchbäume

Definition 18
Ein natürlicher binärer Suchbaum über einem durch $\leq$ total geordneten Universum U ist ein als interner Suchbaum organisierter Binärbaum (also: Schlüssel an den internen Knoten, Blätter entsprechen leeren Knoten und werden nur wenn nötig angegeben).

Sortierungsbedingung:
Für jeden Knoten v und alle Knoten u im linken und alle Knoten w im rechten Unterbaum von v gilt

$$k(u) < k(v) < k(w)\,.$$



Lemma 19

Die In-Order-Linearisierung eines binären Suchbaumes mit obigen Suchwegbedingungen ergibt die Folge der Schlüssel in (strikt) aufsteigender Folge.

Beweis:

Klar! □

Die Wörterbuch-Operationen:

- *IsElement*(k, T):

$v :=$ Wurzel von T;
while $v \neq$ Blatt **do**
  if $k(v) = k$ **then**
    return $v(v)$
  elif $k(v) > k$ **then**
    $v :=$linkes Kind von v
  else
    $v :=$rechtes Kind von v
  fi
od
return NIL

- *Insert*(k, T):
  Führe *IsElement*(k, T) aus;
  if k nicht in T **then**
  IsElement ergibt Blatt w (NIL-Pointer);
  Füge neuen internen Knoten v mit $k(v) = k$ an Stelle von w ein
  fi

- *Delete*(k, T):
  Führe *IsElement*(k, T) aus;
  if k in T enthalten **then**
  IsElement führt zu Knoten w mit $k = k(w)$
  Falls w keine internen Kinder hat: klar
  Falls w nur ein internes Kind hat: Ersetze w durch dieses
  Ansonsten ersetze w durch seinen In-Order-Vorgänger oder -Nachfolger
  fi

Problem bei natürlichen Suchbäumen:
Bei bestimmten Abfolgen der Wörterbuchoperationen entarten natürliche Suchbäume stark, z.B. bei Einfügen der Schlüssel in monoton aufsteigender bzw. absteigender Folge zu einer linearen Liste der Tiefe n.

Daraus ergibt sich für die Wörterbuch-Operationen *Insert*, *Delete*, *IsElement* eine *worst case*-Komplexität von

$$\Theta(n)\,.$$

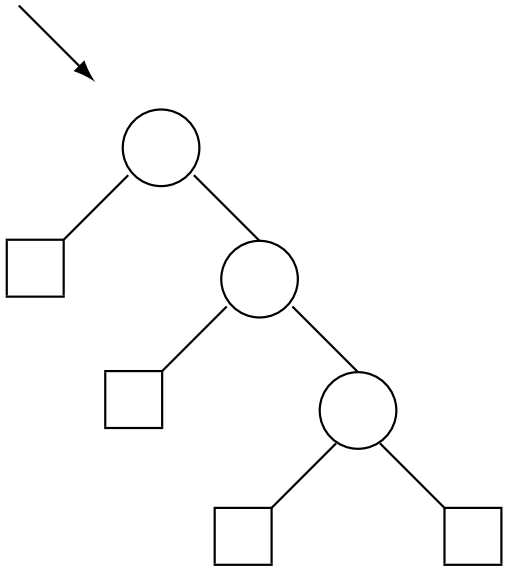
Falls alle Einfügefolgen gleichwahrscheinlich sind, gilt für die Höhe des Suchbaums

$$\mathbb{E}[h] = \mathcal{O}(\log n)\,.$$

Falls alle topologischen (von der Form her gleichaussehenden) Bäume der Größe n gleichwahrscheinlich sind, dann gilt

$$\mathbb{E}[h] = \Theta(\sqrt{n})\,.$$

Ohne Beweis!



3.2 Höhenbalancierte binäre Suchbäume (AVL-Bäume)

Definition 20

AVL-Bäume sind (interne) binäre Suchbäume, die die folgende Höhenbalancierung erfüllen:

Für jeden Knoten v gilt:

$$|\text{Höhe(linker UB}(v)) - \text{Höhe(rechter UB}(v))| \leq 1\,.$$

Bemerkung: AVL-Bäume sind nach ihren Erfindern G. Adelson-Velskii und Y. Landis (1962) benannt.

Satz 21

Ein AVL-Baum der Höhe h enthält mindestens $F_{h+2} - 1$ und höchstens $2^h - 1$ interne Knoten, wobei F_n die n-te Fibonacci-Zahl ($F_0 = 0, F_1 = 1$) und die Höhe die maximale Anzahl von Kanten auf einem Pfad von der Wurzel zu einem (leeren) Blatt ist.

Beweis:

Die obere Schranke ist klar, da ein Binärbaum der Höhe h höchstens

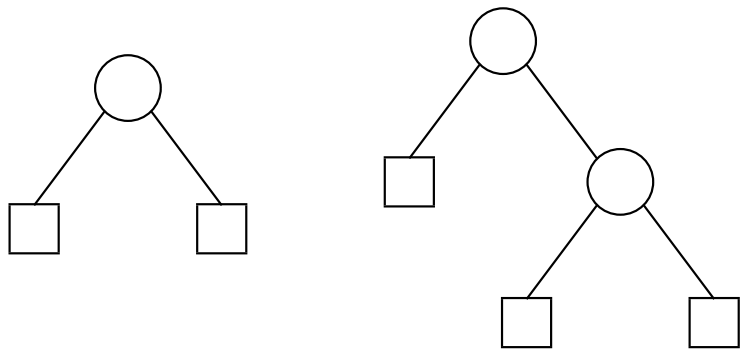
$$\sum_{j=0}^{h-1} 2^j = 2^h - 1$$

interne Knoten enthalten kann.

Beweis:

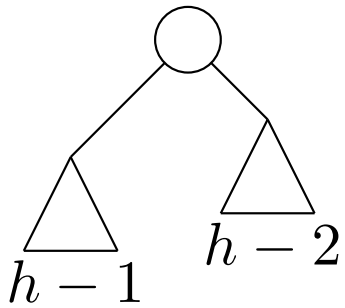
Induktionsanfang:

1. ein AVL-Baum der Höhe $h = 1$ enthält mindestens einen internen Knoten, $1 \geq F_3 - 1 = 2 - 1 = 1$
2. ein AVL-Baum der Höhe $h = 2$ enthält mindestens zwei Knoten, $2 \geq F_4 - 1 = 3 - 1 = 2$



Beweis:

Induktionsschluss: Ein AVL-Baum der Höhe $h \geq 2$ mit minimaler Knotenzahl hat als Unterbäume der Wurzel einen AVL-Baum der Höhe $h-1$ und einen der Höhe $h-2$, jeweils mit minimaler Knotenzahl.



Beweis:

Induktionsschluss: Ein AVL-Baum der Höhe $h \geq 2$ mit minimaler Knotenzahl hat als Unterbäume der Wurzel einen AVL-Baum der Höhe $h-1$ und einen der Höhe $h-2$, jeweils mit minimaler Knotenzahl. Sei

$$f_h := 1 + \text{minimale Knotenanzahl eines AVL-Baums der Höhe } h\,.$$

Dann gilt demgemäß

$$\begin{aligned} f_1 &= 2 && = F_3 \\ f_2 &= 3 && = F_4 \\ f_h - 1 &= 1 + f_{h-1} - 1 + f_{h-2} - 1\,, && \text{also} \\ f_h &= f_{h-1} + f_{h-2} && = F_{h+2} \end{aligned}$$

□

Bemerkung:

Da

$$F(n) \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n ,$$

hat ein AVL-Baum mit n internen Knoten eine Höhe

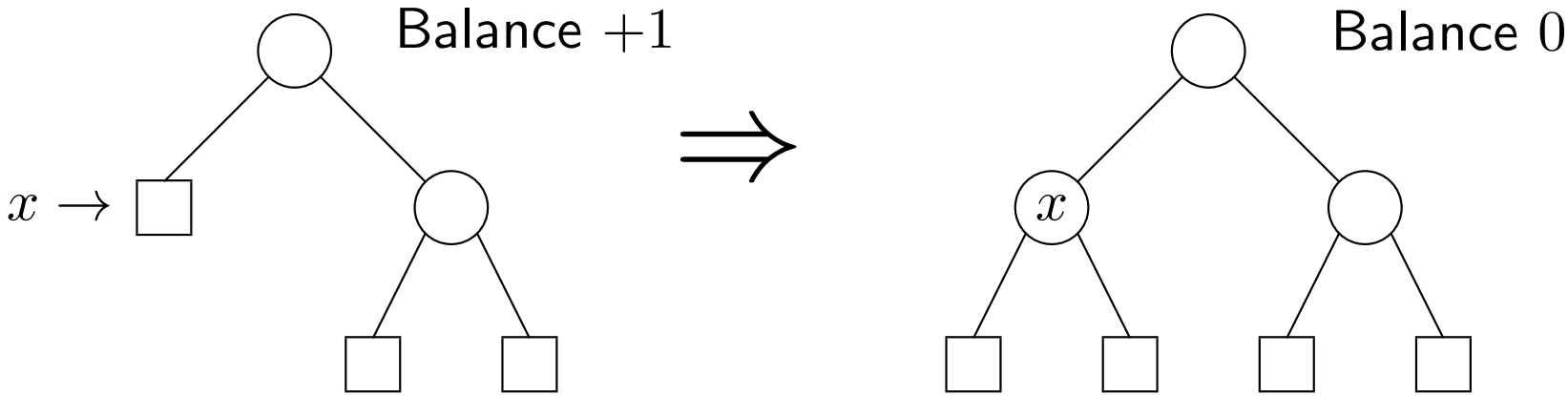
$$\Theta(\log n) .$$

Dies ist ein bedeutender Fortschritt gegenüber der bisweilen bei natürlichen Suchbäumen entstehenden, im *worst-case* linearen Entartung.

Die Operationen auf AVL-Bäumen:

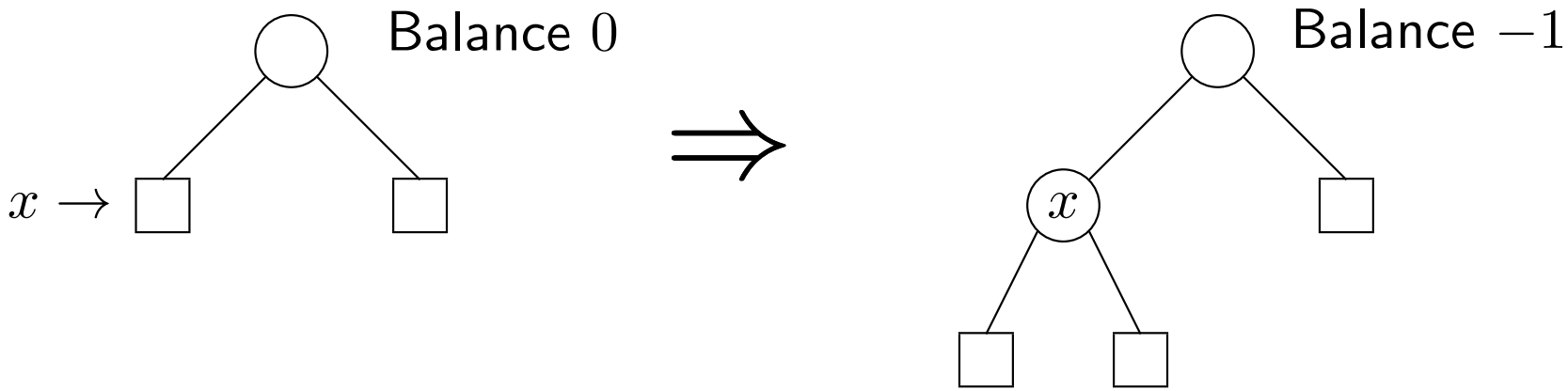
1. *IsElement*: Diese Operation wird wie bei den natürlichen Suchbäumen implementiert. Wie oben ausgeführt, haben aber AVL-Bäume mit n Schlüsseln eine Höhe von $\mathcal{O}(\log n)$, woraus logarithmische Zeit für das Suchen folgt.
2. *Insert*: Zunächst wird eine *IsElement*-Operation ausgeführt, die für den Fall, dass das einzufügende Element nicht bereits enthalten ist, zu einem Blatt führt. Dies ist die Stelle, an der das neue Element einzufügen ist. Dabei ergeben sich 2 Fälle:

1. Fall:



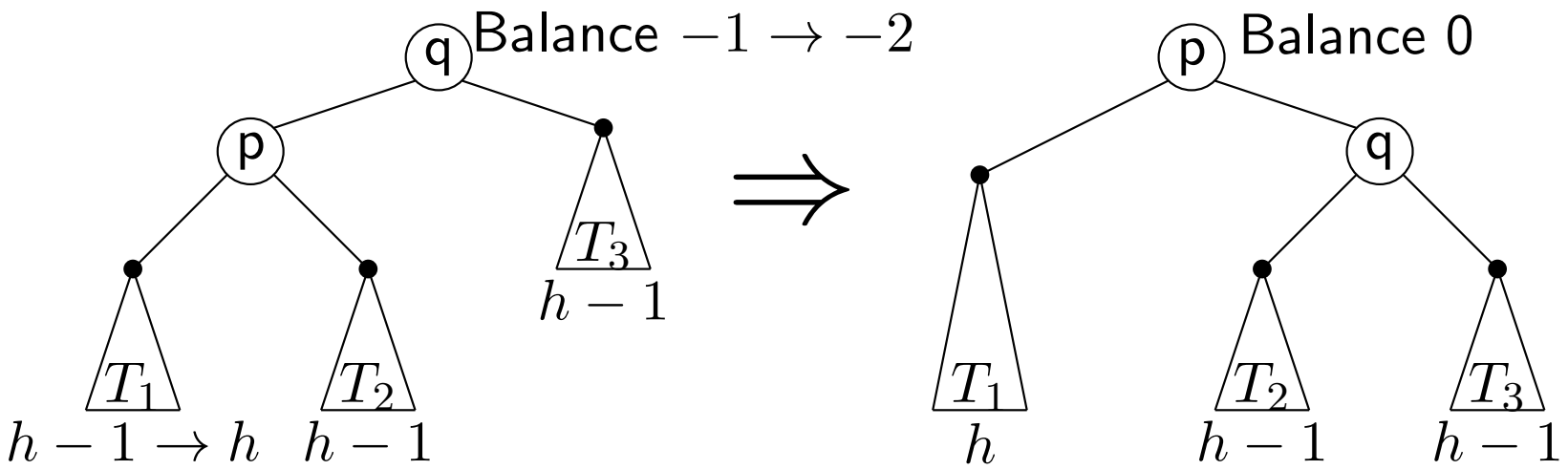
(Balance $-1 \rightarrow 0$: symmetrisch)

2. Fall:

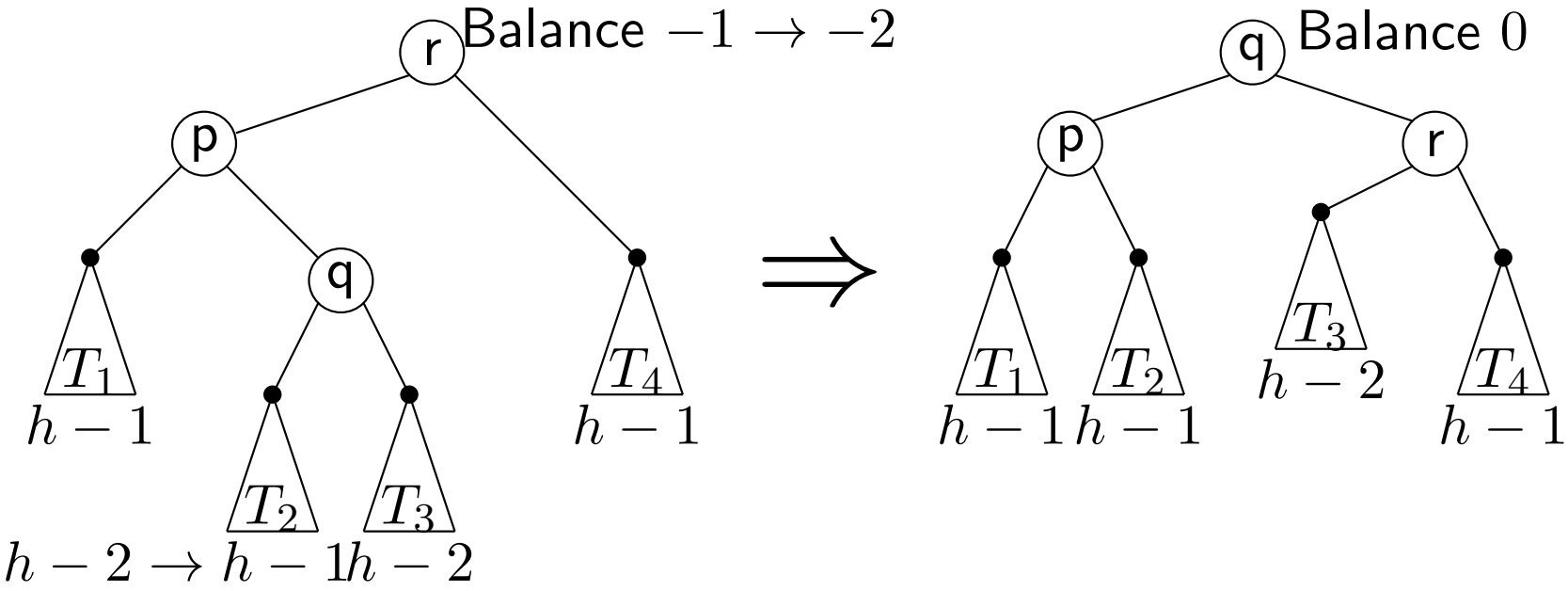


Hier ist eventuell eine Rebalancierung auf dem Pfad zur Wurzel notwendig, denn die Höhe des Unterbaums ändert sich.

Fall 2a:


q
Balance $-1 \to -2$
p
T_3
$h-1$
T_1
T_2
$h-1 \to h$
$h-1$
$\Rightarrow$
p
Balance 0
q
T_1
T_2
T_3
h
$h-1$
$h-1$

Fall 2b:


r
Balance $-1 \to -2$
p
q
T_1
$h-1$
T_2
T_3
$h-2 \to h-1 h-2$
T_4
$h-1$
$\Rightarrow$
q
Balance 0
p
r
T_1
$h-1$
T_2
$h-1$
T_3
$h-2$
T_4
$h-1$

3. *Delete*: Auch diese Operation wird wie bei natürlichen Suchbäumen implementiert. Jedoch hat am Ende ggf. noch eine Rebalancierung auf dem Pfad von dem Blatt, das an die Stelle des zu löschenden Elements geschrieben wird, zur Wurzel zu erfolgen.

Satz 22

Bei AVL-Bäumen sind die Operationen IsElement, Insert, und Delete so implementiert, dass sie die Zeitkomplexität $\mathcal{O}(\log n)$ haben, wobei n die Anzahl der Schlüssel ist.

Beweis:

Klar! □

Im Grundsatz gelten folgende Bemerkungen für AVL-Bäume:

1. Sie haben in der Theorie sehr schöne Eigenschaften, auch zur Laufzeit.
2. Sie sind in der Praxis sehr aufwändig zu implementieren.

Weitere Informationen:


Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman:
The design and analysis of computer algorithms,
Addison-Wesley Publishing Company: Reading (MA), 1974


Robert Sedgewick, Philippe Flajolet:
An introduction to the analysis of algorithms,
Addison-Wesley Publishing Company, 1996

3.3 Gewichtsbalancierte Bäume

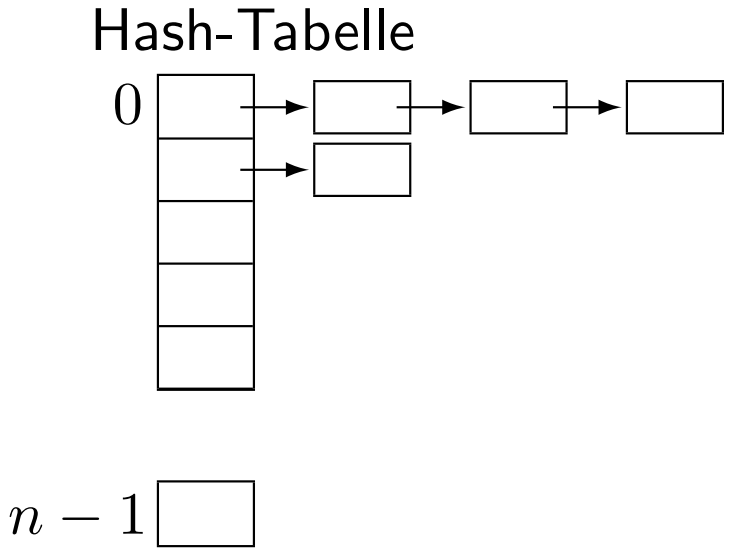
Siehe zu diesem Thema Seite 189ff in

Kurt Mehlhorn:
Data structures and algorithms 1: Sorting and searching,
EATCS Monographs on Theoretical Computer Science,
Springer Verlag: Berlin-Heidelberg-New York-Tokyo, 1984

4. Hashing

4.1 Grundlagen

- Universum U von Schlüsseln, z.B. $\subseteq \mathbb{N}_0$, $|U|$ groß
- Schlüsselmenge $S \subseteq U$, $|S| = m \leq n$
- array $T[0..n-1]$ Hashtabelle
- Hashfunktion $h : U \to [0..n-1]$



Problemstellung: Gegeben sei eine Menge M von Elementen, von denen jedes durch einen Schlüssel k aus der Menge U bezeichnet sei. Die Problemstellung ist: Wie muss die Speicherung der Elemente aus M bzw. der zugehörigen Schlüssel organisiert werden, damit jedes Element anhand seines Schlüssels möglich schnell lokalisiert werden kann?
Gesucht ist also eine Abbildung

$$h : U \to T$$

von der Menge aller Schlüssel in den Adressraum T der Maschine. Hierbei soll jedoch eine bisher nicht beachtete Schwierigkeit berücksichtigt werden: Die Menge U der möglichen Schlüsselwerte ist wesentlich größer als der Adressraum. Folglich kann die Abbildung h nicht injektiv sein, es gibt Schlüssel $k_1, k_2, \ldots$ mit $h(k_1) = h(k_2) = \ldots$.

Wir werden sehen, dass aufgrund dieses Umstandes die Speicherstelle eines Elements mit Schlüssel k von einem anderen Element mit einem anderen Schlüssel l besetzt sein kann und mit einer gewissen Wahrscheinlichkeit auch sein wird: Es treten Kollisionen auf.

Satz 23

In einer Hashtabelle der Größe n mit m Objekten tritt mit Wahrscheinlichkeit

$$\geq 1 - e^{\frac{-m(m-1)}{2n}} \approx 1 - e^{\frac{-m^2}{2n}}$$

mindestens eine Kollision auf, wenn für jeden Schlüssel jede Hashposition gleich wahrscheinlich ist.

Beweis:

Sei A_m das Ereignis, dass unter m Schlüsseln keine Kollision auftritt. Dann gilt

$$\Pr[A_m] = \prod_{j=0}^{m-1} \frac{n-j}{n} = \prod_{j=0}^{m-1} \left(1 - \frac{j}{n}\right)$$

$$\leq \prod_{j=0}^{m-1} e^{-\frac{j}{n}} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \quad .$$

Es folgt die Behauptung. □

Korollar 24

Hat eine Hashtabelle der Größe n mindestens $\omega(\sqrt{n})$ Einträge und ist für jeden Schlüssel jede Hashposition gleich wahrscheinlich, so tritt mit Wahrscheinlichkeit $1 - o(1)$ mindestens eine Kollision auf.

Um die Kollisionszahl möglichst gering zu halten, müssen Hashfunktionen gut streuen.

Definition 25

1. Eine Hashfunktion

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

heißt perfekt für $S \subseteq U$, wenn für alle $j, k \in S, j \neq k$ gilt

$$h(j) \neq h(k) \ .$$

2. Eine Klasse $\mathcal{H}$ von Hashfunktionen $h : U \to \{0, 1, \ldots, n-1\}$ heißt perfekt, falls $\mathcal{H}$ für jedes $S \subseteq U$ mit $|S| = n$ eine für S perfekte Hashfunktion enthält.

Grundsätzliche Fragestellungen:

1. Wie schwierig ist es, perfekte Hashfunktionen darzustellen (also: was ist ihre Programmgröße)?
2. Wie schwierig ist es, gegeben S, eine für S perfekte Hashfunktion zu finden?
3. Wie schwierig ist es, gegeben $k \in S$, $h(k)$ für eine für S perfekte Hashfunktion auszuwerten?

Typische „praktische“ Hashfunktionen:

$$h(k) = k \bmod n \qquad \text{(Teilermethode)}$$

$$h(k) = \lfloor n(ak - \lfloor ak \rfloor) \rfloor \text{ für } a < 1 \qquad \text{(Multiplikationsmethode)}$$

Wir betrachten zunächst Methoden der Kollisionsbehandlung.

4.2 Methoden zur Kollisionsauflösung

Wir unterscheiden grundsätzlich

- geschlossene und
- offene Hashverfahren.

Bei geschlossenen Hashverfahren werden Kollisionen nicht wirklich aufgelöst.

4.2.1 Geschlossene Hashverfahren (Chaining)

Die Hashtabelle ist ein Array von n linearen Listen, wobei die i-te Liste alle Schlüssel k beinhaltet, für die gilt:

$$h(k) = i\,.$$

Zugriff: Berechne $h(k)$ und durchsuche die Liste $T[h(k)]$.

Einfügen: Setze neues Element an den Anfang der Liste.

Sei

$$\delta_h(k_1, k_2) = \begin{cases} 1 \text{ falls } h(k_1) = h(k_2) \text{ und } k_1 \neq k_2 \\ 0 \text{ sonst} \end{cases}$$

und

$$\delta_h(k, S) = \sum_{j \in S} \delta_h(j, k), \text{ Anzahl Kollisionen von } k \text{ mit } S\,.$$

Die Zugriffskosten sind:

$$\boxed{\mathcal{O}(1 + \delta_h(k, S))}$$

Sei A eine Strategie zur Kollisionsauflösung. Wir bezeichnen im Folgenden mit

- A^+ den mittleren Zeitbedarf für eine erfolgreiche Suche unter Verwendung von A;
- A^- den mittleren Zeitbedarf für eine erfolglose Suche unter Verwendung von A;
- $\alpha := \frac{m}{n}$ den Füllfaktor der Hashtabelle.

Sondierungskomplexität für Chaining

Falls auf alle Elemente in der Hashtabelle mit gleicher Wahrscheinlichkeit zugegriffen wird, ergibt sich

- A^-: mittlere Länge der n Listen; da $\frac{m}{n} = \alpha$, folgt

$$A^- \leq 1 + \alpha\,.$$

- A^+:

$$\begin{aligned} A^+ &= \frac{1}{m}\sum_{i=1}^{m}\left(1 + \frac{i-1}{n}\right) \\ &= \frac{1}{m}\sum_{i=0}^{m-1}\left(1 + \frac{i}{n}\right) \\ &= 1 + \frac{m(m-1)}{2nm} \\ &\leq 1 + \frac{m}{2n} = 1 + \frac{\alpha}{2} \end{aligned}$$

Für festen Füllfaktor α ergibt sich also im Mittel Laufzeit $\Theta(1)$.

4.2.2 Hashing mit offener Adressierung

Beispiele:

- Lineares Sondieren (linear probing)
- Quadratisches Sondieren
- Double Hashing
- Robin-Hood-Hashing
- ...

Bei dieser Methode werden die Elemente nicht in der Liste, sondern direkt in der Hash-Tabelle gespeichert. Wird bei *Insert* oder *IsElement*, angewendet auf Schlüssel k, ein Element mit Schlüssel $\neq k$ an der Adresse $h(k)$ gefunden, so wird auf deterministische Weise eine alternative Adresse berechnet. Für jeden Schlüssel $k \in U$ wird somit eine Reihenfolge (Sondierungsfolge) von Positionen in $T[\,]$ betrachtet, um k zu speichern bzw. zu finden.

Sondieren:

Sei $s(j,k) : [0..n-1] \times U \to [0..n-1]$;
Definiere $h(j,k) = (h(k) - s(j,k)) \bmod n$; $\quad (0 \leq j \leq n-1)$
Starte mit $h(0,k)$, dann, falls $h(0,k)$ belegt, $h(1,k)$, ...

Grundsätzliches Problem:
Sei $h(k) = h(k')$ für zwei Schlüssel $k, k' \in S$. Werde zunächst k eingefügt, dann k', dann k gelöscht. Wie findet man k'?
(Beachte: k' steht nicht unmittelbar an $h(k')$.)

Lösungsvorschlag: Markiere k als gelöscht, entferne es aber nicht! Wenn Speicher gebraucht wird, k überschreiben.

Beispiele für Sondierungen

Lineares Sondieren:
Setze $s(j,k) = j$ d.h. sondiere gemäß
$h(k), h(k)-1, \ldots, 0, n-1, .., h(k)+1$.
Es wird für *IsElement* solange rückwärts gesucht, bis entweder das Element mit Schlüssel k oder eine freie Position gefunden ist. Im letzteren Fall ist das gesuchte Element nicht in der Hash-Tabelle enthalten.

Problem: Es entstehen primäre Häufungen (primary clustering) um diejenigen Schlüssel herum, die beim Einfügen eine Kollision hervorgerufen haben.

Satz 26

Die durchschnittliche Anzahl der Schritte beim linearen Sondieren ist

$$\mathbb{E}[\#\ \textit{Sondierungsschritte}] = \begin{cases} \frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right) & \textit{erfolgreich} \\ \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) & \textit{erfolglos} \end{cases}$$

Einige Werte:

α	erfolgreich	erfolglos
0.5	1.5	2.5
0.9	5.5	50.5
0.95	10.5	200.5

Beispiele für Sondierungen

Quadratisches Sondieren:
Setze $s(j,k) = (-1)^j \lceil \frac{j}{2} \rceil^2$, d.h. sondiere nach $h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \ldots$.

Frage: Ist das überhaupt eine Permutation von $[0..n-1]$? Ist $s(j,k)$ geeignet, alle Positionen zu erreichen?

Man kann zeigen, dass für Primzahlen n von der Form $4i+3$ die Sondierungsgröße $(h(k) - s(j,k)) \bmod n$ eine Permutation von $[0..n-1]$ liefert.

Problem: Sekundäres Clustering

Satz 27

Die durchschnittliche Anzahl der Schritte bei quadratischem Sondieren ist

$$\mathbb{E}[\#\ \textit{Sondierungsschritte}] = \begin{cases} 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2} & \textit{erfolgreich} \\ \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right) & \textit{erfolglos} \end{cases}$$

Einige Werte:

α	erfolgreich	erfolglos
0.5	1.44	2.19
0.9	2.85	11.4
0.95	3.52	22.05

Beispiele für Sondierungen

Double Hashing:
Setze $s(j,k) = jh'(k)$, wobei h' eine zweite Hashfunktion ist. $h'(k)$ muss relativ prim zu n gewählt werden, damit $(h(k) - s(j,k)) \bmod n$ eine Permutation der Hashadressen wird.

Satz 28

Die durchschnittliche Anzahl der Sondierungen bei Double Hashing ist

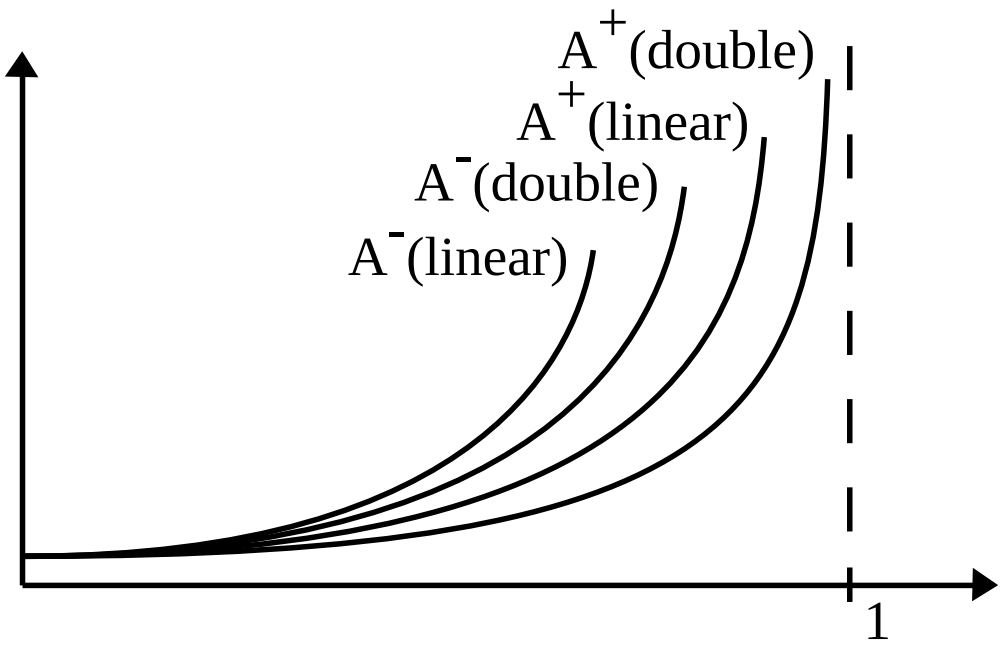
$$\mathbb{E}[\#\ \textit{Sondierungsschritte}] = \begin{cases} \frac{1}{\alpha} \ln \frac{1}{1-\alpha} & \textit{erfolgreich} \\ \frac{1}{1-\alpha} & \textit{erfolglos} \end{cases}$$

Einige Werte:

α	erfolgreich	erfolglos
0.5	1.39	2
0.9	2.55	10
0.95	3.15	20

Zum Beispiel: $h'(k) = 1 + k \bmod (n-2)$ (mit $n > 2$ prim).

Beispiele für Sondierungen



Sondierungskomplexität

4.3 Universelles Hashing

Definition 29

Eine Klasse $\mathcal{H}$ von Hashfunktionen von U nach $[0..n-1]$ heißt universell, falls für alle $x, y \in U$ mit $x \neq y$ gilt

$$\frac{|\{h \in \mathcal{H};\ h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{1}{n}\,.$$

Satz 30

Sei $\mathcal{H}$ eine universelle Klasse von Hashfunktionen für eine Hashtabelle der Größe n und sei $h \in \mathcal{H}$ zufällig gleichverteilt gewählt. Für eine Menge S von $m \leq n$ Schlüsseln ist dann die erwartete Anzahl von Kollisionen eines festen Schlüssels $x \in S$ mit anderen Elementen aus S kleiner als 1.

Beweis:

Sei x fest. Setze

$$C_x(y) =_{\text{def}} \begin{cases} 1 & \text{falls } h(x) = h(y)\,; \\ 0 & \text{sonst.} \end{cases}$$

Dann gilt

$$\begin{aligned} \mathbb{E}[C_x(y)] &= 0 \cdot \Pr[h(x) \neq h(y)] + 1 \cdot \Pr[h(x) = h(y)] \\ &= \Pr[h(x) = h(y)] \leq \frac{1}{n}\,. \end{aligned}$$

Für $C_x =_{\text{def}} \sum C_x(y)$ folgt damit

$$\mathbb{E}[C_x] = \sum_{y \in S \setminus \{x\}} \mathbb{E}[C_x(y)] \leq \frac{m-1}{n} < 1\,.$$

□

Sei $U = \{0, 1, \ldots, n-1\}^{r+1}$, für eine Primzahl n. Definiere

$$\mathcal{H} =_{\text{def}} \{h_\alpha;\ \alpha \in U\},$$

wobei

$$h_\alpha : U \ni (x_0, x_1, \ldots, x_r) \mapsto \sum_{i=0}^{r} \alpha_i x_i \bmod n \in \{0, 1, \ldots, n-1\}.$$

Lemma 31

$\mathcal{H}$ ist universell.

Beweis:

Seien $x, y \in U$ mit $x \neq y$. Wir nehmen o.B.d.A. an, dass $x_0 \neq y_0$. Ist $h_\alpha(x) = h_\alpha(y)$ für ein $\alpha \in U$, so gilt

$$\alpha_0(y_0 - x_0) = \sum_{i=1}^{r} \alpha_i(x_i - y_i) \bmod n\,.$$

Da n prim ist, ist $\mathbb{Z}_n$ ein Körper, und es gibt, bei vorgegebenen x, y und $\alpha_1, \ldots, \alpha_r$, genau ein α, so dass $h_\alpha(x) = h_\alpha(y)$.

Für festes x und y gibt es damit genau n^r Möglichkeiten, α zu wählen, so dass $h_\alpha(x) = h_\alpha(y)$.

Damit:

$$\frac{|\{h_\alpha \in \mathcal{H};\ h_\alpha(x) = h_\alpha(y)\}|}{|\mathcal{H}|} = \frac{n^r}{n^{r+1}} = \frac{1}{n}\,.$$

□

Wie groß müssen universelle Klassen von Hashfunktionen sein?

- Aus dem Beispiel:

$$|\mathcal{H}| = n^{r+1} = |U|\,.$$

- Es gibt Konstruktionen für Klassen der Größe $n^{\log(|U|)}$ bzw. $|U|^{\log n}$.

Satz 32

Sei $\mathcal{H}$ eine universelle Klasse von Hashfunktionen $h : U \to \{0, 1, \ldots, n-1\}$. Dann gilt

$$|\mathcal{H}| \geq n \left\lfloor \frac{\log(|U|) - 1}{\log n} \right\rfloor .$$

Beweis:

Sei $\mathcal{H} = \{h_1, h_2, \ldots, h_t\}$. Betrachte die Folge $U = U_0 \supseteq U_1 \supseteq U_2 \supseteq \cdots \supseteq U_t$, die definiert ist durch

$$U_i =_{\text{def}} U_{i-1} \cap h_i^{-1}(y_i)\,,$$

wobei $y_i \in \{0, 1, \ldots, n-1\}$ so gewählt ist, dass $|U_i|$ maximiert wird. Damit gilt

- h_j ist auf U_i konstant, für $j = 1, \ldots, i$,
- $|U_i| \geq \frac{|U_{i-1}|}{n}$, d.h. $|U_i| \geq \frac{|U|}{n^i}$.

Sei nun $\bar{t} = \left\lfloor \frac{\log(|U|)-1}{\log n} \right\rfloor$. Dann folgt

$$\log|U_{\bar{t}}| \geq \log|U| - \bar{t}\log n \geq \log|U| - \left(\frac{\log(|U|)-1}{\log n}\right) \cdot \log n = 1\,.$$

Beweis:

Sei $\mathcal{H} = \{h_1, h_2, \ldots, h_t\}$. Betrachte die Folge $U = U_0 \supseteq U_1 \supseteq U_2 \supseteq \cdots \supseteq U_t$, die definiert ist durch

$$U_i =_{\text{def}} U_{i-1} \cap h_i^{-1}(y_i)\,,$$

wobei $y_i \in \{0, 1, \ldots, n-1\}$ so gewählt ist, dass $|U_i|$ maximiert wird. Damit gilt

- h_j ist auf U_i konstant, für $j = 1, \ldots, i$,
- $|U_i| \geq \frac{|U_{i-1}|}{n}$, d.h. $|U_i| \geq \frac{|U|}{n^i}$.

Seien $x, y \in U_{\bar{t}}$, $x \neq y$. Dann ist

$$\bar{t} \leq |\{h \in \mathcal{H};\ h(x) = h(y)\}| \leq |\mathcal{H}|/n$$

und damit

$$|\mathcal{H}| \geq n\bar{t} = n \left\lfloor \frac{\log(|U|) - 1}{\log n} \right\rfloor .$$

□

4.4 Perfektes Hashing

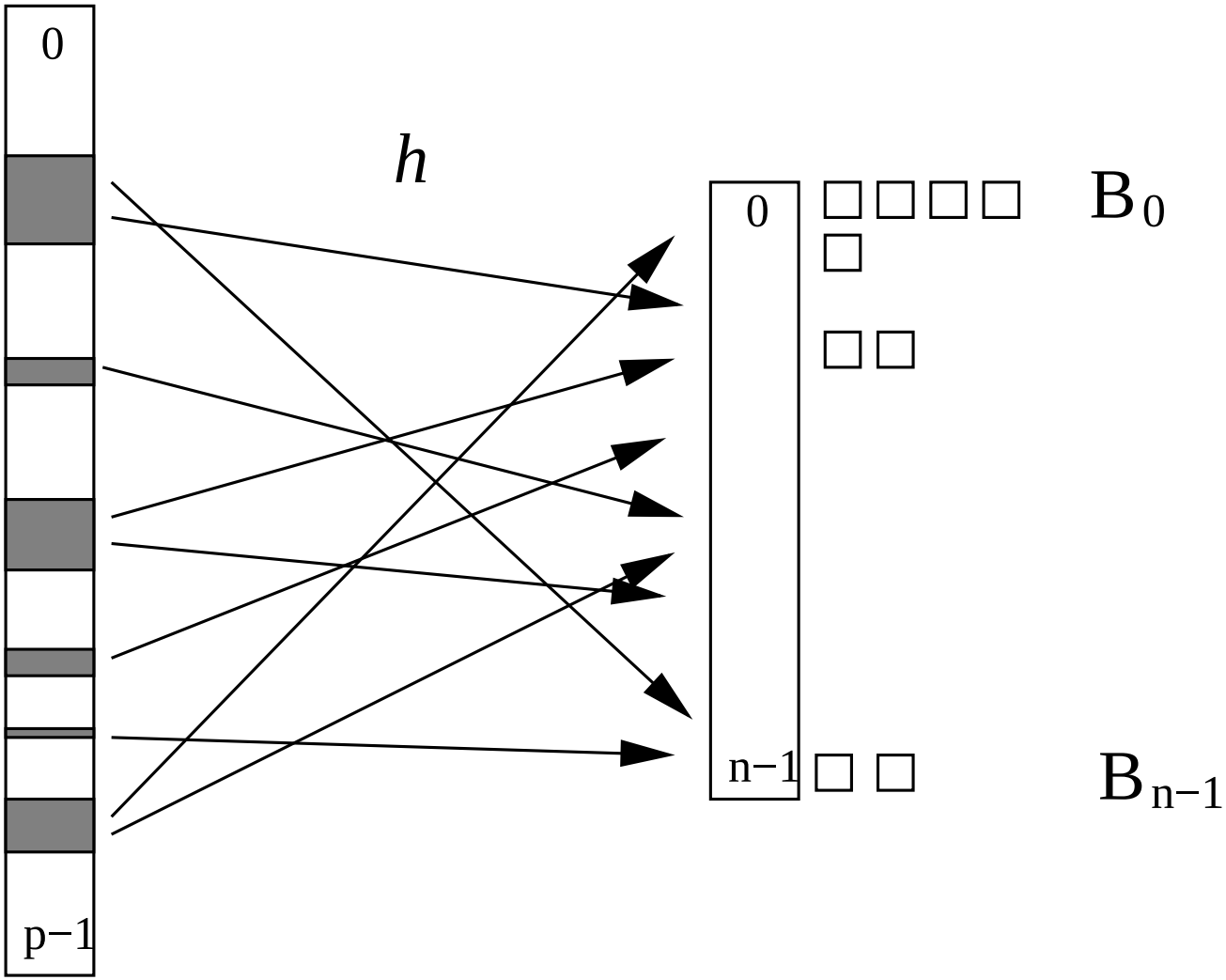
Das Ziel des perfekten Hashings ist es, für eine Schlüsselmenge eine Hashfunktion zu finden, so dass keine Kollisionen auftreten. Die Größe der Hashtabelle soll dabei natürlich möglichst klein sein.

4.4.1 Statisches perfektes Hashing

Sei $U = \{0, 1, \ldots, p-1\}$, p prim, das Universum, $n \in \mathbb{N}$ die Größe des Bildbereichs $\{0, 1, \ldots, n-1\}$ der Hashfunktionen und $S \subseteq U$, $|S| = m \leq n$, eine Menge von Schlüsseln.

Eine Hashfunktion $h : U \to \{0, 1, \ldots, n-1\}$ partitioniert S in „Buckets“

$$B_i = \{x \in S;\ h(x) = i\},\ \text{für } i = 0, 1, \ldots, n-1\,.$$



Hashfunktion h mit Buckets B_i

Definition 33

$\mathcal{H} = \mathcal{H}_{2,n}$ bezeichne die Klasse aller Funktionen

$$h_{a,b} : U \to \{0, 1, \ldots, n-1\}$$

mit

$$h_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod n \text{ für alle } x \in U\,,$$

wobei $0 < a < p$ und $0 \leq b < p$.

Lemma 34

$\mathcal{H}$ ist universell, d.h. für alle $x, y \in U$ mit $x \neq y$ gilt

$$\Pr[h(x) = h(y)] \leq \frac{1}{n}\,,$$

wenn h zufällig und gleichverteilt aus $\mathcal{H}$ gewählt wird.

Beweis:

Sei $h_{a,b}(x) = h_{a,b}(y) = i$. Dann ist

$$i = \underbrace{(ax + b) \bmod p}_{\alpha} = \underbrace{(ay + b) \bmod p}_{\beta} \pmod{n}$$

Sei $\alpha \in \{0, \ldots, p-1\}$ fest. Dann gibt es in der obigen Kongruenz $\lceil p/n \rceil - 1$ Möglichkeiten für β, nämlich

$$\beta \in \{i, i+n, i+2n, \ldots\} \setminus \{\alpha\},$$

da $\alpha \neq \beta$ und $x \neq y$ gilt.

Beweis:

Also gibt es höchstens

$$p \cdot \left(\left\lceil \frac{p}{n} \right\rceil - 1 \right) = p \cdot \left(\left(\left\lfloor \frac{p-1}{n} \right\rfloor + 1 \right) - 1 \right) \leq \frac{p(p-1)}{n}$$

Möglichkeiten für das Paar (α, β). Jedes Paar (α, β) bestimmt aber genau ein Paar (a, b), da $\mathbb{Z}_p$ ein Körper ist.

Weil es insgesamt $p(p-1)$ Paare (a, b) gibt und h uniform zufällig aus $\mathcal{H}$ ausgewählt wird, folgt

$$\Pr[h(x) = h(y)] \leq \frac{p(p-1)/n}{p(p-1)} = \frac{1}{n}$$

für jedes Paar $x, y \in U$ mit $x \neq y$. □

Lemma 35

Sei $S \subseteq U$, $|S| = m$. Dann gilt:

1. $$\mathbb{E}\left[\sum_{i=0}^{n-1} \binom{|B_i|}{2}\right] \leq \frac{m(m-1)}{2n}$$

2. $$\mathbb{E}[\sum_{i=0}^{n-1} |B_i|^2] \leq \frac{m(m-1)}{n} + m$$

3. $$\Pr[h_{a,b} \textit{ ist injektiv auf } S] \geq 1 - \frac{m(m-1)}{2n}$$

4. $$\Pr[\sum_{i=0}^{n-1} |B_i|^2 < 4m] > \frac{1}{2}, \textit{ falls } m \leq n$$

Beweis:

Definiere die Zufallsvariablen $X_{\{x,y\}}$ für alle $\{x,y\} \subseteq S$ gemäß

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

Wegen Lemma 34 gilt $\mathbb{E}[X_{\{x,y\}}] = \Pr[h(x) = h(y)] \leq 1/n$ für alle Paare $\{x,y\} \subseteq S$. Weiter ist

$$\mathbb{E}\left[\sum_{i=0}^{n-1} \binom{|B_i|}{2}\right] = |\{\{x,y\} \subseteq S;\ h(x) = h(y)\}|$$
$$\leq \binom{m}{2} \cdot \frac{1}{n}\,.$$

Beweis (Forts.):

Da $x^2 = 2 \cdot \binom{x}{2} + x$ für alle $x \in \mathbb{N}$, folgt

$$\mathbb{E}[\sum_{i=0}^{n-1} |B_i|^2] = \mathbb{E}\left[\sum_{i=0}^{n-1} \left(2 \cdot \binom{|B_i|}{2} + |B_i|\right)\right]$$
$$\stackrel{(1)}{\leq} 2 \cdot \frac{m(m-1)}{2n} + m\,.$$

Aus der Markov-Ungleichung ($\Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$ für alle $t > 0$) folgt

$$\Pr[h_{a,b} \text{ nicht injektiv auf } S] = \Pr\left[\sum_{i=0}^{n-1} \binom{|B_i|}{2} \geq 1\right]$$
$$\stackrel{(1)}{\leq} \frac{m(m-1)}{2n}\,.$$

Beweis (Forts.):

Für $m \leq n$ folgt aus (2), dass

$$\mathbb{E}[\sum_{i=0}^{n-1} |B_i|^2] \leq m + m = 2m\,.$$

Also folgt, wiederum mit Hilfe der Markov-Ungleichung, dass

$$\Pr\left[\sum_{i=0}^{n-1} |B_i|^2 > 4m\right] \leq \frac{1}{4m} \cdot 2m = \frac{1}{2}\,.$$

□

Die Struktur der perfekten Hashtabelle nach

Michael L. Fredman, János Komlós, Endre Szemerédi:
Storing a sparse table with $\mathcal{O}(1)$ worst case access time,
Journal of the ACM **31**(3), p. 538–544 (1984)

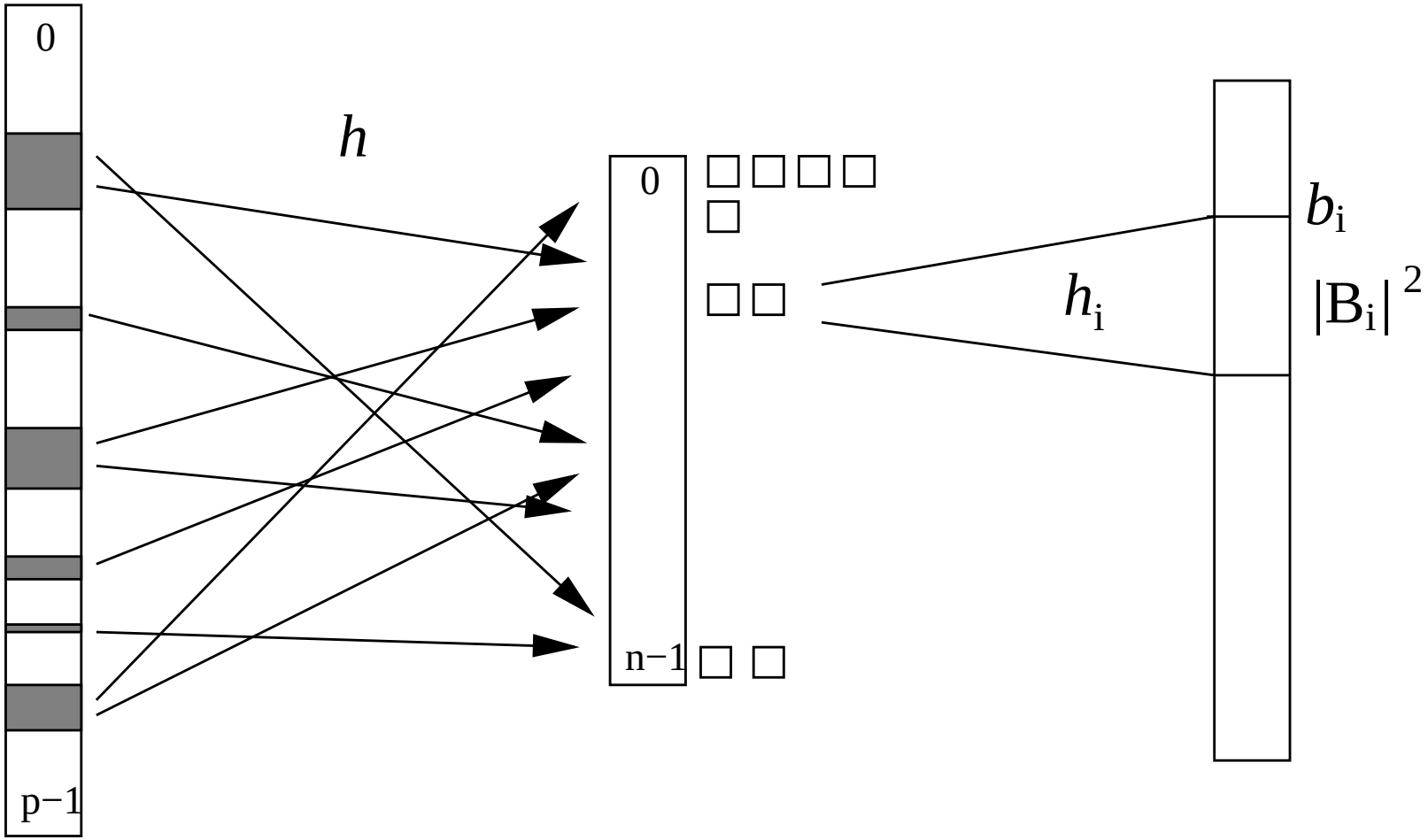
verwendet ein zweistufiges Hashverfahren.

Für einen gegebenen Schlüssel x wird zunächst $i = h(x)$ berechnet, um über den Tabellenplatz $T[i]$, b_i, $|B_i|$ und $h_i \in \mathcal{H}_{2,|B_i|^2}$ zu ermitteln. Dann wird im Tabellenplatz $T'[b_i + h_i(x)]$ nachgeschaut, ob x da abgespeichert ist. Falls ja, wird **true** ausgegeben und sonst **false**.

Falls

$$\sum_{i=0}^{n-1} |B_i|^2 < 4n$$

ist, so wird nur $\mathcal{O}(n)$ Platz verwendet.



Zweistufige Hashtabelle nach Fredman, Komlós und Szemerédi

Algorithmus für Hashtabelle nach FKS:

Eingabe: $S \subseteq U$, $|S| = m \leq n$
Ausgabe: Hashtabelle nach FKS

1. Wähle $h \in \mathcal{H}_{2,n}$ zufällig. Berechne $h(x)$ für alle $x \in S$.
2. Falls $\sum_i |B_i|^2 \geq 4m$, dann wiederhole 1.
3. Konstruiere die Mengen B_i für alle $0 \leq i < n$.
4. **for** $i = 0$ **to** $n - 1$ **do**
   (a) wähle $h_i \in \mathcal{H}_{2,|B_i|^2}$ zufällig
   (b) falls h_i auf B_i nicht injektiv ist, wiederhole (a)

Ein Durchlauf der Schleife bestehend aus den Schritten 1. und 2. benötigt Zeit $\mathcal{O}(n)$. Gemäß Lemma 35 ist die Wahrscheinlichkeit, dass Schritt 1. wiederholt werden muss, $\leq 1/2$ für jedes neue h.

Die Anzahl der Schleifendurchläufe ist also geometrisch verteilt mit Erfolgswahrscheinlichkeit $\geq 1/2$, und es ergibt sich

$$\mathbb{E}[\#\text{ Schleifendurchläufe}] \leq 2\,.$$

Also ist der Zeitaufwand für diese Schleife $\mathcal{O}(n)$. Schritt 3. kostet offensichtlich ebenfalls Zeit $\mathcal{O}(n)$.

Für jedes $i \in \{0, \ldots, n-1\}$ gilt, ebenfalls gemäß Lemma 35, dass

$$\Pr[h_i \text{ ist auf } B_i \text{ injektiv}] \geq 1 - \frac{|B_i|(|B_i|-1)}{2|B_i|^2} > \frac{1}{2}\,.$$

Damit ist auch hier die erwartete Anzahl der Schleifendurchläufe ≤ 2 und damit der erwartete Zeitaufwand

$$\mathcal{O}(|B_i|^2)\,.$$

Insgesamt ergibt sich damit für Schritt 4. wie auch für den gesamten Algorithmus ein Zeitaufwand von

$$\mathcal{O}(n)\,.$$

4.4.2 Dynamisches perfektes Hashing

Sei $U = \{0, \ldots, p-1\}$ für eine Primzahl p. Zunächst einige mathematische Grundlagen.

Definition 36

$\mathcal{H}_{k,n}$ bezeichne in diesem Abschnitt die Klasse aller Polynome $\in \mathbb{Z}_p[x]$ vom Grad $< k$, wobei mit $\vec{a} = (a_0, \ldots, a_{k-1}) \in U^k$

$$h_{\vec{a}}(x) = \left(\left(\sum_{j=0}^{k-1} a_j x^j \right) \bmod p \right) \bmod n \text{ für alle } x \in U\,.$$

Definition 37

Eine Klasse $\mathcal{H}$ von Hashfunktionen von U nach $\{0, \ldots, n-1\}$ heißt (c, k)-universell, falls für alle paarweise verschiedenen $x_0, x_1, \ldots, x_{k-1} \in U$ und für alle $i_0, i_1, \ldots, i_{k-1} \in \{0, \ldots, n-1\}$ gilt, dass

$$\Pr[h(x_0) = i_0 \wedge \cdots \wedge h(x_{k-1}) = i_{k-1}] \leq \frac{c}{n^k} ,$$

wenn $h \in \mathcal{H}$ gleichverteilt gewählt wird.

Satz 38

$\mathcal{H}_{k,n}$ ist (c,k)-universell mit $c = (1+\frac{n}{p})^k$.

Beweis:

Da $\mathbb{Z}_p$ ein Körper ist, gibt es für jedes Tupel $(y_0,\ldots,y_{k-1}) \in U^k$ genau ein Tupel $(a_0,\ldots,a_{k-1}) \in \mathbb{Z}_p^k$ mit

$$\sum_{j=0}^{k-1} a_j x_r^j = y_r \bmod p \qquad \text{für alle } 0 \le r < k.$$

Da es sich hier um eine Vandermonde-Matrix handelt, folgt, dass

$$\begin{aligned}
&|\{\vec{a};\ h_{\vec{a}}(x_r) = i_r \bmod n \text{ für alle } 0 \le r < k\}| \\
&= |\{(y_0,\ldots,y_{k-1}) \in U^k;\ y_r = i_r \bmod n \text{ für alle } 0 \le r < k\}| \\
&\le \left\lceil \frac{p}{n} \right\rceil^k .
\end{aligned}$$

Beweis (Forts.):

Da es insgesamt p^k Möglichkeiten für $\vec{a}$ gibt, folgt

$$\begin{aligned}\Pr[h(x_r) = i_r \text{ für alle } 0 \le r < k] &\le \left\lceil \frac{p}{n} \right\rceil^k \cdot \frac{1}{p^k} \\ &= \left(\left\lceil \frac{p}{n} \right\rceil \cdot \frac{n}{p} \right)^k \cdot \frac{1}{n^k} \\ &< \left(1 + \frac{n}{p} \right)^k \cdot \frac{1}{n^k} \,.\end{aligned}$$

□

Kuckuck-Hashing für dynamisches perfektes Hashing

Kuckuck-Hashing arbeitet mit zwei Hashtabellen, T_1 und T_2, die je aus den Positionen $\{0, \ldots, n-1\}$ bestehen. Weiterhin benötigt es zwei $(1+\delta, \mathcal{O}(\log n))$-universelle Hashfunktionen h_1 und h_2 für ein genügend kleines $\delta > 0$, die die Schlüsselmenge U auf $\{0, \ldots, n-1\}$ abbilden.

Jeder Schlüssel $x \in S$ wird entweder in Position $h_1(x)$ in T_1 oder in Position $h_2(x)$ in T_2 gespeichert, aber nicht in beiden. Die *IsElement*-Operation prüft einfach, ob x an einer der beiden Positionen gespeichert ist.

Die *Insert*-Operation verwendet nun das Kuckucksprinzip, um neue Schlüssel einzufügen. Gegeben ein einzufügender Schlüssel x, wird zunächst versucht, x in $T_1[h_1(x)]$ abzulegen. Ist das erfolgreich, sind wir fertig.

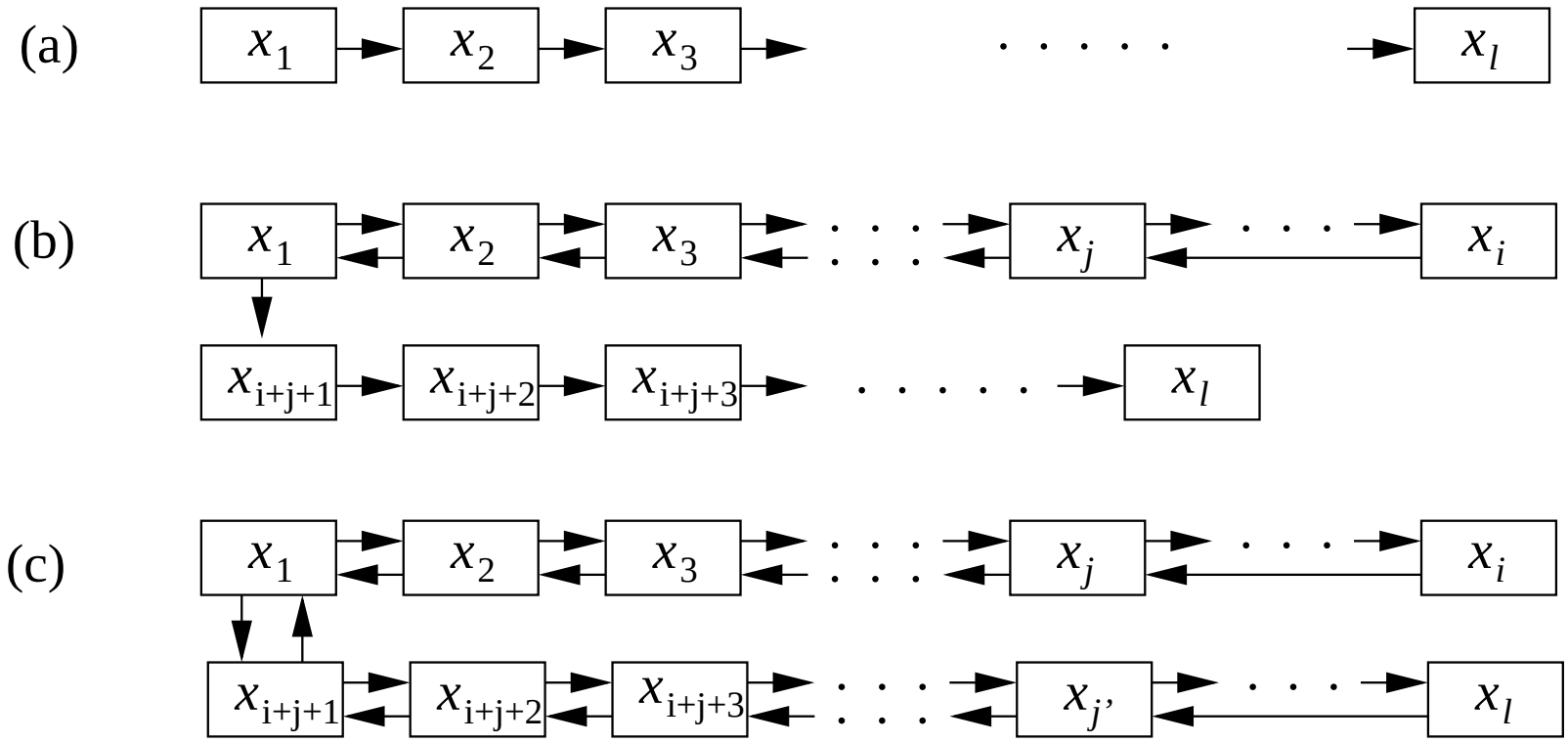
Falls aber $T_1[h_1(x)]$ bereits durch einen anderen Schlüssel y besetzt ist, nehmen wir y heraus und fügen stattdessen x in $T_1[h_1(x)]$ ein. Danach versuchen wir, y in $T_2[h_2(y)]$ unterzubringen. Gelingt das, sind wir wiederum fertig. Falls $T_2[h_2(y)]$ bereits durch einen anderen Schlüssel z besetzt ist, nehmen wir z heraus und fügen stattdessen y in $T_2[h_2(y)]$ ein. Danach versuchen wir, z in $T_1[h_1(z)]$ unterzubringen, und so weiter, bis wir endlich den zuletzt angefassten Schlüssel untergebracht haben. Formal arbeitet die Insert-Operation wie folgt:

if $T_1[h_1(x)] = x$ **then return fi**
repeat MaxLoop times
(a) exchange x und $T_1[h_1(x)]$
(b) **if** $x =$ NIL **then return fi**
(c) exchange x und $T_2[h_2(x)]$
(d) **if** $x =$ NIL **then return fi**
od
rehash(); *Insert*(x)

Für die Analyse der Zeitkomplexität nehmen wir an, dass die Schleife t-mal durchlaufen wird (wobei $t \leq$ MaxLoop).

Es gilt, die folgenden zwei Fälle zu betrachten:

1. Die *Insert*-Operation gerät während der ersten t Runden in eine Endlosschleife
2. Dies ist nicht der Fall



Insert bei Kuckuck-Hashing; Endlosschleife im Fall (c)

Erster Fall: Sei $v \leq l$ die Anzahl der verschiedenen angefassten Schlüssel. Dann ist die Anzahl der Möglichkeiten, eine Endlosscheife zu formen, höchstens

$$v^3 \cdot n^{v-1} \cdot m^{v-1} ,$$

da es maximal v^3 Möglichkeiten für die Werte i, j und l gibt, n^{v-1} viele Möglichkeiten für die Positionen der Schlüssel, und m^{v-1} viele Möglichkeiten für die Schlüssel außer x_1.

Angenommen, wir haben $(1, v)$-universelle Hashfunktionen h_1 und h_2, dann passiert jede Möglichkeit nur mit einer Wahrscheinlichkeit von n^{-2v}. Falls $n \geq (1+\delta)m$ für eine Konstante $\delta > 0$, dann ist die Wahrscheinlichkeit für den Fall 1 höchstens

$$\sum_{v=3}^{l} v^3 n^{v-1} m^{v-1} n^{-2v} \leq \frac{1}{nm} \sum_{v=3}^{\infty} v^3 (m/n)^v = \mathcal{O}(\frac{1}{m^2}) .$$

Zweiter Fall:

Lemma 39

Im zweiten Fall gibt es eine Schlüsselfolge der Länge mindestens $l/3$ in $x_1, \ldots, x_l$, in der alle Schlüssel paarweise verschieden sind.

Beweis:

Nehmen wir an, dass die Operation zu einer bereits besuchten Position zurückkehrt, und seien i und j so definiert wie in der Abbildung.

Es muss eine der Folgen $x_1, \ldots, x_i$ bzw. $x_{i+j+1}, \ldots, x_l$ die Länge mindestens $l/3$ haben.

Beweis (Forts.):

Sei also $x'_1, \ldots, x'_v$ eine solche Folge verschiedener Schlüssel in $x_1, \ldots, x_{2t}$ der Länge $v = \lceil (2t-1)/3 \rceil$. Dann muss entweder für $(i_1, i_2) = (1, 2)$ oder für $(i_1, i_2) = (2, 1)$ gelten, dass

$$h_{i_1}(x'_1) = h_{i_1}(x'_2), h_{i_2}(x'_2) = h_{i_2}(x'_3), h_{i_1}(x'_3) = h_{i_1}(x'_4), \ldots$$

Gegeben x'_1, so gibt es m^{v-1} mögliche Folgen von Schlüsseln $x'_2, \ldots, x'_v$. Für jede solche Folge gibt es zwei Möglichkeiten für (i_1, i_2). Weiterhin ist die Wahrscheinlichkeit, dass die obigen Positionsübereinstimmungen gelten, höchstens $n^{-(v-1)}$, wenn die Hashfunktionen aus einer $(1, v)$-universellen Familie stammen. Also ist die Wahrscheinlichkeit, dass es irgendeine Folge der Länge v gibt, so dass Fall 2 eintritt, höchstens

$$2(m/n)^{v-1} \leq 2(1+\delta)^{-(2t-1)/3+1}.$$

Diese Wahrscheinlichkeit ist polynomiell klein in m, falls $t = \Omega(\log m)$ ist. □

Beweis (Forts.):

Zusammen ergibt sich für die Laufzeit von *Insert*:

$$1 + \sum_{t=2}^{\text{MaxLoop}} (2(1+\delta)^{-(2t-1)/3+1} + \mathcal{O}(1/m^2))$$

$$\leq 1 + \mathcal{O}\left(\frac{\text{MaxLoop}}{m^2}\right) + \mathcal{O}\left(\sum_{t=0}^{\infty}((1+\delta)^{-2/3})^t\right)$$

$$= \mathcal{O}\left(1 + \frac{1}{1-(1+\delta)^{-2/3}}\right) = \mathcal{O}(1+1/\delta)\,.$$

□

Beweis (Forts.):

Überschreitet m irgendwann einmal die Schranke $n/(1+\delta)$, so wird n hochgesetzt auf $(1+\delta)n$ und neu gehasht. Unterschreitet auf der anderen Seite m die Schranke $n/(1+\delta)^3$, so wird n verringert auf $n/(1+\delta)$ und neu gehasht. Auf diese Weise wird die Tabellengröße linear zur Anzahl momentan existierender Schlüssel gehalten. Der Aufwand für ein komplettes Rehashing ist $\mathcal{O}(n)$, so dass amortisiert über $\Theta(n)$ Einfügungen und Löschungen der Aufwand nur eine Konstante ist. □

Originalarbeiten zu Hashverfahren:

J. Lawrence Carter, Mark N. Wegman:
Universal Classes of Hash Functions,
Proc. STOC 1977, pp. 106–112 (1977)

Gaston H. Gonnet:
Expected Length of the Longest Probe Sequence in Hash Code Searching,
Journal of the ACM **28**(2), pp. 289–304 (1981)

Martin Dietzfelbinger et al.:
Dynamic Perfect Hashing: Upper and Lower Bounds,
SIAM J. Comput. **23**(4), pp. 738–761 (1994)

Und weiter:

Rasmus Pagh, Flemming Friche Rodler:
Cuckoo Hashing,
Proc. ESA 2001, LNCS **2161**, pp. 121–133 (2001)

Luc Devroye, Pat Morin, Alfredo Viola:
On Worst Case Robin-Hood Hashing,
McGill Univ., TR **0212** (2002)

5. Vorrangwarteschlangen - Priority Queues

Priority Queues unterstützen die Operationen
Insert(), *Delete()*, *ExtractMin()*, *FindMin()*, *DecreaseKey()*, *Merge()*.

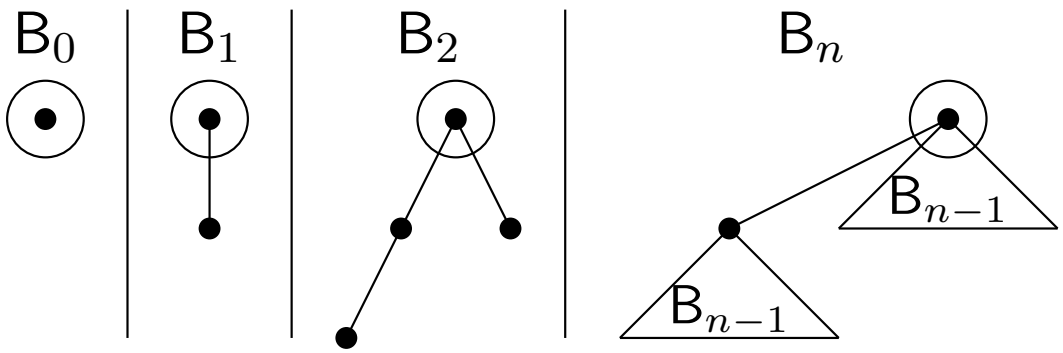
Priority Queues per se sind nicht für *IsElement()*-Anfragen, also zum Suchen geeignet. Falls benötigt, muss dafür eine passende Wörterbuch-Datenstruktur parallel mitgeführt werden.

5.1 Binomial Queues (binomial heaps)

Binomialwarteschlangen (Binomial Queues/Binomial Heaps) werden mit Hilfe von Binomialbäumen konstruiert.

Definition 40

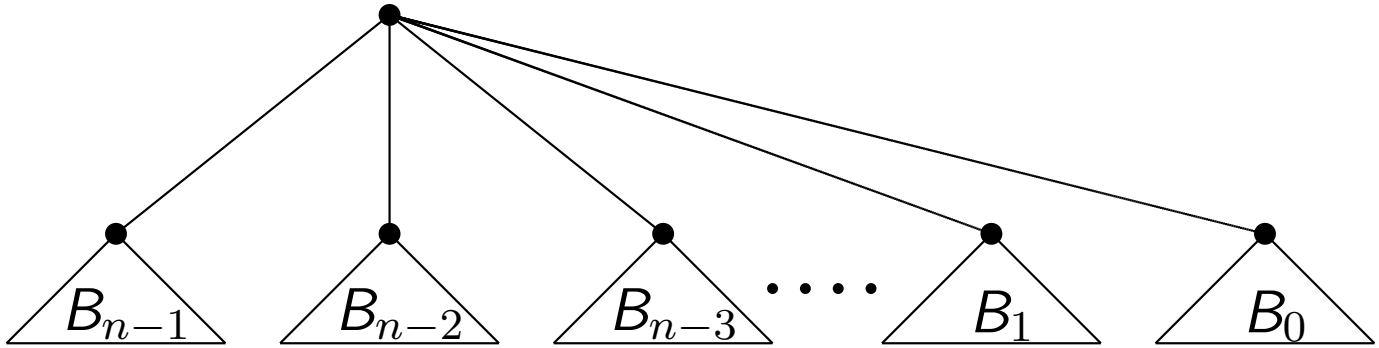
Die Binomialbäume B_n, $n \geq 0$, sind rekursiv wie folgt definiert:



Achtung: Binomialbäume sind offensichtlich keine Binärbäume!

Lemma 41

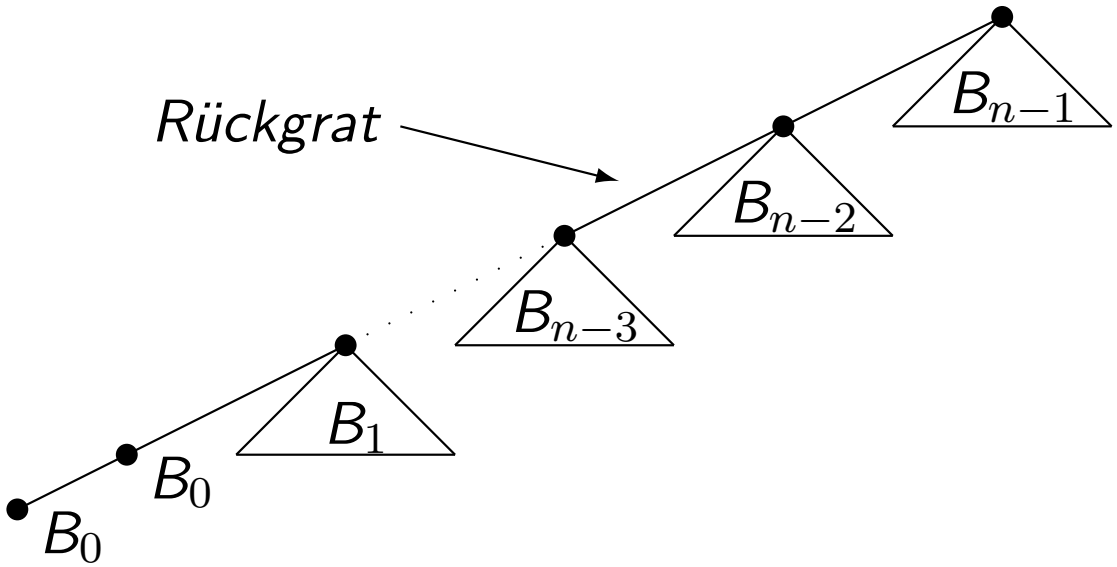
Ein B_n lässt sich wie folgt zerlegen:



Erste Zerlegung von B_n: der Breite nach

Lemma 41

Ein B_n lässt sich wie folgt zerlegen:



Zweite Zerlegung von B_n: der Tiefe nach

Satz 42

Für den Binomialbaum B_n gilt:

1. *B_n hat 2^n Knoten.*
2. *Die Wurzel von B_n hat Grad n.*
3. *B_n hat Höhe/Tiefe n.*
4. *B_n hat $\binom{n}{i}$ Knoten in Tiefe i.*

Beweis:

- zu 1: Induktion unter Verwendung der Definition
- zu 2: Siehe erste Zerlegung von Binomialbäumen
- zu 3: Siehe zweite Zerlegung von Binomialbäumen
- zu 4: Induktion über n:

  (I) $n = 0 : B_0$ hat 1 Knoten in Tiefe $i = 0$ und 0 Knoten in Tiefe $i > 0$, also $\binom{0}{i}$ Knoten in Tiefe i.

  (II) Ist N_i^n die Anzahl der Knoten in Tiefe i im B_n, so gilt unter der entsprechenden Induktionsannahme für B_n, dass

$$N_i^{n+1} = N_i^n + N_{i-1}^n = \binom{n}{i} + \binom{n}{i-1} = \binom{n+1}{i},$$

  woraus wiederum per Induktion die Behauptung folgt.

□

Bemerkung:

Eine mögliche Implementierung von Binomialbäumen ist das Schema „Pointer zum ersten Kind und Pointer zum nächsten Geschwister“.

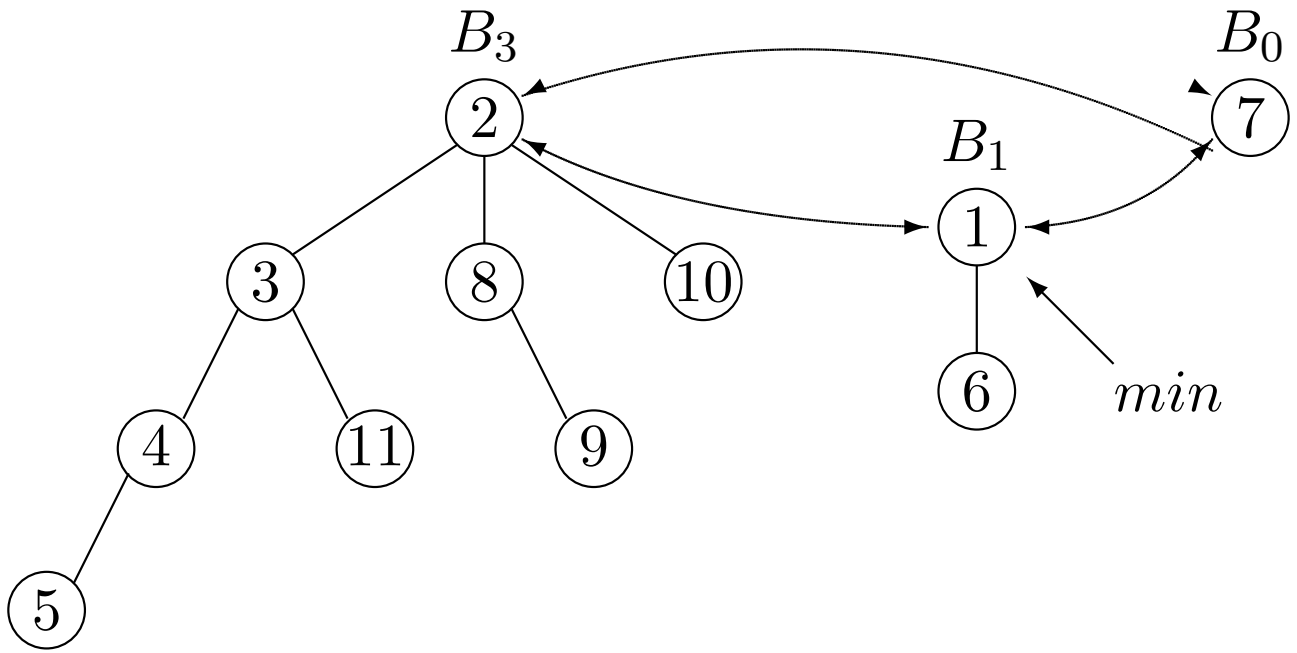
Definition 43

Eine Binomial Queue mit n Elementen wird wie folgt aufgebaut:

1. Betrachte die Binärdarstellung von n.
2. Für jede Position i mit einem 1-Bit wird ein Binomialbaum B_i benötigt (der 2^i Knoten hat).
3. Verbinde die (Wurzeln der) Binomialbäume in einer doppelt verketteten zirkulären Liste.
4. Beachte, dass innerhalb jedes Binomialbaums die Heap-Bedingung erfüllt sein muss. Dadurch enthält die Wurzel eines jeden Binomialbaums gleichzeitig sein minimales Element.
5. Richte einen Min-Pointer auf das Element in der Wurzel-Liste mit minimalem Schlüssel ein.

Beispiel 44

Beispiel: $n = 11 = (1011)_2$**:**



Operationen für Binomial Queues:

- *IsElement*: Die Heap-Bedingung wirkt sich nur auf die Anordnung der Datenelemente innerhalb jedes einzelnen Binomialbaums aus, regelt aber nicht, in welchem Binomialbaum ein gegebenes Element gespeichert ist. Tatsächlich kann ein Element in jedem der vorhandenen Binomialbäume stehen. Das Suchen ist hier nicht effizient implementierbar, denn im worst-case müsste jedes Element der Binomialbäume angeschaut werden.

  Also wären zum Suchen eines Elements schlimmstenfalls $2^0 + 2^1 + \cdots + 2^{\lfloor \log n \rfloor - 1} = \Theta(n)$ Elemente zu betrachten.

  Daher wird eine gesonderte Datenstruktur, etwa ein Suchbaum, für die *IsElement*-Operation verwendet. Damit ist Suchen mit Zeitaufwand $\mathcal{O}(\log n)$ möglich.

Operationen für Binomial Queues:

- *Merge*: Das Vorgehen für das Merge (disjunkte Vereinigung) zweier Binomial Queues entspricht genau der Addition zweier Binärzahlen: Ein einzelnes B_i wird übernommen, aus zwei B_i's wird ein B_{i+1} konstruiert. Damit die Heap-Bedingung erhalten bleibt, wird als Wurzel des entstehenden B_{i+1} die Wurzel der beiden B_i mit dem kleineren Schlüssel genommen.

  Allerdings kann ein solcher „Übertrag“ dazu führen, dass im nächsten Verschmelzungsschritt drei B_{i+1} zu verschmelzen sind. Dann wird unter Beachtung obiger Regel ein B_{i+2} gebildet und einer der B_{i+1} unverändert übernommen.

Algorithmus:

for $i := 0, 1, 2, 3, \ldots$ **do**
  if ($\exists$ genau 3 B_is) **then**
    verbinde zwei der B_i's zu einem B_{i+1} und behalte das dritte B_i
  elif ($\exists$ genau 2 B_is) **then**
    verbinde sie zu einem B_{i+1}
  elif ($\exists$ genau ein B_i) **then**
    übernimm es
  fi
od

Zeitkomplexität:

$$\boxed{\mathcal{O}(\log n) = \mathcal{O}(\log(n_1 + n_2))}$$

Operationen für Binomial Queues:

- *Insert*: Die *Insert*-Operation wird einfach durch eine *Merge*-Operation mit einem B_0 implementiert.

Beispiel 45

1	0	0	1	0	1	1	1
(B_7)			(B_4)		(B_2)	(B_1)	(B_0)
							1
1	0	0	1	1	0	0	0

Zeitkomplexität: $\mathcal{O}(\log n)$

Operationen für Binomial Queues:

- Initialisierung einer BQ durch n sukzessive *Insert*-Operationen: Hier ergäbe die obige Abschätzung einen Gesamtaufwand von $\mathcal{O}(n \log n)$, was allerdings schlecht abgeschätzt ist.

Wir sind an den Kosten zum sukzessiven Aufbau einer Binomial Queue mit n Elementen interessiert, die übrigens identisch sind zum Aufwand für das binäre Zählen von 0 bis n, wenn jeder Zählschritt und jeder Übertrag jeweils eine Zeiteinheit kosten:

$$\underbrace{\underbrace{0+1}_{1}+1}_{2\text{ Schritte}}+1+1+1$$

$$\underbrace{\qquad\qquad}_{1}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{3}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{1}$$

Sei a_n die Anzahl der Schritte (Einfügen des neuen Elements und anschließende Verschmelzungen) beim Mergen eines Knotens zu einer Queue mit $n-1$ Elementen. Dann gilt für a_n:

n	a_n
1	1
2	2
... 4	1 3
... 8	1 2 1 4
... 16	1 2 1 3 1 2 1 5
... 32	1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 6

Man erkennt sofort, dass jede Zeile (außer den beiden ersten) doppelt so lange ist wie die vorhergehende und dabei die Folge aller vorhergehenden Zeilen enthält, wobei das letzte Element noch um eins erhöht ist.

Damit ergibt sich für den Gesamtaufwand der sukzessiven Erzeugung einer Binomial Queue mit n Elementen

$$\begin{aligned} T_n &= \sum_{i=1}^{n} a_i \\ &\leq n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \left\lfloor \frac{n}{8} \right\rfloor + \cdots \\ &\leq 2n\,. \end{aligned}$$

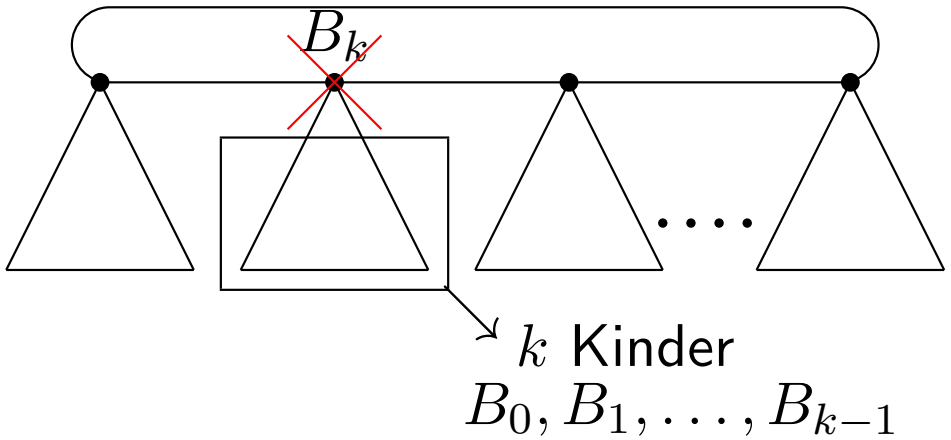
Operationen für Binomial Queues:

- *FindMin*: Diese Operation ist trivial ausführbar, denn es ist ein Pointer auf das minimale Element gegeben.

Zeitkomplexität: $\mathcal{O}(1)$

Operationen für Binomial Queues:

- *ExtractMin*:Das Minimum ist auf Grund der Heapbedingung Wurzel eines Binomialbaums B_k in der Liste. Wird es gelöscht, so zerfällt der Rest in k Teilbäume $B_0, B_1, \ldots, B_{k-1}$:



Die Teilbäume $B_0, B_1, \ldots, B_{k-1}$ sind alle zur verbleibenden Queue zu mergen. Außerdem muss der Min-Pointer aktualisiert werden.
Der Zeitaufwand für die *ExtractMin*-Operation ist daher:

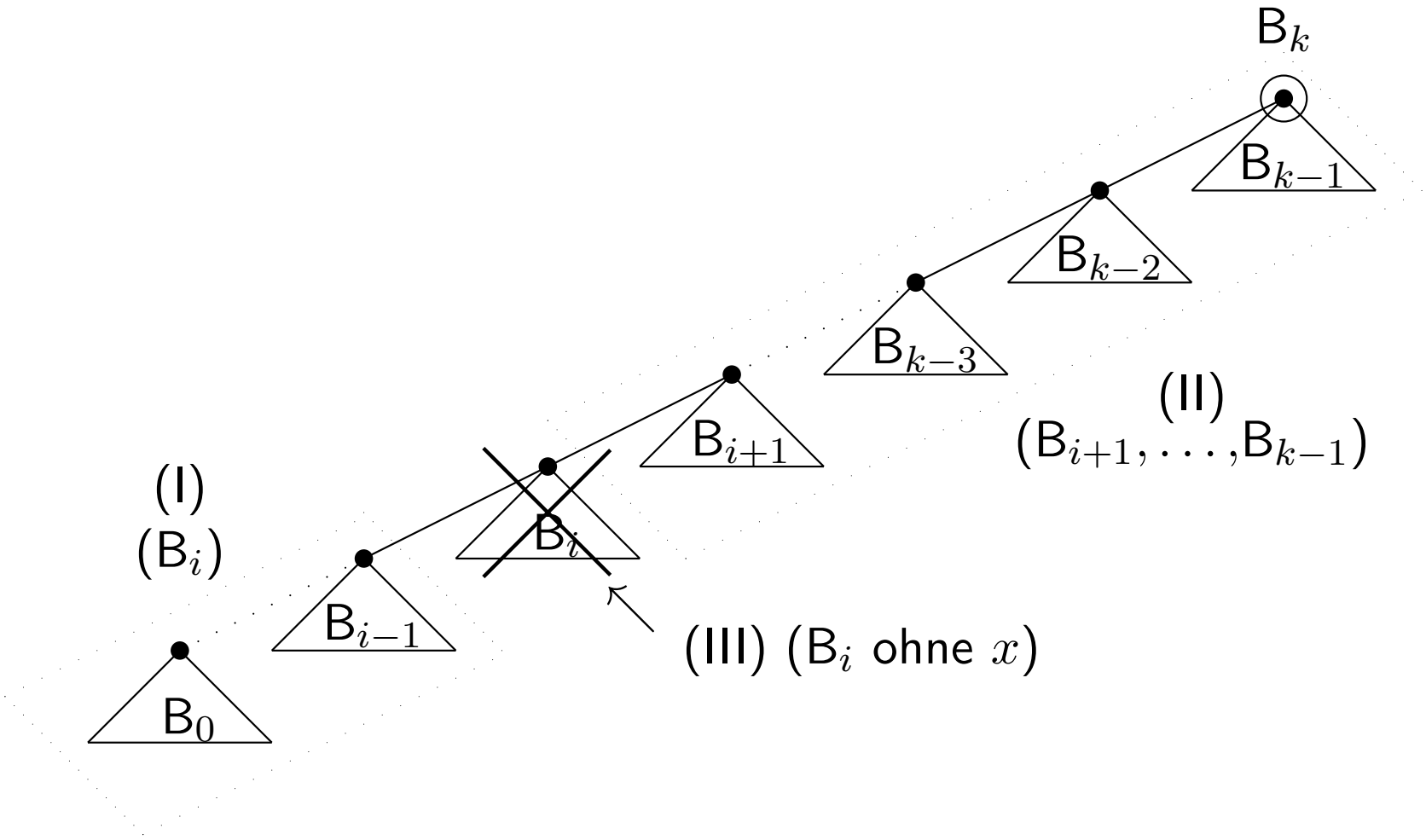
$$\boxed{\mathcal{O}(\log n)}$$

Operationen für Binomial Queues:

- *Delete*: Lösche Knoten x:

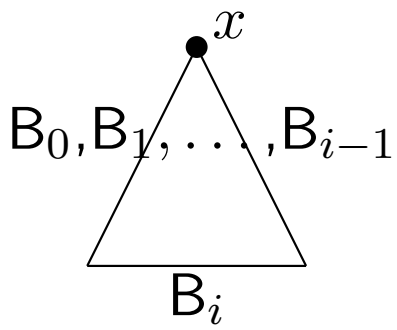
1. Fall: x ist Min-Wurzel: s.o.
2. Fall: x ist eine andere Wurzel in der Wurzelliste. Analog zu oben, ohne den Min-Pointer zu aktualisieren.
3. Fall: x ist nicht Wurzel eines Binomialbaumes.

Angenommen, x ist in einem B_k enthalten:



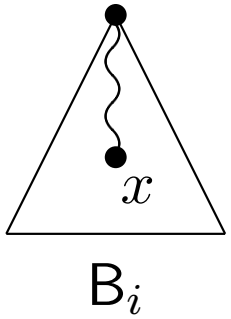
Es ergeben sich im 3. Fall folgende Unterfälle:

(3a) x ist Wurzel von B_i:



Verfahre wie im 2. Fall.

(3b) x ist nicht Wurzel des B_i:



Wiederhole rekursiv Fall 3, bis Fall (3a) eintritt.

Insgesamt muss die Binomial Queue ohne B_k mit einer aus einer Teilmenge von $\{B_0, \ldots, B_{k-1}\}$ bestehenden Binomial Queue vereinigt werden.

Zeitkomplexität: $\mathcal{O}(\log n)$

Operationen für Binomial Queues:

- *DecreaseKey*: Verkleinere $k(x)$

1. Fall: x ist die Min-Wurzel: keine Strukturänderung nötig
2. Fall: x ist eine andere Wurzel: keine Strukturänderung nötig, ggf. Aktualisierung des Min-Pointers
3. Fall: Sonst wie *Delete*(x), aber Einfügen/Merge des Baumes mit Wurzel x und dem neuen reduzierten Schlüssel $k(x)$ in die Wurzelliste.

Zeitkomplexität: $\mathcal{O}(\log n)$

Satz 46

Binomial Queues haben für die Operationen Insert, Delete, ExtractMin, FindMin, Merge und Initialize jeweils worst-case-Kosten von

$$\mathcal{O}(\log n)\,.$$

Die ursprüngliche Literatur zu Binomial Queues:

Jean Vuillemin:

A data structure for manipulating priority queues,
Commun. ACM **21**(4), pp. 309–315 (1978)

Mark R. Brown:

Implementation and analysis of binomial queue algorithms,
SIAM J. Comput. **7**(3), pp. 298–319 (1978)

5.2 Fibonacci-Heaps

Vorbemerkungen:

1. Fibonacci-Heaps stellen eine Erweiterung der Binomial Queues und eine weitere Möglichkeit zur Implementierung von Priority Queues dar. Die amortisierten Kosten für die Operationen *Delete()* und *ExtractMin()* betragen hierbei $\mathcal{O}(\log n)$, die für alle anderen Heap-Operationen lediglich $\mathcal{O}(1)$.
   Natürlich können die worst-case-Gesamtkosten für n *Insert* und n *ExtractMin* nicht unter $\Omega(n \log n)$ liegen, denn diese Operationen zusammengenommen stellen einen Sortieralgorithmus mit unterer Schranke $\Omega(n \log n)$ dar.

Vorbemerkungen:

2. Die Verwendung von Fibonacci-Heaps erlaubt eine Verbesserung der Komplexität der Algorithmen für minimale Spannbäume sowie für Dijkstra's kürzeste Wege-Algorithmus, denn diese verwenden relativ häufig die *DecreaseKey*-Operation, welche durch Fibonacci-Heaps billig zu implementieren ist. Bei einem Algorithmus, bei dem die *Delete*- und *ExtractMin*-Operationen nur einen geringen Anteil der Gesamtoperationen darstellen, können Fibonacci-Heaps asymptotisch schneller als Binomial Queues sein.

5.2.1 Die Datenstruktur

- Die Schlüssel sind an den Knoten von Bäumen gespeichert.
- Jeder Knoten hat folgende Größen gespeichert:
  - Schlüssel und Wert
  - Rang (= Anzahl der Kinder)
  - Zeiger zum ersten Kind, zum Vater (NIL im Falle der Wurzel), zu doppelt verketteter Liste der Kinder
  - Markierung $\in \{0, 1\}$ (außer Wurzel)

Bemerkung: Die doppelte Verkettung der Kinder- bzw. Wurzellisten in Heaps erlaubt das Löschen eines Listeneintrages in Zeit $\mathcal{O}(1)$.

Binomial-Queue vs. Fibonacci-Heap:

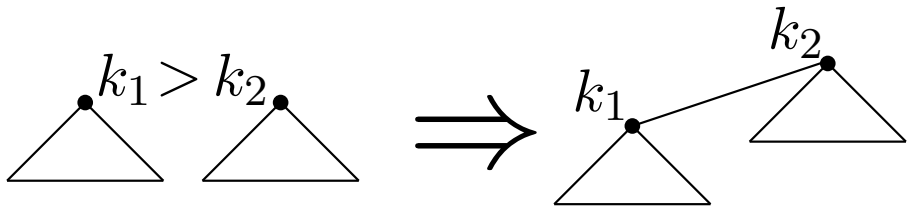
- Beide sind Wälder von Bäumen, innerhalb derer die Heap-Bedingung gilt.
- *a priori* keine Einschränkung für die Topologie der Bäume, aber ohne *Delete*- oder *DekreaseKey*-Operationen (u.ä.) bleibt ein Binomialwald ein Binomialwald und ein Fibonacci-Heap ein Wald aus Binomialbäumen.
- Fibonacci-Heaps kennen keine Invariante der Form „Nur Bäume verschiedenen Wurzel-Rangs“.
- Fibonacci-Heaps: lazy merge.
- Fibonacci-Heaps: lazy delete.

Überblick:

Operationen	worst case	amortisiert
Insert	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Merge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
FindMin	$\mathcal{O}(1)$	$\mathcal{O}(1)$
DecreaseKey	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Delete	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
ExtractMin	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

Linken von Bäumen:

Zwei Bäume desselben Wurzel-Rangs werden unter Einhaltung der Heap-Bedingung verbunden. Sind k_1 und k_2 die Wurzeln der zwei zu linkenden Bäume, so wird ein neuer Baum aufgebaut, dessen Wurzel bzgl. der Schlüssel das Minimum von k_1 und k_2 ist. Sei dies k_2, dann erhält der Baum mit Wurzel k_2 als zusätzlichen Unterbaum den Baum mit Wurzel k_1, d.h. der Rang von k_2 erhöht sich um 1.



Dies ist gerade der konstruktive Schritt beim Aufbau von Binomialbäumen.

Zur Verbesserung der amortisierten Zeitschranken:

Kaskadierendes Abschneiden:

- Ein Knoten ist im „Anfangszustand“, wenn
  - i) er durch einen Linkschritt Kind eines anderen Knotens wird oder
  - ii) er in die Wurzelliste eingefügt wird.

  Sein Markierungsbit wird in jedem dieser Fälle zurückgesetzt.
- Wenn ein Knoten ab seinem Anfangszustand zum zweitenmal ein Kind verliert, wird er samt seines Unterbaums aus dem aktuellen Baum entfernt und in die Wurzelliste eingefügt.
- Der kritische Zustand (ein Kind verloren) wird durch Setzen des Markierungsbits angezeigt.

Algorithmus:

co x verliert Kind **oc**
while x markiert **do**
    entferne x samt Unterbaum
    entferne Markierung von x
    füge x samt Unterbaum in Wurzelliste ein
    if $P(x)$=NIL **then return fi**
    $x := P(x)$ **co** (Vater von x) **oc**
od
if x nicht Wurzel **then** markiere x **fi**

Operationen in Fibonacci-Heaps:

1. *Insert*(x): Füge B_0 (mit dem Element x) in die Wurzelliste ein. Update Min-Pointer.

   Kosten $\mathcal{O}(1)$

2. *Merge()*: Verbinde beide Listen und aktualisiere den Min-Pointer.

   Kosten $\mathcal{O}(1)$

3. *FindMin()*: Es wird das Element ausgegeben, auf das der Min-Pointer zeigt. Dabei handelt es sich sicher um eine Wurzel.

   Kosten $\mathcal{O}(1)$

Operationen in Fibonacci-Heaps:

4. *Delete*(x)
   - i) Falls x Min-Wurzel, *ExtractMin*-Operator (s.u.) benutzen
   - ii) Sonst:

füge Liste der Kinder von x in die Wurzelliste ein; lösche x
if $P(x)$=NIL **then return fi** **co** x ist Wurzel **oc**
while true do
    $x := P(x)$
    if $P(x)$=NIL **then return fi** **co** x ist Wurzel **oc**
    if Markierung(x)=0 **then** Markierung(x):=1; **return**
    else
        hänge x samt Unterbaum in Wurzelliste
        entferne Markierung von x (da x nun Wurzel)
    fi
od

Kosten: $\mathcal{O}(1 + \#\text{kask. Schnitte})$

Operationen in Fibonacci-Heaps:

5. *ExtractMin()*: Diese Operation hat auch Aufräumfunktion und ist daher recht kostspielig. Sei x der Knoten, auf den der Min-Pointer zeigt.

   entferne x aus der Liste
   konkateniere Liste der Kinder von x mit der Wurzelliste
   while $\exists \geq 2$ Bäume mit gleichem Wurzel-Rang i **do**
       erzeuge Baum mit Wurzel-Rang $i+1$
   od
   update Min-Pointer

Man beachte, dass an jedem Knoten, insbesondere jeder Wurzel, der Rang gespeichert ist. Zwar vereinfacht dies die Implementierung, doch müssen noch immer Paare von Wurzeln gleichen Rangs effizient gefunden werden.

Wir verwenden dazu ein Feld (Array), dessen Positionen je für einen Rang stehen. Die Elemente sind Zeiger auf eine Wurzel dieses Rangs. Es ist garantiert, dass ein Element nur dann unbesetzt ist, wenn tatsächlich keine Wurzel entsprechenden Rangs existiert. Nach dem Entfernen des Knoten x aus der Wurzelliste fügen wir die Kinder eines nach dem anderen in die Wurzelliste ein und aktualisieren in jedem Schritt die entsprechende Feldposition. Soll eine bereits besetzte Position des Arrays beschrieben werden, so wird ein Link-Schritt ausgeführt und versucht, einen Pointer auf die neue Wurzel in die nächsthöhere Position im Array zu schreiben. Dies zieht evtl. weitere Link-Schritte nach sich. Nach Abschluss dieser Operation enthält der Fibonacci-Heap nur Bäume mit unterschiedlichem Wurzel-Rang.

Kosten: $\mathcal{O}(\text{max. Rang } + \#\text{Link-Schritte})$

Operationen in Fibonacci-Heaps:

6. *DecreaseKey*(x, Δ):

   entferne x samt Unterbaum
   füge x mit Unterbaum in die Wurzelliste ein
   $k(x) := k(x) - \Delta$; aktualisiere Min-Pointer
   if $P(x)$=NIL **then return fi** **co** x ist Wurzel **oc**
   while true do
      $x := P(x)$
      if $P(x)$=NIL **then return fi** **co** x ist Wurzel **oc**
      if Markierung(x)=0 **then** Markierung(x):=1; **return**
      else
         hänge x samt Unterbaum in Wurzelliste
         entferne Markierung von x (da x nun Wurzel)
      fi
   od

Kosten: $\mathcal{O}(1 + \#\text{kask. Schnitte})$

Bemerkung:
Startet man mit einem leeren Fibonacci-Heap und werden ausschließlich die aufbauenden Operationen *Insert*, *Merge* und *FindMin* angewendet, so können nur Binomialbäume enstehen. In diesem natürlichen Fall liegt also stets ein Binomialwald vor, der jedoch i.a. nicht aufgeräumt ist. Das heißt, es existieren ev. mehrere Binomialbäume B_i desselben Wurzelgrads, die nicht paarweise zu B_{i+1}-Bäumen verschmolzen sind.

Dies geschieht erst bei der Operation *ExtractMin*. Man beachte, dass nur nach einer solchen Operation momentan ein Binomial Heap vorliegt, ansonsten nur ein Binomialwald.

Treten auch *DecreaseKey*- und/oder *Delete*-Operationen auf, so sind die Bäume i.a. keine Binomialbäume mehr.

5.2.2 Amortisierte Kostenanalyse für Fibonacci-Heaps

Kostenanalyse für Folgen von Operationen:

i) Summieren der worst-case-Kosten wäre zu pessimistisch. Der resultierende Wert ist i.a. zu groß.

ii) average-case:
- Aufwand für Analyse sehr hoch
- welcher Verteilung folgen die Eingaben?
- die ermittelten Kosten stellen keine obere Schranke für die tatsächlichen Kosten dar!

iii) amortisierte Kostenanalyse:

average-case-Analyse über worst-case-Operationenfolgen

Definition 47

Wir führen für jede Datenstruktur ein Bankkonto ein und ordnen ihr eine nichtnegative reelle Zahl bal, ihr Potenzial (bzw. Kontostand) zu. Die amortisierten Kosten für eine Operation ergeben sich als Summe der tatsächlichen Kosten und der Veränderung des Potenzials (Δbal), welche durch die Operation verursacht wird:

$t_i =$ tatsächliche Kosten der i-ten Operation

$\Delta bal_i = bal_i - bal_{i-1}$: Veränderung des Potenzials durch die i-te Operation

$a_i = t_i + \Delta bal_i$: amortisierte Kosten der i-ten Operation

$m =$ Anzahl der Operationen

Falls $bal_m \geq bal_0$ (was bei $bal_0 = 0$ stets gilt):

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Delta bal_i) = \sum_{i=1}^{m} t_i + bal_m - bal_0 \geq \sum_{i=1}^{m} t_i$$

In diesem Fall sind die amortisierten Kosten einer Sequenz eine obere Schranke für die tatsächlichen Kosten.

Anwendung auf Fibonacci-Heaps:
Wir setzen

$$bal := \#\text{ Bäume} + 2\#(\text{markierte Knoten} \neq \text{Wurzel})$$

Lemma 48

Sei x ein Knoten im Fibonacci-Heap mit Rang$(x) = k$. Seien die Kinder von x sortiert in der Reihenfolge ihres Anfügens an x. Dann ist der Rang des i-ten Kindes $\geq i - 2$.

Beweis:
Zum Zeitpunkt des Einfügens des i-ten Kindes ist Rang$(x) = i - 1$.

Das einzufügende i-te Kind hat zu dieser Zeit ebenfalls Rang $i - 1$.

Danach kann das i-te Kind höchstens eines seiner Kinder verloren haben

$$\Rightarrow \text{Rang des } i\text{-ten Kindes} \geq i - 2\,.$$

□

Satz 49

Sei x Knoten in einem Fibonacci-Heap, Rang$(x) = k$. Dann enthält der (Unter-)Baum mit Wurzel x mindestens F_{k+2} Elemente, wobei F_k die k-te Fibonacci-Zahl bezeichnet.

Da

$$F_{k+2} \geq \Phi^k$$

für $\Phi = (1+\sqrt{5})/2$ (Goldener Schnitt), ist der Wurzelrang also logarithmisch in der Baumgröße beschränkt.

Wir setzen hier folgende Eigenschaften der Fibonacci-Zahlen F_k voraus:

$$F_{k+2} \geq \Phi^k \text{ für } \Phi = \frac{1+\sqrt{5}}{2} \approx 1,618034\,;$$

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i\,.$$

Beweis:

Sei f_k die minimale Anzahl von Elementen in einem „Fibonacci-Baum“ mit Wurzel-Rang k.
Aus dem vorangehenden Lemma folgt:

$$f_k \geq f_{k-2} + f_{k-3} + \ldots + f_0 + \underbrace{1}_{\text{1. Kind}} + \underbrace{1}_{\text{Wurzel}},$$

also (zusammen mit den offensichtlichen Anfangsbedingungen $f_0 = 1$ bzw. $f_1 = 2$ und den obigen Eigenschaften der Fibonacci-Zahlen):

$$f_k \geq F_{k+2}$$

□

Satz 50

Die amortisierten Kosten der Fibonacci-Heap-Operationen ($a_i = t_i + \Delta bal_i$) sind:

1. *Insert:* $t = \mathcal{O}(1), \Delta bal = +1 \Rightarrow a = \mathcal{O}(1)$.
2. *Merge:* $t = \mathcal{O}(1), \Delta bal = 0 \Rightarrow a = \mathcal{O}(1)$.
3. *FindMin:* $t = \mathcal{O}(1), \Delta bal = 0 \Rightarrow a = \mathcal{O}(1)$.

Satz 50

Die amortisierten Kosten der Fibonacci-Heap-Operationen ($a_i = t_i + \Delta bal_i$) sind:

4. *Delete (nicht Min-Knoten):*

Einfügen der Kinder von x in Wurzelliste:
$\Delta bal =$ *Rang*(x)
Jeder kask. Schnitt erhöht #Bäume um 1
$\Delta bal = \#$*kask. Schnitte*
Jeder Schnitt vernichtet eine Markierung
$\Delta bal = -2 \cdot \#$*kask. Schnitte*
Letzter Schnitt erzeugt ev. eine Markierung
$\Delta bal = 2$
$\Rightarrow$ *Jeder kask. Schnitt wird vom Bankkonto bezahlt und verschwindet amortisiert*

$\Rightarrow a = \mathcal{O}(\log n)$

Satz 50

Die amortisierten Kosten der Fibonacci-Heap-Operationen ($a_i = t_i + \Delta bal_i$) sind:

5. *ExtractMin:*

Einfügen der Kinder von x in Wurzelliste:
$\Delta bal =$ *Rang*(x)
Jeder Link-Schritt verkleinert #Bäume um 1
$\Delta bal = -$*#Link-Schritte*
$\Rightarrow$ *Jeder Link-Schritt wird vom Bankkonto bezahlt und verschwindet amortisiert*

$\Rightarrow a = \mathcal{O}(\log n)$

Satz 50

Die amortisierten Kosten der Fibonacci-Heap-Operationen ($a_i = t_i + \Delta bal_i$) *sind:*

6. *DecreaseKey: Es ist* $\Delta bal \leq 4 - (\#$*kaskadierende Schnitte*$)$.

$$\left.\begin{array}{l} \text{Jeder kask. Schnitt erhöht } \#\text{Bäume um } 1 \\ \Delta bal = \#\text{kask. Schnitte} \\ \text{Jeder Schnitt vernichtet eine Markierung} \\ \Delta bal = -2 \cdot \#\text{kask. Schnitte} \\ \text{Letzter Schnitt erzeugt ev. eine Markierung} \\ \Delta bal = 2 \end{array}\right\} \Rightarrow a = \mathcal{O}(1)$$

Beweis:

s.o. □

Literatur zu Fibonacci-Heaps:

Michael L. Fredman, Robert Endre Tarjan:
Fibonacci heaps and their uses in improved network optimization algorithms
Journal of the ACM **34**(3), pp. 596–615 (1987)

James R. Driscoll, Harold N. Gabow, Ruth Shrairman, Robert E. Tarjan:
Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation
Commun. ACM **31**(11), pp. 1343–1354 (1988)

Mikkel Thorup:
Equivalence between priority queues and sorting
Journal of the ACM **54**(6), Article 28 (2007)

6. Sich selbst organisierende Datenstrukturen

6.1 Motivation

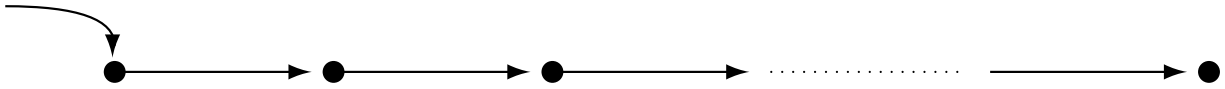
- einfach, wenig Verwaltungsoverhead
- effizient im amortisierten Sinn

6.2 Sich selbst organisierende lineare Listen

Bei der Implementierung von Wörterbüchern ist auf möglichst effiziente Weise eine Menge von bis zu n Elementen zu organisieren, dass die folgenden Operationen optimiert werden:

- Einfügen
- Löschen
- Vorhandensein überprüfen

Dabei soll die Effizienz der Implementierung bei beliebigen Sequenzen von m Operationen untersucht werden. Die Menge wird als unsortierte Liste dargestellt.



Operationen:

1. *Access*(x): Suchen des Elements x:
   Die Liste wird von Anfang an durchsucht, bis das gesuchte Element X gefunden ist. Ist x das i-te Element in der Liste, so betragen die tatsächlichen Kosten i Einheiten.
2. *Insert*(x): Einfügen des Elements x:
   Die Liste wird von Anfang an durchsucht, und wenn sie vollständig durchsucht wurde und das Element nicht enthalten ist, so wird es hinten angefügt. Dies erfordet $(n' + 1)$ Schritte, wobei n' die aktuelle Länge der Liste ist.
3. *Delete*(x): Löschen des Elements x:
   Wie bei *Access* wird die Liste zuerst durchsucht und dann das betreffende Element gelöscht. Dies kostet im Falle des i-ten Elements i Schritte.

Nach Abschluss einer jeden Operation können Umordnungsschritte stattfinden, die spätere Operationen ev. beschleunigen.

Annahme: Wir nehmen an, dass nach Ausführung jeder *Access*- oder *Insert*-Operation, die auf Element i angewandt wird, dieses Element kostenlos um beliebig viele Positionen in Richtung Listenanfang geschoben werden kann. Wir nennen jede daran beteiligte Elementvertauschung einen kostenlosen Tausch. Jeder andere Tausch zweier Elemente ist ein bezahlter Tausch und kostet eine Zeiteinheit.

Ziel:
Es soll eine einfache Regel gefunden werden, durch die mittels obiger Vertauschungen die Liste so organisiert wird, dass die Gesamtkosten einer Folge von Operationen minimiert werden. Wir sprechen von selbstorganisierenden linearen Listen und untersuchen hierzu folgende Regeln:

- MFR (Move-to-Front-Rule): *Access* und *Insert* stellen das betreffende Element vorne in die Liste ein und verändern ansonsten nichts.
- TR (Transposition-Rule): *Access* bringt das Element durch eine Vertauschung um eine Position nach vorne, *Insert* hängt es ans Ende der Liste an.

Bemerkung:
Wir werden im Folgenden sehen, dass MFR die Ausführung einer Sequenz von Operationen *Access*, *Insert* und *Delete* mit einer amortisierten Laufzeit erlaubt, die um höchstens den konstanten Faktor 2 schlechter ist als ein optimaler Listenalgorithmus.

Für TR dagegen gibt es keinen solchen konstanten Faktor. Sie kann verglichen mit MTR bzw. einem optimalen Algorithmus beliebig schlecht sein.

Für die Algorithmen nehmen wir an:

1. die Kosten für *Access*(x) und *Delete*(x) sind gleich der Position von x in der Liste (vom Anfang an gezählt);
2. die Kosten für *Insert*(x) (x nicht in Liste) sind die Länge der Liste nach der *Insert*-Operation (d.h., neue Elemente werden zunächst am Listenende angehängt);
3. Ein Algorithmus kann die Liste jederzeit umorganisieren (die Reihenfolge ändern):
   (a) kostenlose Vertauschung: nach einer *Access*(x)- oder *Insert*(x)-Operation kann x beliebig näher an den Anfang der Liste verschoben werden;
   (b) bezahlte Vertauschung: jede andere Vertauschung benachbarter Listenelemente kostet eine Einheit.

Bemerkung: MFR (und auch TR) benutzen nur kostenlose Vertauschungen. 3(a) ist pessimistisch für MFR.

Satz 51

Sei s eine beliebige Folge von Access-, Insert- und Delete-Operationen, beginnend mit einer leeren Liste. Sei A ein beliebiger (optimaler) Wörterbuch-Algorithmus mit obigen Einschränkungen. Sei $C_A(s)$ der Zeitaufwand von A auf s, ohne bezahlte Vertauschungen. Sei $X_A(s)$ die Anzahl der bezahlten Vertauschungen. Sei $C_{MFR}(s)$ der Gesamtaufwand der Move_to_Front-Heuristik auf s. Sei $F_A(s)$ die Anzahl der kostenlosen Vertauschungen, die A auf s ausführt. Sei $m := |s|$ die Länge von s. Dann

$$C_{MFR}(s) \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m$$
$$\left(\leq 2 \cdot (C_A(s) + X_A(s)) \, . \right)$$

Beweis:

L_{A} bzw. L_{MFR} seien die Listen, A bzw. MFR seien die Algorithmen.

Potenzial := Anzahl der Inversionen in L_{MFR} im Vergleich zu L_{A}, d.h., falls o.B.d.A. $L_{\mathsf{A}} = (x_1, x_2, \ldots, x_n)$, dann ist das Potenzial

$$bal(L_{\mathsf{MFR}}, L_{\mathsf{A}}) = |\{(i,j);\ i < j,\ x_j \text{ ist in } L_{\mathsf{MFR}} \text{ vor } x_i\}|$$

Beweis (Forts.):

Wir betrachten im Folgenden *Insert*(x) als ein *Access* auf ein (fiktives) $n+1$-tes Element. Eine Operation (*Access*(x_i) bzw. *Delete*(x_i)) kostet für A dann i Einheiten und für MFR t Einheiten, wobei t die Position von x_i in L_{MFR} ist.

i) A ändert (bei *Access* bzw. *Delete*) L_{A} in kanonischer Weise. Für Element x_i sind die Kosten dann i.

Beweis (Forts.):

ii) Die amortisierten Kosten einer Operation (*Access*(x_i) oder *Delete*(x_i)) für MFR sind $\leq 2i-1$:

L_A : $x_1\ x_2\ \ldots\ x_i\ \ldots\ldots\ldots x_n$

L_{MFR} : $\ldots\ldots\ldots\ x_j\ \ldots\ x_i\ \ldots$

$\uparrow$

t

Sei $k = \#$Elemente x_j, $j > i$, die in L_{MFR} vor x_i stehen.
Sei t die Position von x_i im L_{MFR}. Dann

$$t \leq i + k; \qquad \Rightarrow \qquad k \geq t - i$$

$$\Delta bal \leq -k + (i-1) \leq 2i - 1 - t \qquad \text{(bei } \textit{Access}\text{)}$$

$$\Delta bal = -k \leq i - t \qquad \text{(bei } \textit{Delete}\text{)}$$

Also: Amortisierte Kosten $\leq t + \Delta bal \leq 2i - 1$.

Beweis (Forts.):

iii) Die amortisierten Kosten einer kostenlosen Vertauschung durch Algorithmus A sind ≤ -1:

L_A : $x_1\ x_2 \ldots \overset{\frown}{x_{i-1}\ x_i} \ldots x_n$ eine Inversion weniger

L_{MFR} : $x_i \ldots\ldots x_{i-1} \ldots\ldots$

iv) Die amortisierten Kosten, die Algorithmus A durch eine bezahlte Vertauschung verursacht, sind ≤ 1:

L_A : $x_1\ x_2 \ldots \overset{\frown}{x_{i-1}\ x_i} \ldots x_n$ plus eine Inversion

L_{MFR} : $\ldots x_{i-1} \ldots x_i \ldots\ldots$

Beweis (Forts.):

Ist das Anfangspotenzial $= 0$, so ergibt sich

$$\mathsf{C}_{\mathsf{MFR}}(s) \leq \text{amort. Kosten MFR } \leq 2 \cdot \mathsf{C}_{\mathsf{A}}(s) + \mathsf{X}_{\mathsf{A}}(s) - \mathsf{F}_{\mathsf{A}}(s) - m$$

□

Anmerkung: Wenn wir nicht mit einer leeren Liste (oder zwei identischen Listen, d.h. $\mathsf{L}_{\mathsf{A}} = \mathsf{L}_{\mathsf{MFR}}$) starten, ergibt sich:

$$\mathsf{C}_{\mathsf{MFR}} \leq 2\mathsf{C}_{\mathsf{A}}(s) + \mathsf{X}_{\mathsf{A}}(s) - \mathsf{F}_{\mathsf{A}}(s) - m + n^2/2\,.$$

Daniel D. Sleator, Robert E. Tarjan:
Amortized efficiency of list update and paging rules
Commun. ACM **28**(2), pp. 202–208 (1985)

6.3 Sich selbst organisierende Binärbäume

Wir betrachten hier Binärbäume für Priority Queues.

Definition 52

Ein Leftist-Baum (Linksbaum) ist ein (interner) binärer Suchbaum, so dass für jeden Knoten gilt, dass ein kürzester Weg zu einem Blatt (externer Knoten!) über das rechte Kind führt [s. Knuth].

Zur Analyse von Linksbäumen verweisen wir auf

Knuth, Donald E.:
The Art of Computer Programming. Volume 3 / Sorting and Searching
Addison-Wesley Publishing Company: Reading, MA (1973)
pp. 150ff [pdf (54MB), djvu (8MB)]

Wir betrachten nun eine durch Linksbäume motivierte, sich selbst organisierende Variante als Implementierung von Priority Queues. Die dabei auftretenden Bäume brauchen keine Linksbäume zu sein!

Operationen:

- *Insert*
- *FindMin*
- *ExtractMin*
- *Delete*
- *Merge*

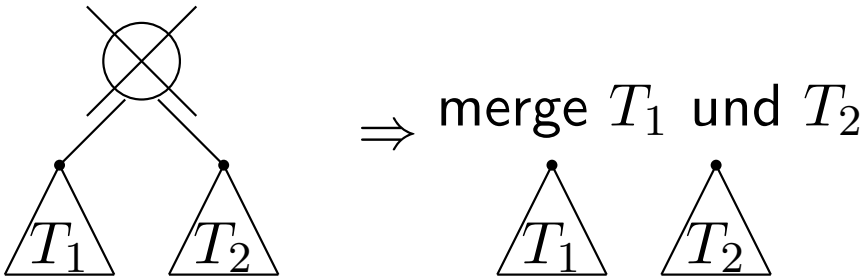
} alle Operationen lassen sich auf *Merge* zurückführen

Operationen:

1. *Insert*(T, x): neuer Baum mit x als einzigem Element, merge diesen mit T

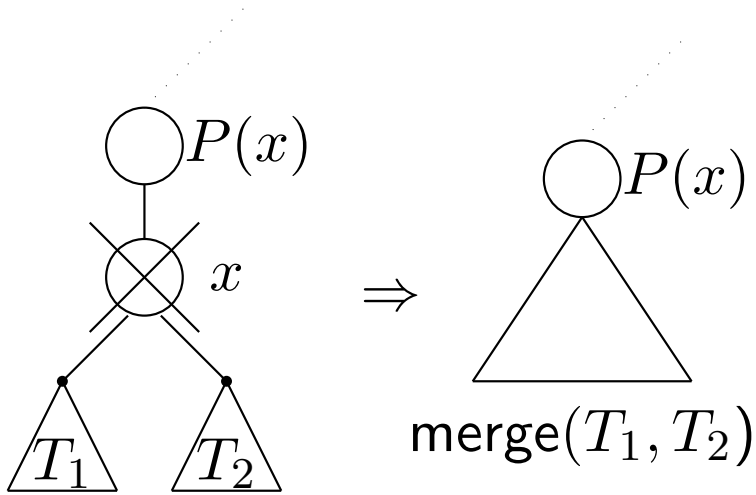
2. *FindMin*: $\surd$ (Minimum an der Wurzel, wegen Heap-Bedingung)

3. *ExtractMin*:



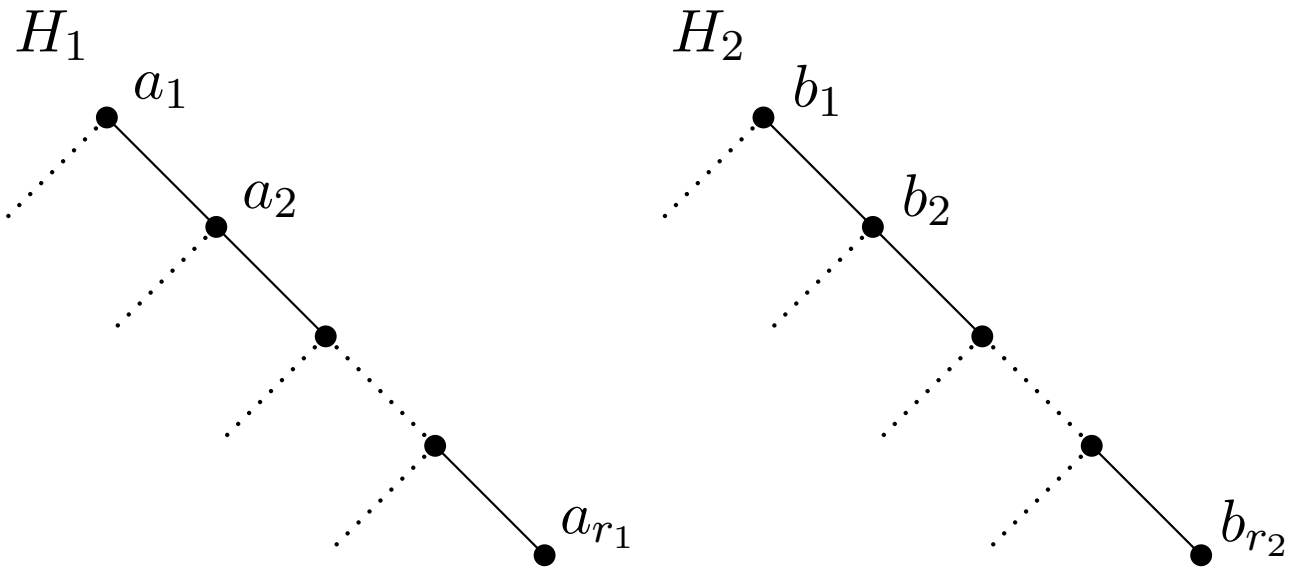
Operationen:

4. *Delete*:



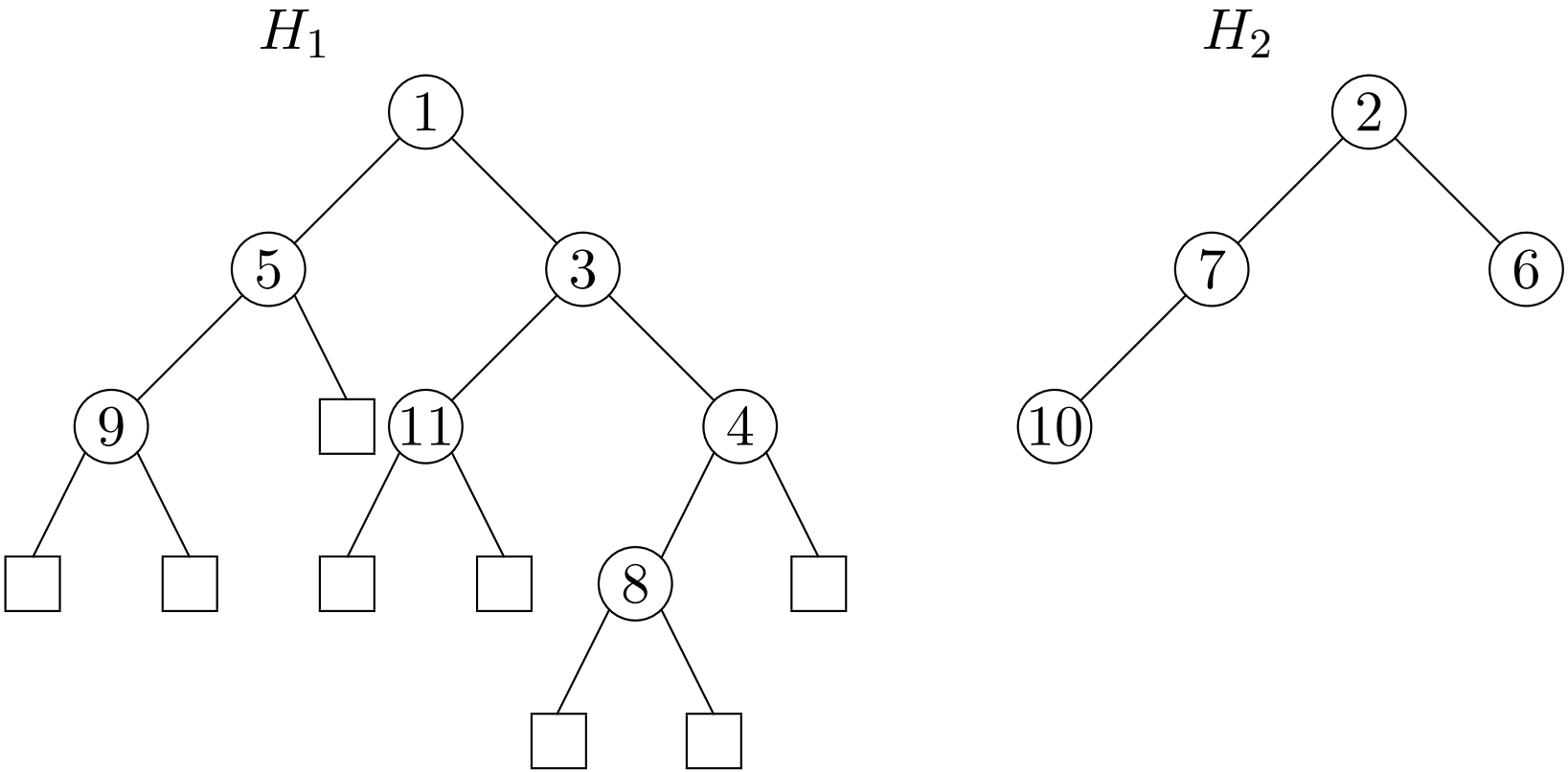
Operationen:

4. *Merge*: Seien zwei Heaps H_1 und H_2 gegeben, sei $a_1, \ldots, a_{r_1}$ die Folge der Knoten in H_1 von der Wurzel zum rechtesten Blatt, sei $b_1, \ldots, b_{r_2}$ die entsprechende Folge in H_2.



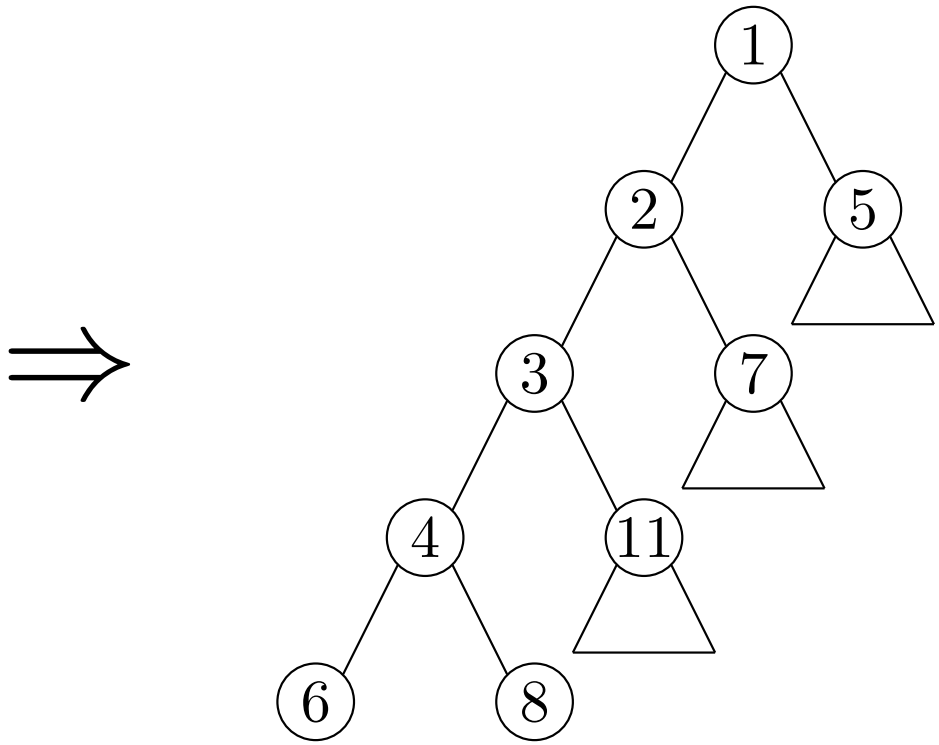
Sei $(c_1, c_2, \ldots, c_{r_1+r_2})$ die durch Merging aus $(a_1, \ldots, a_{r_1})$ und $(b_1, \ldots, b_{r_2})$ entstehende (sortierte) Folge. Mache $(c_1, \ldots, c_{r_1+r_2})$ zum linkesten Pfad im neuen Baum und hänge den jeweiligen anderen (d.h. linken) UB a_i bzw. b_j als rechten UB des entsprechenden c_k an.

Beispiel 53



(Bemerkung: H_1 ist kein Linksbaum!)

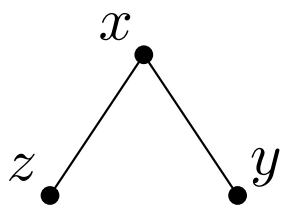
Beispiel 53



(Bemerkung: H_1 ist kein Linksbaum!)

Amortisierte Kostenanalyse der *Merge*-Operation:

Sei
$w(x) :=$ Gewicht von $x = \#$Knoten im UB von x einschließlich x



(x, y) schwer $\Rightarrow (x, z)$ leicht

Kante (x, y) heißt leicht, falls $2 \cdot w(y) \leq w(x)$
Kante (x, y) heißt schwer, falls $2 \cdot w(y) \geq w(x)$

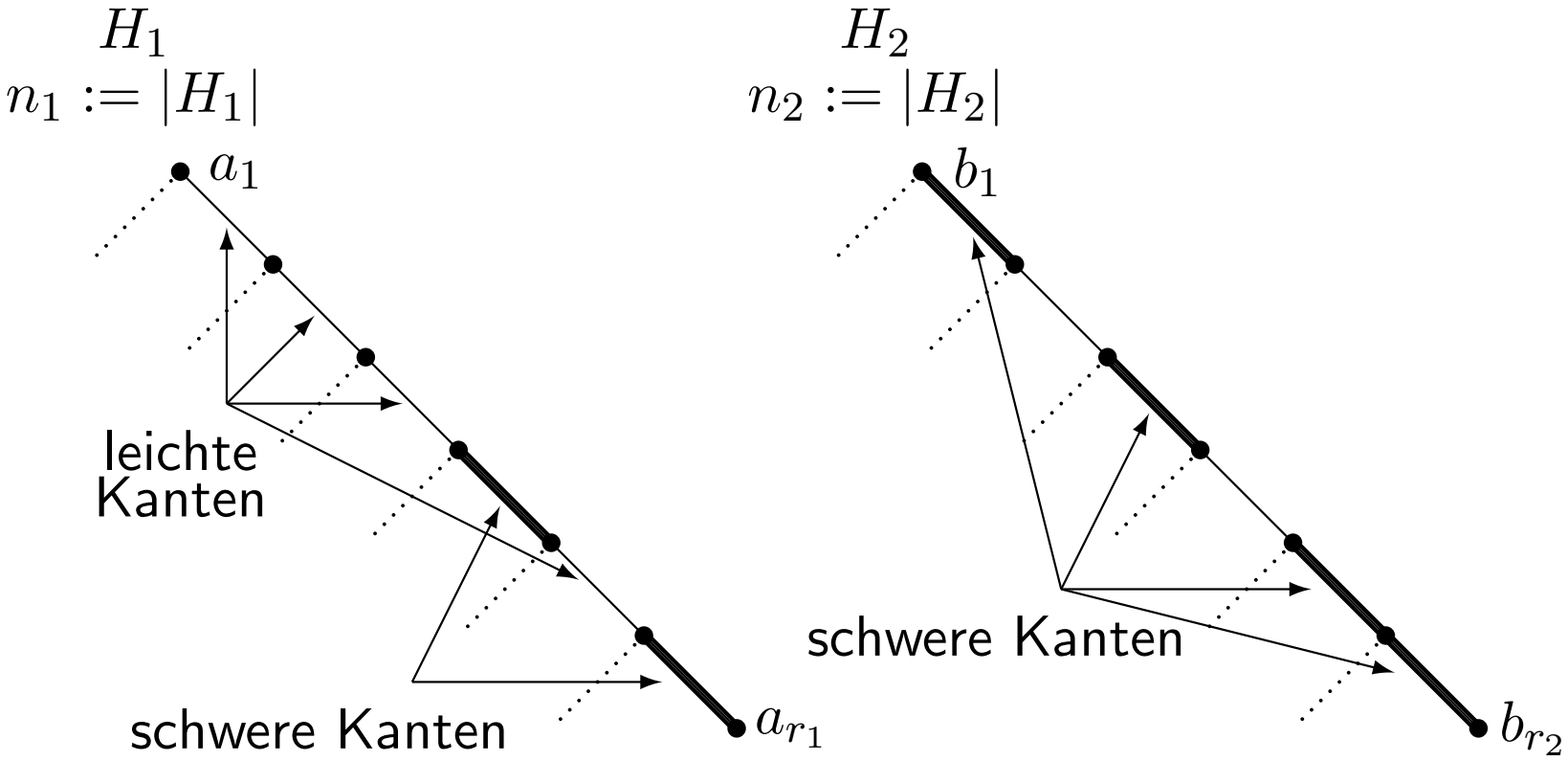
Es gilt: Es ist immer ein Kind über eine leichte, höchstens eins über eine schwere Kante erreichbar (d.h. die beiden Fälle oben schließen sich wechselseitig aus!). .

Beobachtungen:

1. (x, y) schwer $\Rightarrow$ (x, z) leicht
2. Da sich über eine leichte Kante das Gewicht mindestens halbiert, kann ein Pfad von der Wurzel zu einem Blatt höchstens $\mathrm{ld}\, n$ leichte Kanten enthalten.

Setze

$$\text{Potenzial} := \#\text{ schwere Kanten zu rechten Kindern}$$


H_1
$n_1 := |H_1|$
a_1
leichte
Kanten
schwere Kanten
a_{r_1}
H_2
$n_2 := |H_2|$
b_1
schwere Kanten
b_{r_2}

Sei $s_1 :=$ die Zahl der schweren Kanten auf dem rechten Pfad in H_1, s_2 analog.

$$r_1 \leq s_1 + \mathrm{ld}\, n_1$$
$$r_2 \leq s_2 + \mathrm{ld}\, n_2$$

Amortisierte Kosten für *Merge:*

$$\begin{aligned} &\leq r_1 + r_2 + \mathrm{ld}(n_1 + n_2) - s_1 - s_2 \\ &\leq \mathrm{ld}\, n_1 + \mathrm{ld}\, n_2 + \mathrm{ld}(n_1 + n_2) \\ &= \mathcal{O}(\log(n_1 + n_2))\ , \end{aligned}$$

da die s_1 schweren Kanten auf dem rechten Pfad in H_1 nun zu linken Kindern verlaufen (ebenso für die in H_2) und höchstens $\mathrm{ld}(n_1 + n_2)$ Knoten auf dem linken Pfad im neuen Baum eine schwere Kante zum rechten (und damit eine leichte Kante zum linken) Kind haben können.

Satz 54

Eine Folge von m Merge-Operationen benötigt Zeit

$$\mathcal{O}(\sum_{i=1}^{m} \log(n_i)) \,,$$

wobei n_i die Größe des Baumes ist, der in der i-ten Merge-Operation geschaffen wird.

Beweis:

s.o. □

6.4 Splay-Trees als Suchbäume

In diesem Abschnitt untersuchen wir Splay-Trees, eine Datenstruktur, die den MFR-Ansatz auf Bäume überträgt:

> *Wird auf ein Element durch eine Operation zugegriffen, so wird dieses im Splay-Tree in geeigneter Weise zur Wurzel befördert, um, sollten weitere Zugriffe auf dieses Element folgen, diese zu beschleunigen.*

Wir untersuchen hier Splay-Trees als Suchbäume, d.h. die Schlüssel stammen aus einem total geordneten Universum, und innerhalb des Splay-Trees soll die Invariante

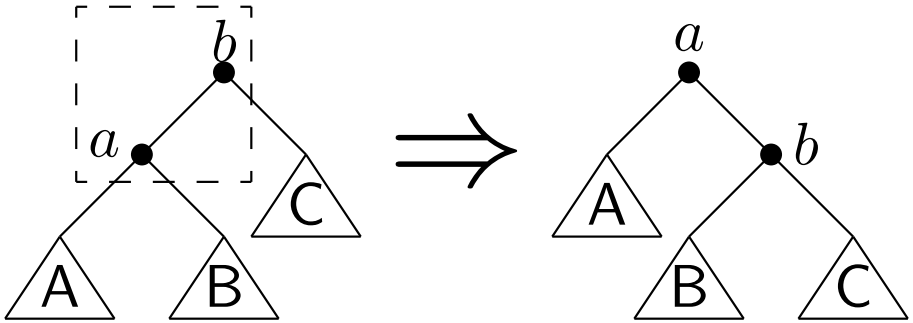
$$\text{“Knoten im lUb} \leq k(x) \leq \text{Knoten im rUb“}$$

gelten.

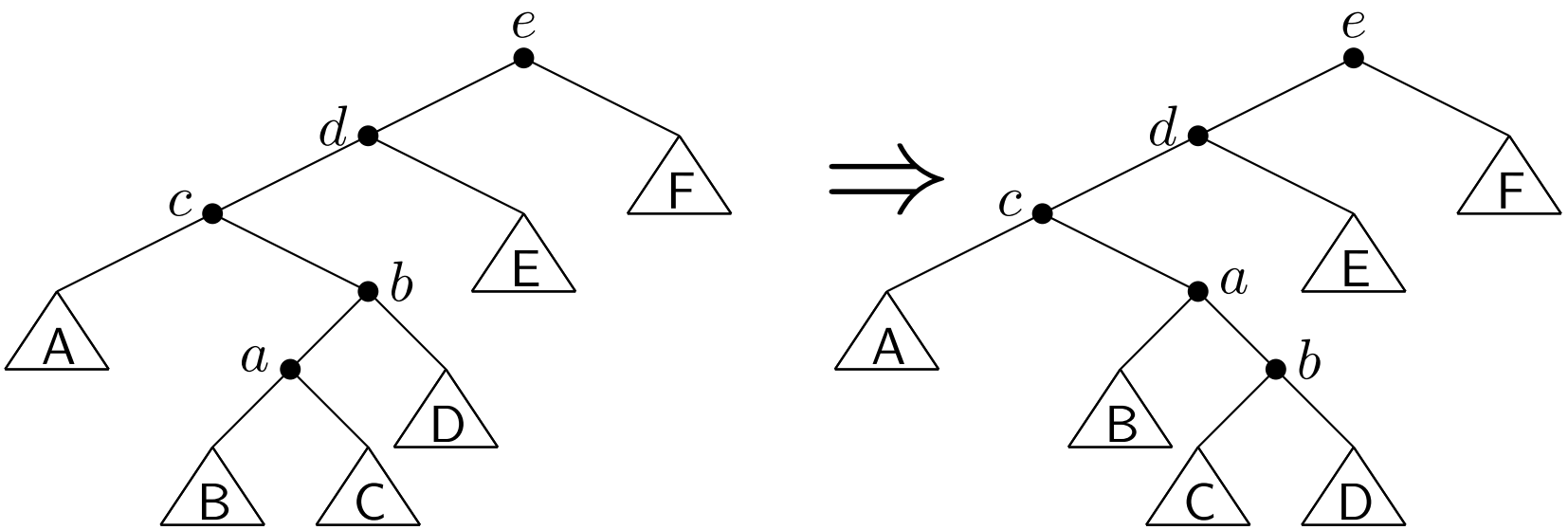
Ansonsten ist ein Splay-Tree ein interner binärer Suchbaum.

Wir benutzen Rotationen, um unter Beibehaltung der Invariante einen Schlüssel näher zur Wurzel zu bewegen, und zwar Einfach- und Doppelrotationen.

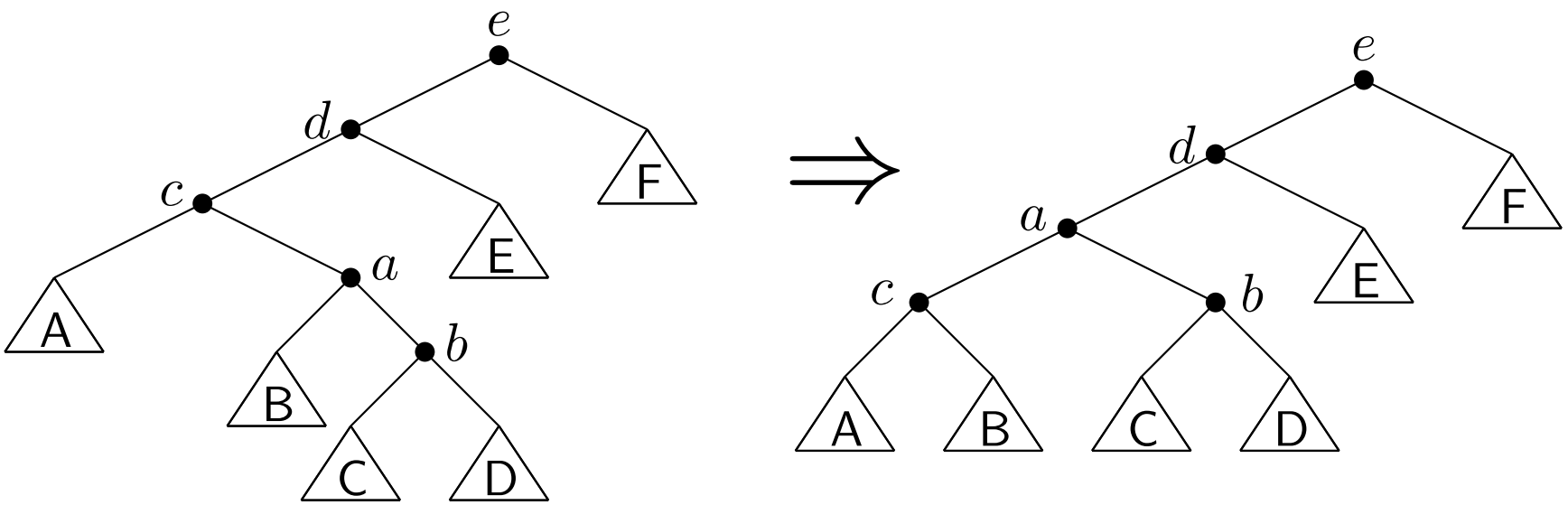
Beispiel 55



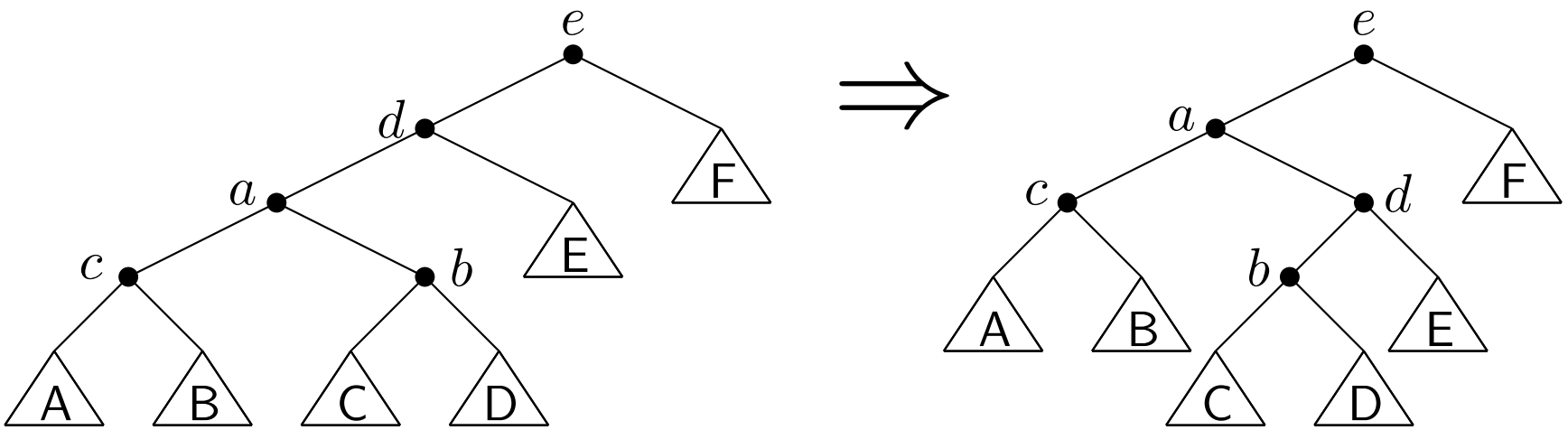
Beispiel 55



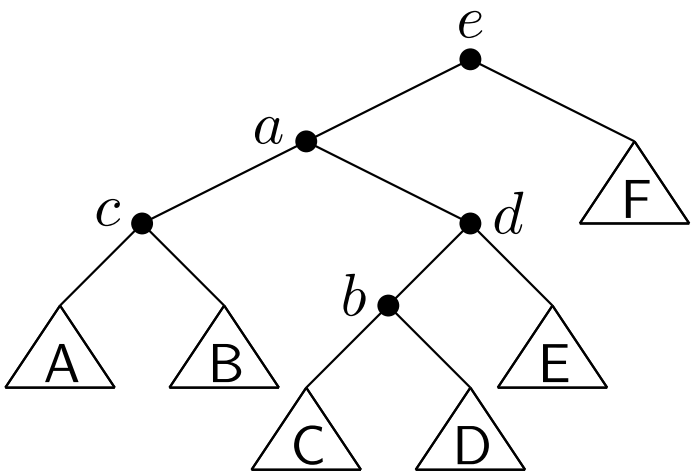
Beispiel 55

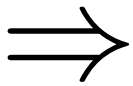


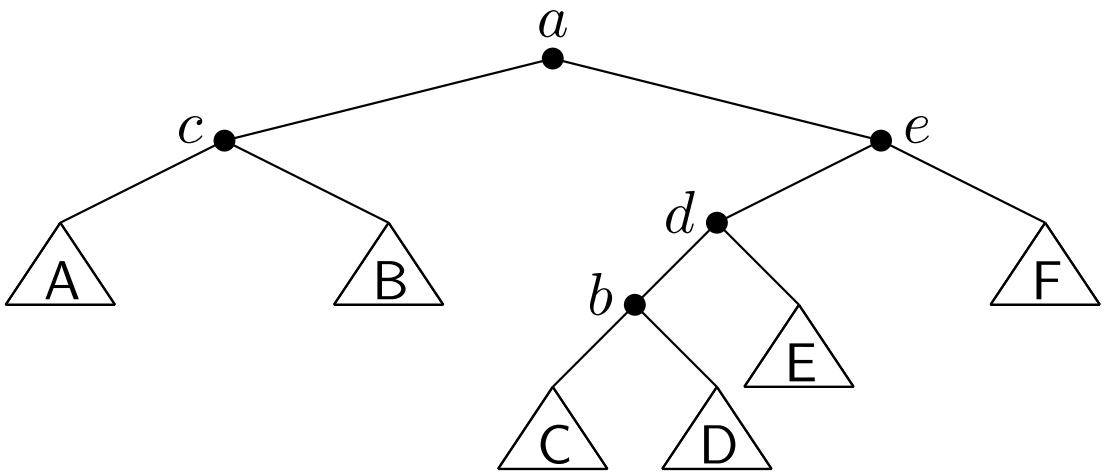
Beispiel 55



Beispiel 55





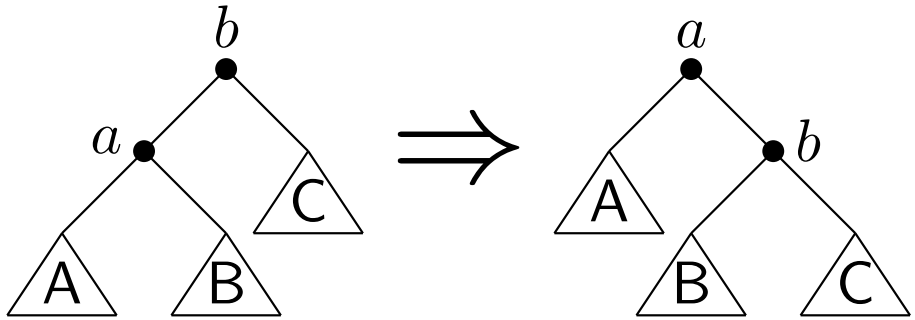


6.4.1 Die Splaying-Operation

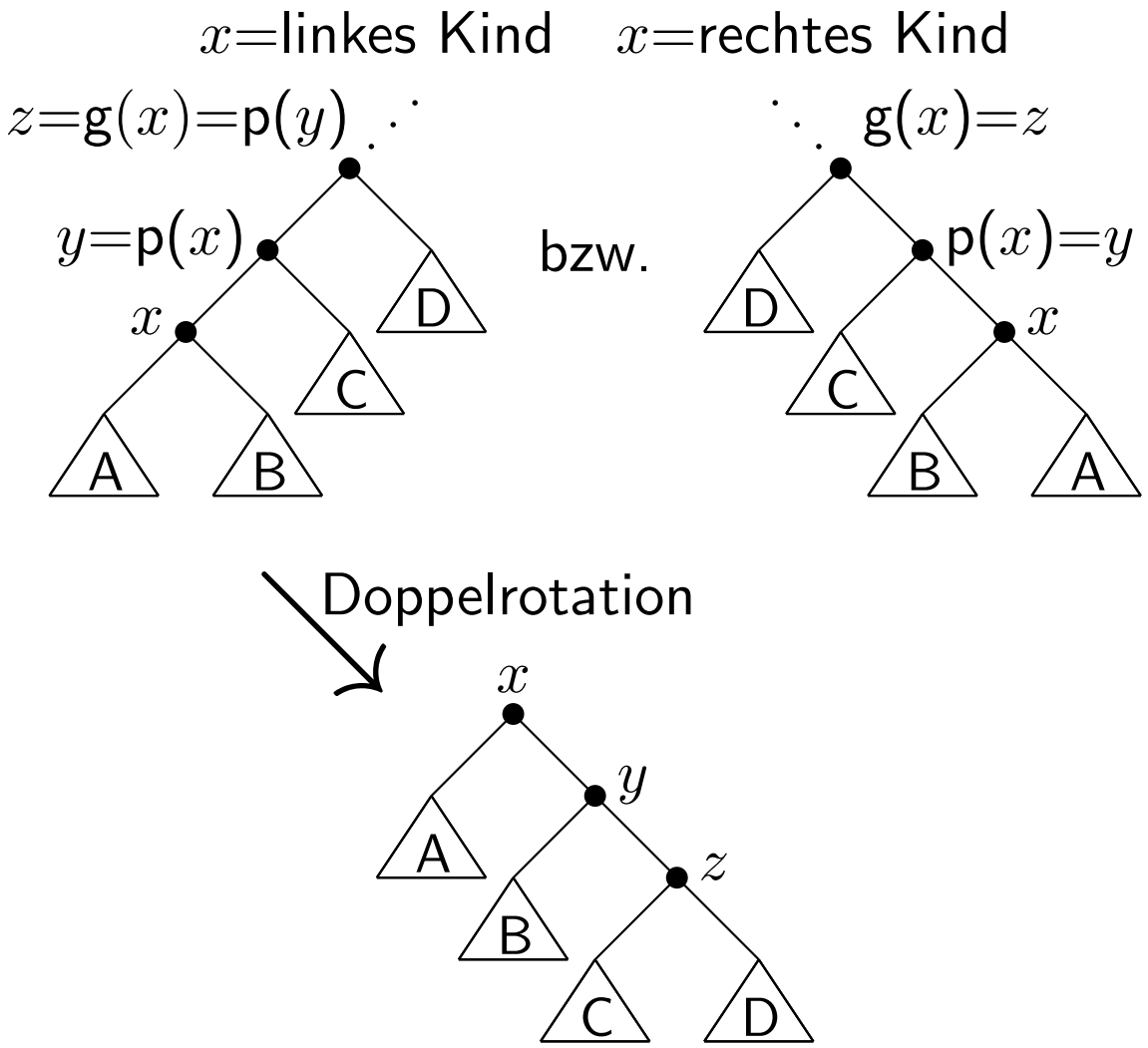
Ein Knoten, auf den zugegriffen wird (Splay(x, T)), wird durch eine Folge von einfachen und doppelten Rotationen an die Wurzel bewegt.

Wir unterscheiden die folgenden Fälle:

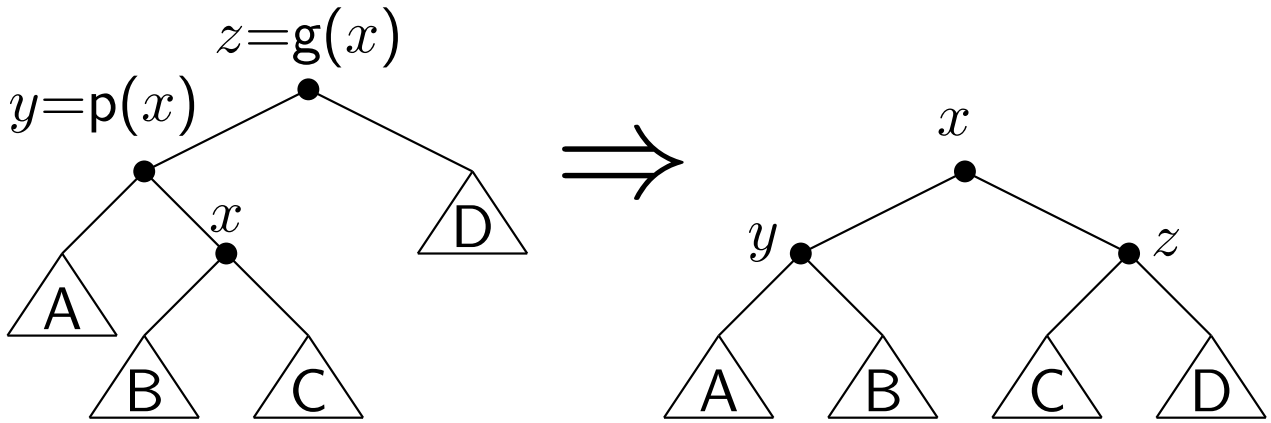
1. (zig): x ist Kind der Wurzel von T: einfache Rotation



2. (zig-zig): x hat Großvater g(x) und Vater p(x); x und p(x) sind jeweils linke (bzw. rechte) Kinder ihres Vaters.



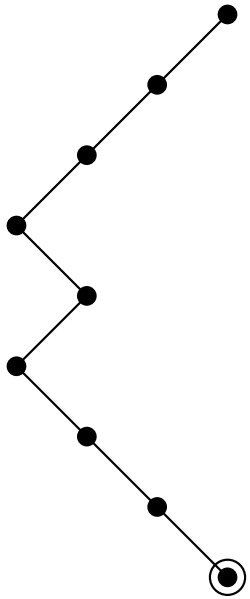
3. (zig-zag): x hat Großvater g(x) und Vater p(x), x ist linkes (rechtes) Kind von p(x), p(x) ist rechtes (linkes) Kind von g(x).



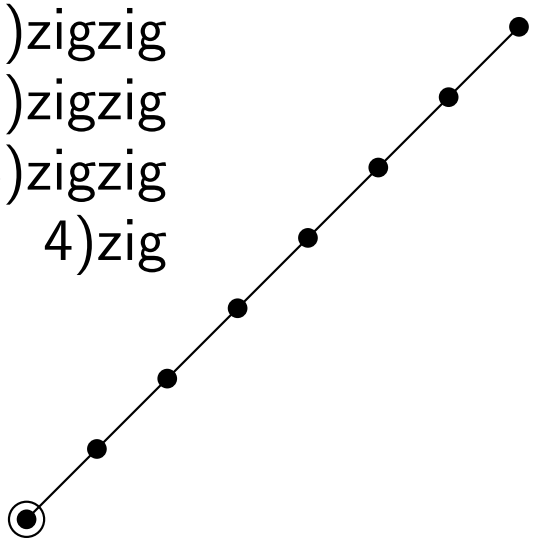
Beispiel 56

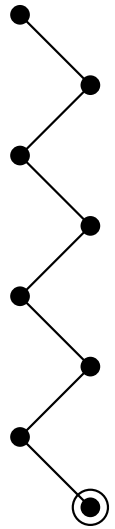
Führe die Splaying-Operation jeweils mit dem eingekreisten Element durch:

1)zigzig
2)zigzag
3)zigzag
4)zigzig



1)zigzig
2)zigzig
3)zigzig
4)zig





1)zigzag
2)zigzag
3)zigzag
4)zig

6.4.2 Amortisierte Kostenanalyse der Splay-Operation

Jeder Knoten habe ein Gewicht $w(x) > 0$. Das Gewicht $tw(x)$ des Unterbaums mit Wurzel x ist die Summe der Gewichte aller Knoten im Unterbaum. Setze

$$\text{Rang } r(x) = \log(tw(x))$$

$$\text{Potenzial eines Baumes } T = \sum_{x \in T} r(x)$$

Lemma 57

Sei T ein Splay-Tree mit Wurzel u, x ein Knoten in T. Die amortisierten Kosten für Splay(x, T) sind

$$\leq 1 + 3(r(u) - r(x)) = \mathcal{O}\left(\log \frac{tw(u)}{tw(x)}\right) .$$

Beweis:

Induktion über die Folge von (Doppel)Rotationen:

Berechne r und r', tw und tw', die Rang- bzw. Gewichtsfunktion vor und nach einem Rotationsschritt. Wir zeigen, dass die amortisierten Kosten im

$$\text{Fall 1 (zig)} \leq 1 + 3(r'(x) - r(x))$$

und in den

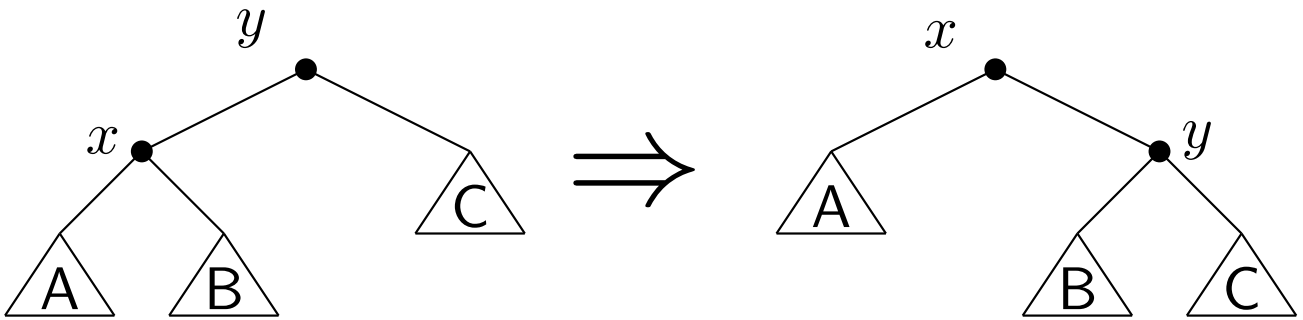
$$\text{Fällen 2 und 3 (zig-zig bzw. zig-zag)} \leq 3(r'(x) - r(x))$$

sind.

y sei der Vater von x, z der Großvater (falls er existiert).

Beweis (Forts.):

1. Fall:

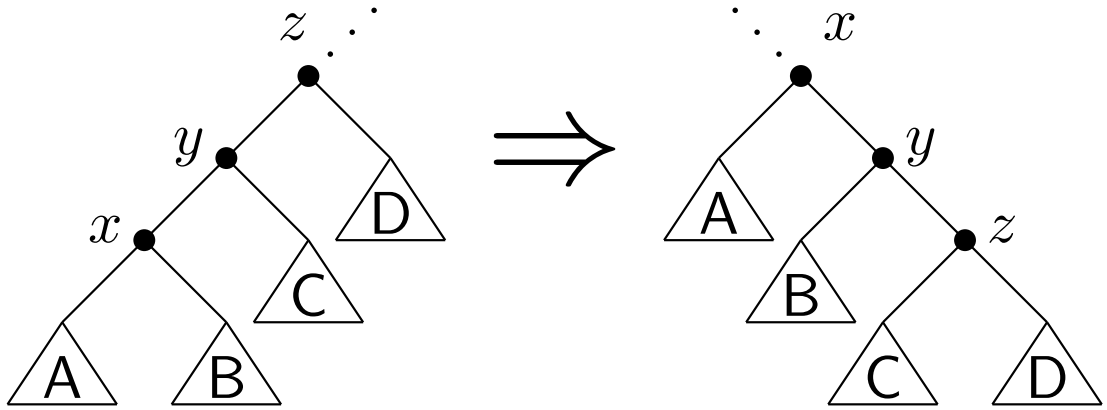


Amortisierte Kosten:

$$
\begin{aligned}
&\leq 1 + r'(x) + r'(y) - r(x) - r(y) \\
&\leq 1 + r'(x) - r(x), && \text{da } r'(y) \leq r(y) \\
&\leq 1 + 3(r'(x) - r(x)), && \text{da } r'(x) \geq r(x)
\end{aligned}
$$

Beweis (Forts.):

2. Fall:



Amortisierte Kosten:

$$
\begin{aligned}
&\leq 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
&= 2 + r'(y) + r'(z) - r(x) - r(y), \quad \text{da } r'(x) = r(z) \\
&\leq 2 + r'(x) + r'(z) - 2r(x), \text{ da } r'(x) \geq r'(y) \text{ und } r(y) \geq r(x)
\end{aligned}
$$

Beweis (Forts.):

Es gilt, dass

$$2 + r'(x) + r'(z) - 2r(x) \leq 3(r'(x) - r(x)) \,,$$

d.h.

$$2r'(x) - r(x) - r'(z) \geq 2 \,.$$

Betrachte dazu die Funktion

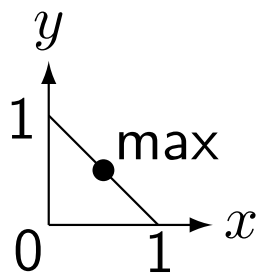
$$f(x, y) = \log x + \log y$$

in dem Bereich

$$x, y > 0, \;\; x + y \leq 1 \,.$$

Beweis (Forts.):

Behauptung: $f(x,y)$ nimmt sein eindeutiges Maximum im Punkt $\left(\frac{1}{2},\frac{1}{2}\right)$ an.



Beweis der Behauptung: Da die $\log$-Funktion streng monoton wachsend ist, kann sich das Maximum der Funktion $f(x,y) = \log x + \log y$ nur auf dem Geradensegment $x + y = 1$, $x, y > 0$ befinden. Dadurch erhalten wir ein neues Maximierungsproblem für die Funktion $g(x) = \log(x) + \log(1-x)$ auf diesem Geradensegment. Da $g(x)$ an den Rändern gegen $-\infty$ strebt, muss es sich um ein lokales Maximum handeln.

Beweis (Forts.):

Die einzige Nullstelle der Ableitung

$$g'(x) = \frac{1}{\ln a}\left(\frac{1}{x} - \frac{1}{1-x}\right),$$

wenn $\log = \log_a$, ist $x = 1/2$ (unabhängig von a).

Weiter ist

$$g''(x) = -\frac{1}{\ln a}\left(\frac{1}{x^2} + \frac{1}{(1-x)^2}\right).$$

Da $g''(0.5) < 0$ ist, nimmt $g(x)$ sein globales Maximum in $x = 0.5$ an. Insgesamt folgt, dass die Funktion $f(x, y) = \log x + \log y$ ihr globales Maximum im Bereich $x, y > 0$, $x + y \leq 1$ an der Stelle $(0.5, 0.5)$ annimmt.

Damit ist die obige Behauptung gezeigt. Wir fahren mit dem Beweis der Abschätzung im Lemma fort.

Beweis (Forts.):

Damit gilt im 2. Fall:

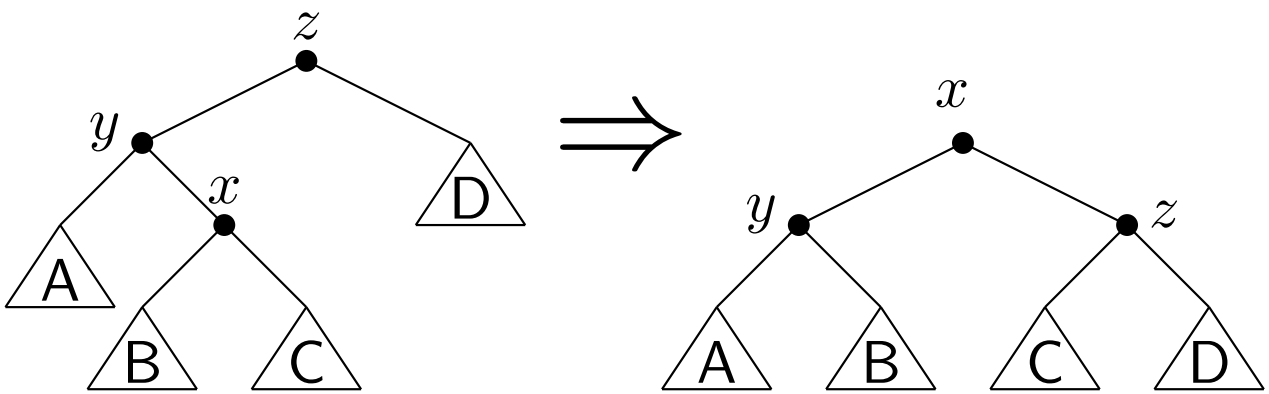
$$r(x) + r'(z) - 2r'(x) = \log\left(\frac{tw(x)}{tw'(x)}\right) + \log\left(\frac{tw'(z)}{tw'(x)}\right) \leq -2\,,$$

da

$$tw(x) + tw'(z) \leq tw'(x)\,.$$

Beweis (Forts.):

3. Fall:



Amortisierte Kosten:

$$
\begin{aligned}
&\leq 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
&\leq 2 + r'(y) + r'(z) - 2r(x), && \text{da } r'(x) = r(z) \text{ und } r(x) \leq r(y) \\
&\leq 2(r'(x) - r(x)), && \text{da } 2r'(x) - r'(y) - r'(z) \geq 2\,.
\end{aligned}
$$

(Letzteres folgt aus der Behauptung über $f(x, y)$ wie im 2. Fall.)

Beweis (Forts.):

Die Gesamtbehauptung des Lemmas folgt dann durch Aufaddieren der amortisierten Kosten für die einzelnen Schritte (Teleskop-Summe). □

Sei T ein Splay-Tree mit n Knoten $x_1, \ldots, x_n$. Falls sich die Gewichte der Knoten nicht ändern, ist die Verringerung des Potenzials durch eine (beliebige) Folge von Splay-Operationen beschränkt durch

$$\sum_{i=1}^{n}(\log W - \log w_i) = \sum_{i=1}^{n} \log \frac{W}{w_i} \,,$$

wobei

$$W := \sum_{i=1}^{n} w_i \,,$$

$$w_i = \text{Gewicht von } x_i \,,$$

da das Gewicht des Unterbaums mit Wurzel x_i immer mindestens w_i und höchstens W ist.

Satz 58

Die gesamten Kosten für die m Zugriffe im Splay-Tree sind

$$\mathcal{O}((m+n)\log n + m)\,.$$

Beweis:

Wähle $w_i = \frac{1}{n}$ für alle Knoten. Dann sind die amortisierten Kosten für einen Zugriff $\leq 1 + 3\log n$, da $W = \sum_{i=1}^{n} w_i = 1$.

Die Verringerung des Potenzials ist

$\leq \sum_{i=1}^{n} \log \frac{W}{w_i} = \sum_{i=1}^{n} \log n = n \log n.$

Damit sind die reellen Kosten $\leq m(1 + 3\log n) + n\log n.$ □

Satz 59

Sei $q(i)$ die Anzahl der Zugriffe auf das Element x_i (in einer Folge von m Zugriffen). Falls auf jedes Element zugegriffen wird (also $q(i) \geq 1$ für alle i), dann sind die (reellen) Gesamtkosten für die Zugriffe

$$\mathcal{O}\left(m + \sum_{i=1}^{n} q(i) \cdot \log\left(\frac{m}{q(i)}\right)\right).$$

Beweis:

Setze das Gewicht des i-ten Knotens gleich $\frac{q(i)}{m}$.

$$\Rightarrow \mathsf{W} = \sum_{i=1}^{n} \frac{q(i)}{m} = 1\,.$$

Der Rest folgt wie zuvor. □

Satz 60

Betrachte eine Folge von Zugriffsoperationen auf eine n-elementige Menge. Sei t die dafür nötige Anzahl von Vergleichen in einem optimalen statischen binären Suchbaum. Dann sind die Kosten in einem (anfangs beliebigen) Splay-Tree für die Operationenfolge $\mathcal{O}(t + n^2)$.

Beweis:

Sei U die Menge der Schlüssel, d die Tiefe eines (fest gewählten) optimalen statischen binären Suchbaumes. Für $x \in U$ sei weiter $d(x)$ die Tiefe von x in diesem Suchbaum. Setze

$$tw(x) := 3^{d-d(x)}\,.$$

Sei T ein beliebiger Splay-Tree für U, $|U| =: n$.

$$bal(T) \leq \sum_{x\in U} r(x) = \sum_{x\in U} \log(3^{d-d(x)}) = \sum_{x\in U} (\log 3)(d-d(x)) =$$

$$= (\log 3) \sum_{x\in U} (d-d(x)) = \mathcal{O}(n^2);$$

$$\sum_{x\in U} tw(x) = \sum_{x\in U} 3^{d-d(x)} \leq \sum_{i=0}^{d} 2^i 3^{d-i} \leq 3^d \frac{1}{1-\frac{2}{3}} = 3^{d+1}$$

$$\Rightarrow \log \frac{tw(T)}{tw(x)} \leq \log \frac{3^{d+1}}{3^{d-d(x)}} = \log 3^{d(x)+1}.$$

Beweis (Forts.):

Damit ergibt sich für die amortisierten Kosten von Splay(x, T)

$$\mathcal{O}(\log \frac{tw(T)}{tw(x)}) = \mathcal{O}(d(x) + 1)\,.$$

Die amortisierten Kosten sind damit

$\leq c \cdot$ Zugriffskosten (#Vergleiche) im optimalen Suchbaum

(wo sie $d(x) + 1$ sind).

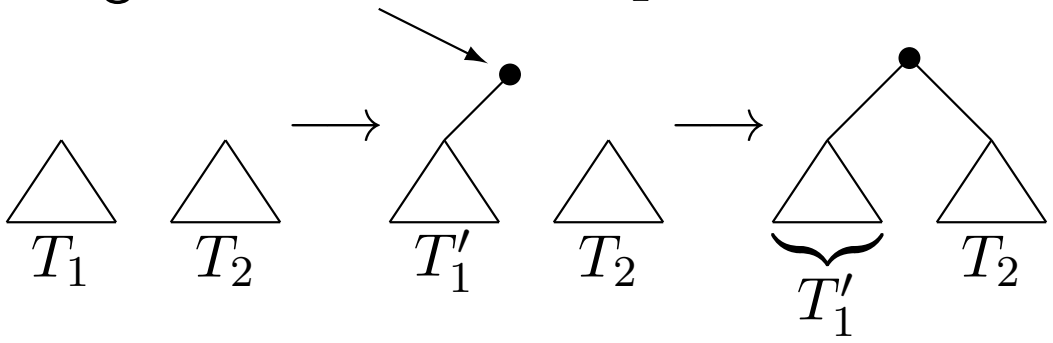
Die gesamten amortisierten Kosten für die Zugriffsfolge sind daher $\leq c \cdot t$.
Die reellen Kosten ergeben sich zu $\leq$ amort. Kosten + Verringerung des Potenzials, also $\mathcal{O}(t + n^2)$. □

6.4.3 Wörterbuchoperationen in Splay-Trees

Alle folgenden Operationen werden mit Hilfe von Splay implementiert.

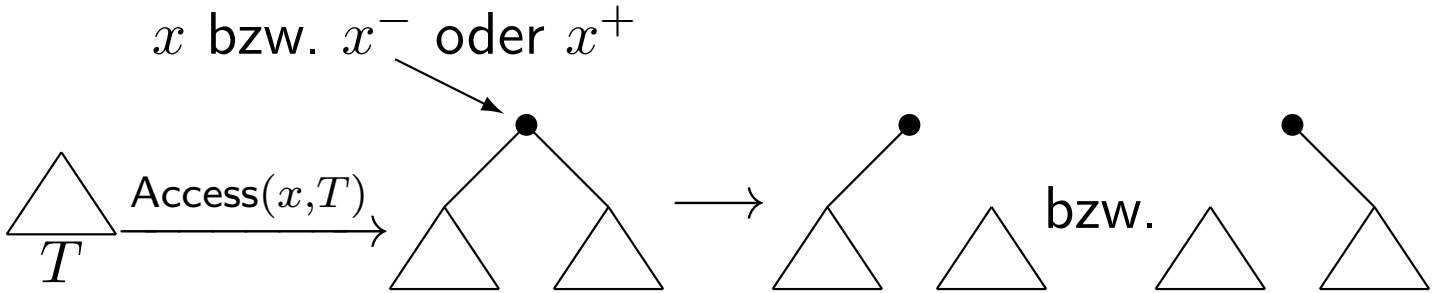
- *Access*(x, T): $\surd$ (siehe oben)
- *Join*(T_1, T_2):

  größtes Element in T_1

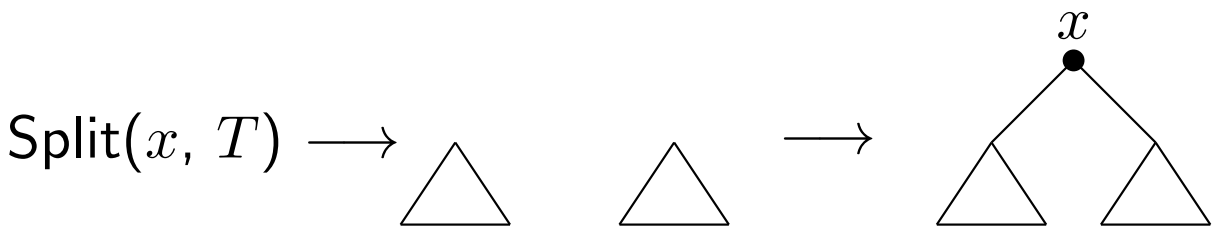


  Beachte: Falls $x \in T_1$, $y \in T_2$, dann $x < y$.

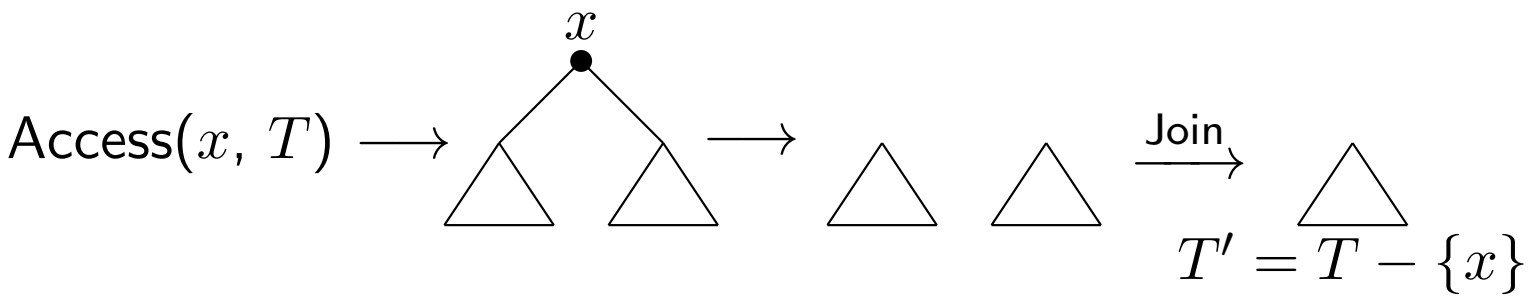
- *Split*(x, T):



- *Insert*(x, T):



- *Delete*(x, T):



Sei $x \in U$, T ein Splay-Tree. Dann bezeichnen wir mit x^- (bzw. x^+) den Vorgänger (bzw. den Nachfolger) von x in U (falls diese existieren). U ist ja total geordnet. Falls x^- bzw. x^+ undefiniert sind, so setzen wir $w(x^-) = \infty$ bzw. $w(x^+) = \infty$.

Weiterhin sei W das Gesamtgewicht aller an einer Wörterbuch-Operation beteiligten Knoten.

Satz 61

Für die amortisierten Kosten der Wörterbuch-Operationen in Splay-Trees gelten die folgenden oberen Schranken ($T,\ T_1,\ T_2 \neq \emptyset$):

$$\textit{Access}(x,T): \begin{cases} 3\log\left(\frac{W}{w(x)}\right) + \mathcal{O}(1), & \textit{falls } x \in T \\ 3\log\left(\frac{W}{\min\{w(x^-),\ w(x^+)\}}\right) + \mathcal{O}(1), & \textit{falls } x \notin T \end{cases}$$

$$\textit{Split}(x,T): \begin{cases} 3\log\left(\frac{W}{w(x)}\right) + \mathcal{O}(1), & \textit{falls } x \in T \\ 3\log\left(\frac{W}{\min\{w(x^-),\ w(x^+)\}}\right) + \mathcal{O}(1), & \textit{falls } x \notin T \end{cases}$$

Satz 61

Für die amortisierten Kosten der Wörterbuch-Operationen in Splay-Trees gelten die folgenden oberen Schranken ($T,\ T_1,\ T_2 \neq \emptyset$):

$$\textit{Join}(T_1, T_2) : 3\log\left(\frac{W}{w(i)}\right) + \mathcal{O}(1),\ x \text{ maximal in } T_1$$

$$\underset{x \notin T}{\textit{Insert}(x, T)} : 3\log\left(\frac{W - w(x)}{\min\{w(x^-), w(x^+)\}}\right) + \log\left(\frac{W}{w(x)}\right) + \mathcal{O}(1)$$

$$\underset{i \in T}{\textit{Delete}(x, T)} : 3\log\left(\frac{W}{w(x)}\right) + 3\log\left(\frac{W - w(x)}{w(x^-)}\right) + \mathcal{O}(1),$$

falls x nicht minimal in T

Literatur zu Splay-Trees:

Daniel D. Sleator, Robert E. Tarjan:
Self-adjusting binary search trees
Journal of the ACM **32**(3), pp. 652–686 (1985)

6.5 Weitere Arten wichtiger Datenstrukturen

Dynamische Datenstrukturen:

Samuel W. Bent:
Dynamic weighted data structures
TR STAN-CS-82-916, Department of Computer Science, Stanford University, 1982

Persistente Datenstrukturen:

J.R. Driscoll, N. Sarnak, D.D. Sleator, R.E. Tarjan:
Making data structures persistent
Proceedings of the 18th Annual ACM Symposium on Theory of Computing (STOC), pp. 109–121 (1986)

Probabilistische Datenstrukturen:

William Pugh:
Skip Lists: A Probabilistic Alternative to Balanced Trees
Commun. ACM, 33(6):668-676, 1990

7. Radix-basierte Priority Queues

7.1 Buckets

Eine relativ einfache Möglichkeit, Vorrangwarteschlangen zu implementieren, stellen Buckets dar. Diese Implementierung beinhaltet einen Array von Buckets, wobei der i-te Bucket alle Elemente x mit dem Schlüssel $k(x) = i$ enthält. Sobald der Schlüssel eines Elements sich ändert, wird das Element vom alten Bucket entfernt und entsprechend dem neuen Schlüssel in dem neuen Bucket eingefügt.

Dazu müssen folgende Annahmen erfüllt sein:

- Schlüssel sind ganzzahlig
- Zu jedem Zeitpunkt gilt für die zu speichernden Elemente:

$$\boxed{\text{größter Schlüssel - kleinster Schlüssel} \leq C}$$

Diese Bedingungen sind zum Beispiel beim Algorithmus von Dijkstra erfüllt, falls die Kantengewichte natürliche Zahlen $\leq C$ sind.

7.1.1 1-Level-Buckets

1-Level-Buckets bestehen aus:

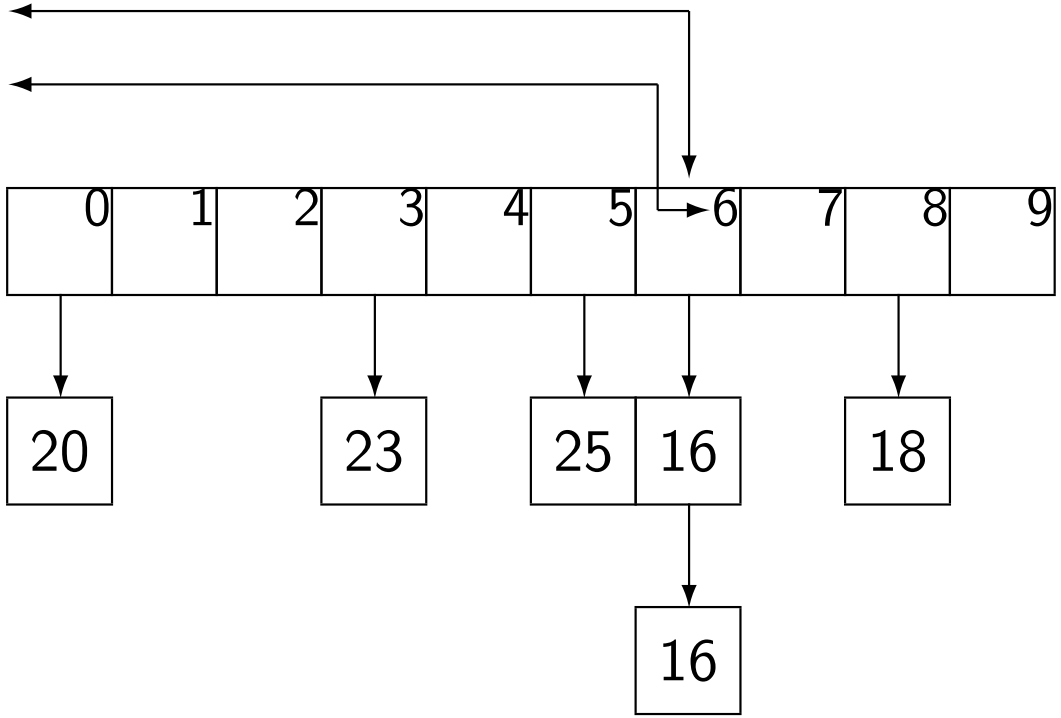
- einem Array $b[0..C]$ zur Aufnahme der Buckets. Jedes b_i enthält einen Pointer auf die Liste der Elemente im Bucket i;
- einer Zahl $minvalue$, die gleich dem kleinsten gespeicherten Schlüssel ist;
- einer Zahl $0 \leq minpos \leq C$, die den Index des Buckets mit dem kleinsten Schlüssel enthält, und
- der Zahl n der gespeicherten Elemente.

$C = 9$

$minvalue = 16$

$minpos = 6$

$n = 6$



Wie bei jeder Vorrangwarteschlange müssen drei Operationen unterstützt werden:

- *Insert*(x): fügt das Element x in die Vorrangswarteschlange ein. Falls der Schlüssel des neuen Elements kleiner als der $minvalue$ ist, werden $minpos$ und $minvalue$ aktualisiert.
- *ExtractMin*: liefert und löscht eines der kleinsten Elemente der Vorrangswarteschlange (falls das Element das kleinste ist, müssen $minpos$ und $minvalue$ noch aktualisiert werden).
- *DecreaseKey*(x, k): verringert Schlüssel des Elements x auf den Wert k (falls nötig, werden $minpos$ und $minvalue$ aktualisiert).

Dazu kommt noch eine Initialisierung *Initialize*.

Für 1-Level-Buckets sehen die Operationen wie folgt aus:

1. *Initialize*(x):

   $n := 0$; $minvalue := \infty$
   reserviere Platz für b (C sei bekannt)
   initialisiere b

2. *Insert*:

   füge x in $b[k(x) \bmod (C+1)]$ ein
   $n := n + 1$
   if $k(x) < minvalue$ **then**
   co x ist jetzt das Element mit dem kleinsten Schlüssel **oc**
   $minpos := k(x) \bmod (C+1)$
   $minvalue := k(x)$
   fi

Für 1-Level-Buckets sehen die Operationen wie folgt aus:

3. *ExtractMin*: Entferne ein beliebiges Element aus $b[minpos]$

co wir nehmen $n > 0$ an **oc**
extrahiere beliebiges Element in $b[minpos]$
$n := n - 1$
if $n > 0$ **then**
    co suche neues Element mit kleinstem Schlüssel **oc**
    while $b[minpos]$ ist leer **do**
        $minpos := (minpos + 1) \bmod (C + 1)$
    od
    $minvalue :=$ Schlüssel eines Elements in $b[minpos]$
else
    $minvalue := \infty$
fi

Für 1-Level-Buckets sehen die Operationen wie folgt aus:

4. *DecreaseKey*(x, k): verringert Schlüssel des Elements x auf den Wert k

entferne $k(x)$ aus Bucket $k(x) \bmod (C+1)$
$k(x) := k$
füge x in $b[k(x) \bmod (C+1)]$ ein
if $k(x) < minvalue$ **then**
    $minpos := k(x) \bmod (C+1)$
    $minvalue := k(x)$
fi

Bei geeigneter Implementierung der Buckets, z.B. als doppelt verkettete Listen, gilt:

Satz 62

Die worst-case (reellen) Kosten sind $\mathcal{O}(1)$ für Insert und DecreaseKey, und sie sind $\mathcal{O}(C)$ für Initialize und ExtractMin.

Beweis:

Wird x am Anfang der Liste eingefügt, so müssen bei *Insert* nur einige Zeiger umgehängt sowie n, $minpos$ und $minvalue$ angepasst werden, was wieder nur ein paar Zeigeroperationen sind. Die Aktualisierung von n, $minpos$ und $minvalue$ benötigt auch nur konstante Zeit. Für das Finden des nächstkleinsten Elements müssen aber möglicherweise alle weiteren Buckets betrachtet werden, im schlimmsten Falle C. Da bei *DecreaseKey* das Element x direkt übergeben wird, sind neben dem Einfügen nur wenige Zeigeroperationen und je eine Zuweisung an n und $k(x)$ nötig. □

7.1.2 2-Level-Buckets

Bei einem großen Wertebereich der zu speichernden Schlüssel, d.h. bei großem C, und einer geringen Anzahl tatsächlich abgelegter Datenelemente sind 1-Level-Buckets in zweifacher Hinsicht ungünstig:

- Das Feld b belegt statisch Speicherplatz der Größe $\Theta(C)$, obwohl nur ein kleiner Teil davon wirklich gebraucht wird.
- Der Zeitbedarf für ein *ExtractMin* nähert sich der worst-case-Komplexität $\Theta(C)$, da der nächste nicht-leere Bucket ziemlich weit entfernt sein kann.

2-Level-Buckets versuchen diesen Problemen mit folgender Idee abzuhelfen:

Es gibt einen Array $btop$, bestehend aus $B+1$ top-Buckets, mit $B := \lceil \sqrt{C+1} \rceil$.
Zu jedem Bucket i in $btop$ gibt es noch einen weiteren Array $bbot_i$, der ebenfalls aus B bottom-Buckets besteht.
$bbot_i$ nimmt Elemente auf, deren Schlüssel im Interval $[iB, (i+1)B-1]$ liegen. Um ein Element in einen Bucket einzufügen, wird zuerst der passende Bucket in $btop$ gesucht. Dann wird in dem dazugehörigen $bbot_i$ das Element (wie bei 1-Level-Buckets) eingefügt.

Um sowohl Platz als auch Zeit zu sparen, kann man durch leichte Modifizierung mit einem einzigen Array von Bottom-Buckets auskommen:

Wir verwenden zwei Arrays (Top-Buckets und Bottom-Buckets). Dabei enthält der Array der Top-Buckets in

$$B = \left\lceil \sqrt{C+1} \right\rceil$$

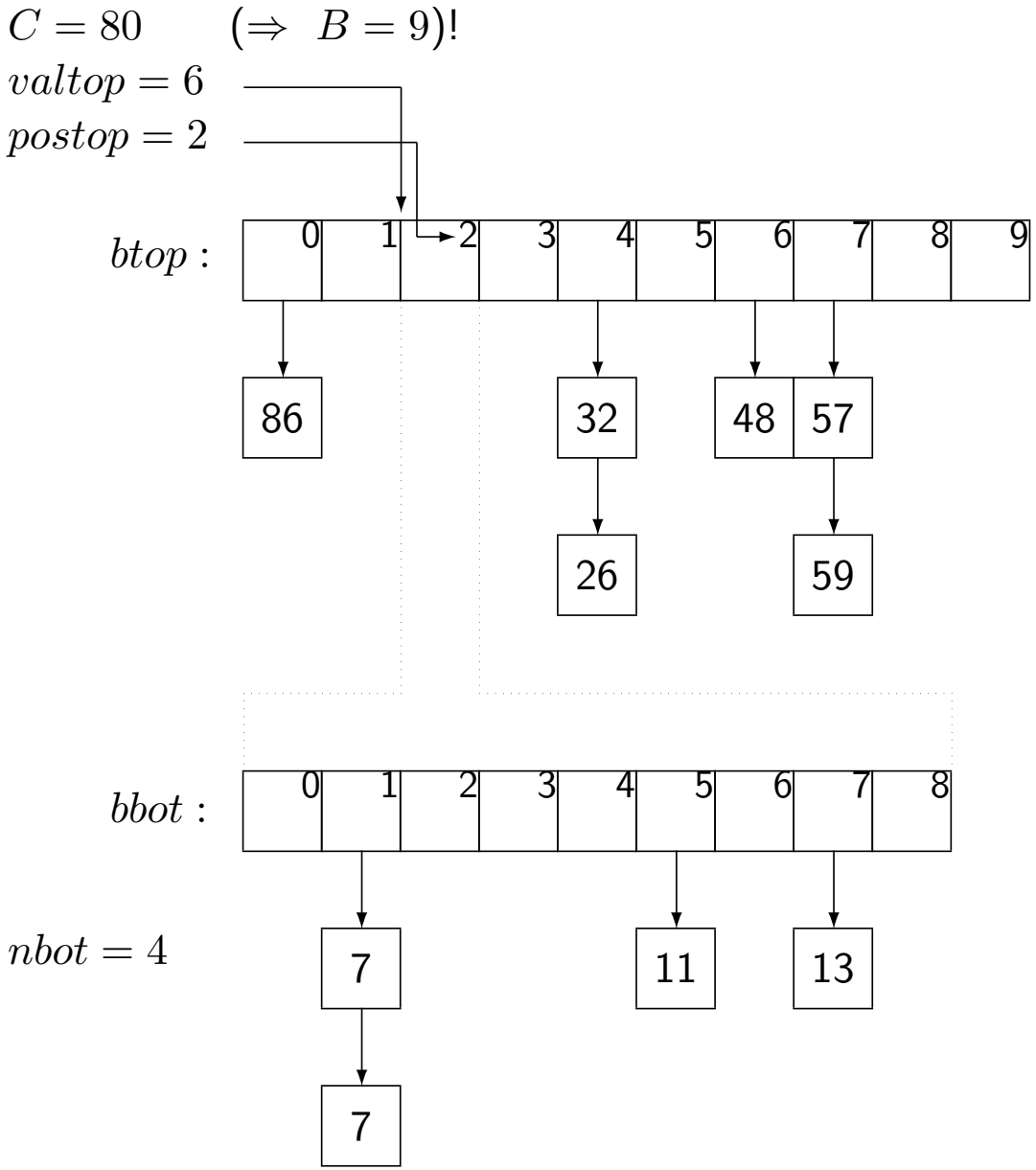
Buckets die meisten Elemente in grob vorsortierter Form (ein Top-Bucket wird nicht benützt!), nur die Elemente mit den kleinsten Schlüsseln werden im Array für die Bottom-Buckets, der die Länge

$$B = \left\lceil \sqrt{C+1} \right\rceil$$

hat, dann endgültig sortiert vorgehalten.

2-Level-Buckets können mit folgenden Elementen aufgebaut werden:

- Der Array $btop[0..B]$ nimmt in jedem Bucket Elemente mit Schlüsseln aus einem Teilintervall der Länge B auf; die Schlüssel stammen aus allen Teilintervallen außer dem niedrigsten.
- Der Array $bbot[0..B-1]$ nimmt in jedem Bucket Elemente mit genau einem Schlüssel auf; die Schlüssel stammen nur aus dem niedrigsten Teilintervall.
- $valtop$ ist die linke Intervallgrenze des niedrigsten Teilintervalls.
- $postop$ enthält den Index des Buckets in $btop$ für das niedrigste Teilintervall (das aber in $bbot$ gespeichert wird).
- $nbot$ ist die Anzahl der Elemente in $bbot$.
- n ist die Anzahl der Elemente in $bbot$ und $btop$.


$C = 80 \quad (\Rightarrow \ B = 9)!$
$valtop = 6$
$postop = 2$
$btop:$
0 1 2 3 4 5 6 7 8 9
86
32
48
57
26
59
$bbot:$
0 1 2 3 4 5 6 7 8
$nbot = 4$
7
11
13
7

Dabei gilt:

- $btop[(postop + i) \bmod B]$ enthält alle Elemente x mit

$$valtop + iB \leq k(x) < valtop + (i+1)B, \text{ wobei } 1 \leq i \leq B\,;$$

- $bbot[i]$ enthält alle Elemente x mit

$$k(x) = valtop + i, \text{ wobei } 0 \leq i \leq B - 1\,.$$

Dazu ist allerdings folgende Annahme zu machen:
Hat ein *ExtractMin* ein Element mit Schlüssel k zurückgeliefert, werden bis zum nächsten Aufruf von *ExtractMin* nur Elemente mit Schlüsseln $\geq valtop$ eingefügt. Dies stellt sicher, dass ein Element aus $bbot$ nie nach $btop$ wandert, und ist z.B. in Dijkstra's Algorithmus für kürzeste Pfade erfüllt.

Lemma 63

Zu jedem Zeitpunkt haben alle Schlüssel Platz in der Datenstruktur.

Beweis:

Am meisten Platz wird benötigt, wenn der kleinste Schlüssel ganz rechts in $bbot$ gespeichert oder dort entfernt worden ist. Er hat dann den Wert $valtop + B - 1$. Der größte Schlüssel, der nun vorkommen kann und also Platz finden muss, ist $valtop + B - 1 + C$.

Beweis (Forts.):

größtmöglicher Schlüssel in $btop$

$$
\begin{aligned}
&= valtop + (B+1) \cdot B - 1 \\
&= valtop + B + B \cdot B - 1 \\
&= valtop + B + \lceil\sqrt{C+1}\rceil\lceil\sqrt{C+1}\rceil - 1 \\
&\geq valtop + B + C + 1 - 1 \\
&> valtop + B - 1 + C \\
&= \text{größtmöglicher erlaubter Schlüssel}
\end{aligned}
$$

□

Annahme: Vor dem ersten *ExtractMin* werden nur Elemente x mit $0 \leq k(x) \leq C$ eingefügt.

i) *Initialize*:

$valtop := postop := nbot := n := 0$
initialisiere $btop$, $bbot$

ii) *Insert*(x):

überprüfe Invarianten;

$i := \left(\left\lfloor \frac{k(x) - valtop}{B} \right\rfloor + postop \right) \bmod (B+1)$

if $i = postop$ **then**

füge x in $bbot[k(x) - valtop]$ ein

$nbot := nbot + 1$

else

füge x in $btop[i]$ ein

fi

$n := n + 1$

iii) *ExtractMin*

co suche kleinstes Element in $bbot$ **oc**
$j := 0$; **while** $bbot[j] = \emptyset$ **do** $j := j + 1$ **od**
entferne ein (beliebiges) Element aus $bbot[j]$ und gib es zurück
$nbot := nbot - 1$; $n := n - 1$
if $n = 0$ **then return fi**
if $nbot = 0$ **then**
    while $btop[postop] = \emptyset$ **do**
        $postop := (postop + 1) \bmod (B + 1)$
        $valtop := valtop + B$
    od
    while $btop[postop] \neq \emptyset$ **do**
        entferne beliebiges Element x aus $btop[postop]$
        füge x im $bbot[k(x) - valtop]$ ein
        $nbot := nbot + 1$
    od
fi

iv) *DecreaseKey*(x, k)

entferne x aus seinem momentanen Bucket
falls nötig, aktualisiere $nbot$
$k(x) := k$
überprüfe Invarianten;
$i := \left(\left\lfloor \frac{k(x) - valtop}{B} \right\rfloor + postop \right) \bmod (B+1)$
if $i = postop$ **then**
    füge x in $bbot[k(x) - valtop]$ ein
    $nbot := nbot + 1$
else
    füge x in $btop[i]$ ein
fi

Lemma 64

Die worst-case (reelle) Laufzeit bei 2-Level-Buckets ist für Insert und DecreaseKey $\mathcal{O}(1)$, für ExtractMin $\mathcal{O}(n+\sqrt{C})$ und für Initialize $\mathcal{O}(\sqrt{C})$.

Beweis:

— *Insert*: $\mathcal{O}(1)+1=\mathcal{O}(1)$

— *DecreaseKey*: $\mathcal{O}(1)$

— *ExtractMin*

$$\mathcal{O}\left(\sqrt{C}+\sqrt{C}+\#\text{Elemente}\begin{array}{c} btop \\ \downarrow \\ bbot \end{array}\right)$$

□

Setze Potenzial := Anzahl der Elemente in Buckets in $btop$.

Satz 65

Die amortisierten Kosten für 2-Level-Buckets sind $\mathcal{O}(1)$ bei Insert und DecreaseKey und $\mathcal{O}(\sqrt{C})$ bei ExtractMin.

Beweis:

- *Insert*: s.o.
- *DecreaseKey*: s.o.
- *ExtractMin*:

$$\mathcal{O}\left(\sqrt{C} + \#\text{Elemente}_{\substack{btop \\ \downarrow \\ bbot}}\right) - \#\text{Elemente}_{\substack{btop \\ \downarrow \\ bbot}} = \mathcal{O}(\sqrt{C})$$

□

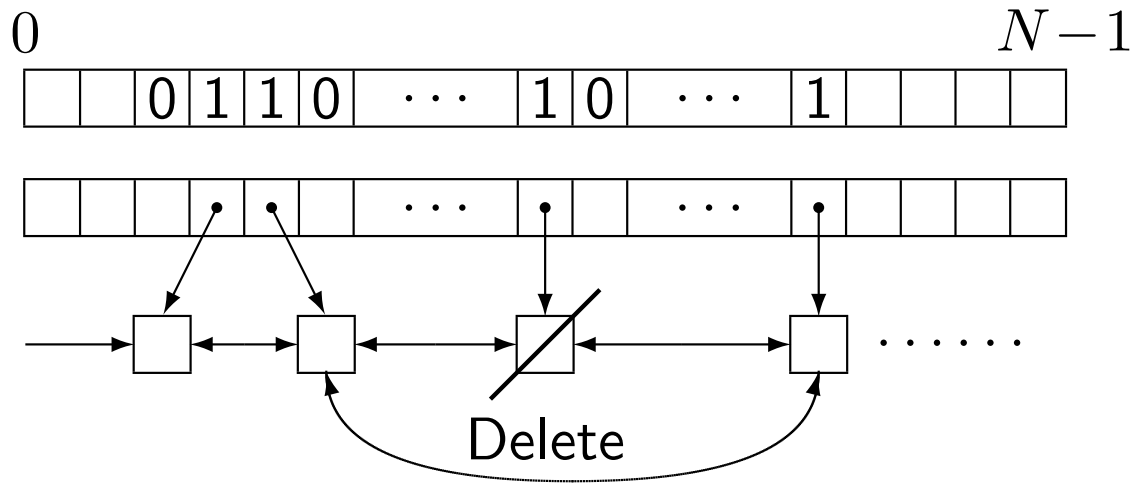
7.1.3 k-Level-Buckets

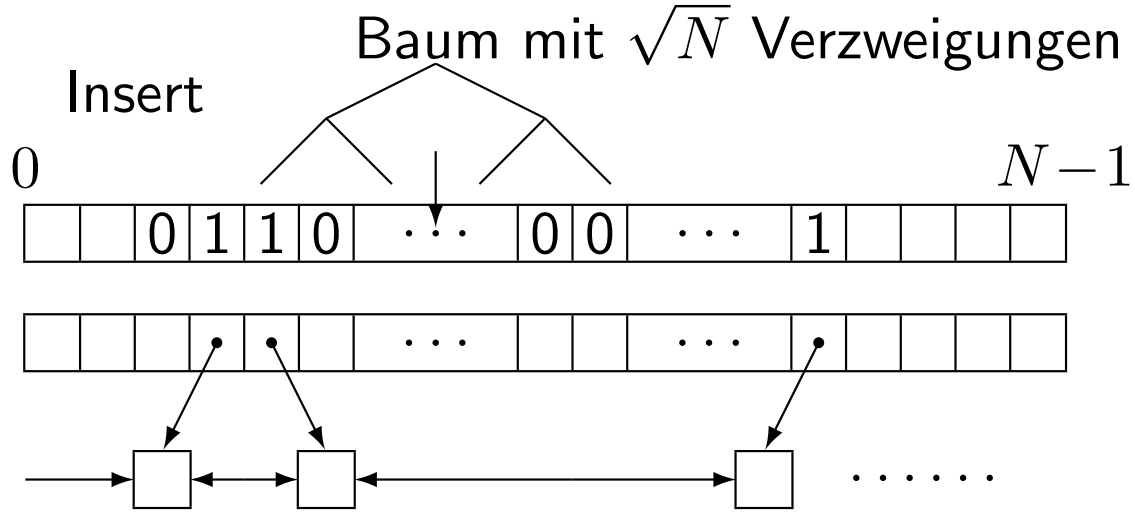
Die Verallgemeinerung von 2-Level-Buckets führt zu k-Level-Buckets. Diese bestehen dann aus k Arrays der Länge $\left\lceil \sqrt[k]{C} \right\rceil$. Dadurch lassen sich die Speicher- und Zeitkomplexität weiter verbessern, der Implementierungsaufwand steigt jedoch stark an.

7.2 van Emde Boas-Priority Queues

Universum U, $|U| = N$, $U \subset \mathbb{N}$.
Hier: $U = \{0, 1, \ldots, N-1\}$.
Wir untersuchen hier eine bessere Priority Queue für den Fall $m \geq \log N$.

Allerdings ist der Platzbedarf meist impraktikabel.


0
$N-1$
0 1 1 0 ··· 1 0 ··· 1
Delete


Baum mit $\sqrt{N}$ Verzweigungen
Insert
0
$N-1$
0 1 1 0 ··· 0 0 ··· 1

Notation

Sei

$$k \in \mathbb{N}, k \geq 2$$

$$k' = \left\lceil \frac{k}{2} \right\rceil$$

$$k'' = \left\lfloor \frac{k}{2} \right\rfloor$$

$$x \in [0..2^k - 1] \qquad (x \text{ hat } \leq k \text{ Bits})$$

$$x' = \left\lfloor \frac{x}{2^{k''}} \right\rfloor \qquad (x' \text{ vordere Hälfte von } x)$$

$$x'' = x \bmod 2^{k''} \qquad (x'' \text{ hintere Hälfte von } x)$$

Sei

$$S = \{x_1, \ldots, x_m\} \subseteq [0..2^k - 1] = U\,.$$

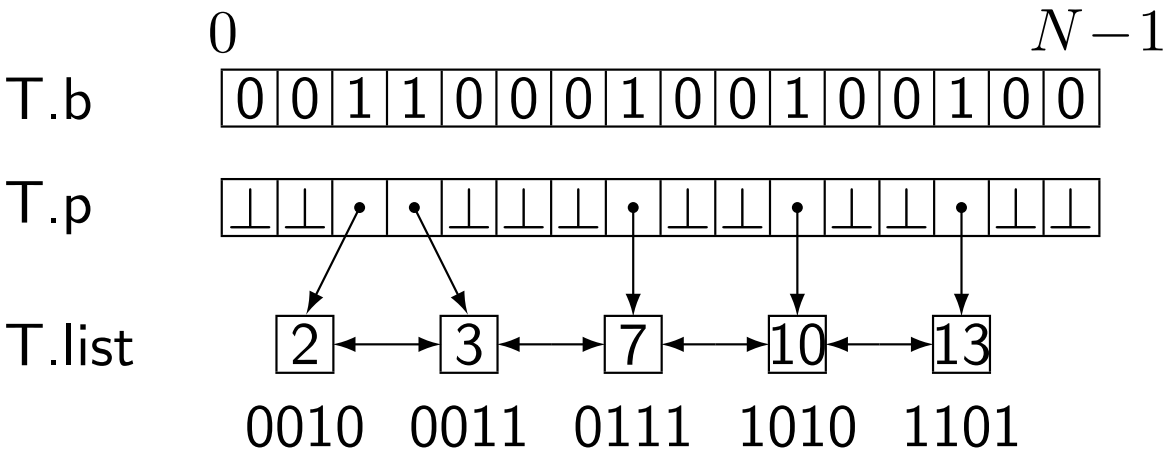
Definition 66

Eine k-Struktur T für S besteht aus:

1. der Zahl T.size $= |\{x_1, \ldots, x_m\}| = |S| = m$;
2. einer doppelt verketteten Liste T.list, die die Elemente von S in aufsteigender Reihenfolge enthält;
3. einem Bitvektor T.b$[0..2^k - 1]$ mit T.b$[i] = \begin{smallmatrix} 0 \\ 1 \end{smallmatrix}$, falls $\begin{smallmatrix} i \notin S \\ i \in S \end{smallmatrix}$ und einem Zeiger (Pointer) Vektor T.p$[0 \ldots 2^k - 1]$. Falls T.b$[i] = 1$, dann zeigt T.p$[i]$ auf i in der Liste aus 2.
4. einer k'-Struktur T.top und einem Feld T.bottom$[0 \ldots 2^{k'} - 1]$ von k''-Strukturen. Falls $m = 1$, dann T.top, T.bottom und die zugehörigen k''-Strukturen leer, T.size $= 1$. T.list $= \{x\}$, der Bitvektor wird nicht benötigt. Falls $m > 1$, dann ist T.top eine k'-Struktur für die durch $\{x'_1, x'_2, \ldots, x'_m\}$ gegebene Menge, und für jedes $y \in [0 \ldots 2^{k'} - 1]$ ist T.bottom$[y]$ eine k''-Struktur für die Menge $\{x''_i;\ 1 \leq i \leq m;\ y = x'_i\}$

Beispiel 67

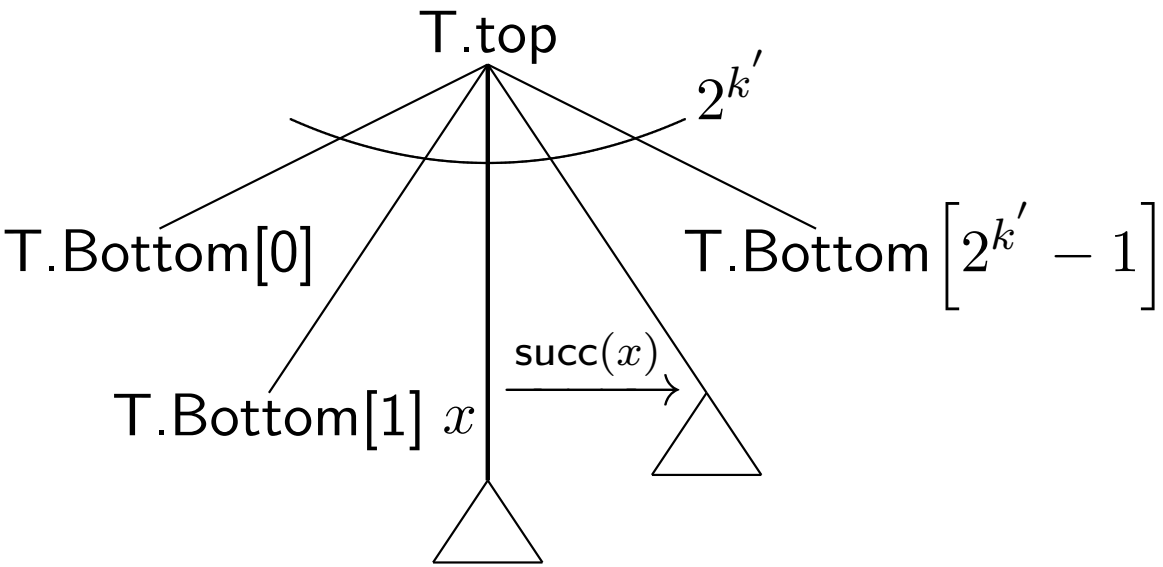
$k = 4$, $S = \{2, 3, 7, 10, 13\}$, T.size $= 5$:



T.top ist 2-Struktur für $\{0, 1, 2, 3\}$
T.bottom$[0]$ ist eine 2-Struktur für $\{2, 3\}$
T.bottom$[1]$ ist eine 2-Struktur für $\{3\}$
T.bottom$[2]$ ist eine 2-Struktur für $\{2\}$
T.bottom$[3]$ ist eine 2-Struktur für $\{1\}$

Operation *Succ(x)* findet $\min\{y \in S;\ y > x\}$ in der k-Struktur T.

if $k = 1$ **or** T.Size ≤ 2 **then**
  naive Suche
elif $x \geq \max$ in T.list **then**
    return *Succ(x)* gibt's nicht
else
  $x' := \left\lfloor \frac{x}{2^{k''}} \right\rfloor$;
  $x'' := x \bmod 2^{k''}$;
  if T.top.b$[x'] = 1$ **and** $x'' < \max\{$T.bottom$[x']\}$ **then**
    return $x' \cdot 2^{k''} +$ *Succ(x'', T.bottom[x'])*
  else
    $z' :=$*Succ(x', T.top)*; **return** $z' \cdot 2^{k''} + \min\{$T.bottom$[z']\}$
  fi
fi


T.top
$2^{k'}$
T.Bottom[0]
T.Bottom $\left[2^{k'}-1\right]$
succ(x)
T.Bottom[1] x

Kosten:

$$\mathsf{T}(k) \leq c + \mathsf{T}(\left\lceil \frac{k}{2} \right\rceil) = \mathcal{O}(\log k)\,.$$

Lemma 68

Die Succ-Operation hat Zeitbedarf $\mathcal{O}(\log k)$.

Beweis:

$\surd$ □

Insert Operation:

- falls x bereits in Priority Queue, dann fertig
- bestimme *Succ*(x, T), füge x davor ein
- bestimme x' und x''
- behandle die entsprechenden Unterstrukturen rekursiv: *Insert*(x',$T.top$), *Insert*(x'',$T.bottom[x']$) (nur ein nicht trivialer rekursiver Aufruf)

Zeitbedarf: naiv $\mathcal{O}(\log^2 k)$, Optimierung: das oberste *Succ* tut alles $\Rightarrow \mathcal{O}(\log k)$.

Delete Operation:
Komplexität von *Delete* in k-Struktur: $\mathcal{O}(\log k)$

Kosten der Initialisierung: $\sim$ Größe der Datenstruktur

Platzbedarf für k-Struktur: Sei $S(k)$ der Platzbedarf für eine k-Struktur. Dann gilt:

$$S(1) = c$$
$$S(k) = c2^k + S(k') + 2^{k'} S(k'') \text{ für } k \geq 2$$

Wir ersetzen zur Vereinfachung:

$$S(k) = c2^k + S(\frac{k}{2}) + 2^{\frac{k}{2}} S(\frac{k}{2})$$

Lemma 69

$$S(k) = \mathcal{O}(2^k \cdot \log k)$$

Beweis:

Zeige: $\mathsf{S}(k) := c'2^k \log k$ funktioniert.

Für $k = 1$ ist das klar.

Beweis (Forts.):

Rekursionsschritt:

$$\begin{aligned}
\text{Platz für } k\text{-Struktur} &\leq c2^k + c'2^{\frac{k}{2}}(\log k - 1) + 2^{\frac{k}{2}}c'2^{\frac{k}{2}}(\log k - 1) \\
&= c2^k + c'2^{\frac{k}{2}}(1 + 2^{\frac{k}{2}})(\log k - 1) \\
&= c2^k + c'2^{\frac{k}{2}}(2^{\frac{k}{2}}\log k + \log k - 2^{\frac{k}{2}} - 1) \\
&\leq c2^k + c'2^{\frac{k}{2}}(2^{\frac{k}{2}}\log k + \log k - 2^{\frac{k}{2}}) \\
&\leq c'2^k \log k\,, \text{ falls}
\end{aligned}$$

$$c2^k + c'2^k \log k + c'2^{\frac{k}{2}} \log k - c'2^k \leq c'2^k \log k$$

$$\Leftrightarrow c'2^{\frac{k}{2}} \log k \leq (c' - c)2^k$$

$$\Leftrightarrow \frac{c' - c}{c'} \geq \frac{\log k}{2^{\frac{k}{2}}} \text{ (gilt für } c' \text{ groß genug!)}$$

□

Satz 70

Sei $N = 2^k$, Universum $U = \{0, \ldots, N-1\}$. Wird eine Teilmenge $S \subseteq U$ durch eine k-Struktur dargestellt, dann benötigen die Operationen Insert, Delete und FindMin jeweils Zeit $\mathcal{O}(\log\log N)$, die Initialisierung Zeit $\mathcal{O}(N \log\log N)$. Der Platzbedarf ist ebenfalls $\mathcal{O}(N \log\log N)$.

Beweis:

s.o. □

Literatur zu van Emde Boas-Priority Queue:

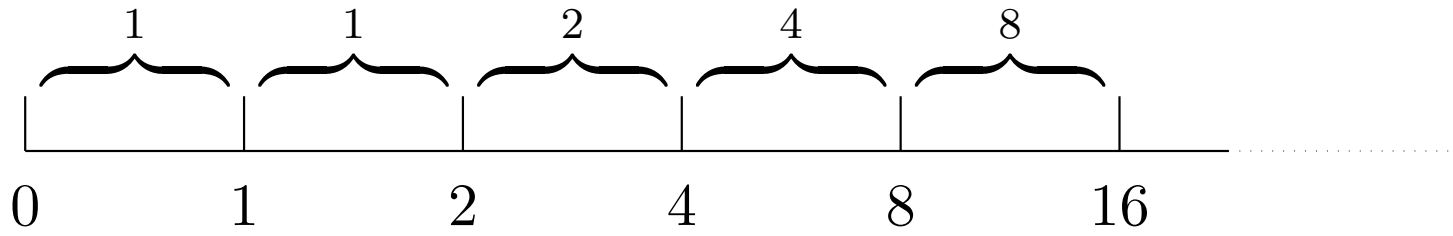
Kurt Mehlhorn:
Data structures and algorithms 1: Sorting and searching, pp. 290–296,
EATCS Monographs on Theoretical Computer Science,
Springer Verlag: Berlin-Heidelberg-New York-Tokyo, 1984

P. van Emde Boas, R. Kaas, E. Zijlstra:
Design and implementation of an efficient priority queue
Math. Systems Theory 10 (1976), pp. 99–127

7.3 Radix-Heaps

Radix-Heaps stellen eine Möglichkeit zur effizienten Realisierung von Priority Queues dar, wobei ähnliche Randbedingungen wie bei den 2-Level-Buckets vorausgesetzt werden. Dabei wird die amortisierte Laufzeit der langsamsten Zugriffsfunktion im Vergleich zu diesen verbessert, nämlich von $\mathcal{O}(\sqrt{C})$ auf $\mathcal{O}(\log C)$. C bezeichne wie bei den 2-Level-Buckets die maximale Differenz zwischen zwei Schlüsseln im Heap.

Die Grundidee besteht darin, anstelle einer Hierarchie von Buckets konstanter Größe solche mit exponentiell zunehmender Größe zu verwenden. Somit sind nur noch $\mathcal{O}(\log C)$ Buckets zur Verwaltung der Elemente im Heap nötig. Wie bei 2-Level-Buckets hängt die Laufzeit der "teueren" Operationen direkt von der Anzahl der Buckets ab.



$B = \lceil \log(C+1) \rceil + 1$ Buckets

Randbedingungen:

- Schlüssel $\in \mathbb{N}_0$
- max. Schlüssel $-$ min. Schlüssel stets $\leq C$
- Monotonie von *ExtractMin*

Implementierung:

- $B := \lceil \log(C+1) \rceil + 1$
- Buckets $b[0..B-1]$
- (untere) Schranken für Buckets $u[0..B]$
- Index $b\_no[x]$ des aktuellen Buckets für x

Invarianten:

i) $u[i] \leq$ Schlüssel in $b[i] < u[i+1]$

ii) $u[0] = 0$, $u[1] = u[0] + 1$, $u[B] = \infty$;
$0 \leq u[i+1] - u[i] \leq 2^{i-1}$; für $i = 1, \ldots, B-1$

Operationen:

1. *Initialize*: Leere Buckets erzeugen und die Bucketgrenzen $u[i]$ entsprechend der Invariante ii) setzen:

for $i := 0$ **to** B **do** $b[i] := \emptyset$ **od**
$u[0] := 0$; $u[1] = 1$
for $i := 2$ **to** $B - 1$ **do**
    $u[i] := u[i-1] + 2^{i-2}$
od
$u[B] := \infty$

Operationen:

2. *Insert*(x): Um ein neues Element einzufügen, wird linear nach dem richtigen Bucket gesucht, beginnend beim letzten Bucket:

if #Elemente in Radix-Heap $= 0$ (d.h. $b[0] = \emptyset$) **then**
füge x in $b[0]$ ein
$u[0] :=$ Schlüssel von x
passe $u[i]$, $i = 1, \ldots, B-1$, gemäß Invariante ii) an
return
fi
$i := B - 1$
while $u[i] > k(x)$ **do** $i := i - 1$ **od**
füge x in $b[i]$ ein

Operationen:

3. *DecreaseKey*(x, δ): Hierbei wird analog zur Prozedur *Insert* linear nach dem Bucket gesucht, in den das Element mit dem veränderten Schlüssel eingefügt werden muss. Der einzige Unterschied besteht darin, dass die Suche in diesem Fall beim aktuellen Bucket des Elements beginnen kann, da der Schlüssel erniedrigt wird und das Element deshalb nie in einen größeren Bucket wandern kann. Aus diesem Grund ist es jedoch notwendig, zu jedem Element x die dazugehörige aktuelle Bucketnummer in $b\_no[x]$ zu speichern:

   $i := b\_no[x]$
   entferne x aus $b[i]$
   $k(x) := k := k(x) - \delta$; **co** überprüfe Invarianten! **oc**
   while $(u[i] > k)$ **do** $i := i - 1$ **od**
   füge x in $b[i]$ ein

Operationen:

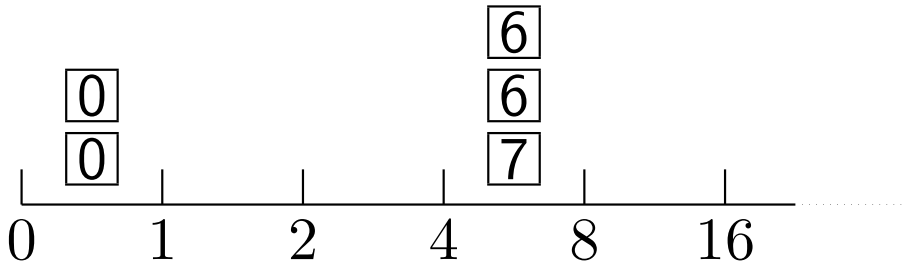
4. *ExtractMin*:
   Es wird ein Element aus $b[0]$ entfernt (und zurückgegeben). Falls $b[0]$ nun leer ist, wird nach dem ersten nicht leeren Bucket $b[i]$, $i > 0$, gesucht und der kleinste dort enthaltene Schlüssel k festgestellt. Es wird $u[0]$ auf k gesetzt, und die Bucketgrenzen werden gemäß der Invarianten neu gesetzt. Danach werden die Elemente in $b[i]$ auf die davorliegenden Buckets verteilt:

Operationen:

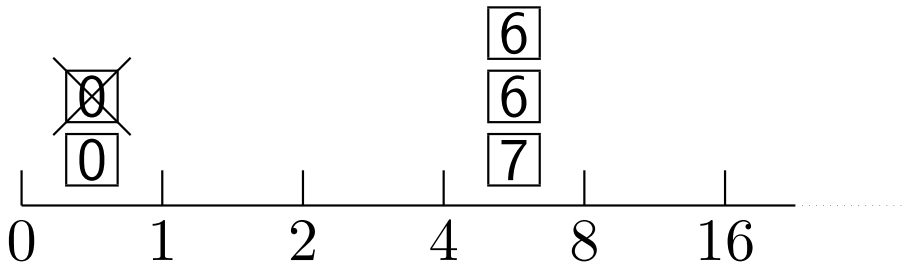
4. *ExtractMin*:

entferne (und gib zurück) ein beliebiges Element in $b[0]$
if #Elemente in Radix-Heap $= 0$ **then return fi**
if $b[0]$ nicht leer **then return fi**
$i := 1$
while $b[i] = \emptyset$ **do** $i := i + 1$ **od**
$k :=$ kleinster Schlüssel in $b[i]$
$u[0] := k$
$u[1] := k + 1$
for $j := 2$ **to** i **do**
$\quad u[j] = \min\{u[j-1] + 2^{j-2}, u[i+1]\}$
od
verteile alle Elemente aus $b[i]$ auf $b[0], b[1], \ldots, b[i-1]$

Beispiel 71

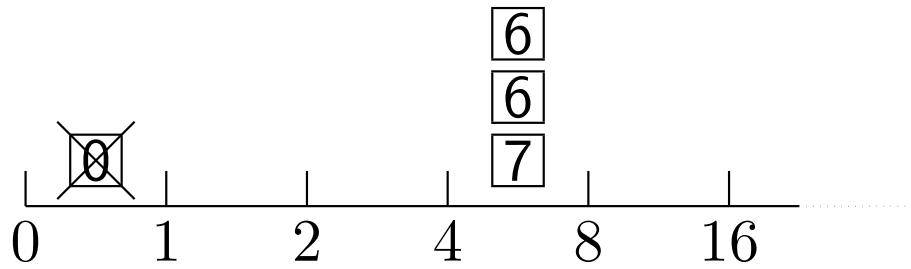


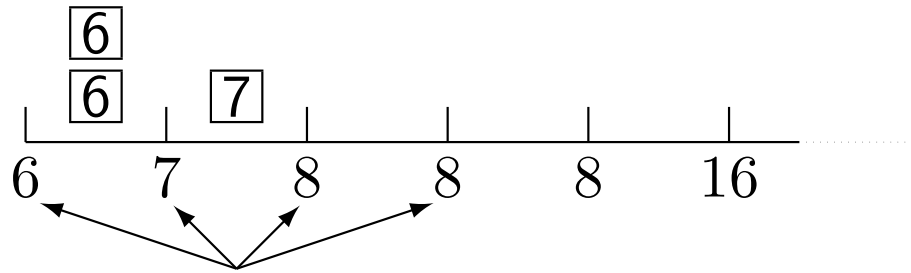
1. ExtractMin



Beispiel 71

2. ExtractMin





neu gesetzte Intervallgrenzen

Korrektheit von ExtractMin:
Es gilt: $b[i] \neq \emptyset \Rightarrow$ Intervallgröße $\leq 2^{i-1}$.

Unterhalb $b[i]$ stehen i Buckets zur Verfügung mit Intervallen

$$[k, k+1[,\ [k+1, k+2[,\ [k+2, k+4[, \ldots, [k+2^{i-2}, k+2^{i-1}[$$

(wobei alle Intervallgrenzen jedoch höchstens $u[i+1]$ sein können).

Da alle Schlüssel in $b[i]$ aus dem Intervall
$[k, \min\{k+2^{i-1}-1, u[i+1]-1\}]$ sind, passen sie alle in
$b[0], b[1], \ldots, b[i-1]$.

Analyse der amortisierten Kosten:

$$\text{Potenzial: } c \cdot \sum_{x \in \text{Radix-Heap}} b\_no[x]\,,$$

für ein geeignetes $c > 0$.

Amortisierte Komplexität:

i) *Initialize*: $\mathcal{O}(B)$

ii) *Insert*: $\mathcal{O}(B)$

iii) *DecreaseKey*:$1 + \mathcal{O}(\Delta b\_no[x] - c \cdot \Delta b\_no[x]) = \mathcal{O}(1)$

iv) *ExtractMin*:
$\mathcal{O}(i + |b[i]| + \sum_{x \in b[i]} \Delta b\_no[x] - c \cdot \sum_{x \in b[i]} \Delta b\_no[x]) = \mathcal{O}(1)$

Satz 72

Ausgehend von einem leeren Heap beträgt die worst-case (reelle) Laufzeit für k Insert-, l DecreaseKey und l ExtractMin-Operationen bei Radix-Heaps

$$\mathcal{O}(k \log C + l)\,.$$

Beweis:

s.o. □

7.3.1 Literatur:

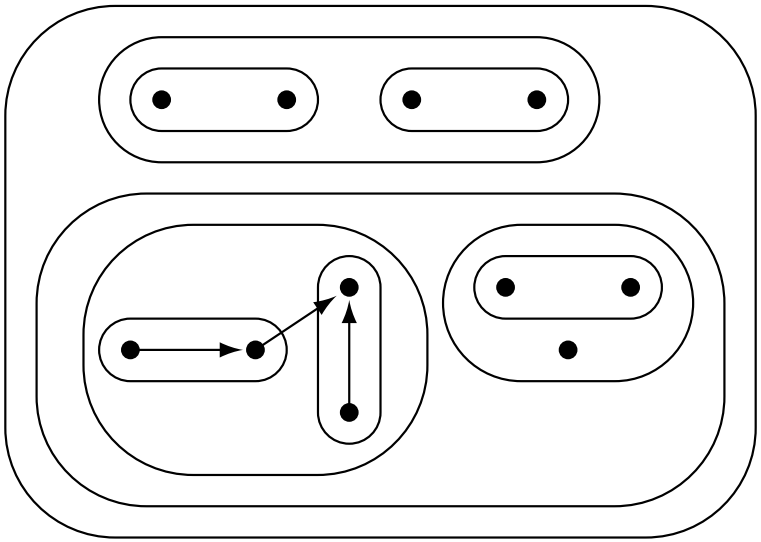

R.K. Ahuja, K. Mehlhorn, J.B. Orlin, R.E. Tarjan:
Faster Algorithms for the Shortest Path Problem
J.ACM **37**, pp. 213–223 (1990)


B.V. Cherkassky, A.V. Goldberg, T. Radzig:
Shortest Path Algorithms: Theory and Experimental Evaluation
Math. Prog. **73**, pp. 129–174 (1996)


B.V. Cherkassky, A.V. Goldberg, C. Silverstein:
Buckets, Heaps, Lists and Monotone Priority Queues
Proc. 8th SODA, ACM, pp. 83–92 (1997)

8. Union/Find-Datenstrukturen

8.1 Motivation

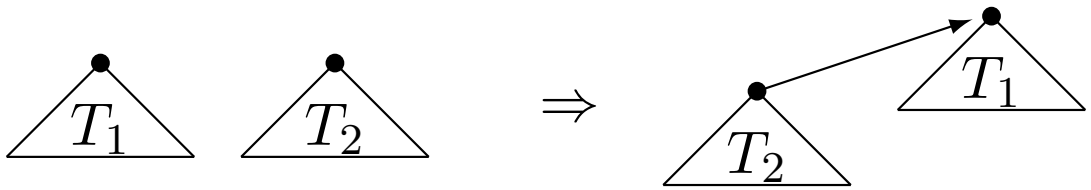


- *Union*(T_1, T_2): Vereinige T_1 und T_2
  $T_1 \cap T_2 = \emptyset$
- *Find*(x): Finde den Repräsentanten der (größten) Teilmenge, in der sich x gerade befindet.

8.2 Union/Find-Datenstruktur

8.2.1 Intrees

1. Initialisierung: $x \to \bullet x$: Mache x zur Wurzel eines neuen (einelementigen) Baumes.
2. *Union*(T_1, T_2):



3. *Find*: Suche Wurzel des Baumes, in dem sich x befindet.

Bemerkung: Naive Implementation: worst-case-Tiefe $= n$

- Zeit für *Find* $= \Omega(n)$
- Zeit für *Union* $= \mathcal{O}(1)$

8.2.2 Gewichtete Union (erste Verbesserung)

Mache die Wurzel des kleineren Baumes zu einem Kind der Wurzel des größeren Baumes. Die Tiefe des Baumes ist dann $\mathcal{O}(\log n)$.

- Zeit für *Find* $= \mathcal{O}(\log n)$
- Zeit für *Union* $= \mathcal{O}(1)$

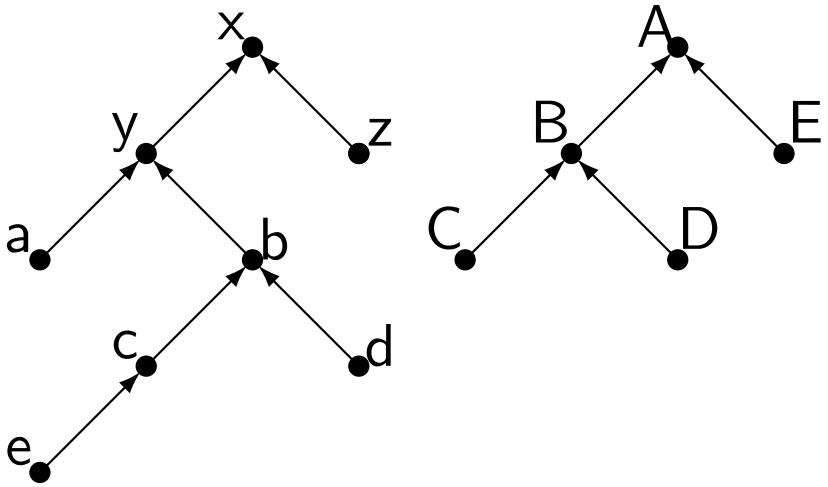
Es gilt auch: Tiefe des Baumes im worst-case:

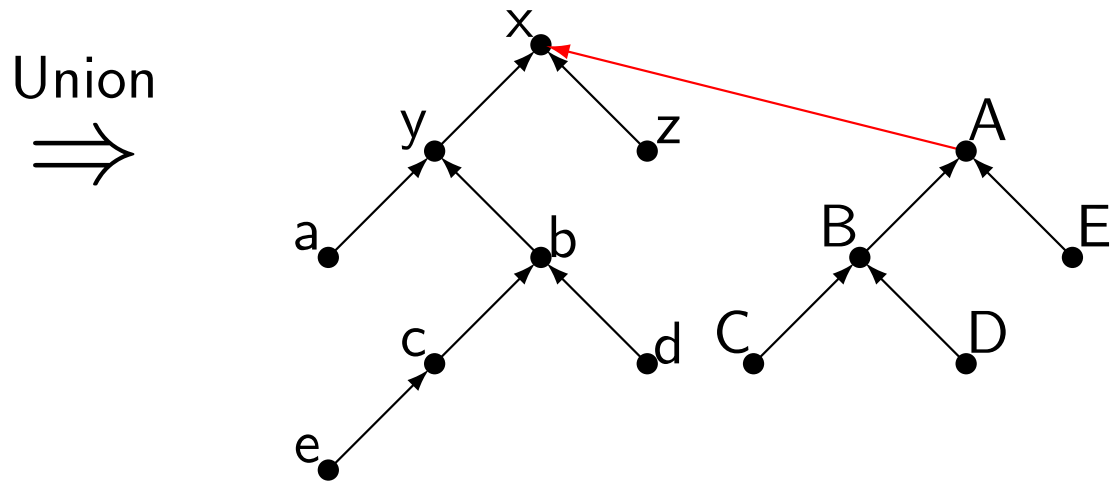
$$\boxed{\Omega(\log n)}$$

8.2.3 Pfad-Kompression mit gewichteter Union (zweite Verbesserung)

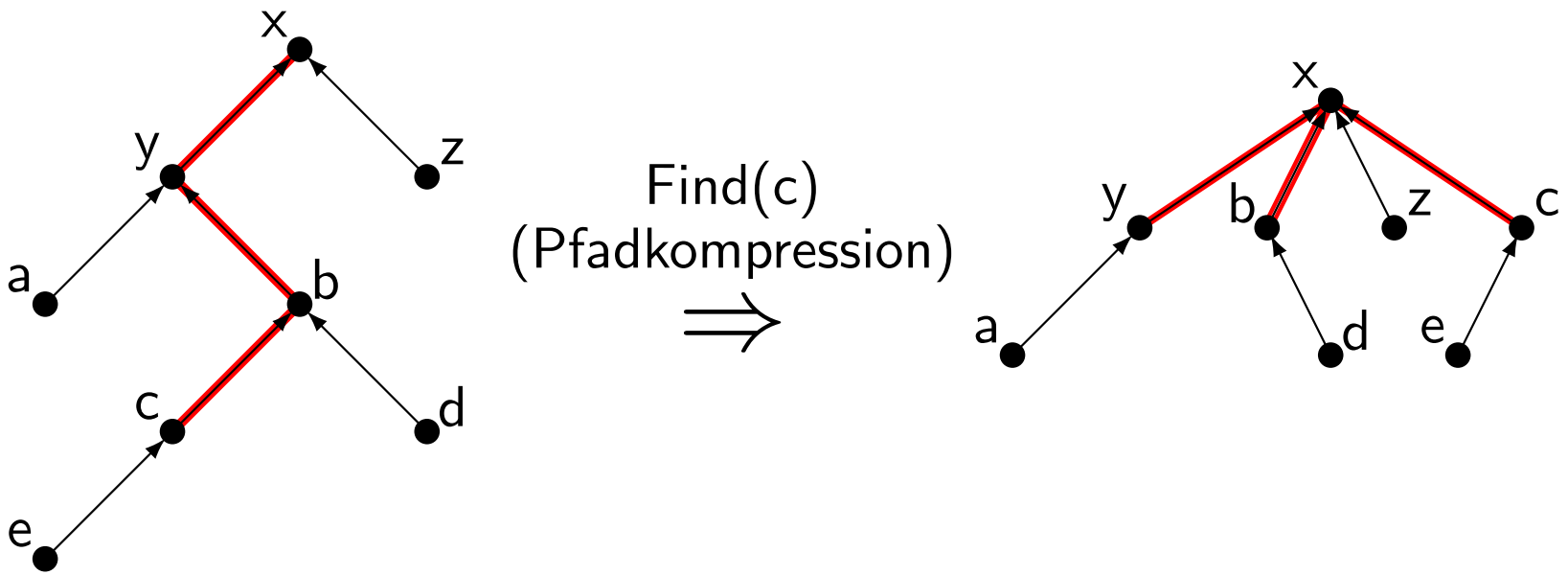
Wir betrachten eine Folge von k *Find*- und *Union*-Operationen auf einer Menge mit n Elementen, darunter $n-1$ *Union*.

Implementierung: Gewichtete *Union* für Pfad-Kompression:





Implementierung: *Find* für Pfad-Kompression:



Bemerkung:
Nach Definition ist

$$\log^* n = \min\{i \geq 0; \underbrace{\log\log\log\ldots\log n}_{i \text{ log 's}} \leq 1\}$$

Beispiel 73

$$\log^* 0 = \log^* 1 = 0$$
$$\log^* 2 = 1$$
$$\log^* 3 = 2$$
$$\log^* 16 = 3 \qquad \text{da } 16 = 2^{2^2}$$
$$\log^* 2^{65536} = 5 \qquad \text{da } 2^{65536} = 2^{2^{2^{2^2}}}$$

Satz 74

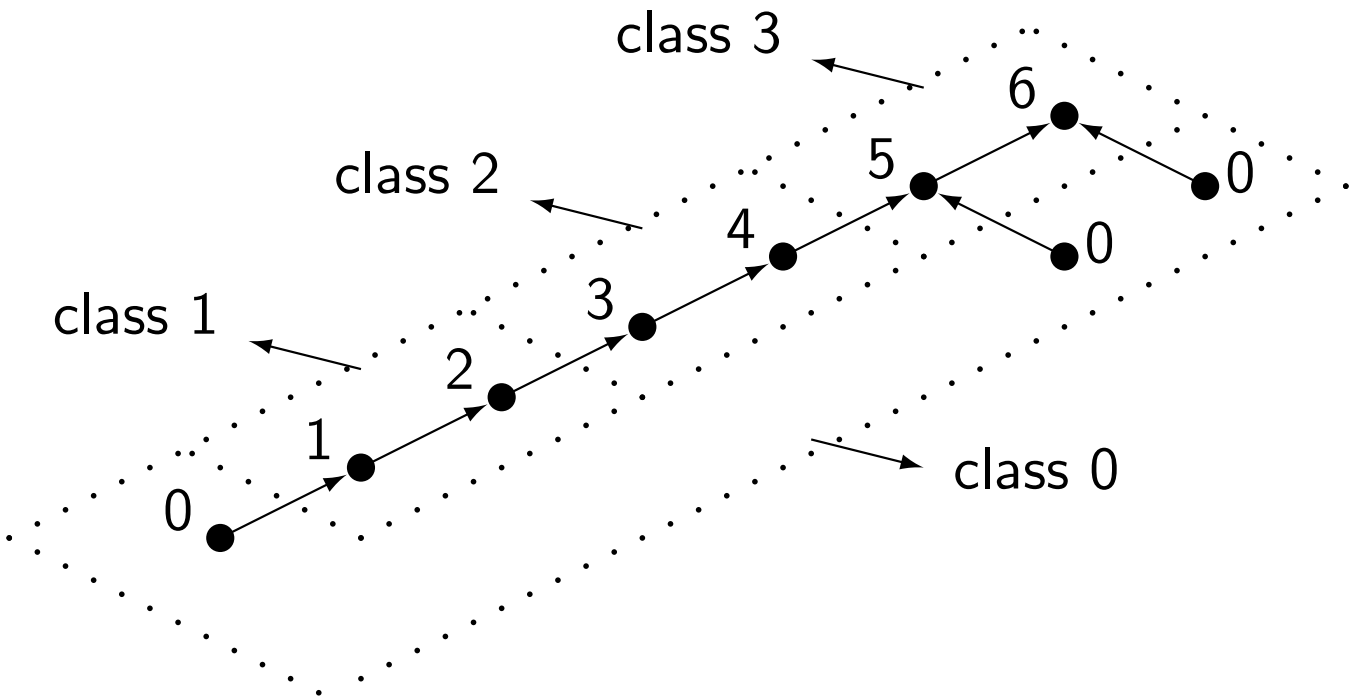
Bei der obigen Implementierung ergibt sich eine amortisierte Komplexität von $\mathcal{O}(\log^* n)$ pro Operation.

Beweis:

Sei T' der (endgültige) In-Baum, der durch die Folge der *Union*'s, ohne die *Find*'s, entstehen würde (also keine Pfad-Kompression).
Ordne jedem Element x drei Werte zu:

- rank(x):= Höhe des Unterbaums in T' mit Wurzel x
- class(x):= $\begin{cases} i \geq 1 & \text{falls } a_{i-1} < \text{ rank}(x) \leq a_i \text{ ist } (i \geq 1) \\ 0 & \text{falls rank}(x) = 0 \end{cases}$

  Dabei gilt: $a_0 = 0, a_i = \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{i \text{ 2'en}}$ für $i \geq 1$.
  Setze zusätzlich $a_{-1} := -1$.


class 3
class 2
class 1
class 0
0
1
2
3
4
5
6
0
0

Beweis (Forts.):

- dist(x) ist die Distanz von x zu einem Vorfahr y im momentanen Union/Find-Baum (mit Pfad-Kompression), so dass class(y) $>$ class(x) bzw. y die Wurzel des Baumes ist.

Definiere die Potenzialfunktion

$$\text{Potenzial} := c \sum_x \text{dist}(x), \quad c \text{ eine geeignete Konstante } > 0$$

Beweis (Forts.):

Beobachtungen:

i) Sei T ein Baum in der aktuellen Union/Find-Struktur (mit Pfad-Kompression), seien x, y Knoten in T, y Vater von x. Dann ist class(x) $\leq$ class(y).

ii) Aufeinander folgende *Find(x)* durchlaufen (bis auf eine) verschiedene Kanten. Diese Kanten sind (im wesentlichen) eine Teilfolge der Kanten in T' auf dem Pfad von x zur Wurzel.

Beweis (Forts.):

Amortisierte Kosten Find(x):

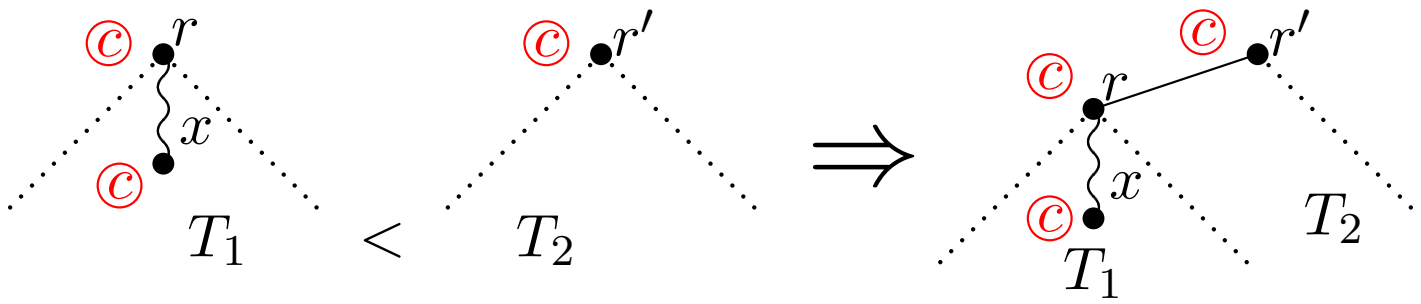
Sei $x_0 \to x_1 \to x_2 \ldots x_k = r$ der Pfad von x_0 zur Wurzel. Es gibt höchstens $\log^* n$-Kanten (x_{i-1}, x_i) mit class$(x_{i-1}) <$ class(x_i). Ist class$(x_{i-1}) =$ class(x_i) und $i < k$ (also $x_i \neq r$), dann ist dist(x_{i-1}) vor der *Find(x)*-Operation ≥ 2, nachher gleich 1.

Damit können die Kosten für alle Kanten (x_{i-1}, x_i) mit class(x_{i-1}) $=$ class(x_i) aus der Potenzialverringerung bezahlt werden. Es ergeben sich damit amortisierte Kosten

$$\boxed{\mathcal{O}(\log^* n)}$$

Beweis (Forts.):

Amortisierte Gesamtkosten aller $(n-1)$-Union's:



Die gesamte Potenzialerhöhung durch alle *Union*'s ist nach oben durch das Potenzial von T' beschränkt (Beobachtung ii).

Beweis (Forts.):

$$\begin{aligned}
\text{Potenzial}(T') &\leq c \cdot \sum_{i=0}^{\log^* n} \sum_{\text{rank}(x)=j=a_{i-1}+1}^{a_i} \text{dist}(x) \\
&\leq c \cdot \sum_{i=0}^{\log^* n} \sum_{\text{rank}(x)=j=a_{i-1}+1}^{a_i} \frac{n}{2^j} a_i \\
&\leq c \cdot n \sum_{i=0}^{\log^* n} a_i \frac{1}{2^{a_{i-1}}} = c \cdot n \sum_{i=0}^{\log^* n} 1 \\
&= \mathcal{O}(n \log^* n) \,.
\end{aligned}$$

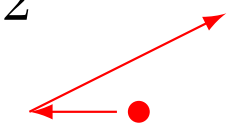
Die zweite Ungleichung ergibt sich, da alle Unterbäume, deren Wurzel x $\text{rank}(x) = j$ hat, disjunkt sind und jeweils $\geq 2^j$ Knoten enthalten. □

8.2.4 Erweiterungen

1) Bessere obere Schranke $\alpha(k,n), k \geq n$. Betrachte die (Variante der) Ackermannfunktion $A(m,n)$ mit:

$$A(0,n) = 2n; \qquad n \geq 0$$
$$A(m,0) = 2; \qquad m \geq 1$$
$$A(m+1,n+1) = A(m, A(m+1,n))$$

$m \downarrow$ / $n \rightarrow$	0	2	4	6	8
	2	4	8	16	32
	2	8	2^9		
	2				
	2				
	$\vdots$				



Die Ackermannfunktion $A(\cdot,\cdot)$ steigt asymptotisch schneller als jede primitiv-rekursive Funktion.

Definition 75

Die Klasse der primitiv-rekursiven Funktionen (auf den natürlichen Zahlen) ist induktiv wie folgt definiert:

1. Alle konstanten Funktionen sind primitiv-rekursiv.
2. Alle Projektionen sind primitiv-rekursiv.
3. Die Nachfolgerfunktion auf den natürlichen Zahlen ist primitiv-rekursiv.
4. Jede Funktion, die durch Komposition von primitiv-rekursiven Funktionen entsteht, ist primitiv-rekursiv.
5. Jede Funktion, die durch sog. primitive Rekursion aus primitiv-rekursiven Funktionen entsteht, ist primitiv-rekursiv. Primitive Rekursion bedeutet folgendes Schema für die Definition von f:

$$f(0, \cdots) = g(\cdots)$$
$$f(n+1, \cdots) = h(f(n, \cdots), \cdots)$$

wobei g, h bereits primitiv-rekursive Funktionen sind.

Weiter wird gesetzt:

$$\alpha(k,n) := \min\{i \geq 1; A(i, \left\lfloor \frac{k}{n} \right\rfloor) > \log n\}$$

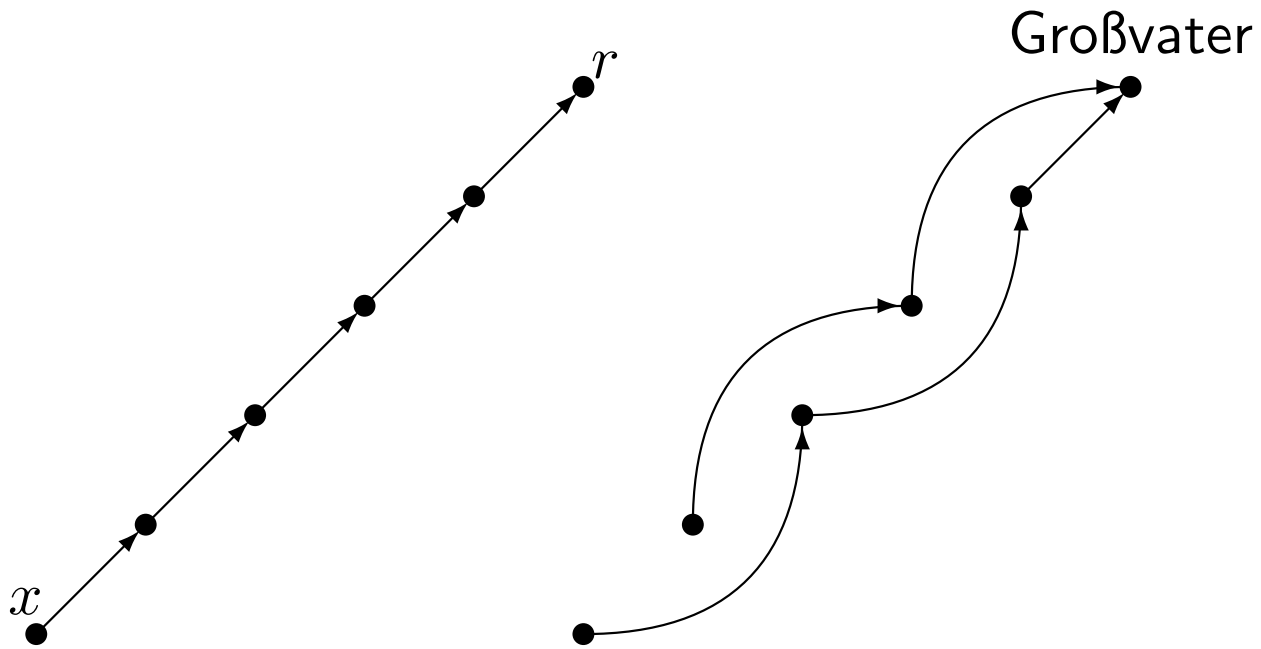
Dann gilt: Der Zeitbedarf für eine Folge von k *Find*- und *Union*-Operationen auf einer Menge mit n Elementen, darunter $n-1$ *Union*, ist

$$\mathcal{O}(k\alpha(k,n)) .$$

Es gilt auch eine entsprechende untere Schranke.

Robert E. Tarjan:
Data Structures and Network Algorithms
SIAM CBMS-NSF Regional Conference Series in Applied Mathematics Bd. 44 (1983)

2) Variante der Pfadkompression:



Diese Variante der Pfadhalbierung erfüllt ebenfalls die $\mathcal{O}(k\alpha(k,n))$ Schranke.

Kapitel III Selektieren und Sortieren

1. Einleitung

Gegeben: Menge S von n Elementen aus einem total geordneten Universum U, $i \in \mathbb{N}$, $1 \leq i \leq n$.
Gesucht: i-kleinstes Element in S.

Die Fälle $i = 1$ bzw. $i = n$ entsprechen der Suche nach dem Minimum bzw. Maximum.
Der Standardalgorithmus dafür benötigt $n - 1$ Vergleiche.

Satz 76

Die Bestimmung des Minimums/Maximums von n Elementen benötigt mindestens $n-1$ Vergleiche.

Beweis:

Interpretiere Algorithmus als Turnier. Ein Spiel wird jeweils vom kleineren Element gewonnen. Wir beobachten: Jedes Element außer dem Gesamtsieger muss mindestens ein Spiel verloren haben $\Rightarrow n-1$ Vergleiche notwendig. □

Bestimmung des Vize-Meisters bzw. des zweitkleinsten Elements

Satz 77

Das zweitkleinste von n Elementen kann mit

$$n + \lceil \log_2 n \rceil - 2$$

Vergleichen bestimmt werden.

Beweis:

Wir betrachten wiederum ein KO-Turnier: $(n-1)$ Vergleiche genügen zur Bestimmung des Siegers (Minimum).
Das zweitkleinste Element ist unter den „Verlierern“ gegen das Minimum zu suchen. Deren Anzahl ist $\leq \lceil \log_2 n \rceil$. Man bestimme nun unter diesen Elementen wiederum das Minimum und erhält damit das zweitkleinste Element in $\leq \lceil \log_2 n \rceil - 1$ weiteren Vergleichen. □


Lewis Carroll:
Lawn Tennis Tournaments
St. Jones Gazette (Aug. 1, 1883), pp. 5–6
Reprinted in The Complete Work of Lewis Carroll. Modern Library, New York (1947)


Vaughan R. Pratt, Frances F. Yao:
On lower bounds for computing the i-th largest element
Proc. 14th Ann. IEEE SWAT, pp. 70–81 (1973)


Donald E. Knuth:
The art of computer programming. Vol. 3: Sorting and searching,
3. Auflage, Addison-Wesley Publishing Company: Reading (MA), 1997

2. Der Blum-Floyd-Pratt-Rivest-Tarjan Selektions-Algorithmus

Definition 78

Sei $n \in \mathbb{N}$. Der Median (das „mittlere“ Element) einer total geordneten Menge von n Elementen ist deren i-kleinstes Element, wobei

$$i = \left\lceil \frac{n}{2} \right\rceil .$$

Bemerkung: Für gerade n wird manchmal auch $i = \left\lfloor \frac{n}{2} \right\rfloor + 1$ benutzt.

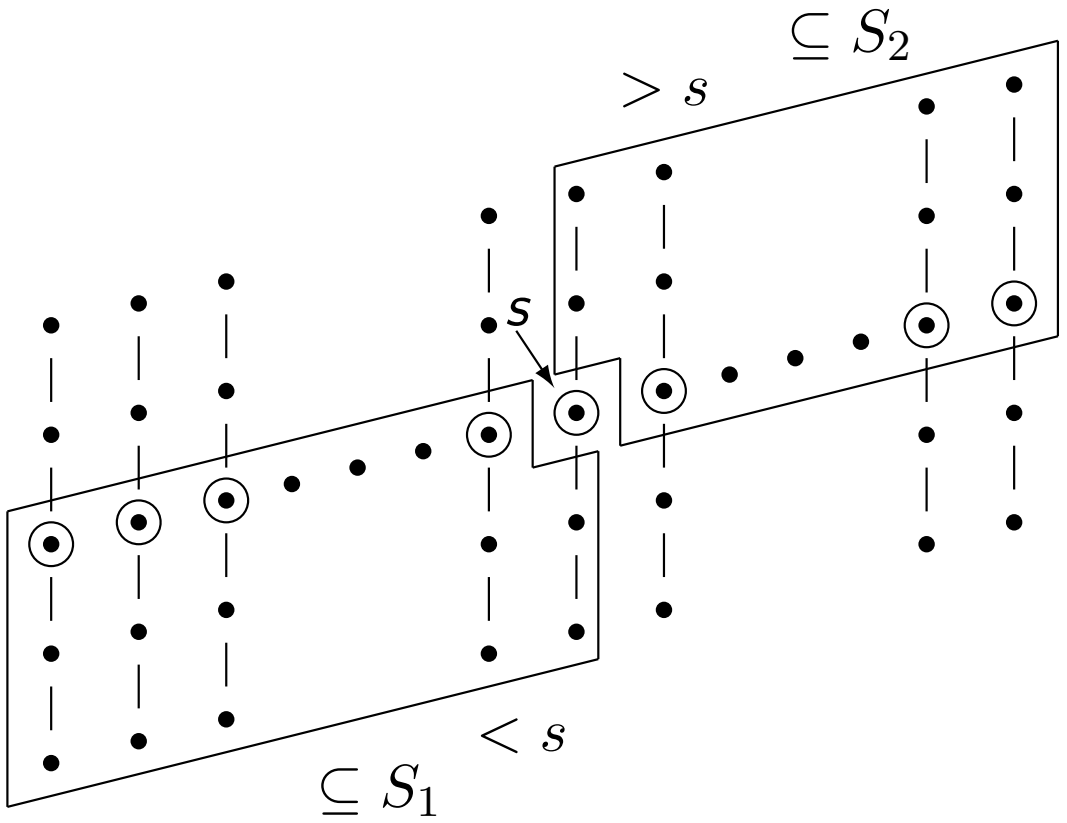
Sei m eine kleine ungerade Zahl (etwa $5 \leq m \leq 21$). Sei $S := \{a_1, \ldots, a_n\}$ eine Menge von n paarweise verschiedenen Elementen. Zur Bestimmung des i-kleinsten Element in S betrachten wir folgenden Algorithmus BFPRT.

Der BFPRT-Selektions-Algorithmus (1/3)

1. Teile S in $\left\lceil \frac{n}{m} \right\rceil$ Blöcke auf, $\left\lfloor \frac{n}{m} \right\rfloor$ davon mit je m Elementen
2. Sortiere jeden dieser Blöcke
3. Sei S' die Menge der $\left\lceil \frac{n}{m} \right\rceil$ Mediane der Blöcke. Bestimme rekursiv den Median s dieser Mediane (also das $\left\lceil \frac{|S'|}{2} \right\rceil$-kleinste Element von S').

Der BFPRT-Selektions-Algorithmus (2/3)

4. Partitioniere $S - \{s\}$ in $S_1 := \{x \in S : x < s\}$, $S_2 := \{x \in S : x > s\}$. Bemerkung: $|S_1|,\ |S_2| \geq \frac{n}{4}$, falls $n \geq m^2 + 4m$.



Der BFPRT-Selektions-Algorithmus (3/3)

5. Falls $i \leq |S_1|$, bestimme rekursiv das i-kleinste Element in S_1.
   Falls $i = |S_1| + 1$, gib s als Lösung zurück.
   Ansonsten bestimme rekursiv das $(i - |S_1| - 1)$-kleinste Element in S_2.

Sei $T(n)$ die worst-case Anzahl von Vergleichen für $|S| = n$ des Algorithmus BFPRT. Sei C_m die # von Vergleichen, um m Elemente zu sortieren (z.B. $C_5 = 7$, $C_{11} = 26$). Es gilt:

$$T(n) \leq \underbrace{T\left(\left\lceil \frac{n}{m} \right\rceil\right)}_{3.} + \underbrace{T\left(\left\lfloor \frac{3}{4}n \right\rfloor\right)}_{5.} + \underbrace{\left\lceil \frac{n}{m} \right\rceil}_{2.} C_m + \underbrace{\left\lfloor \frac{n}{2} \right\rfloor}_{4.}$$

Satz 79

Der Selektions-Algorithmus BFPRT bestimmt das i-kleinste Element von n Elementen mit $\mathcal{O}(n)$ Vergleichen (und Zeit).

Beweis:

Annahme: $T(n) \leq c \cdot n$, wobei $c = c(m)$ konstant ist.
Die Annahme ist ok, falls $T(n) \leq$

$$T\left(\left\lceil \frac{n}{m} \right\rceil\right) + T\left(\left\lfloor \frac{3}{4}n \right\rfloor\right) + \left\lceil \frac{n}{m} \right\rceil C_m + \left\lfloor \frac{n}{2} \right\rfloor \leq cn; \text{ dies gilt, falls}$$

$$\left\lceil \frac{n}{m} \right\rceil c + \left\lfloor \frac{3}{4}n \right\rfloor c + \left\lceil \frac{n}{m} \right\rceil C_m + \left\lfloor \frac{n}{2} \right\rfloor \leq cn \qquad \left| \cdot \frac{1}{n} \right. \quad \text{(IA)}$$

$$\Leftrightarrow \text{(bis auf } \lceil\rceil, \lfloor\rfloor \text{)} \ \frac{c}{m} + \frac{3}{4}c + \frac{C_m}{m} + \frac{1}{2} \leq c$$

$$\Leftrightarrow -\frac{c}{m} - \frac{3}{4}c + c \geq \frac{C_m}{m} + \frac{1}{2}$$

$$\Leftrightarrow c \geq \frac{\frac{C_m}{m} + \frac{1}{2}}{1 - \frac{3}{4} - \frac{1}{m}}$$

□

Bemerkung: $m = 11 \rightsquigarrow c = c(m) \approx 20$.

Literatur:

Vaughan R. Pratt, Frances F. Yao:
On lower bounds for computing the i-th largest element
Proc. 14th Ann. IEEE SWAT, pp. 70–81 (1973)

Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ron L. Rivest, Robert E. Tarjan:
Time bounds for selection
J. Comput. Syst. Sci. **7**, pp. 448–461 (1973)

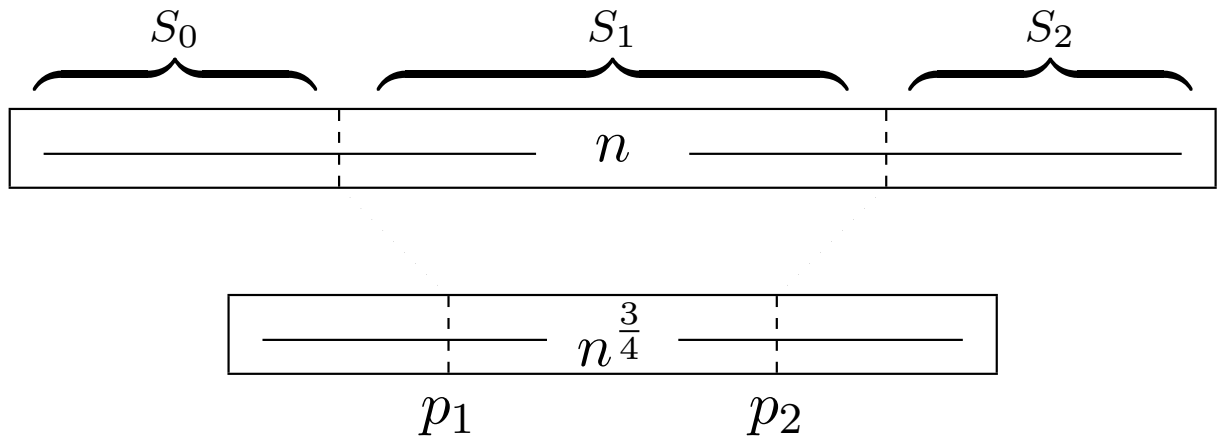
3. Randomisierter Median-Algorithmus

Problemstellung: Bestimme den Median von n Elementen

1. Wähle $n^{\frac{3}{4}}$ Elemente zufällig und gleichverteilt aus den n Elementen aus.
2. Sortiere diese $n^{\frac{3}{4}}$ Elemente mit einem (Standard-) $n \log n$-Algorithmus.
3. Setze

   $p_1 := \max\{\frac{n^{\frac{3}{4}}}{2} - \sqrt{n}, 1\}$-kleinstes Element der $n^{\frac{3}{4}}$ Elemente.

   $p_2 := \min\{\frac{n^{\frac{3}{4}}}{2} + \sqrt{n}, n^{\frac{3}{4}}\}$-kleinstes Element der $n^{\frac{3}{4}}$ Elemente.



4. Partitioniere die n Elemente in

   $S_0 := \{\text{Elemente} < p_1\}$
   $S_1 := \{p_1 \leq \text{Elemente} \leq p_2\}$
   $S_2 := \{p_2 < \text{Elemente}\}$

5. Falls $|S_0| \geq \lceil \frac{n}{2} \rceil$ oder $|S_2| \geq \lceil \frac{n}{2} \rceil$ oder $|S_1| \geq 4 \cdot n^{\frac{3}{4}}$, dann wiederhole den Algorithmus;
   ansonsten sortiere S_1 und liefere das $(\lceil \frac{n}{2} \rceil - |S_0|)$-kleinste Element davon ab.

Satz 80

Obiger randomisierter Algorithmus bestimmt den Median von n Elementen mit einer erwarteten Anzahl von $\frac{3}{2}n + o(n)$ Vergleichen.

Beweis:

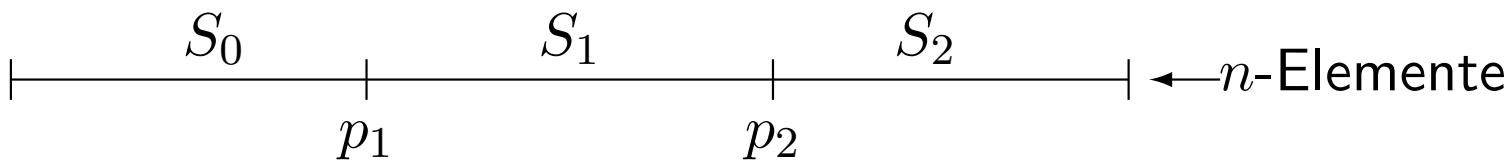
i) Korrektheit: klar.

Beweis (Forts.):

ii) Anzahl der Vergleiche in einer Iteration:

$$\mathcal{O}(n^{\frac{3}{4}} \log n^{\frac{3}{4}}) + \text{Kosten der Partitionierung}$$

Für die Partitionierung ist der naive Ansatz zu ungünstig, stattdessen:



Wähle zuerst jeweils mit Wahrscheinlichkeit $\frac{1}{2}$ aus, ob Element x mit p_1 oder p_2 verglichen wird, mache zweiten Vergleich nur, falls nötig.

Beweis (Forts.):

Die erwartete Anzahl von Vergleichen ist dann

$$= \frac{n}{2}\left(\frac{|S_0|}{n}\cdot 1 + \frac{|S_1|+|S_2|}{n}\cdot 2\right) + \frac{n}{2}\left(\frac{|S_2|}{n}\cdot 1 + \frac{|S_0|+|S_1|}{n}\cdot 2\right)$$

$$= \frac{n}{2}\left(\frac{|S_0|+|S_2|}{n} + 2\frac{\overbrace{|S_0|+|S_1|+|S_2|}^{n}+|S_1|}{n}\right)$$

$$= \frac{n}{2}(3+\frac{|S_1|}{n}) = \frac{3}{2}n + o(n)$$

Wir zeigen nun, dass der Algorithmus mit Wahrscheinlichkeit $\geq 1 - \mathcal{O}(n^{-\frac{1}{4}})$ nur eine Iteration benötigt (daraus folgt dann, dass insgesamt die Anzahl der Vergleiche $\leq \frac{3}{2}n + o(n)$ ist).

Beweis (Forts.):

Dafür verwenden wir Hilfsmittel aus der Wahrscheinlichkeitstheorie/Stochastik:

- **Bernoulli-Zufallsvariable** (ZV): X, Werte $\in \{0,1\}$ mit

$$X = \begin{cases} 1 \text{ mit WS } p \\ 0 \text{ mit WS } q = 1-p \end{cases}$$

- **Erwartungswert** einer ZV:
  $\mathbb{E}[X] = \sum_{x \in \text{Wertebereich}} x \cdot \Pr[X = x]$
  (X ist diskret, d.h. der Wertebereich von X ist endlich)
- **Markov-Ungleichung**: $\Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$ für X nicht negativ
- **Chebyshev-Ungleichung**: $\Pr[|X - \mathbb{E}[X]| \geq t] \leq \frac{\text{Var}(X)}{t^2}$

Beweis (Forts.):

- **Binomialverteilung**: Seien $X_1, \ldots, X_n$ unabhängige, identisch verteilte Bernoulli-Zufallsvariablen mit $\Pr[X_i = 1] = p$.

$$X := \sum_{i=1}^{n} X_i \; .$$

X ist binomial verteilt, mit Wertebereich $\{0, 1, \ldots, n\}$.

$$\Pr[X = k] = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\mathbb{E}[X] = n \cdot p$$

$$\mathrm{Var}[X] = n \cdot p \cdot (1-p) = n \cdot p \cdot q \; .$$

In Zeichen: $X \sim \mathsf{B}(n, p)$

Beweis (Forts.):

Die Auswahl der $n^{\frac{3}{4}}$ Elemente wird wiederholt, falls $|S_0| \geq \frac{n}{2}$. Dies passiert gdw wir höchstens $\frac{1}{2}n^{\frac{3}{4}} - \sqrt{n}$ Elemente aus der Hälfte aller Elemente $\leq$ dem Median auswählen.

Wir bestimmen die Wahrscheinlichkeit dafür, dass keine neue Auswahl der $n^{\frac{3}{4}}$ Elemente stattfinden muss.
Setze Bernoulli-Zufallsvariable $X_1, \ldots, X_n$ mit:

$$X_i = \begin{cases} 1, \text{ falls Element } i < \text{Median ausgewählt wird} \\ 0, \text{ sonst} \end{cases}$$

$X := \sum X_i$ ist binomialverteilt mit Parametern n und $\frac{1}{2}n^{-\frac{1}{4}}$, und $\mathbb{E}[X] = \frac{1}{2}n^{\frac{3}{4}}$, $\mathrm{Var}[X] = n \cdot \frac{1}{2}n^{-\frac{1}{4}}(1 - \frac{1}{2}n^{-\frac{1}{4}}) = \frac{1}{2}n^{\frac{3}{4}}(1 - o(1))$.

Beweis (Forts.):

Die Wahrscheinlichkeit hierbei ist

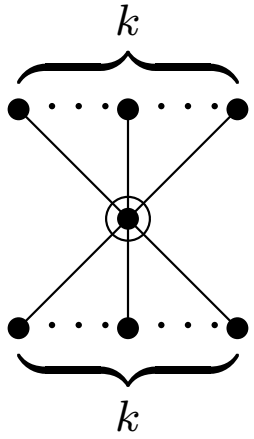
$$\Pr[|S_0| \geq \frac{n}{2}] = \Pr[X \leq \frac{1}{2}n^{\frac{3}{4}} - \sqrt{n}] \leq \Pr[|X - \mathbb{E}[X]| \geq \sqrt{n}]$$
$$\leq \frac{\frac{1}{2}n^{\frac{3}{4}}(1 - o(1))}{n} \leq \frac{1}{2}n^{-\frac{1}{4}}(1 - o(1))$$

Die anderen beiden Wahrscheinlichkeitsbedingungen ($\Pr[|S_2| \geq \lceil \frac{n}{2} \rceil]$ und $\Pr[|S_1| \geq 4 \cdot n^{\frac{3}{4}}]$) ergeben analoge Abschätzungen.
Damit: Wiederholung mit WS $\leq \mathcal{O}(n^{-\frac{1}{4}})$. □

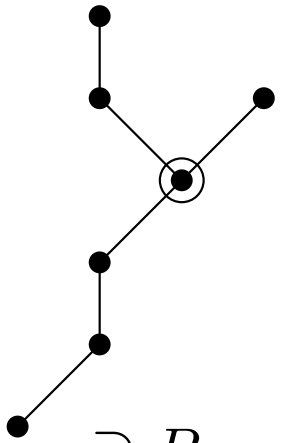
4. Schönhage/Paterson/Pippenger-Median-Algorithmus

Definition 81

Sei $k \in \mathbb{N}\backslash\{0\}$. P_k sei die folgende partielle Ordnung:



$2k+1$ Elemente

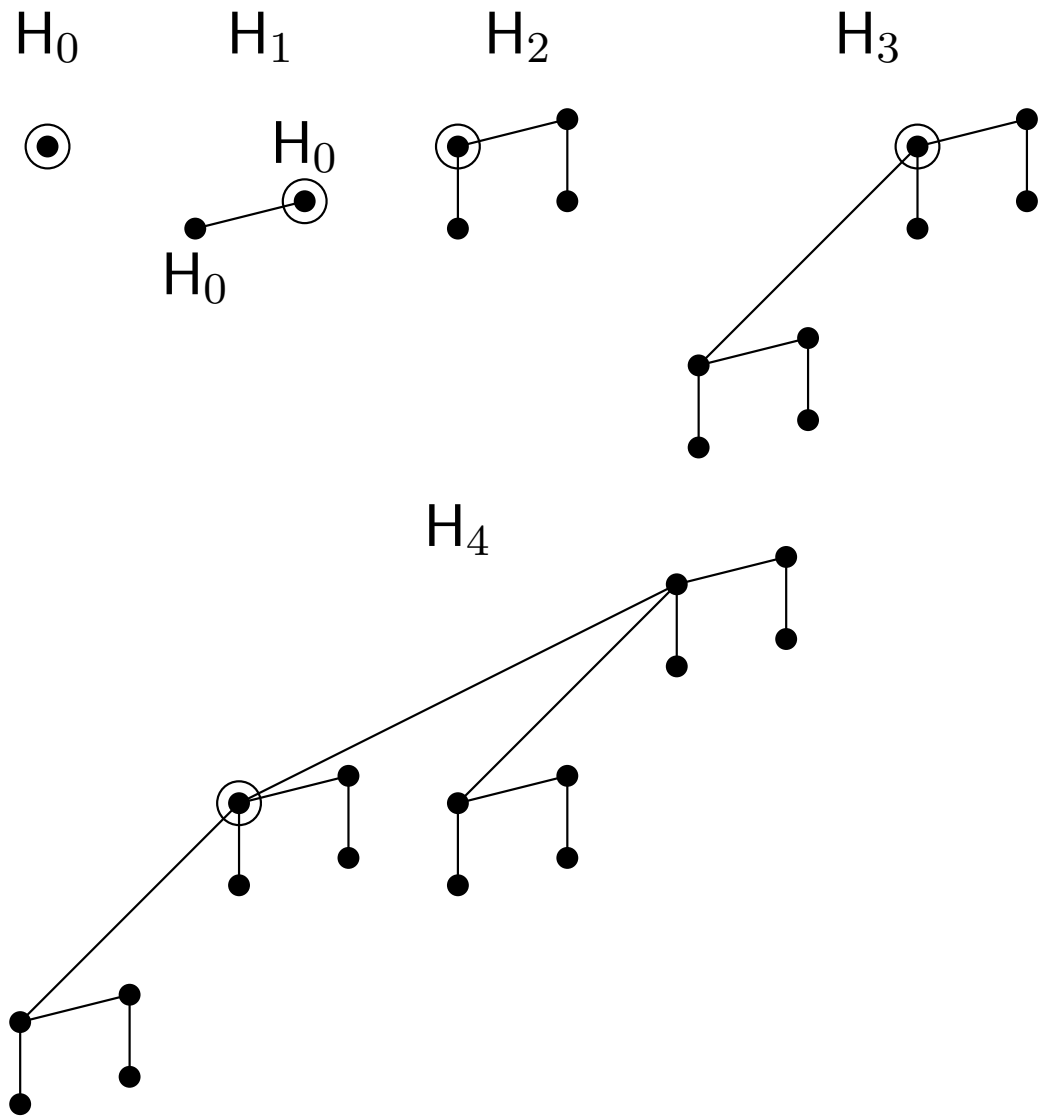


$\supseteq P_3$

Wir betrachten nun spezielle Binomialbäume mit „Zentrum“ (um Ordnungsinformation widerzuspiegeln).

Definition 82

1. Der Baum H_0 besteht aus einem Knoten, und dieser ist auch das Zentrum.
2. $H_{2h}(h > 0)$ besteht aus zwei H_{2h-1}, deren Zentren durch eine neue Kante verbunden sind. Das Zentrum des H_{2h} ist das kleinere der beiden Zentren der H_{2h-1}.
3. $H_{2h+1}(h \geq 0)$ besteht aus zwei H_{2h}, deren Zentren durch eine neue Kante verbunden sind, sein Zentrum ist das größere dieser beiden Zentren.


H0
H1
H0
H0
H2
H3
H4

Lemma 83 (Zerlegungslemma)

a) *H_h hat 2^h Knoten, es werden $2^h - 1$ Vergleiche benötigt, um H_h zu konstruieren.*

b) *H_{2h} kann zerlegt werden in*

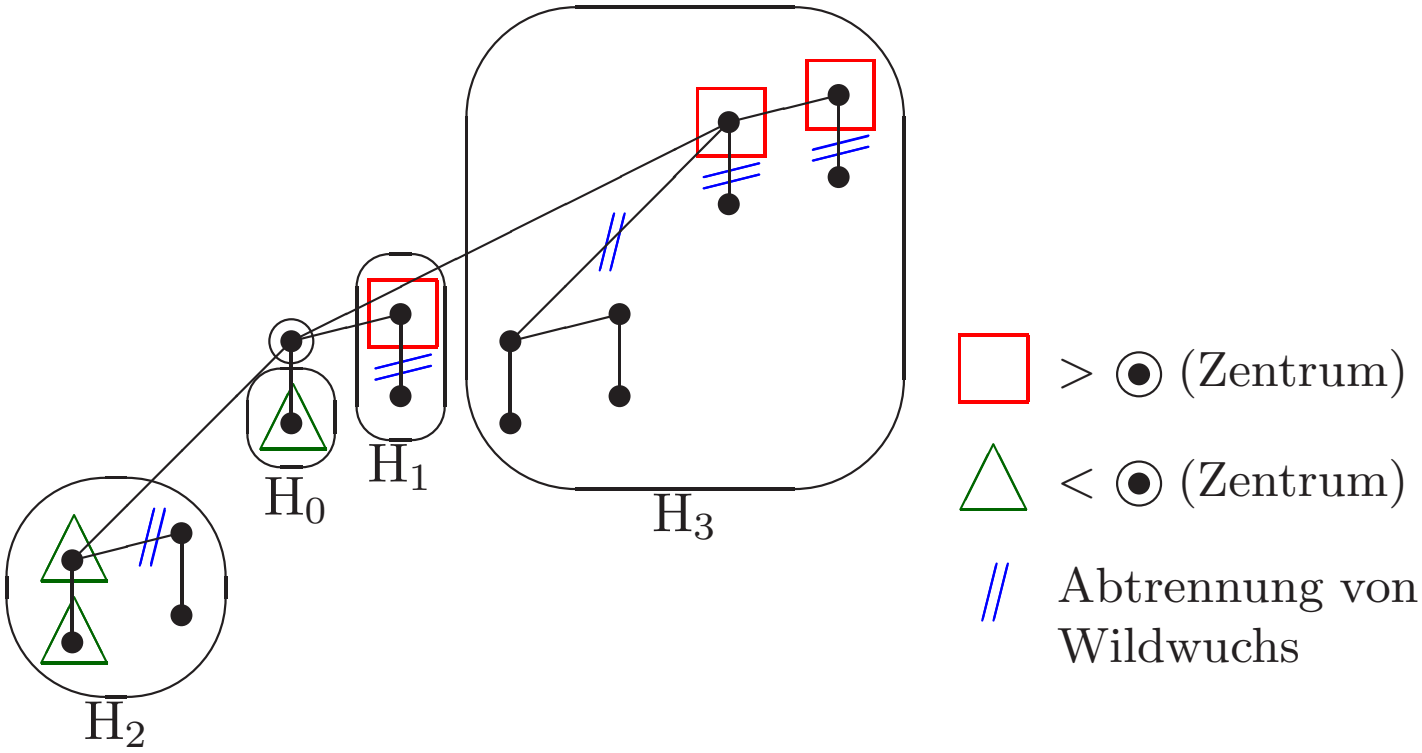
- *sein Zentrum*
- *eine Menge $\{H_1, H_3, \ldots, H_{2h-1}\}$ von disjunkten Teilbäumen, deren Zentren alle größer sind als das Zentrum von H_{2h}.*
- *eine Menge $\{H_0, H_2, H_4, \ldots, H_{2h-2}\}$ von disjunkten Teilbäumen mit Zentren kleiner als das von H_{2h}.*

Lemma 83 (Zerlegungslemma)

c) *H_{2h+1} kann so zerlegt werden, dass die Zusammenhangskomponente des Zentrums genau 2^h Knoten $\geq$ dem Zentrum enthält, indem höchstens $2^{h+1}-1$ Kanten entfernt werden.*
H_{2h} kann so zerlegt werden, dass die Zusammenhangskomponente des Zentrums genau 2^h Knoten enthält, die alle $\leq$ dem Zentrum sind, indem höchstens 2^h-1 Kanten entfernt werden.

d) *Falls $k \leq 2^h-1$, dann kann H_{2h} so zerlegt werden, dass die Zusammenhangskomponente des Zentrums genau $2k+1$ Elemente enthält, von denen k größer und k kleiner als das Zentrum sind ($\Rightarrow P_k$).*
Dazu genügt es, höchstens $3k+2h$ Kanten zu entfernen. Die restlichen Zusammenhangskomponenten sind wieder H_i's.

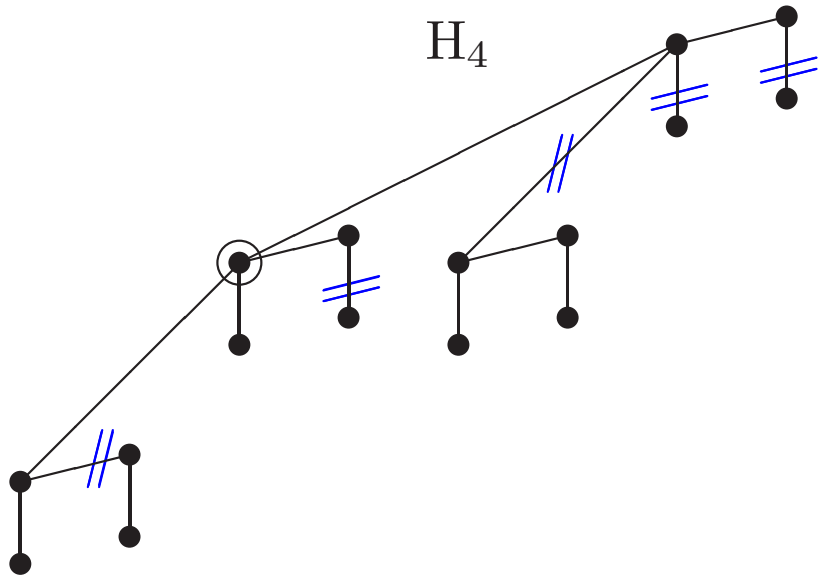
Zerlegungslemma



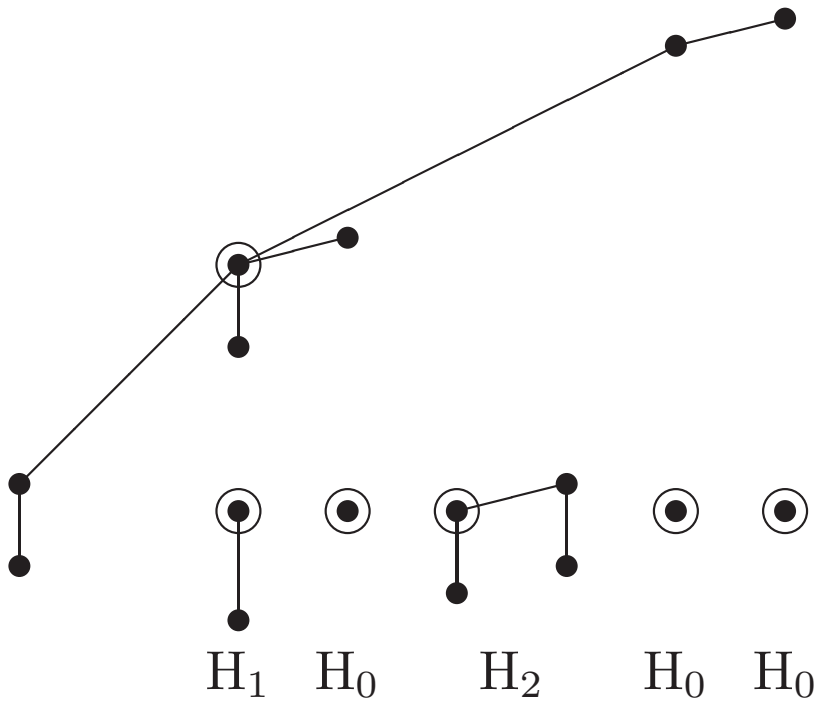
Bemerkung: Bei jedem Konstruktionsschritt wird ein Vergleich durchgeführt, um zu bestimmen, welcher der beiden Teilbäume das kleinere Zentrum hat. Im Algorithmus von Schönhage, Paterson und Pippenger werden aus Teilstücken H_r größere Bäume H_{r+1} zusammengebaut, wodurch schrittweise eine partielle Ordnung auf den Eingabewerten bestimmt wird. Wurde ein Baum H_{2h} hinreichender Größe hergestellt, so wird er durch Zerlegung in einen Baum umgewandelt, der nur noch sein altes Zentrum sowie k darüberliegende und k darunterliegende Elemente enthält, wobei $k \leq 2^h - 1$.

Beispiel 84

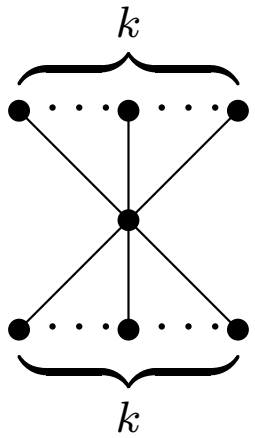
In diesem Beispiel wollen wir H_4 zerlegen und wählen $k = 3$:



Um einen H_4 derart zu zerlegen, müssen wir 5 Kanten aufbrechen. Dabei werden drei H_0, ein H_1 sowie ein H_2 abgespalten.



Übrig bleibt die gewünschte Struktur mit k Knoten über dem Zentrum und k unter dem Zentrum, wodurch eine partielle Ordnung auf $2k+1$ Eingabewerten bestimmt wurde:



$2k+1$ Elemente

Die bei der Zerlegung angefallenen Reststücke werden beim Aufbau weiterer Bäume benutzt. So geht das bereits angesammelte Wissen über die Ordnung der Elemente nicht verloren.

Beweis des Zerlegungslemmas:

Wir beginnen mit Teil a).

Lemma 85

H_r hat 2^r Knoten, es werden $2^r - 1$ Vergleiche benötigt, um H_r aufzubauen.

Beweis:

In jedem der r Konstruktionsschritte wird die Anzahl der Knoten verdoppelt. Da wir mit einem Knoten beginnen, hat H_r folglich 2^r Knoten. Die Anzahl der notwendigen Vergleiche C_r unterliegt folgender Rekursionsgleichung ($r \geq 1$):

$$C_r = 1 + 2C_{r-1} \text{ und } C_0 = 0\,.$$

Damit folgt sofort $C_r = 2^r - 1$. □

Beweis von b):

Lemma 86

H_r kann in folgende disjunkte Bereiche unterteilt werden:

- *sein Zentrum,*
- *eine Reihe $H_1, H_3, \ldots, H_{r-1}$ (falls r gerade) bzw. $H_1, H_3, \ldots, H_{r-2}$ (falls r ungerade) von Unterbäumen, deren Zentren alle über dem von H_r liegen,*
- *eine Reihe $H_0, H_2, \ldots, H_{r-2}$ (falls r gerade) bzw. $H_0, H_2, \ldots, H_{r-1}$ (falls r ungerade) von Unterbäumen, deren Zentren alle unter dem von H_r liegen.*

Beweis von b):

Beweis:

Durch Induktion über r.

Induktionsanfang: für H_0 gilt die Behauptung.

Induktionsannahme: die Behauptung gelte für H_{r-1}.

Beweis von b):

Beweis:

1. Sei $r = 2h, h > 0$.
   H_{2h} besteht aus zwei H_{2h-1}, wobei das kleinere der beiden alten Zentren das neue Zentrum z bildet. Wende auf den H_{2h-1}, der z enthält, die Induktionsannahme an. Wir können diesen Unterbaum also in z sowie $H_1, H_3, \ldots, H_{2h-3}$ (Zentren über z) und $H_0, H_2, \ldots, H_{2h-2}$ (Zentren unter z) partitionieren. Zusammen mit dem H_{2h-1}, dessen Zentrum über z liegt, ergibt sich die Induktionsbehauptung für $H_r = H_{2h}$.

Beweis von b):

Beweis:

2. Sei $r = 2h + 1$, $h \geq 0$. H_{2h+1} besteht aus zwei H_{2h}, wobei das größere der beiden alten Zentren das neue Zentrum z bildet. Wende auf den H_{2h}, der z enthält, die Induktionsannahme an. Wir können diesen Unterbaum also in z sowie $H_1, H_3, \ldots, H_{2h-1}$ (Zentren über z) und $H_0, H_2, \ldots, H_{2h-2}$ (Zentren unter z) partitionieren. Zusammen mit dem H_{2h}, dessen Zentrum unter z liegt, ergibt sich die Induktionsbehauptung für $H_r = H_{2h+1}$.

□

Beweis von c):

Wir bezeichnen im Folgenden mit H_{2h}^- den Baum, der entsteht, wenn wir H_{2h} so zerlegen, dass der verbleibende Rest nur mehr Elemente unterhalb des Zentrums (und das Zentrum selbst) enthält. Mit H_{2h+1}^+ bezeichnen wir den Baum, der entsteht, wenn wir H_{2h+1} so zerlegen, dass der verbleibende Rest nur mehr Elemente über dem Zentrum und dieses selbst enthält.

Lemma 87

H_{2h}^- und H_{2h+1}^+ haben jeweils 2^h Knoten. Bei der Herstellung aus H_{2h} bzw. H_{2h+1} werden $2^h - 1$ bzw. $2^{h+1} - 1$ Kanten aufgebrochen. Die wegfallenden Teile haben die Form H_s, $s < 2h$ bzw. $s < 2h + 1$.

Beweis:

Durch Induktion über r.

Induktionsanfang: für H_0 und H_1 gilt die Behauptung.

Induktionsannahme: die Behauptung gilt für alle H_p, $p < r$.

1. Sei $r = 2h$, $h > 0$. Wir betrachten die Partitionierung von H_{2h} mit Zentrum z wie in Lemma 86. Die Unterbäume H_1, H_3, ..., H_{2h-1} haben ihre Zentren oberhalb von z. Wir trennen sie von H_{2h}, indem wir h Kanten aufbrechen. Die abgetrennten Teile haben offensichtlich die Form H_s, $s < 2h$. Bei den Unterbäumen H_0, H_2, ..., H_{2h-2}, mit Zentren unterhalb von z, wenden wir jeweils die Induktionsannahme an, d.h. wir erzeugen H_0^-, H_2^-, ..., H_{2h-2}^-. Als Ergebnis erhalten wir H_{2h}^-.

Beweis (Forts.):

Damit gilt für die Zahl der aufzubrechenden Kanten $K^-(2h)$ zur Herstellung von H^-_{2h}:

$$K^-(2h) = h + \sum_{i=0}^{h-1} K^-(2i) \overset{I.A.}{=} h + \sum_{i=0}^{h-1}(2^i - 1) = \sum_{i=0}^{h-1} 2^i = 2^h - 1.$$

Für die Zahl $E^-(2h)$ der Elemente in H^-_{2h} gilt:

$$E^-(2h) = 1 + \sum_{i=0}^{h-1} E^-(2i) \overset{I.A.}{=} 1 + \sum_{i=0}^{h-1} 2^i = 1 + \underbrace{\sum_{i=1}^{h} 2^{i-1}}_{2^h-1} = 2^h.$$

Beweis (Forts.):

2. Sei $r = 2h + 1$, $h > 0$. Wir betrachten die Partitionierung von H_{2h+1} mit Zentrum z wie in Lemma 86. Die Unterbäume H_0, H_2, ..., H_{2h} haben ihre Zentren unterhalb von z. Wir trennen sie von H_{2h+1}, indem wir $h + 1$ Kanten aufbrechen. Die abgetrennten Teile haben offensichtlich die Form H_s, $s < 2h + 1$. Bei den Unterbäumen H_1, H_3, ..., H_{2h-1}, mit Zentren oberhalb von z, wenden wir jeweils die Induktionsannahme an, d.h. wir erzeugen H_1^+, H_3^+, ..., H_{2h-1}^+. Als Ergebnis erhalten wir H_{2h+1}^+. Damit gilt für die Zahl der aufzubrechenden Kanten $K^+(2h + 1)$ zur Herstellung von H_{2h+1}^+:

Beweis (Forts.):

$$\begin{aligned} K^+(2h+1) &= h+1+\sum_{i=1}^{h} K^+(2(i-1)+1) \\ &\overset{I.A.}{=} h+1+\sum_{i=1}^{h}(2^i-1) = 1+\sum_{i=1}^{h} 2^i \\ &= 1+\underbrace{\sum_{i=1}^{h+1} 2^{i-1}}_{2^{h+1}-1} - 1 = 2^{h+1}-1. \end{aligned}$$

Für die Zahl $E^+(2h+1)$ der Elemente in H^+_{2h+1} gilt:

$$E^+(2h+1) = 1+\sum_{i=1}^{h} E^+(2(i-1)+1) \overset{I.A.}{=} 1+\underbrace{\sum_{i=1}^{h} 2^{i-1}}_{2^h-1} = 2^h.$$

□

Beweis von d):

Lemma 88

Falls $k \leq 2^h - 1$, dann kann H_{2h} so zerlegt werden, dass die Komponente des Zentrums genau $2k+1$ Elemente enthält, k davon über und k unter dem Zentrum. Dazu müssen $\leq 3k + 2h$ Kanten entfernt werden. Die entfernten Teile sind von der Form H_s, $s < 2h$.

Beweis:

Betrachte die Binärdarstellung von
$k = k_0 2^0 + k_1 2^1 + \cdots + k_{h-1} 2^{h-1}$ und die Partitionierung von H_{2h} mit Zentrum z wie in Lemma 86.

Beweis (Forts.):

Für jedes i mit $k_i = 1$, betrachte H_{2i+1} aus der Sequenz H_1, H_3, ..., H_{2h-1} von Unterbäumen, deren Zentren oberhalb von z liegen, und schneide alle Elemente aus H_{2i+1}, die kleiner als sein Zentrum sind (bilde also H^+_{2i+1}). Dazu müssen höchstens $2k$ Kanten aufgebrochen werden, denn jedes $k_i = 1$ steht für 2^i in k, kostet aber nach Lemma 87 $K^+(2i+1) = 2^{i+1} - 1$ Kanten, also:

$$\sum_{i=0}^{h-1} k_i K^+(2i+1) \leq 2k\,.$$

Für jedes i mit $k_i = 0$, schneide H_{2i+1} ganz weg. Dabei werden $\leq h$ Kanten aufgebrochen. Genau k Elemente oberhalb z bleiben zurück, da jedes $k_i = 1$ für 2^i in k steht, und ein H^+_{2i+1} genau $E^+(2i+1) = 2^i$ Elemente enthält, also:

$$\sum_{i=0}^{h-1} k_i E^+(2i+1) = k\,.$$

Beweis (Forts.):

Für jedes i mit $k_i = 1$, betrachte H_{2i} aus der Sequenz H_0, H_2, ..., H_{2h-2} von Unterbäumen, deren Zentren unterhalb von z liegen, und schneide alle Elemente aus H_{2i}, die größer als sein Zentrum sind (bilde also H_{2i}^-). Dazu müssen höchstens $k-1$ Kanten aufgebrochen werden, denn jedes $k_i = 1$ steht für 2^i in k und kostet uns nach Lemma 87 $K^-(2i) = 2^i - 1$ Kanten, also:

$$\sum_{i=0}^{h-1} k_i(2^i - 1) \leq k - 1\,.$$

Für jedes i mit $k_i = 0$, schneide H_{2i} ganz weg. Dabei werden höchstens h Kanten aufgebrochen. Genau k Elemente unterhalb von z bleiben zurück, da jedes $k_i = 1$ für 2^i in k steht, und ein H_{2i}^- genau $E^-(2i) = 2^i$ Elemente enthält, also:

$$\sum_{i=0}^{h-1} k_i E^-(2i) = k\,.$$

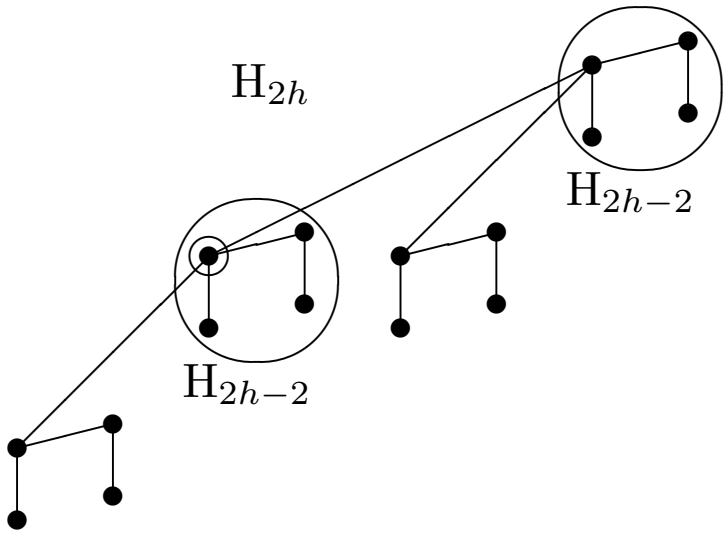
Beweis (Forts.):

Damit ergibt sich für die Gesamtanzahl aufzubrechender Kanten eine obere Schranke von $3k + 2h$. Lemma 87 liefert uns darüber hinaus die gewünschte Aussage über die Form der abgetrennten Teile. □

Beweis von d):

Betrachte H_{2h}.

- „größer“: $H_{2h-1}, H_{2h-3}, \ldots, H_1$
- „kleiner“: $H_{2h-2}, H_{2h-4}, \ldots, H_0$



$U(h) :=$ Anzahl der Elemente in $H_{2h} \geq$ Zentrum:
$U(h) = 2U(h-1) = 2^h;\ U(0) = 1$
$D(h) :=$ Anzahl der Elemente in $H_{2h} \leq$ Zentrum:
$D(h) = 2D(h-1) = 2^h;\ D(0) = 1$

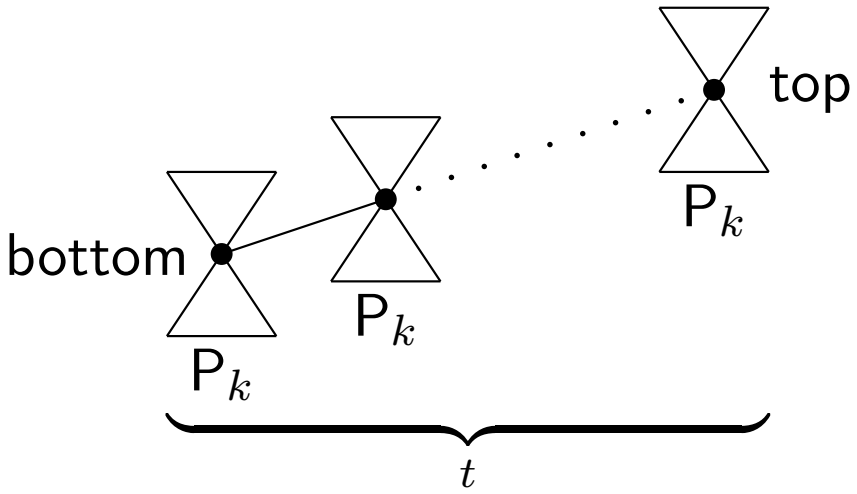
Beweis von d):

Anzahl der Kanten, die entfernt werden müssen:

$$\left.\begin{aligned} C_u(h) &\leq 2 + 2C_u(h-1) \\ &= 2 + 4 + 2^3 + \ldots + 2^h \\ &= 2^{h+1} - 2 \\ C_d(h) &\leq 1 + 2C_d(h-1) \\ &= 2^h - 1 \end{aligned}\right\} C(h) \leq 2^{h+1} - 2 + 2^h - 1 < 3 \cdot 2^h$$

Damit ist der Beweis des Zerlegungslemmas beendet. □

Kette von P_k's:



Gesamtzahl der Elemente:

n

$t(2k+1)$ in den P_k's

$r = n - t(2k+1)$ Rest

Wenn $r < t - 1$, dann wissen wir, dass top größer ist als

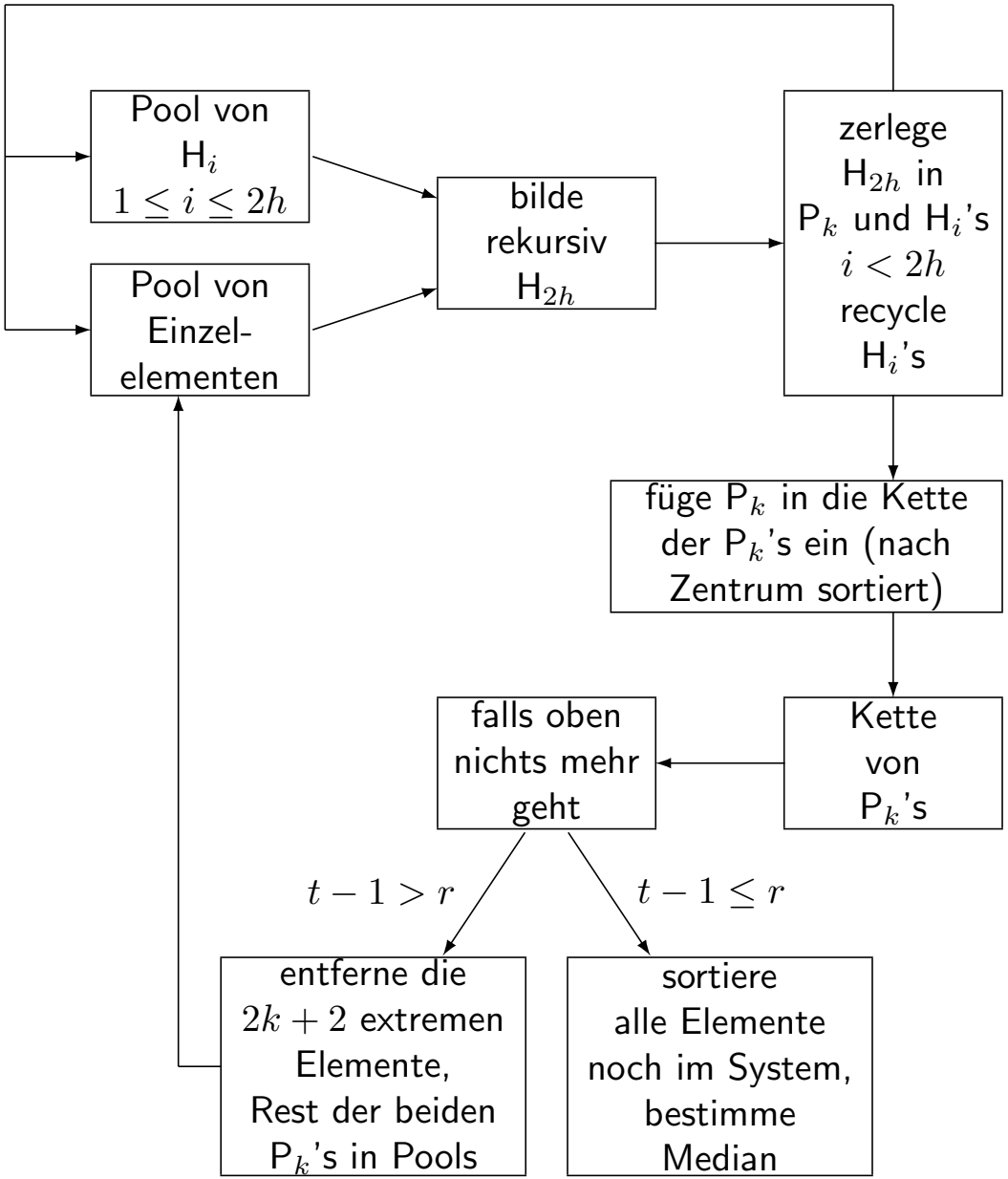
$$k + (t-1)(k+1) > k + (k+1)\left(\frac{n+1}{2k+2} - 1\right) = \frac{n-1}{2}$$

$\Rightarrow$ top $>$ Median (Entsprechendes gilt für das Element bottom)

Setze

$$k := \left\lfloor n^{\frac{1}{4}} \right\rfloor$$

h sdg. $2^{h-1} \leq k < 2^h$


Pool von H_i $1 \leq i \leq 2h$
Pool von Einzel-elementen
bilde rekursiv H_{2h}
zerlege H_{2h} in P_k und H_i's $i < 2h$ recycle H_i's
füge P_k in die Kette der P_k's ein (nach Zentrum sortiert)
Kette von P_k's
falls oben nichts mehr geht
$t-1 > r$
$t-1 \leq r$
entferne die $2k+2$ extremen Elemente, Rest der beiden P_k's in Pools
sortiere alle Elemente noch im System, bestimme Median

Definiere $r :=$ Anzahl der noch im H_{2h}-Produktionsprozess steckenden Elemente (für jedes $i < 2h$ höchstens ein H_i, daher $r \leq \sum_{i=0}^{2h-1} 2^i = 2^{2h} - 1$).

$R :=$ Anzahl der im letzten Schritt zu sortierenden Elemente. Es gilt: $t \leq r + 1$, und damit

$$R = t(2k+1) + r \leq 2^{2h}(2k+1) + 2^{2h} - 1\,.$$

$m :=$ Gesamtzahl der im Algorithmus produzierten P_k's.

$$m = t + 2\frac{n-R}{2(k+1)} = t + \frac{n-R}{k+1}\,.$$

Also: r und t sind $\mathcal{O}(n^{\frac{1}{2}})$, R ist $\mathcal{O}(n^{\frac{3}{4}})$.

Gesamtzahl der vom Algorithmus durchgeführten Vergleiche =

1. Anzahl der Kanten in allen P_k's
2. $+$ Anzahl der Kanten, die gelöscht werden, um die P_k's zu formen
3. $+$ Anzahl der Kanten, die zum Schluss in übriggebliebenen H_i's, $i < 2h$, stecken
4. $+$ Anzahl der Vergleiche, um jedes Zentrum der P_k's in die (sortierte) Kette einzufügen
5. $+$ Anzahl der Vergleiche, um die zum Schluss übriggebliebenenen R Elemente zu sortieren

$$\leq \left(\frac{n-R}{k+1}+t\right)\left[\underbrace{2k}_{1}+\underbrace{3k+2h}_{2}+\underbrace{\log\frac{n}{2k+1}}_{4}\right]+\underbrace{R\log R}_{5}+\underbrace{r}_{3}\,.$$

Mit $k = \left\lfloor n^{\frac{1}{4}} \right\rfloor$, h so, dass $2^{h-1} \leq k < 2^h$, ergibt sich damit

$$\begin{aligned} r &= \mathcal{O}(k^2) \\ t &= \mathcal{O}(k^2) \text{ zum Schluss} \\ R &= \mathcal{O}(k^3), \text{ und damit die} \\ \text{Anzahl der Vergleiche } &= T(n) \leq 5n + o(n)\,. \end{aligned}$$

Verbesserte Version (besseres Zurechtschneiden, bessere Verwertung der Reste):

$$T(n) = 3n + o(n)$$

Bester bekannter Algorithmus (von Dor/Zwick):

$$2,95n + o(n)$$

Literatur:

Arnold Schönhage, Michael Paterson, Nicholas Pippenger:
Finding the median
J. Comput. Syst. Sci. **13**, pp. 184–199 (1976)

Dorit Dor, Uri Zwick:
Selecting the median
SIAM J. Comput. **28**(5), pp. 1722–1758 (1999)

5. Eine untere Schranke für die Medianbestimmung

Satz 89

Jeder (vergleichsbasierte) Medianalgorithmus benötigt im worst-case mindestens $\left\lceil \frac{3n}{2} \right\rceil - 2$ *Vergleiche.*

Beweis:

Gegenspielerargument (adversary argument)

n Elemente, o.B.d.A. n ungerade, alle Elemente paarweise verschieden. Die Menge aller Elemente wird in drei Teilmengen partitioniert: U enthält die Kandidaten für den Median, G enthält Elemente, die sicher größer als der Median sind, und L enthält Elemente, die sicher kleiner als der Median sind. Anfangs sind alle Elemente in U.

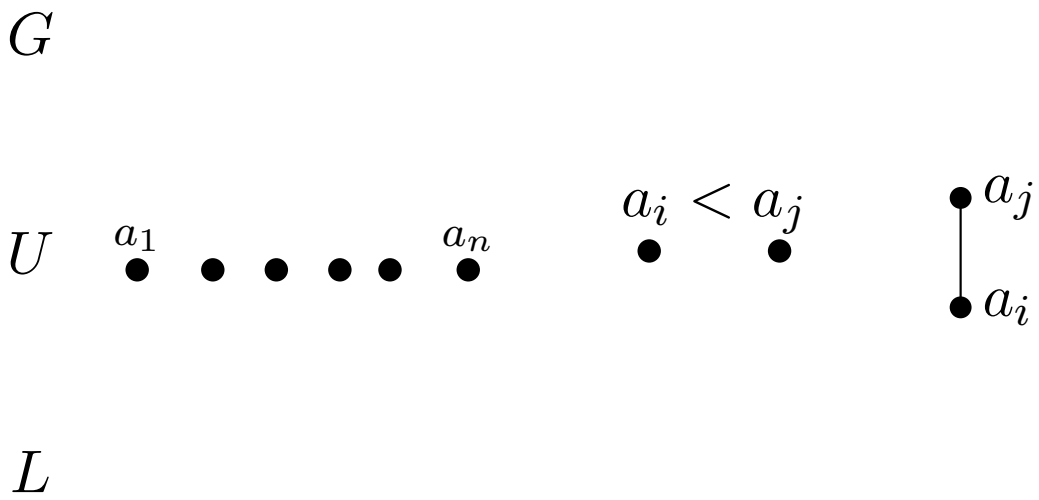
Beweis (Forts.):

Der Algorithmus stellt nun Fragen der Form $a_i < a_j$. Der Gegenspieler gibt konsistente Antworten, die den Algorithmus jedoch dazu zwingen, möglichst viele Fragen stellen zu müssen, bevor die Antwort feststehen kann (d.h. U soll möglichst ungeordnet bleiben).

Durch die (konsistenten!) Antworten des Gegenspielers auf die "$\overset{?}{<}$"-Queries des Algorithmus entsteht ein DAG. Der Gegenspieler hält diesen DAG "einfach", z.B. angenommen $y > z$ und $y > x$, dann soll y „sehr groß" sein $\Rightarrow y \to G$.

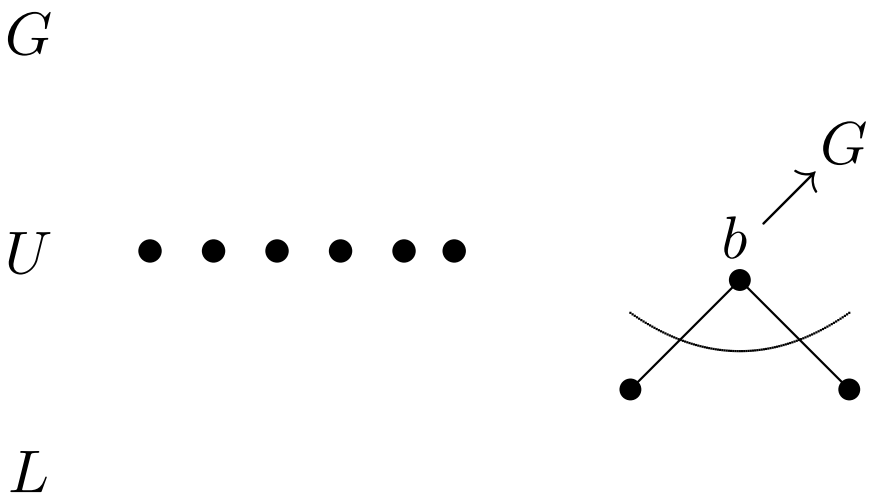
Strategie des Gegenspielers

Beweis (Forts.):



Strategie des Gegenspielers

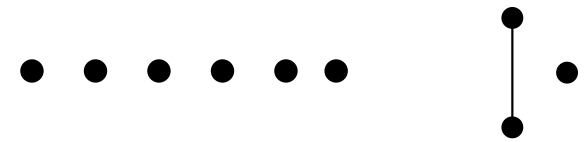
Beweis (Forts.):



Strategie des Gegenspielers

Beweis (Forts.):

$G \quad \leq \frac{n+1}{2} - 1$

U 

$L \quad \leq \frac{n+1}{2} - 1$

Solange ein Element unverglichen ist, ist nicht klar, welches der Median ist.

Beweis (Forts.):

Solange $|L|, |G| < \frac{n+1}{2}$, kann der Algorithmus annehmen, dass der Median in U ist. Solange U mindestens zwei unverglichene Elemente und keine Zusammenhangskomponente mit > 2 Elementen enthält, kann der Algorithmus den Median nicht bestimmen.

Strategie des Gegenspielers

Beweis (Forts.):

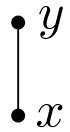
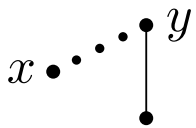
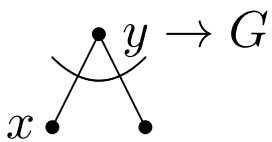
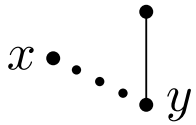
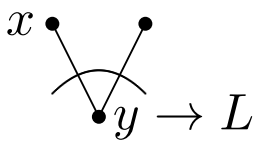
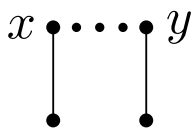
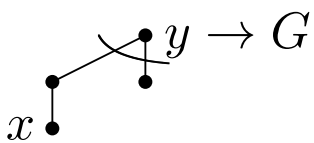
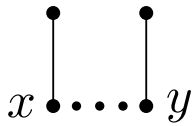
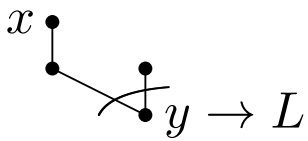
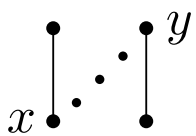
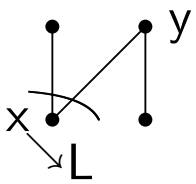
Die Strategie des Gegenspielers hat zwei Phasen.

Erste Phase: Query sei $x \overset{?}{<} y$:

i) $x, y \in G$ bzw. $x, y \in L$: irgendeine konsistente Antwort

ii) $x \in G \wedge y \in L \cup U$ (bzw. $x \in L \wedge y \in G \cup U$):
Antwort: $y < x$ (bzw. $x < y$).

iii) Sei $x, y \in U$.

	Query	Antwort des Gegenspielers	Anzahl der Paare in U	$\lvert U\rvert$	$\lvert L\rvert$	$\lvert G\rvert$
1.			$+1$	—	—	—
2.			-1	-1	0	$+1$
3.			-1	-1	$+1$	0
4.			-1	-1	0	$+1$
5.			-1	-1	$+1$	0
6.			-1	-1	$+1$	0

Die erste Phase endet, wenn $|L| = \frac{n-1}{2}$ oder $|G| = \frac{n-1}{2}$. Während der Phase 1 enthält U mindestens zwei (in U) maximale und mindestens zwei (in U) minimale Elemente (bzgl. des DAGs). $\Rightarrow$ Während Phase 1 kann der Algorithmus den Median mit Sicherheit nicht bestimmen.

Der Gegenspieler beginnt mit Phase 2, sobald

$$|L| \text{ wird } \frac{n-1}{2} \text{ oder } |G| \text{ wird } \frac{n-1}{2}.$$

O.B.d.A.:

$$|L| = \frac{n-1}{2}$$

Der Gegenspieler zwingt nun den Algorithmus, das minimale Element in U bzgl. der gesamten totalen Ordnung zu bestimmen (da dieses unter den Vorgaben der Median ist).

Beweis (Forts.):

Laufzeitanalyse:

$C :=$ Anzahl der Vergleiche

$P :=$ Anzahl der Paare in U

i) In der Phase 1 gilt folgende Invariante: $C - P + 2|U| \geq 2n$.
Dies wird durch vollständige Induktion gezeigt:
Induktionsanfang: $0 - 0 + 2n \geq 2n$;
Induktionsschritt: Gemäß der Tabelle.

ii) Sei C die Anzahl der Vergleiche am Ende der Phase 1. In der Phase 2 werden noch $\geq |U| - 1 - |P|$ Vergleiche nötig. Die Anzahl der Vergleiche für alle Phasen ist damit

$$\geq C + |U| - 1 - |P| \,.$$

Beweis (Forts.):

Am Ende der Phase 1 gilt ja $C \geq 2n + |P| - 2|U|$ (wg. Invariante).
Damit gilt für die Anzahl der Vergleiche:

$$\begin{aligned} &\geq 2n + |P| - 2|U| + |U| - 1 - |P| = 2n - |U| - 1 \\ &\geq \frac{3}{2}n - \frac{3}{2} = \\ &= \left\lceil \frac{3}{2}n - 2 \right\rceil, \text{ da } |U| \leq \frac{n+1}{2} \end{aligned}$$

□

6. Eine bessere untere Schranke

Satz 90

Sei T ein Entscheidungsbaum für die Bestimmung des i-kleinsten von n verschiedenen Elementen, mit $i^2 \geq \log\left[\binom{n}{i}\frac{1}{n-i+1}\right] + 3$. Dann gilt, wenn

$$p := 2\sqrt{\log\left[\binom{n}{i}\frac{1}{n-i+1}\right] + 3} - 2$$

gesetzt wird,

$$\textit{Höhe}(T) \geq \log\left[\binom{n}{i}\frac{2^{n-p}}{n-i+1}\right] .$$

Bemerkung: $i = \lceil \frac{n}{2} \rceil \rightarrow \binom{n}{\lceil \frac{n}{2} \rceil} = \Theta(\frac{2^n}{\sqrt{n}})$; also erhalten wir eine untere Schranke von Höhe$(T) \geq 2n - o(n)$ für die Bestimmung des Medians.

Beweis:

Der Beweis setzt sich aus einem Gegenspieler- und einem Abzählargument zusammen, um zu zeigen

„T hat viele Blätter.“

Sei A Teilmenge der Schlüssel, $|A| = i$. Wir konstruieren einen Teilbaum T_A von T, so dass alle Blätter von T_A auch Blätter von T sind, und zeigen: T_A hat viele Blätter (nämlich 2^{n-p}). Es gibt $\binom{n}{i}$ Möglichkeiten, A zu wählen, jedes Blatt von T kommt in höchstens $n - i + 1$ T_A's vor.

Beweis (Forts.):

Beobachtungen:

Jedes Blatt w von T liefert folgende Informationen:

$$\begin{aligned}
\text{answer}(w) &\xrightarrow{\text{liefert}} i\text{-kleinstes Element } x \\
\text{little}(w) &\xrightarrow{\text{liefert}} i-1 \text{ Elemente } < x \\
\text{big}(w) &\xrightarrow{\text{liefert}} n-i \text{ Elemente } > x
\end{aligned}$$

Beweis (Forts.):

Wähle A als beliebige Teilmenge der n gegebenen Schlüssel, mit $|A| = i$. Wir geben für den Gegenspieler eine Strategie an, welche dazu führt, dass wir durch Zurechtschneiden aus T einen Baum T_A konstruieren können, so dass gilt:

- T_A ist Binärbaum
- jedes Blatt von T_A ist auch Blatt von T, d.h. durch das Zurechtschneiden entstehen keine zusätzlichen Blätter
- für jedes Blatt w von T_A gilt: little$(w) \subset A$

Beweis (Forts.):

Setze $\bar{A}$ gleich dem Komplement von A, sowie

$$r := \sqrt{\log\left[\binom{n}{i}\frac{1}{n-i+1}\right] + 3}$$

$$s := r - 1$$

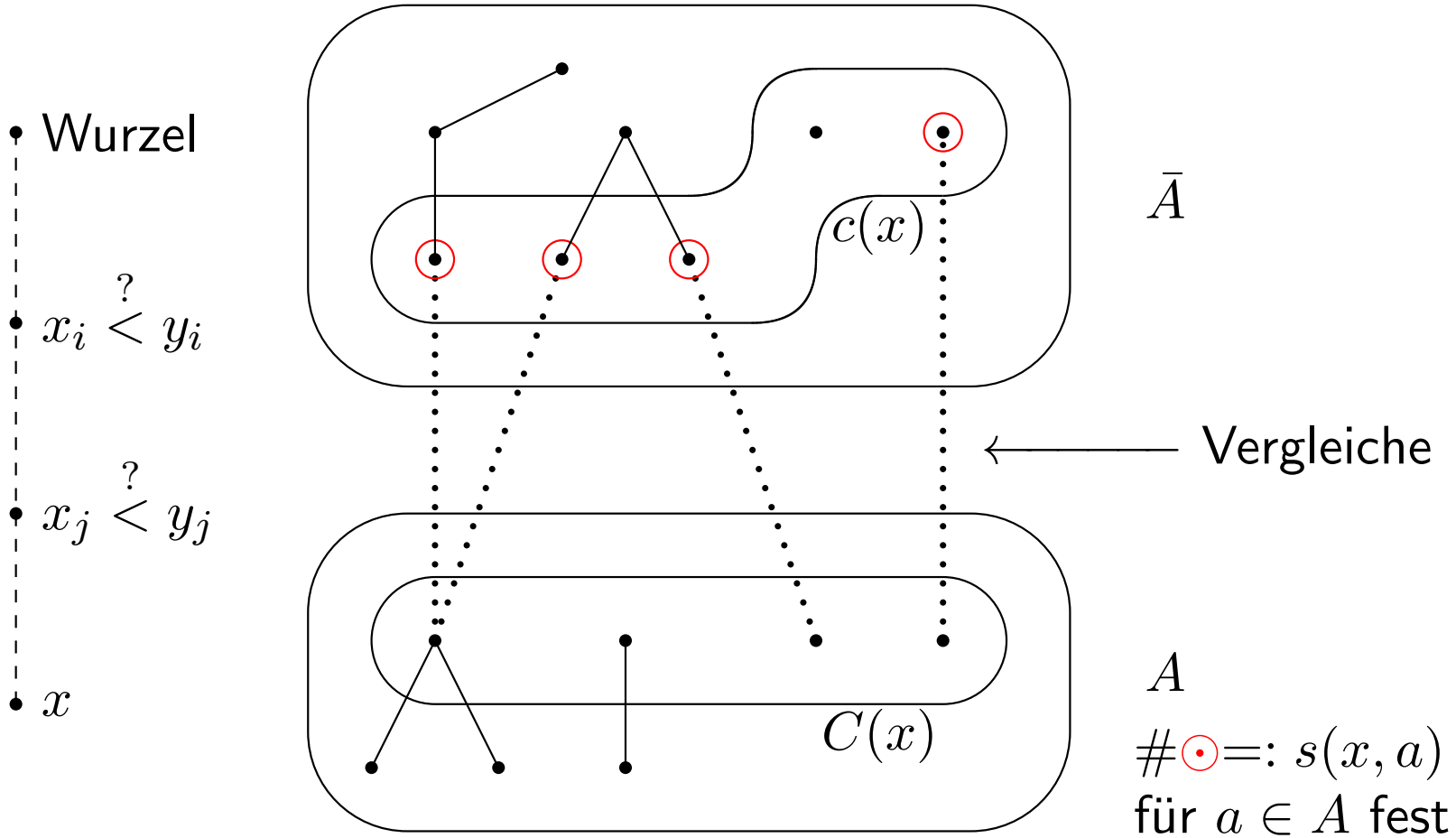
Damit gilt: $p = r + s - 1$.

Die Konstruktion (das "Zurechtstutzen") von T zu T_A erfolgt in zwei Phasen.

Beweis (Forts.):

Erste Phase: Breitensuche von der Wurzel nach unten.
Betrachte Knoten x. Definiere:

- $C(x)$ sind die Elemente $a \in A$, für die es kein $b \in A$ und keinen Knoten auf dem Pfad von der Wurzel von T zu x gibt, an dem a mit b mit dem Ergebnis $a < b$ verglichen wurde.
- $c(x)$ sind entsprechend die im Knoten x bekannten Minima in $\bar{A}$.
- $s(x, a)$ ist, für $a \in A$, die Anzahl der Elemente $\in c(x) \subseteq \bar{A}$, mit denen a auf dem Pfad von der Wurzel zu x verglichen wurde.


Wurzel
$x_i \overset{?}{<} y_i$
$x_j \overset{?}{<} y_j$
x
$c(x)$
$\bar{A}$
Vergleiche
$C(x)$
A
$\#\odot =: s(x,a)$
für $a \in A$ fest

Beweis (Forts.):

Regeln für Phase 1:

Seien a und b die Elemente, die im Knoten x verglichen werden.

1.1: Falls $a \in A$ und $b \in A$, behalte x in T_A bei.

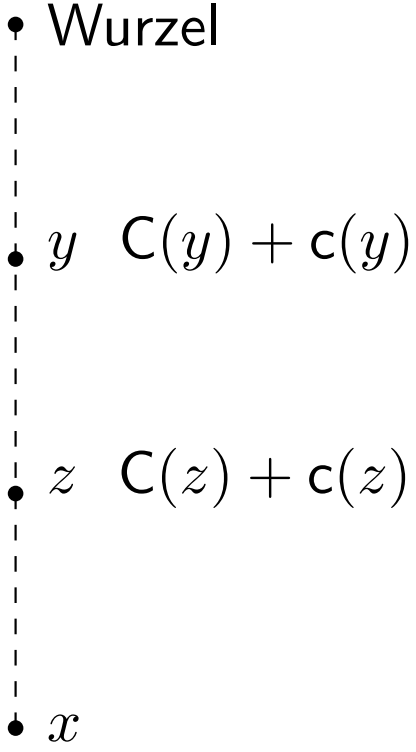
1.2: Falls $a \in \bar{A}$ und $b \in \bar{A}$, behalte x in T_A bei.

1.3: Sei nun o.B.d.A. $a \in A$ und $b \in \bar{A}$. Ersetze den Unterbaum in T mit Wurzel x mit dem Unterbaum, dessen Wurzel das Kind von x ist, das dem Ergebnis „$a < b$“ entspricht (d.h. lasse den Vergleich $a < b$ aus, da gemäß Strategie alle Elemente aus A kleiner als alle Elemente in $\bar{A}$ sind).

Phase 1 läuft, solange $|C(x)| \geq r$. Ein Knoten auf einem Pfad in T von der Wurzel, bei dem $|C(x)|$ erstmals $= r$ wird, heißt kritisch. Jeder Pfad in T von der Wurzel zu einem Blatt enthält genau einen kritischen Knoten.

Beweis (Forts.):

Betrachte in T_A einen Pfad von der Wurzel zu einem kritischen Knoten x. Sei y ein Knoten auf diesem Pfad, z sein Kind. Es gilt:



$$|C(z)| + |c(z)| \geq |C(y)| + |c(y)| - 1$$

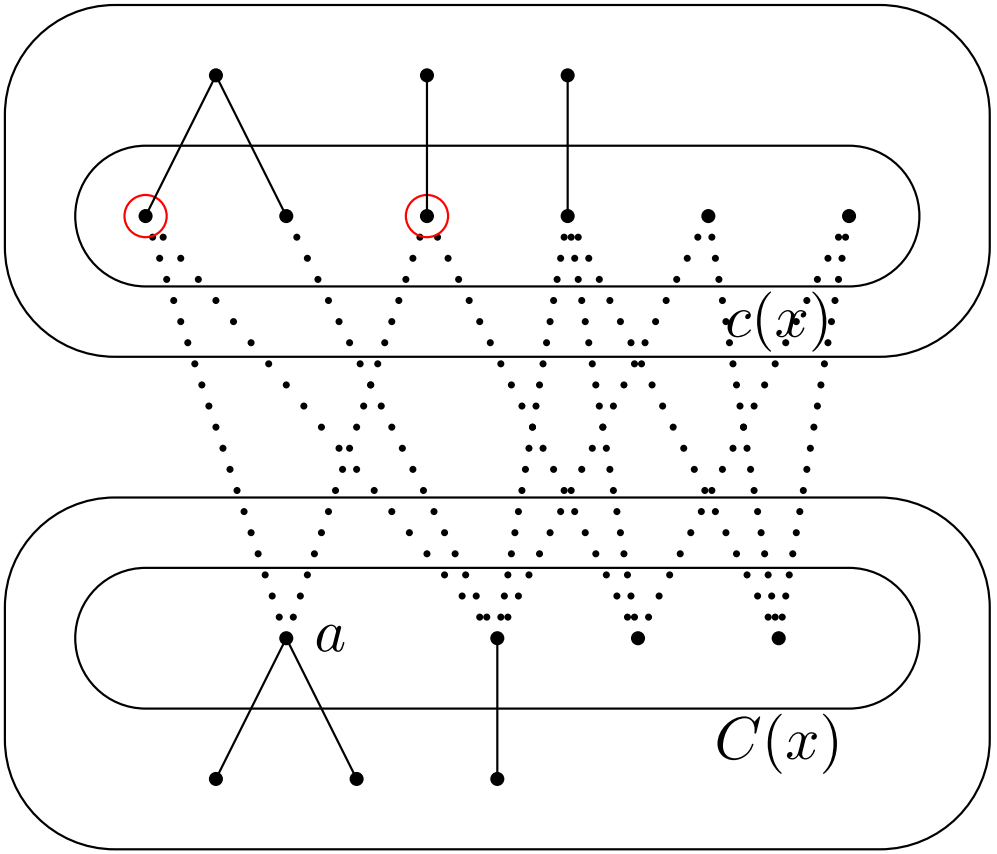
Da $|C(\text{Wurzel})| = |A| = i$ und $|c(\text{Wurzel})| = |\bar{A}| = n - i$, müssen überhalb eines jeden kritischen Knoten x mindestens

$$i - |C(x)| + n - i - |c(x)| = n - r - |c(x)|$$

Vergleiche erfolgen. Von jedem kritischen Knoten abwärts arbeitet der Gegenspieler nach einer Strategie für Phase 2. Sei x ein solcher kritischer Knoten. Dann ist $|C(x)| = r$.

Beweis (Forts.):

Phase 2: Sei $a \in C(x)$ ein Element mit minimalem $s(x,a)$.



$\#\odot = s(x,a)$
für $a \in A$

Beweis (Forts.):

Fall 1: $s(x,a) \geq s$. Betrachte irgendeinen Pfad von der Wurzel durch x zu einem Blatt w. Jeder solche Pfad muss mindestens $n-1$ Vergleiche enthalten, um answer(w) zu verifizieren: $\geq n-i$, für die answer(w) sich (direkt oder indirekt) als das kleinere Element ergibt, und $\geq i-1$, wo es sich als das größere ergibt. Damit sind $\geq (r-1)s$ Vergleiche redundant (nämlich alle die, die zwischen Elementen aus $C(x) \setminus \{\text{answer}(w)\}$ und Elementen in $\bar{A}$ erfolgt sind). Also:

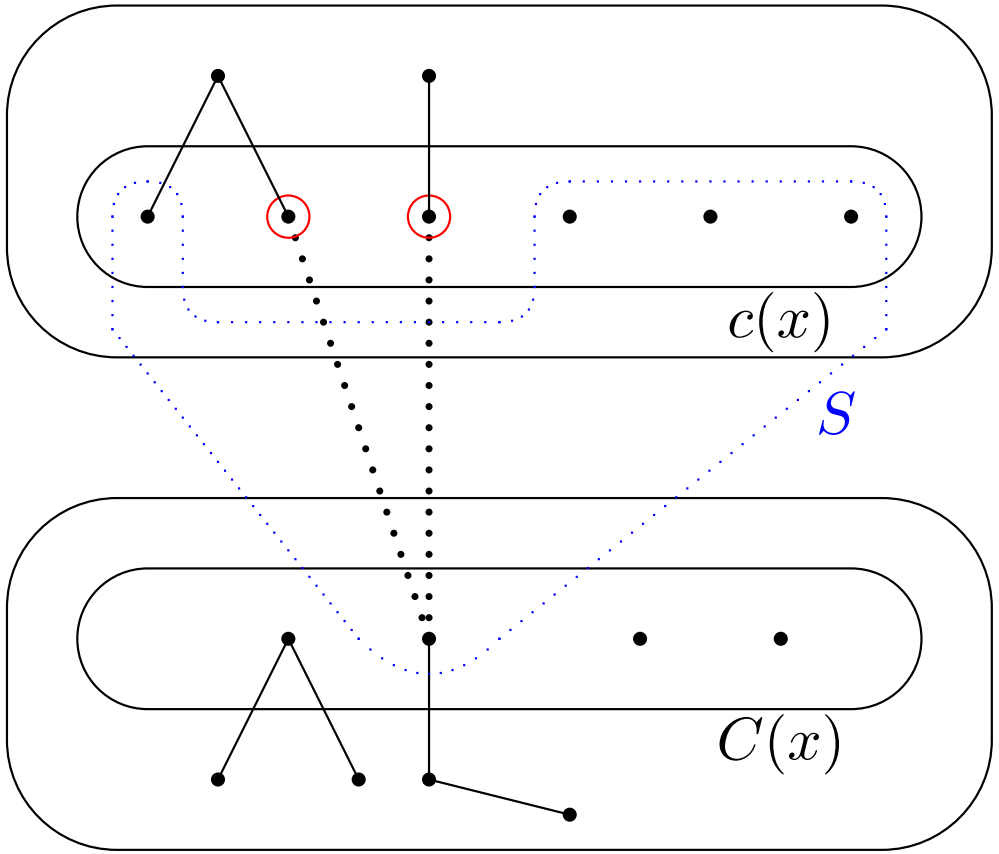
$$\begin{aligned}\text{Höhe}(T) &\geq n-1+s(r-1) = n-r-s-1+(s+1)r \\ &= n-r-s+1+1+\log\left[\frac{\binom{n}{i}}{n-i+1}\right] \\ &> \log\left[\binom{n}{i}\frac{2^{n-p}}{n-i+1}\right].\end{aligned}$$

In diesem Fall folgt also die Behauptung direkt!

Beweis (Forts.):

Fall 2: $s(x,a) < s$.

Fall 2.1: $|c(x)| \geq p$.



$\#\odot = s(x,a)$
für $a \in A$

Beweis (Forts.):

Sei $S := c(x) \cup \{a\} \backslash$ die Elemente, die in s(x, a) gezählt werden. Der Gegenspieler antwortet nun so, dass das Ergebnis das kleinste Element in S wird. Ist w ein Blatt in T_A unter x, so ist little(w)= $A - \{a\}$. Der Entscheidungsbaum T wird also gemäß folgender Regeln gestutzt (sei y der aktuelle Knoten und seien e und f die beiden Elemente, die in y verglichen werden):

2.1: falls $e, f \in S$, dann bleibt y erhalten

2.2: andernfalls sei o.B.d.A. $e \in A \setminus S$ oder $f \in \bar{A} \setminus S$; ersetze y mit seinem Unterbaum durch das Kind von y und dessen Unterbaum, das der Antwort auf $e < f$ entspricht.

Da überhalb des kritischen Knoten x keine zwei Elemente in S verglichen wurden, muss der Unterbaum von T_A unterhalb von x Tiefe $\geq |S| - 1$ haben.

Beweis (Forts.):

Zusammen mit Phase 1 ergibt sich eine Tiefe von T_A:

$$\begin{aligned}
&\geq n - r - |c(x)| + |S| - 1 \\
&\geq n - r - |c(x)| + |c(x)| + 1 - (s-1) - 1 \\
&= n - r - s + 1 \\
&= n - p
\end{aligned}$$

Beweis (Forts.):

Fall 2.2: $|c(x)| < p$. Sei $S := C(x)$.
Die Regeln für Phase 2 sind in diesem Fall so, dass der Algorithmus als Antwort das Maximum von S bestimmt. Damit ergibt sich für die Tiefe von T_A:

$$\begin{aligned} &\geq n - r - |c(x)| + |S| - 1 \\ &\geq n - r - (p-1) + r - 1 \\ &= n - p\,. \end{aligned}$$

Beweis (Forts.):

Insgesamt ergibt sich also: Jeder Pfad in T_A von x zu einem Blatt hat mindestens die Länge $n - p$. Also enthält jedes T_A mindestens 2^{n-p} Blätter (von T).

Alle T_A's zusammen enthalten $\geq \binom{n}{i} 2^{n-p}$ Blätter von T, wobei jedes Blatt von T höchstens $n - i + 1$ mal vorkommt: Sei w Blatt von T, dann ist little(w) eindeutig bestimmt und es muss, falls T_A das Blatt w enthält, little$(w) \subseteq A$ sein.

Für das Element in $A \setminus$ little(w) gibt es $\leq n - i + 1$ Möglichkeiten. Damit ist die Anzahl der Blätter von $T \geq \frac{1}{n-i+1} \binom{n}{i} 2^{n-p}$ und

$$\text{Höhe}(T) \geq \log \left[\binom{n}{i} \frac{2^{n-p}}{n-i+1} \right] .$$

□

Literatur

L. Hyafil:
Bounds for selection
SIAM J. Comput. **5**, pp. 109–114 (1976)

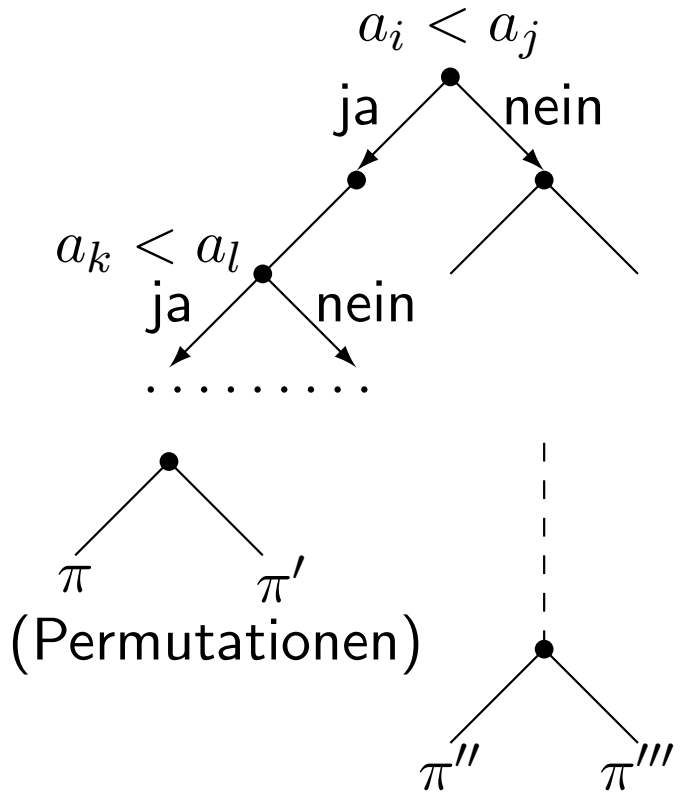
Sam Bent, John W. John:
Finding the median requires $2n$ comparisons
Proc. 17th Annual ACM Symposium on Theory of Computing, pp. 213–216 (1985)

John Welliaveetil John:
A new lower bound for the set-partitioning problem
SIAM J. Comput. **17**, pp. 640–647 (1988)

Dorit Dor, Uri Zwick:
Selecting the median
SIAM J. Comput. **28**(5), pp. 1722–1758 (1999)

7. Untere Schranke für (vergleichsbasiertes) Sortieren

Gegeben n Schlüssel, Queries $a_i \stackrel{?}{<} a_j$. Dies entspricht einem Entscheidungsbaum:



Beobachtung: Jede Permutation $\pi \in S_n$ kommt in mindestens einem Blatt des Entscheidungsbaums vor. Da $|S_n| = n!$, folgt, dass die Tiefe des Entscheidungsbaums

$$\geq \log_2 n! \,.$$

Stirling'sche Approximation: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Also

$$\begin{aligned} \text{Tiefe} \;\geq \log_2\left[\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right] &= n\log_2 n - n\log_2 e + \frac{1}{2}\log_2(2\pi n) \\ &= n\log_2 n - \mathcal{O}(n)\,. \end{aligned}$$

Satz 91

Jeder vergleichsbasierte Sortieralgorithmus benötigt für das Sortieren von n Schlüsseln mindestens

$$n \log_2 n - \mathcal{O}(n)$$

Vergleiche.

8. Bucketsort im Schnitt

Gegeben seien n zufällig und gleichverteilt gewählte Schlüssel im Intervall $]0, 1]$. Um diese zu sortieren:

1. Initialisiere Buckets $b_1, \ldots, b_n$
2. Lege a_i in Bucket $\lceil n \cdot a_i \rceil$
3. Sortiere Schlüssel innerhalb eines jeden Buckets
4. Konkateniere die Buckets

Satz 92

Wird zum Sortieren ein Sortieralgorithmus mit einer Zeitkomplexität von $\mathcal{O}(n^2)$ verwendet, dann hat obiger Bucketsort-Algorithmus im Durchschnitt eine lineare Laufzeit.

Beweis:

Sei n_i die Anzahl der Elemente im Bucket b_i (nach Schritt 2). Die Schritte 1,2 und 4 benötigen zusammen Zeit $\mathcal{O}(n)$. Die Zeit für Schritt 3 beträgt:

$$\mathcal{O}(\sum_{i=1}^{n} n_i^2)\,.$$

Die Erwartungswert für $\sum n_i^2$ lässt sich folgendermaßen abschätzen:

$$\mathbb{E}\left[\sum_{i=1}^{n} n_i^2\right] \leq c \sum_{1\leq j<k\leq n} \Pr[a_j \text{ und } a_k \text{ enden im gleichen Bucket}]$$

$$= c\left(\sum_{1\leq j<k\leq n} \frac{1}{n}\right) = \mathcal{O}(n)\,.$$

□

9. Quicksort

Wie bei vielen anderen Sortierverfahren (Bubblesort, Mergesort, usw.) ist auch bei Quicksort die Aufgabe, die Elemente eines Array $a[1..n]$ zu sortieren.

Quicksort ist ein Divide-and-Conquer-Verfahren.

Divide: Wähle ein Pivot-Element p (z.B. das letzte) und partitioniere $a[l..r]$ gemäß p in zwei Teile $a[l..i-1]$ und $a[i+1..r]$ (durch geeignetes Vertauschen der Elemente), wobei abschließend $a[i] = p$.

Conquer: Sortiere $a[l..i-1]$ und $a[i+1..r]$ rekursiv.

Algorithmus:

proc qs(a, l, r)
  if $l \geq r$ **then return fi**
  #ifdef Variante 2
    vertausche $a[$random$(l, r)]$ mit $a[r]$
  #endif
  $p := a[r]$
  $i := l$; $j := r$
  repeat
    while $i < j$ **and** $a[i] \leq p$ **do** $i++$ **od**
    while $i < j$ **and** $p \leq a[j]$ **do** $j--$ **od**
    if $i < j$ **then** vertausche $a[i]$ und $a[j]$ **fi**
  until $i = j$
  vertausche $a[i]$ und $a[r]$
  qs$(a, l, i-1)$
  qs$(a, i+1, r)$

Bemerkung:
Der oben formulierte Algorithmus benötigt pro Durchlauf für n zu sortierende Schlüssel n oder mehr Schlüsselvergleiche. Durch geschicktes Einstreuen von **if**-Abfragen kann man in jedem Fall mit $n-1$ Schlüsselvergleichen auskommen.

Komplexität von Quicksort:

- Best-case-Analyse: Quicksort läuft natürlich am schnellsten, falls die Partitionierung möglichst ausgewogen gelingt, im Idealfall also immer zwei gleich große Teilintervalle entstehen, das Pivot-Element ist dann stets der Median.

  Anzahl der Schlüsselvergleiche:

$$\leq \sum_{i=1}^{\log n} (n-1) = (n-1)\log n \approx n \log n$$

- Worst-case-Analyse: Z.B. bei einer aufsteigend sortierten Eingabe.

  Anzahl der Schlüsselvergleiche:

$$\Omega(n^2)$$

- Average-case: Da die Laufzeit von Quicksort sehr stark von den Eingabedaten abhängt, kann man die Frage stellen, wie lange der Algorithmus “im Mittel“ zum Sortieren von n Elementen braucht. Um jedoch überhaupt eine derartige Analyse durchführen zu können, muss man zuerst die genaue Bedeutung von “im Mittel“ festlegen. Eine naheliegende Annahme ist, dass alle möglichen Permutationen der Eingabedaten mit gleicher Wahrscheinlichkeit auftreten.

Satz 93

Die durchschnittliche Anzahl von Schlüsselvergleichen von Quicksort beträgt unter der Annahme, dass alle Permutationen für die Eingabe gleichwahrscheinlich sind, höchstens

$$C_n = 2(n+1)(H_{n+1}-1) \approx 2n\ln n - 0.846n + o(n) \approx 1.386n\log n$$

wobei $H_n := \sum_{i=1}^{n} i^{-1}$ *die* n*-te Harmonische Zahl ist.*

Beweis:

(Variante mit $n-1$ Vergleichen pro Durchlauf)
Sei C_n die Anzahl der Vergleiche bei einem Array der Länge n.
$C_0 = C_1 = 0$.

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

Beweis (Forts.):

Da

i) (in beiden Varianten) das j-kleinste Element bestimmt wird und

ii) auch für die rekursiven Aufrufe wieder alle Eingabepermutationen gleichwahrscheinlich sind:

$$\Rightarrow \qquad C_n = n - 1 + \frac{2}{n}\sum_{j=0}^{n-1} C_j;$$

$$nC_n = n^2 - n + 2\sum_{j=0}^{n-1} C_j;$$

$$(n-1)C_{n-1} = (n-1)^2 - (n-1) + 2\sum_{j=0}^{n-2} C_j;$$

Beweis (Forts.):

$$
\begin{aligned}
nC_n - (n-1)C_{n-1} &= 2n - 1 - 1 + 2C_{n-1}; \\
nC_n &= 2n - 2 + (n+1)C_{n-1}; \qquad / \frac{1}{n(n+1)} \\
\frac{C_n}{n+1} &= 2\frac{n-1}{n(n+1)} + \frac{C_{n-1}}{n} = \\
&= 2\frac{n-1}{n(n+1)} + 2\frac{n-2}{(n-1)n} + \frac{C_{n-2}}{n-1} = \\
&= 2\frac{n-1}{n(n+1)} + \cdots + \frac{C_2}{3} = \\
&= 2\left[\sum_{j=2}^{n} \frac{j-1}{j(j+1)}\right] =
\end{aligned}
$$

Beweis (Forts.):

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

$$= 2\left[H_{n+1} - \frac{3}{2} - \sum_{j=2}^{n}\left(\frac{1}{j} - \frac{1}{j+1}\right)\right] =$$

$$= 2\left[H_{n+1} - \frac{3}{2} - \left(\frac{1}{2} - \frac{1}{n+1}\right)\right] =$$

$$= 2\left[H_{n+1} - 2 + \frac{1}{n+1}\right]$$

$\Rightarrow C_n = 2(n+1)\left(H_{n+1} - 2 + \frac{1}{n+1}\right)$;
Mit $H_n \approx \ln n + 0.57721\ldots + o(1)$
$\Rightarrow C_n \approx 2n\ln n - 4n + o(n) \approx 1.386 n \log n$ □

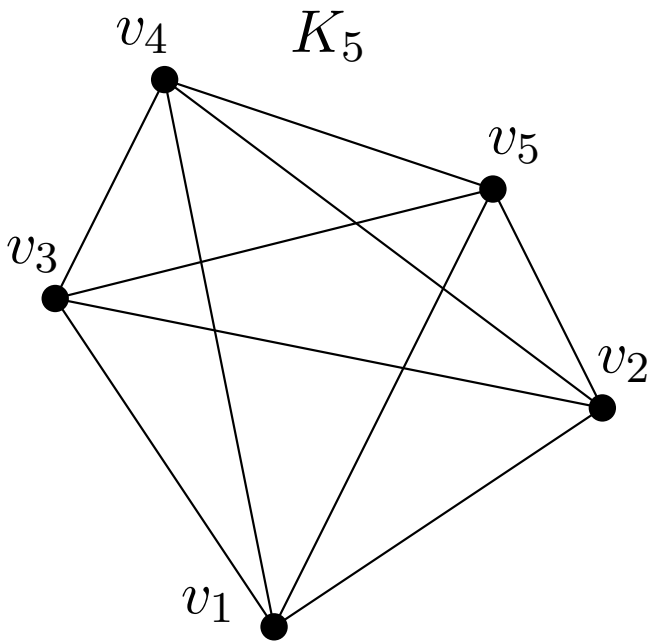
Kapitel IV Minimale Spannbäume

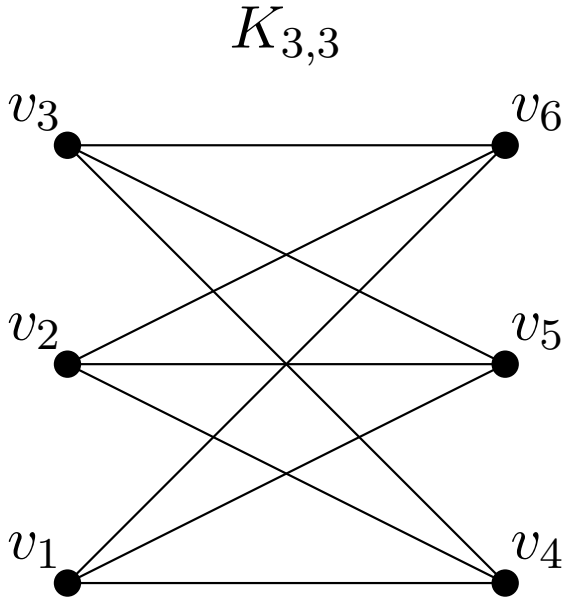
1. Grundlagen

Ein Graph $G = (V, E)$ besteht aus einer Menge V von Knoten und einer Menge E von Kanten. Wir werden nur endliche Knoten- (und damit auch Kanten-) Mengen betrachten. Die Anzahl der Knoten bezeichnen wir mit n ($|V| = n$), die Anzahl der Kanten mit m ($|E| = m$). Eine gerichtete Kante mit den Endknoten u und v wird mit (u, v), eine ungerichtete mit $\{u, v\}$ notiert. Eine Kante $(v, v) \in E$ (bzw. $\{v, v\}$) heißt Schlinge. Falls E eine Multimenge ist, spricht man von Mehrfachkanten. Kanten (u, v) und (v, u) heißen antiparallel. Graphen ohne Schlingen und Mehrfachkanten heißen einfache Graphen (engl. simple). Für einfache ungerichtete Graphen gilt daher:

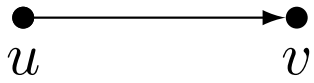
$$E \subseteq \binom{V}{2} := \{X \subseteq V, |X| = 2\}$$

Graphische Darstellung:





Ist $E \subseteq V \times V$, dann heißt G gerichteter Graph (engl. digraph).



Der zu G gehörige ungerichtete Graph ist $G' = (V, E')$. E' erhält man, indem man in E die Richtungen weglässt und Mehrfachkanten beseitigt.
Sei $v \in V$. Unter der Nachbarschaft
$N(v) := \{w; (v, w)$ oder $(w, v) \in E\}$ eines Knotens v versteht man die Menge der direkten Nachbarn von v. Der Grad eines Knotens ist definiert als:

$$\deg(v) = \begin{cases} |N(v)| & ; \text{falls } G \text{ ungerichtet und} \\ \text{indeg}(v) + \text{outdeg}(v) & ; \text{falls } G \text{ gerichtet.} \end{cases}$$

Dabei ist $\text{indeg}(v)$ die Anzahl aller Kanten, die v als Endknoten, und $\text{outdeg}(v)$ die Anzahl aller Kanten, die v als Anfangsknoten haben.

Beobachtung: Für einfache (gerichtete oder ungerichtete) Graphen gilt

$$\sum_{v \in V} \deg(v) = 2|E| \,.$$

Korollar 94

In jedem (einfachen) Graphen ist die Anzahl der Knoten mit ungeradem Grad eine gerade Zahl.

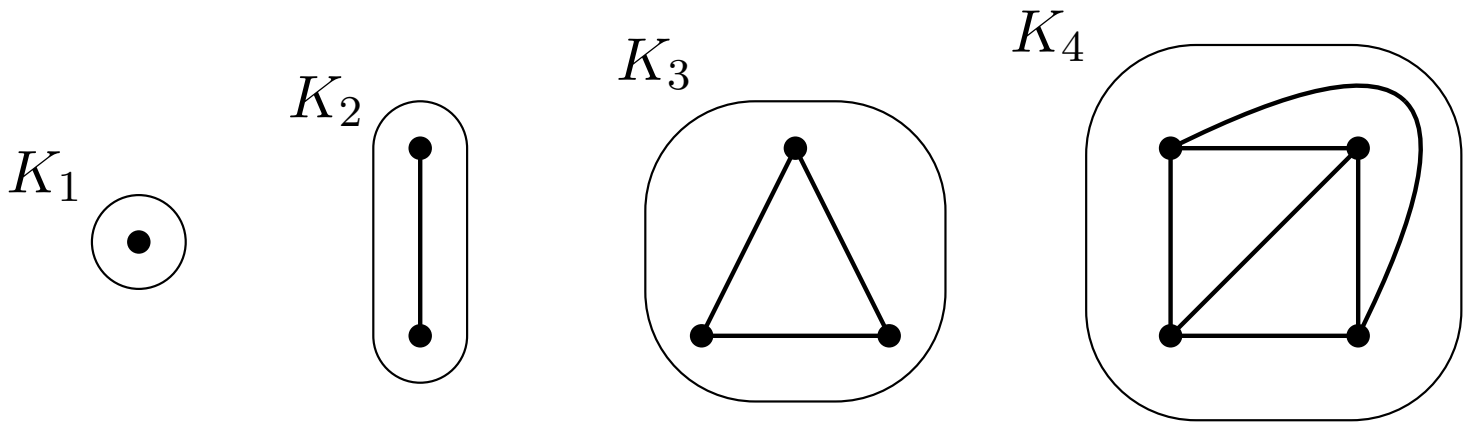
Das Komplement $\bar{G} = (V, \binom{V}{2} \setminus E)$ eines Graphen $G = (V, E)$ besitzt die gleiche Knotenmenge V und hat als Kantenmenge alle Kanten des vollständigen Graphen ohne die Kantenmenge E.

Ein Graph $H = (V', E')$ heißt Teilgraph (aka. Subgraph) von $G = (V, E)$, falls $V' \subseteq V$ und $E' \subseteq E$. H heißt (knoten-) induzierter Teilgraph, falls H Teilgraph von G ist und

$$E' = E \cap \binom{V'}{2}.$$

Ein Kantenzug (oder Pfad) ist eine Folge $e_1 := \{v_0, v_1\}, \ldots, e_l := \{v_{l-1}, v_l\}$. v_0 und v_l sind die Endknoten, l ist die Länge des Kantenzuges. Sind bei einem Pfad alle v_i (und damit erst recht alle e_i) verschieden, so sprechen wir von einem einfachen Pfad. Ein Kantenzug mit $v_l = v_0$ heißt Zykel oder Kreis. Ein Kreis, in dem alle v_i verschieden sind, heißt einfacher Kreis.

Ein (ungerichteter) Graph G heißt zusammenhängend, wenn es für alle $u, v \in V$ einen Pfad gibt, der u und v verbindet. Ein (knoten-)maximaler induzierter zusammenhängender Teilgraph heißt (Zusammenhangs-)Komponente.



Ein Graph G heißt azyklisch, wenn er keinen Kreis enthält. Wir bezeichnen einen solchen ungerichteten Graphen dann als Wald. Ist dieser auch zusammenhängend, so sprechen wir von einem Baum. Ist der Teilgraph $T = (V, E') \subseteq G = (V, E)$ ein Baum, dann heißt T ein Spannbaum von G.

Satz 95

Ist $T = (V, E)$ ein Baum, dann ist $|E| = |V| - 1$.

Beweis:

Induktion über die Anzahl n der Knoten:
$n = 0, 1$: klar.
$n \to n + 1$: Sei $|V| = n + 1$. Da T zusammenhängend ist, ist $\deg(v) \geq 1$ für alle $v \in V$. T muss einen Knoten v mit $\deg(v) = 1$ enthalten, denn ansonsten würde, wie wiederum eine einfache Induktion zeigt, T einen Kreis enthalten. Wende nun die Induktionsannahme auf den durch $V - \{v\}$ induzierten Teilgraphen an. □

Korollar 96

Ein (ungerichteter) Graph G ist zusammenhängend, gdw G einen Spannbaum hat.

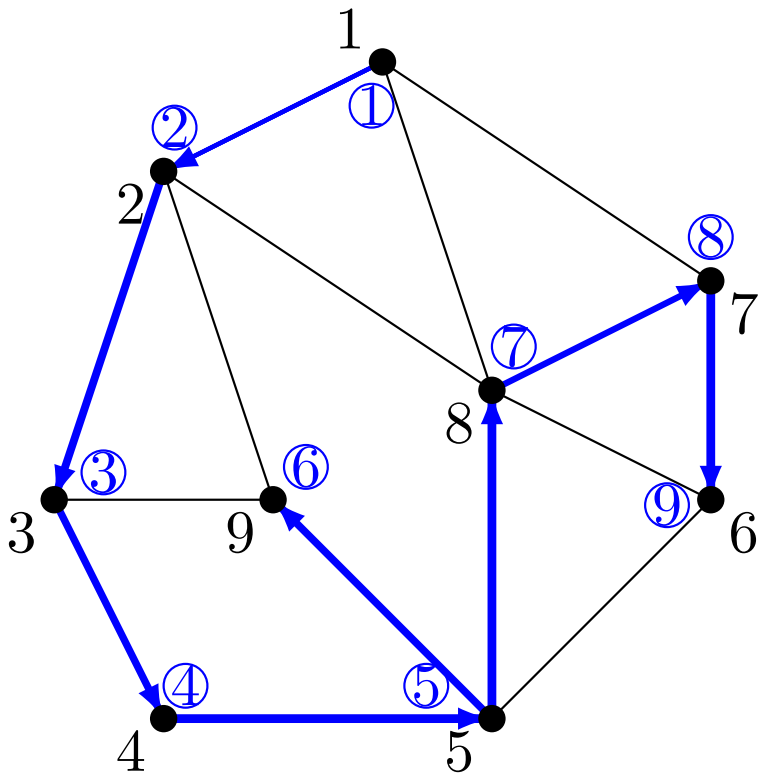
2. Traversierung von Graphen

Sei $G = (V, E)$ ein ungerichteter Graph. Anhand eines Beipiels betrachten wir die zwei Algorithmen DFS (Tiefensuche) und BFS (Breitensuche).

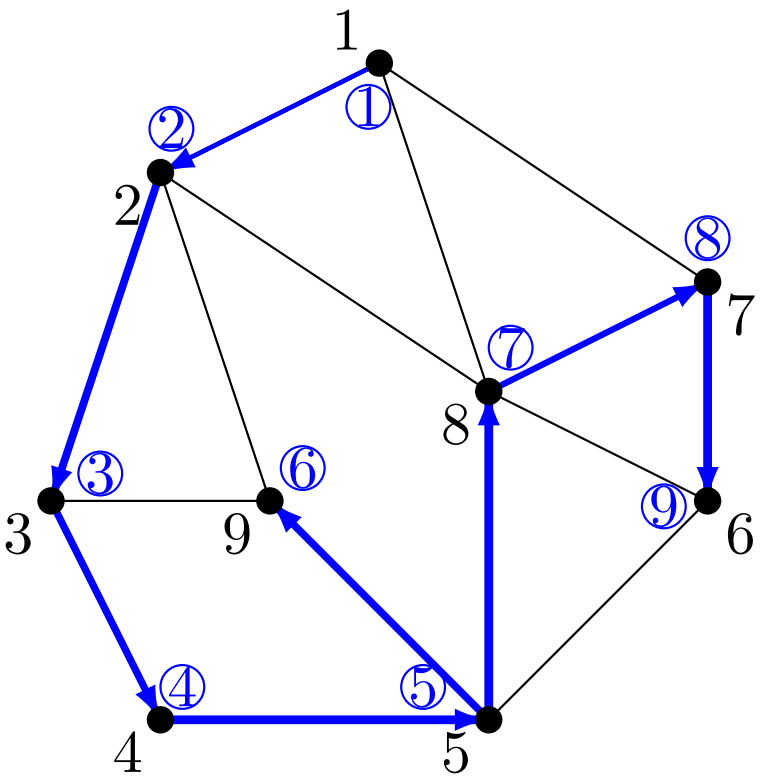
2.1 DFS-Algorithmus

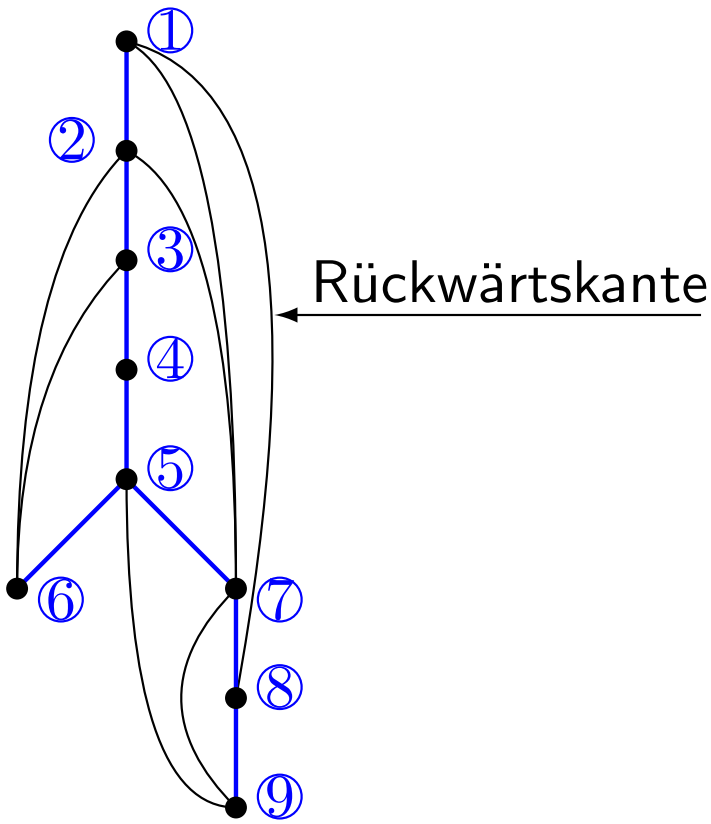
```
while ∃ unvisited v do
    r := pick (random) unvisited node
    push r onto stack
    while stack ≠ ∅ do
        v := pop top element
        if v unvisited then
            mark v visited
            push all neighbours of v onto stack
            perform operations DFS_Ops(v)
        fi
    od
od
```

Beispiel 97



Beobachtung: Die markierten Kanten bilden einen Spannbaum:

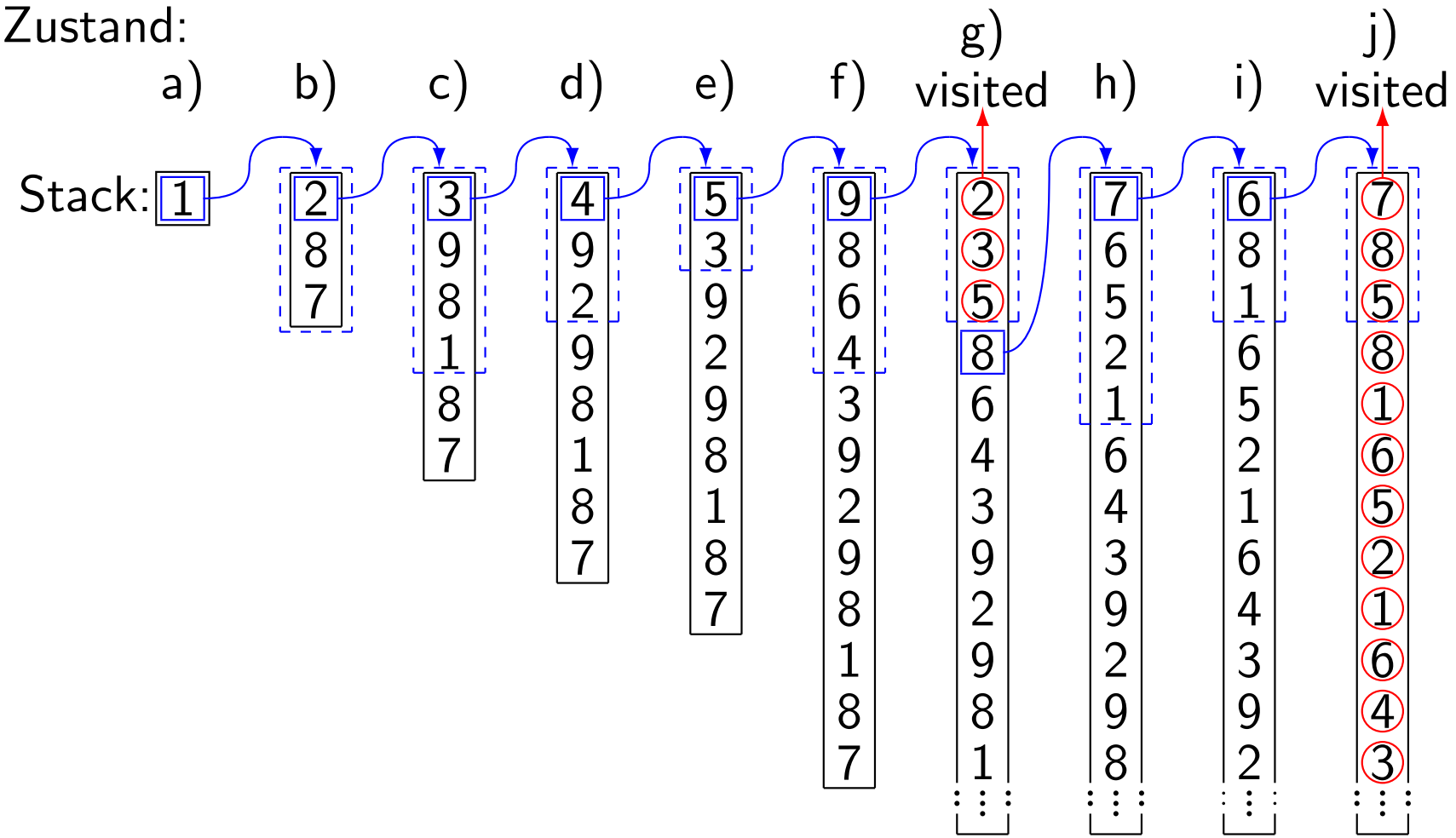




Folge der Stackzustände

: oberstes Stackelement : Nachbarn

: schon besuchte Knoten

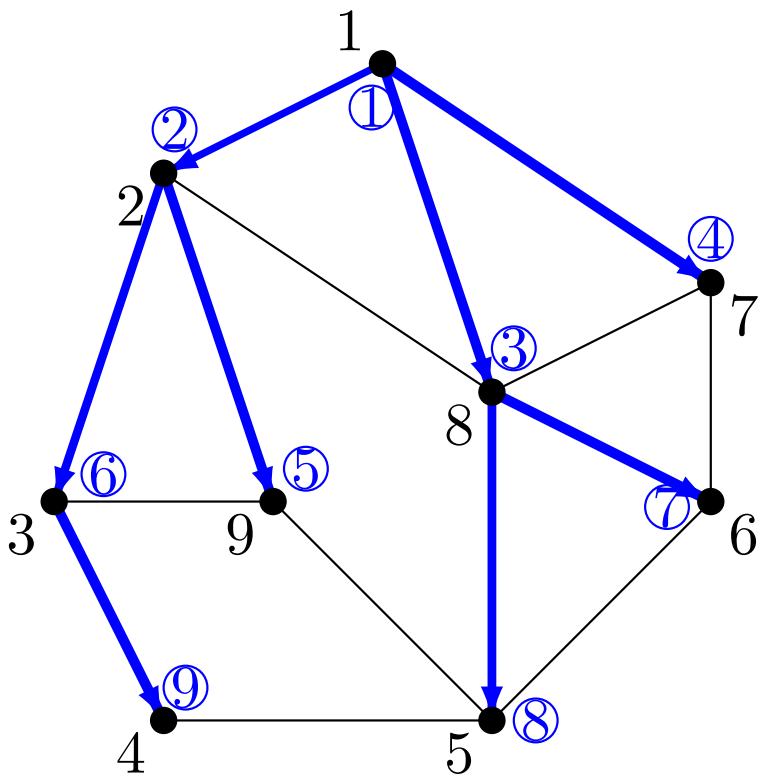


Wir betrachten den Stackzustand:
Im Zustand g) sind die Elemente 2, 3 und 5 als visited markiert (siehe Zustände b), c) und e)). Deswegen werden sie aus dem Stack entfernt, und das Element 8 wird zum obersten Stackelement. Im Zustand j) sind alle Elemente markiert, so dass eins nach dem anderen aus dem Stack entfernt wird.

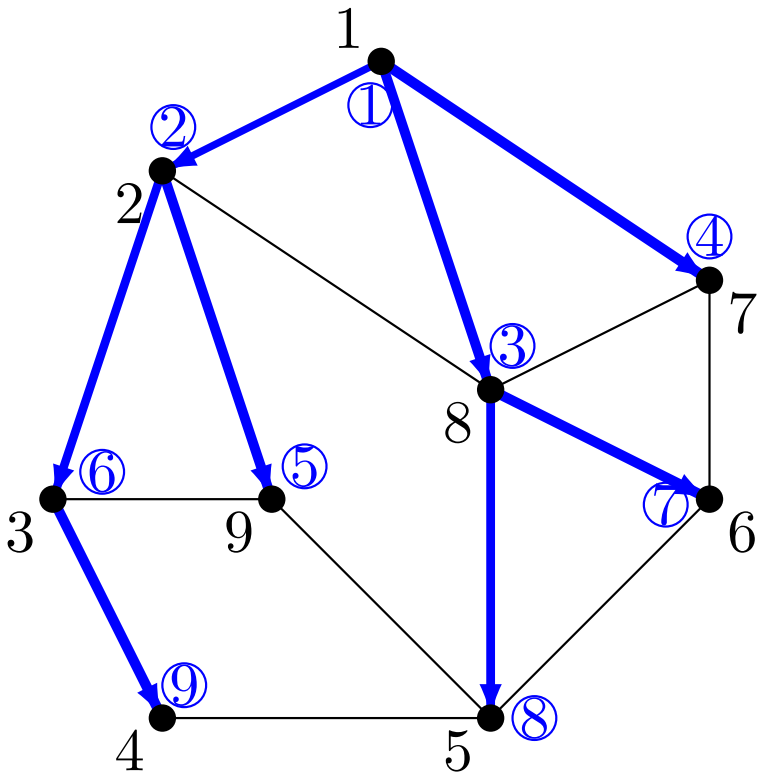
2.2 BFS-Algorithmus

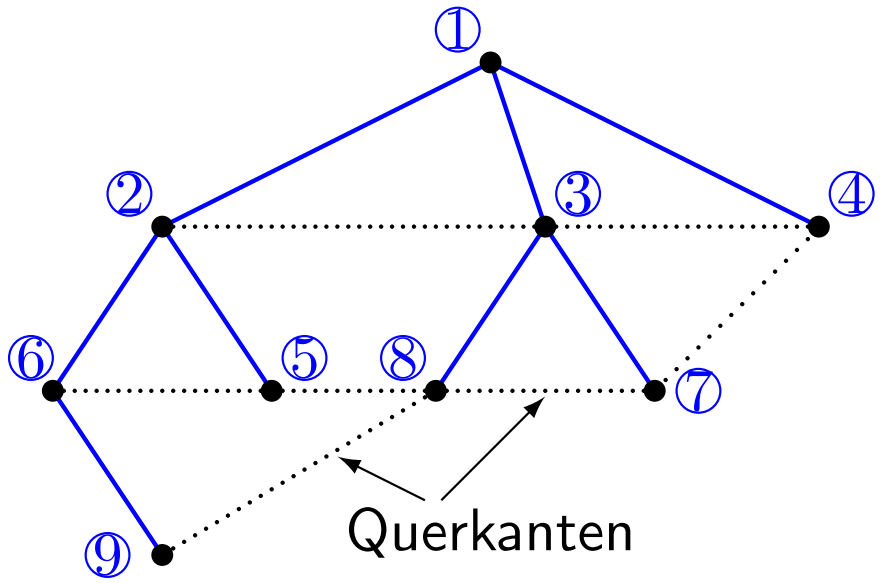
while $\exists$ unvisited v **do**
    $r :=$ pick (random) unvisited node
    push r into (end of) queue
    while queue $\neq \emptyset$ **do**
        $v :=$ remove front element of queue
        if v unvisited **then**
            mark v visited
            append all neighbours of v to end of queue
            perform operations BFS_Ops(v)
        fi
    od
od

Beispiel 98

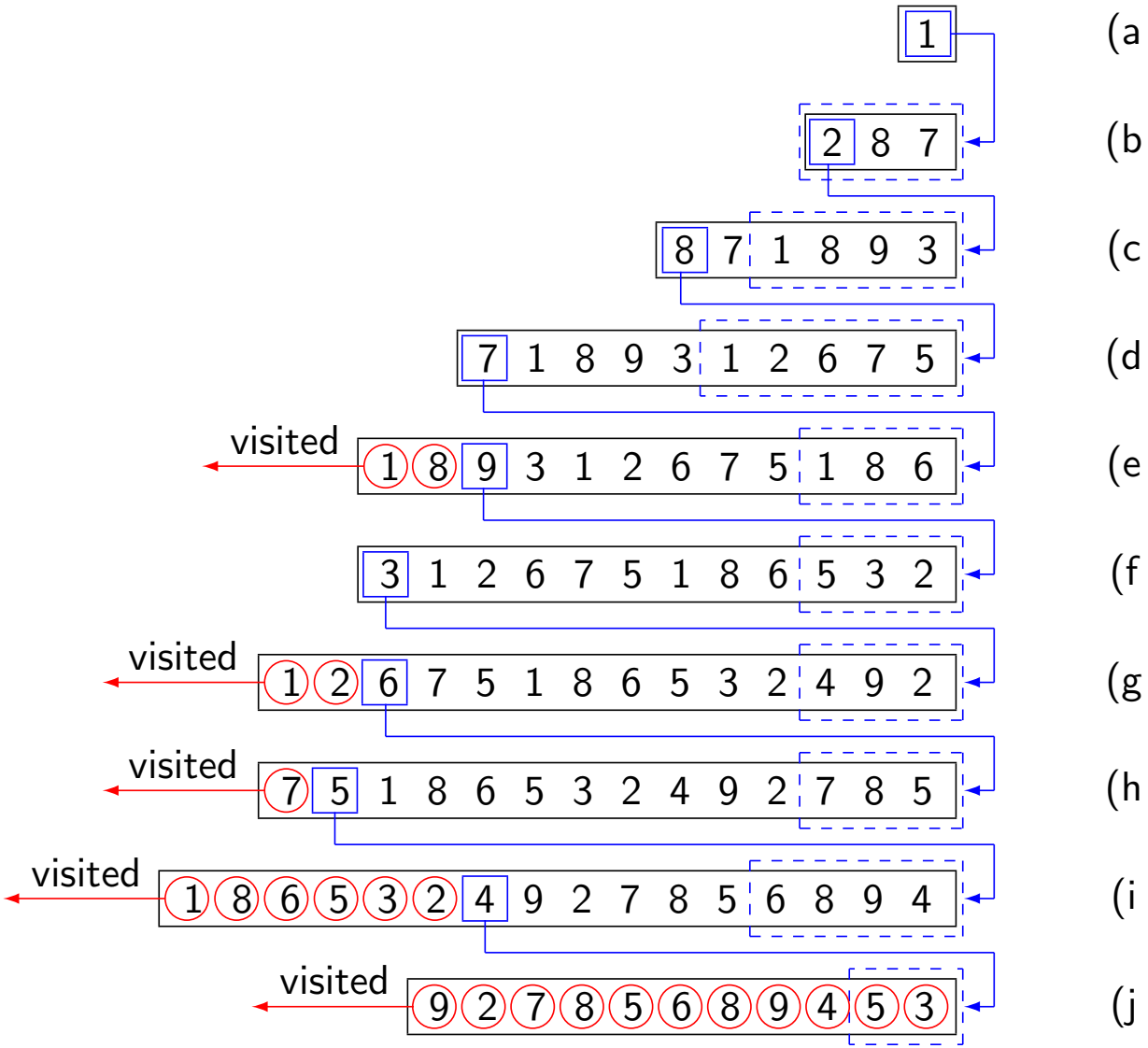


Beobachtung: Die markierten Kanten bilden einen Spannbaum:





Folge der Queuezustände



Wir betrachten die Folge der Queuezustände. Wiederum bedeutet die Notation:

 : vorderstes Queue-Element

 : schon besuchte Knoten

 : Nachbarn

Im Zustand e) sind die Elemente 1 und 8 als visited markiert (siehe Zustände a) und c)). Deswegen werden sie aus der Queue entfernt, und so wird das Element 9 das vorderste Queueelement. Das gleiche passiert in den Zuständen g), h) und i). Im Zustand j) sind alle Elemente markiert, so dass sie eins nach dem anderen aus der Queue entfernt werden.

3. Minimale Spannbäume

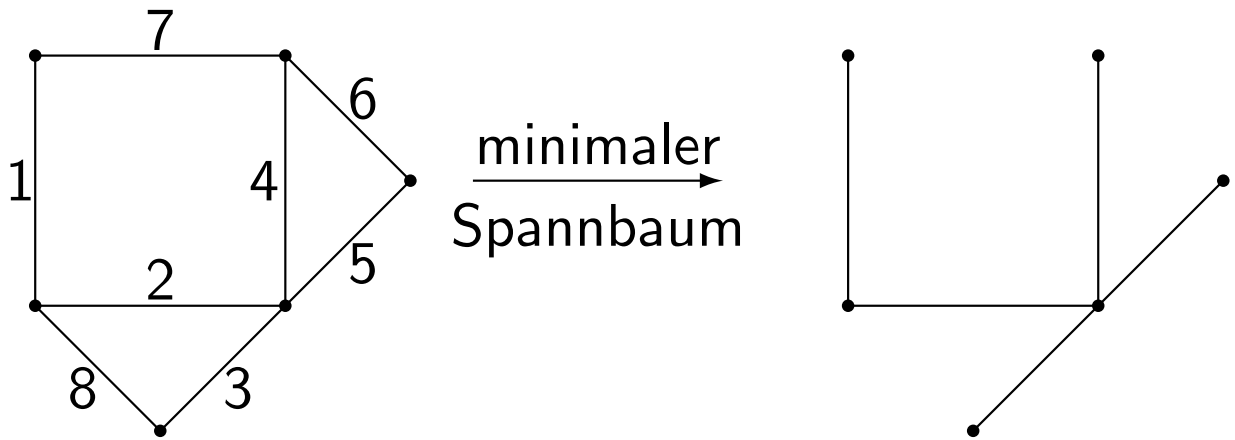
Sei $G = (V, E)$ ein einfacher ungerichteter Graph, der o.B.d.A. zusammenhängend ist. Sei weiter $w : E \to \mathbb{R}$ eine Gewichtsfunktion auf den Kanten von G.
Wir setzen

- $E' \subseteq E$: $w(E') = \sum_{e \in E'} w(e)$,
- $T = (V', E')$ ein Teilgraph von G: $w(T) = w(E')$.

Definition 99
T heißt minimaler Spannbaum (MSB, MST) von G, falls T Spannbaum von G ist und gilt:

$$w(T) \leq w(T') \text{ für alle Spannbäume } T' \text{ von } G.$$

Beispiel 100



Anwendungen:

- Telekom: Verbindungen der Telefonvermittlungen
- Leiterplatinen

3.1 Konstruktion von minimalen Spannbäumen

Es gibt zwei Prinzipien für die Konstruktion von minimalen Spannbäumen (Tarjan):

- „blaue“ Regel
- „rote“ Regel

Satz 101

*Sei $G = (V, E)$ ein zusammenhängender ungerichteter Graph, $w : E \to \mathbb{R}$ eine Gewichtsfunktion, $C = (V_1, V_2)$ ein Schnitt (d.h. $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, $V_1 \neq \emptyset \neq V_2$). Sei weiter $E_C = E \cap (V_1 \times V_2)$ die Menge der Kanten „über den Schnitt hinweg“. Dann gilt: (**„blaue“ Regel**)*

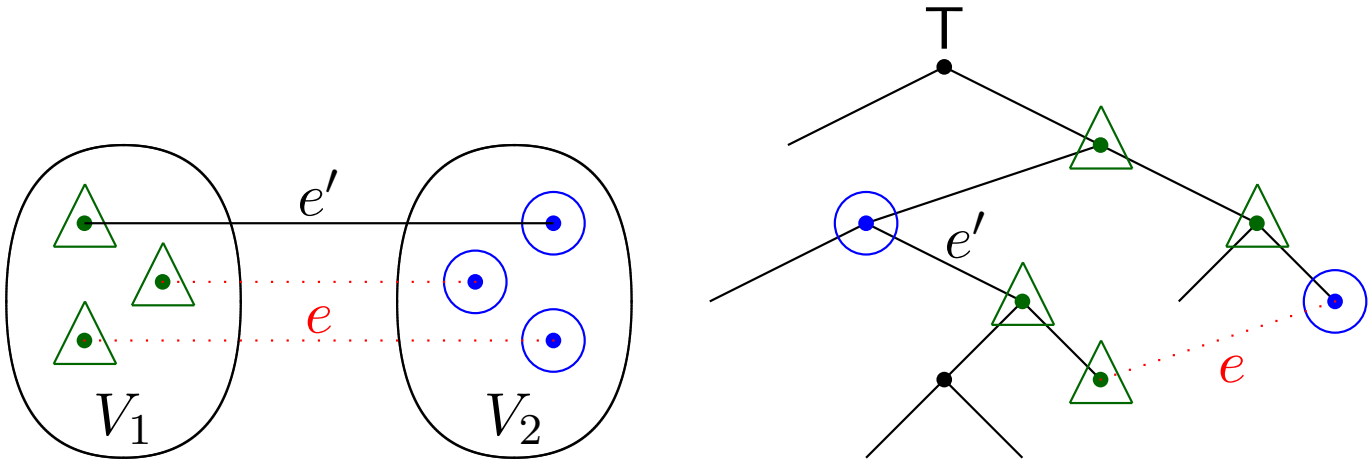
1. *Ist $e \in E_C$ die einzige Kante minimalen Gewichts (über alle Kanten in E_C), dann ist e in jedem minimalen Spannbaum für (G,w) enthalten.*
2. *Hat $e \in E_C$ minimales Gewicht (über alle Kanten in E_C), dann gibt es einen minimalen Spannbaum von (G,w), der e enthält.*

Beweis:

[durch Widerspruch]

1. Sei T ein minimaler Spannbaum von (G, w), sei $e \in E_C$ die minimale Kante. Annahme: $e \notin T$. Da T Spannbaum $\Rightarrow T \cap E_C \neq \emptyset$.
   Sei $T \cap E_C = \{e_1, e_2, \ldots, e_k\}$, $k \geq 1$. Dann enthält $T \cup \{e\}$ einen eindeutig bestimmten Kreis (den sogenannten durch e bzgl. T bestimmten Fundamentalkreis). Dieser Kreis muss mindestens eine Kante $\in E_C \cap T$ enthalten, da die beiden Endpunkte von e auf verschiedenen Seiten des Schnitts C liegen.

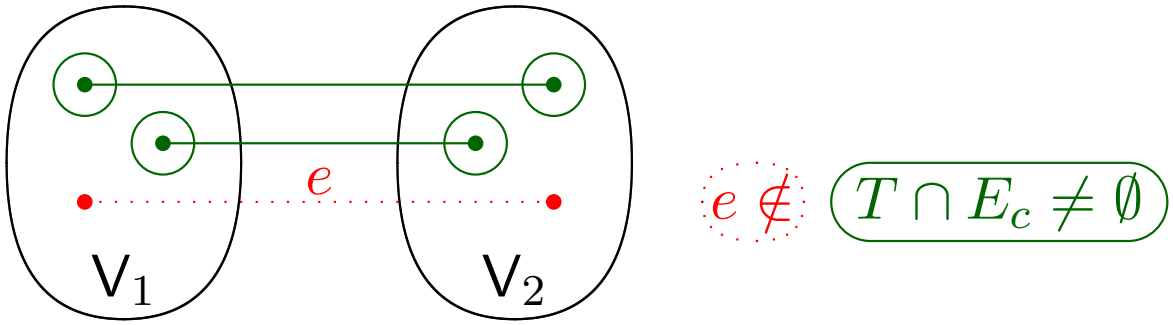
Beweis (Forts.):



Sei $e' \in E_C \cap T$. Dann gilt nach Voraussetzung $w(e') > w(e)$. Also ist $T' := T - \{e'\} \cup \{e\}$ ein Spannbaum von G, der echt kleineres Gewicht als T hat, Widerspruch zu „T ist minimaler Spannbaum“.

Beweis (Forts.):

2. Sei $e \in E_C$ minimal. Annahme: e kommt in keinem minimalen Spannbaum vor. Sei T ein beliebiger minimaler Spannbaum von (G, w).



$e \notin T \cap E_C \neq \emptyset$. Sei $e' \in E_C \cap T$ eine Kante auf dem durch e bezüglich T erzeugten Fundamentalkreis. Dann ist $T' = T - \{e'\} \cup \{e\}$ wieder ein Spannbaum von G, und es ist $w(T') \leq w(T)$. Also ist T' minimaler Spannbaum und $e \in T'$.

□

Satz 102

*Sei $G = (V, E)$ ein ungerichteter, gewichteter, zusammenhängender Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$. Dann gilt: (**„rote“ Regel**)*

1. *Gibt es zu $e \in E$ einen Kreis C in G, der e enthält und $w(e) > w(e')$ für alle $e' \in C \setminus \{e\}$ erfüllt, dann kommt e in keinem minimalen Spannbaum vor.*
2. *Ist $C_1 = e_1, \ldots, e_k$ ein Kreis in G und $w(e_i) = \max\{w(e_j);\ 1 \leq j \leq k\}$, dann gibt es einen minimalen Spannbaum, der e_i nicht enthält.*

Beweis:

1. Nehmen wir an, dass es einen minimalen Spannbaum T gibt, der $e = \{v_1, v_2\}$ enthält. Wenn wir e aus T entfernen, so zerfällt T in zwei nicht zusammenhängende Teilbäume T_1 und T_2 mit $v_i \in T_i$, $i = 1, 2$. Da aber e auf einem Kreis in G liegt, muss es einen Weg von v_1 nach v_2 geben, der e nicht benützt. Mithin gibt es eine Kante $\hat{e} \neq e$ auf diesem Weg, die einen Knoten in T_1 mit T_2 verbindet. Verbinden wir T_1 und T_2 entlang $\hat{e}$, so erhalten wir einen von T verschiedenen Spannbaum $\hat{T}$. Wegen $w(\hat{e}) < w(e)$ folgt $w(\hat{T}) < w(T)$, im Widerspruch zur Minimalität von T.

Beweis (Forts.):

2. Wir nehmen an, e_i liege in jedem minimalen Spannbaum (MSB) von G, und zeigen die Behauptung durch Widerspruch.

   Sei T ein beliebiger MSB von G. Entfernen wir e_i aus T, so zerfällt T in zwei nicht zusammenhängende Teilbäume T_1 und T_2. Da e_i auf einem Kreis $C_1 = e_1, \ldots, e_k$ in G liegt, können wir wie zuvor e_i durch eine Kante e_j des Kreises C_1 ersetzen, die T_1 und T_2 verbindet. Dadurch erhalten wir einen von T verschiedenen Spannbaum $\tilde{T}$, der e_i nicht enthält. Da nach Voraussetzung $w(e_j) \leq w(e_i)$ gilt, folgt $w(\tilde{T}) \leq w(T)$ (und sogar $w(\tilde{T}) = w(T)$, da T nach Annahme ein MSB ist). Also ist $\tilde{T}$ ein MSB von G, der e_i nicht enthält, im Widerspruch zur Annahme, e_i liege in *jedem* MSB von G.

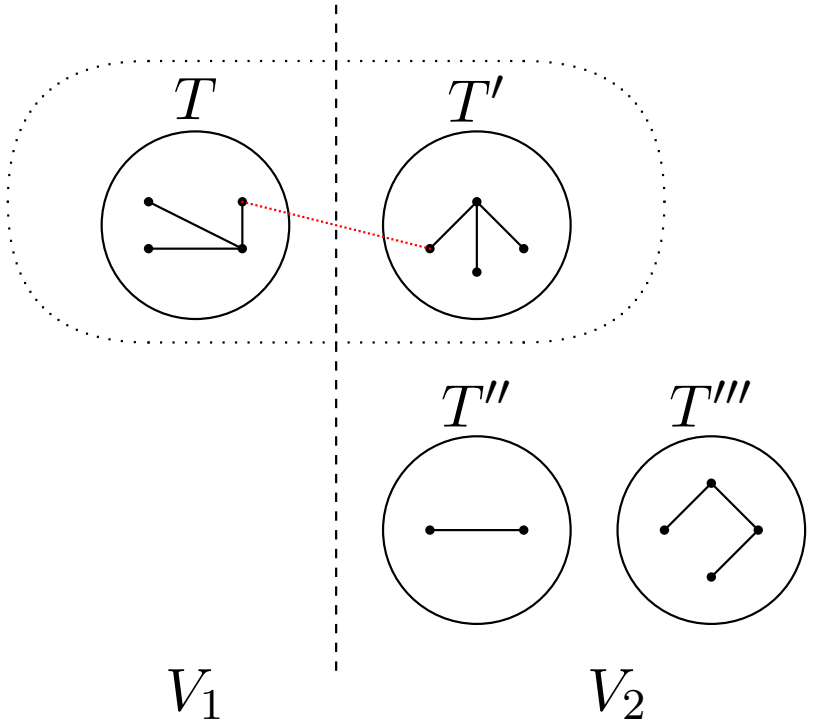
Literatur

Robert E. Tarjan:
Data Structures and Network Algorithms
SIAM CBMS-NSF Regional Conference Series in Applied Mathematics Bd. 44 (1983)

3.2 Generischer minimaler Spannbaum-Algorithmus

Initialisiere Wald F von Bäumen, jeder Baum ist ein singulärer Knoten
(jedes $v \in V$ bildet einen Baum)
while Wald F mehr als einen Baum enthält **do**
    wähle einen Baum $T \in F$ aus
    bestimme eine leichteste Kante $e = \{v, w\}$ aus T heraus
    sei $v \in T$, $w \in T'$
    vereinige T und T', füge e zum minimalen Spannbaum hinzu
od

Generischer MST-Algorithmus



Wald F

3.3 Kruskal's Algorithmus

algorithm Kruskal $(G, w) :=$
  sortiere die Kanten nach aufsteigendem Gewicht in eine Liste L
  initialisiere Wald $F = \{T_i, i = 1, \ldots, n\}$, mit $T_i = \{v_i\}$
  MSB:=$\emptyset$
  for $i := 1$ **to** length(L) **do**
    $\{v, w\} := L_i$
    $x :=$ Baum $\in F$, der v enthält; **co** $x :=$*Find*(v) **oc**
    $y :=$ Baum $\in F$, der w enthält; **co** $y :=$*Find*(w) **oc**
    if $x \neq y$ **then**
      MSB:=MSB $\cup \{\{v, w\}\}$
      Union(x, y) **co** gewichtete Vereinigung **oc**
    fi
  od

Korrektheit: Falls die Gewichte eindeutig sind ($w(\cdot)$ injektiv), folgt die Korrektheit direkt mit Hilfe der "blauen" und "roten" Regel.
Ansonsten Induktion über die Anzahl $|V|$ der Knoten:
Ind. Anfang: $|V|$ klein: $\surd$

Sei $r \in \mathbb{R}$, $E_r := \{e \in E; w(e) < r\}$.
Es genügt zu zeigen:
Sei $T_1, \ldots, T_k$ ein minimaler Spannwald für $G_r := \{V, E_r\}$ (d.h., wir betrachten nur Kanten mit Gewicht $< r$). Sei weiter T ein MSB von G, dann gilt die

Hilfsbehauptung: Die Knotenmenge eines jeden T_i induziert in T einen zusammenhängenden Teilbaum, dessen Kanten alle Gewicht $< r$ haben.

Beweis der Hilfsbehauptung:

Sei $T_i =: (V_i, E_i)$. Wir müssen zeigen, dass V_i in T einen zusammenhängenden Teilbaum induziert. Seien $u, v \in V_i$ zwei Knoten, die in T_i durch eine Kante e verbunden sind. Falls der Pfad in T zwischen u und v auch Knoten $\notin V_i$ enthält (also der von V_i induzierte Teilgraph von T nicht zusammenhängend ist), dann enthält der in T durch Hinzufügen der Kante e entstehende Fundamentalkreis notwendigerweise auch Kanten aus $E \setminus E_r$ und ist damit gemäß der "roten" Regel nicht minimal! Da T_i zusammenhängend ist, folgt damit, dass je zwei Knoten aus V_i in T immer durch einen Pfad verbunden sind, der nur Kanten aus E_r enthält.

Zeitkomplexität: (mit $n = |V|, m = |E|$)

Sortieren	$m \log m = \mathcal{O}(m \log n)$
$\mathcal{O}(m)$ *Find*-Operationen	$\mathcal{O}(m)$
$n - 1$ *Unions*	$\mathcal{O}(n \log n)$

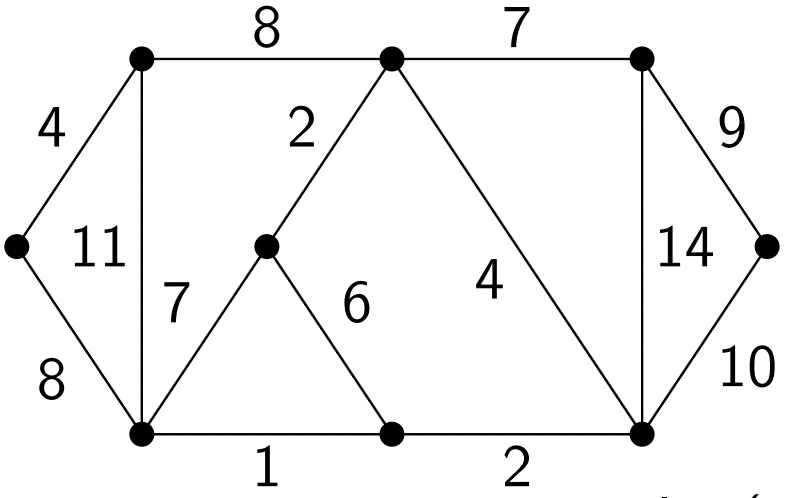
Satz 103

Kruskal's MSB-Algorithmus hat die Zeitkomplexität $\mathcal{O}((m + n) \log n)$.

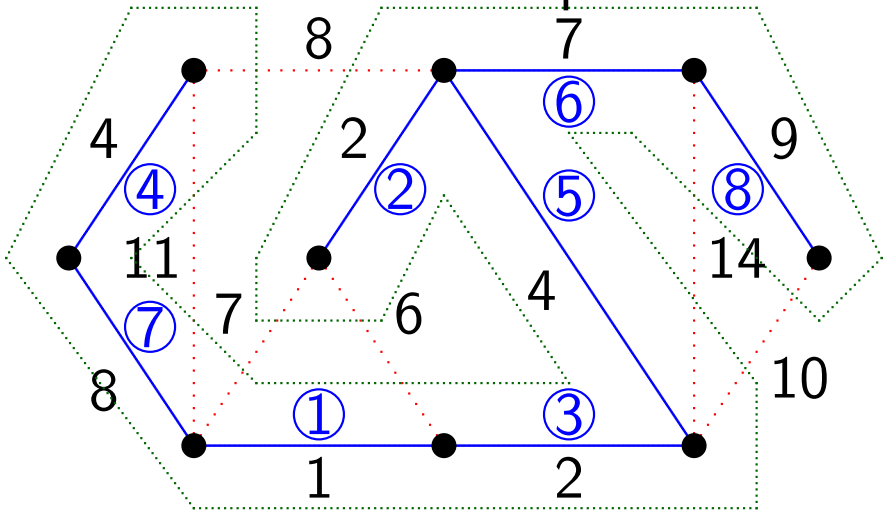
Beweis:

s.o. □

Beispiel 104 (Kruskals Algorithmus)



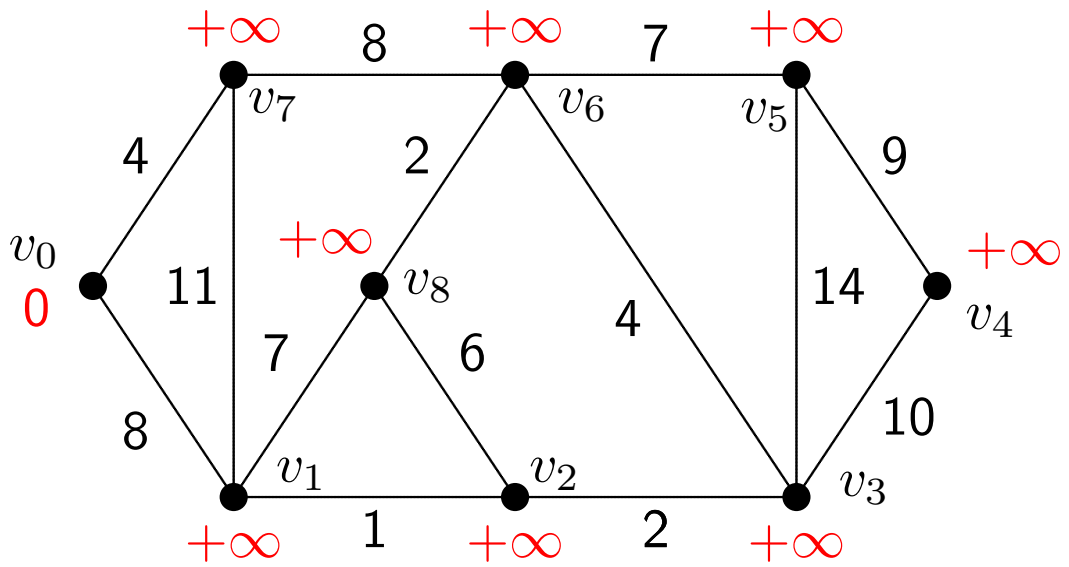
minimaler (nicht eindeutiger) Spannbaum



3.4 Prim's Algorithmus

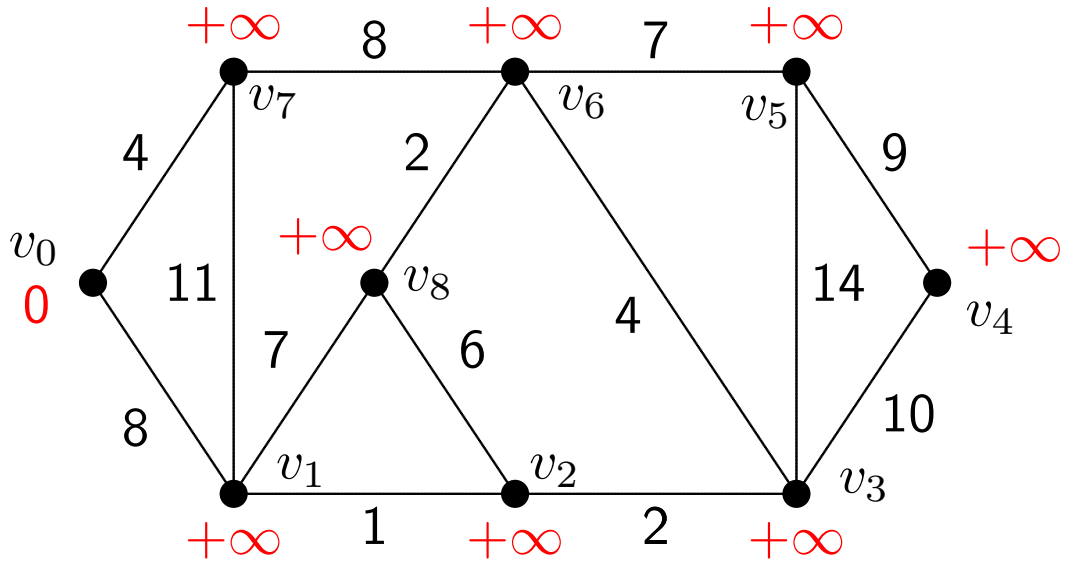
algorithm PRIM-MSB $(G, w) :=$
  initialisiere Priority Queue PQ mit Knotenmenge V und
    Schlüssel $+\infty$, $\forall v \in V$
  wähle Knoten r als Wurzel (beliebig)
  Schlüssel $k[r] := 0$
  Vorgänger$[r] :=$ **nil**
  while $PQ \neq \emptyset$ **do**
    $u :=$ *ExtractMin*(PQ)
    for alle Knoten v, die in G zu u benachbart sind **do**
      if $v \in PQ$ **and** $w(\{u, v\}) < k[v]$ **then**
        Vorgänger$[v] := u$
        $k[v] := w(\{u, v\})$
      fi
    od
  od

Beispiel 105 (Prim's Algorithmus)

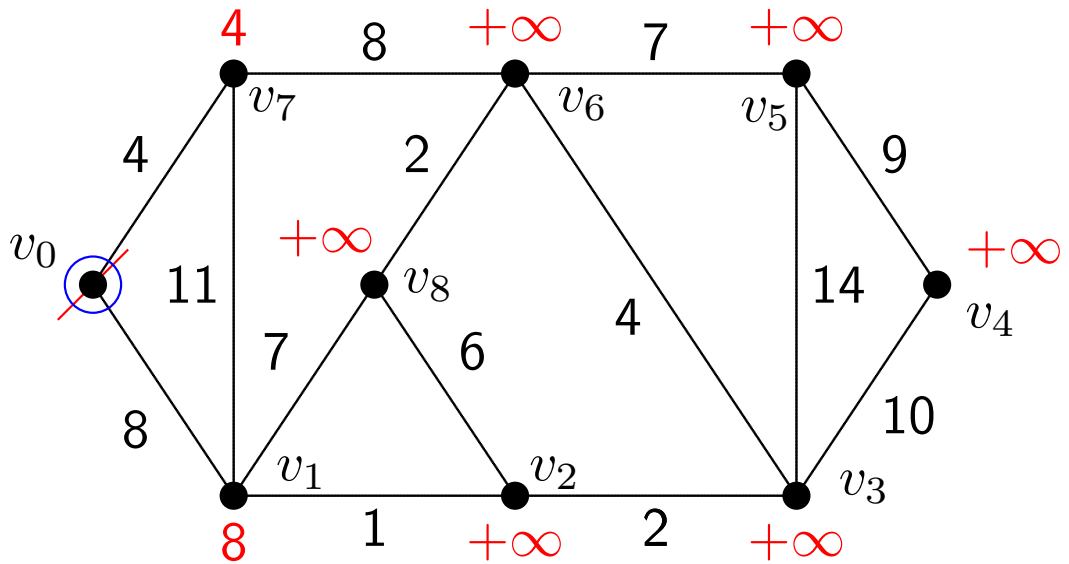


Ausgangszustand:
alle Schlüssel $= +\infty$
aktueller Knoten u: ◉
Startknoten: r $(= v_0)$

Beispiel 105 (Prim's Algorithmus)

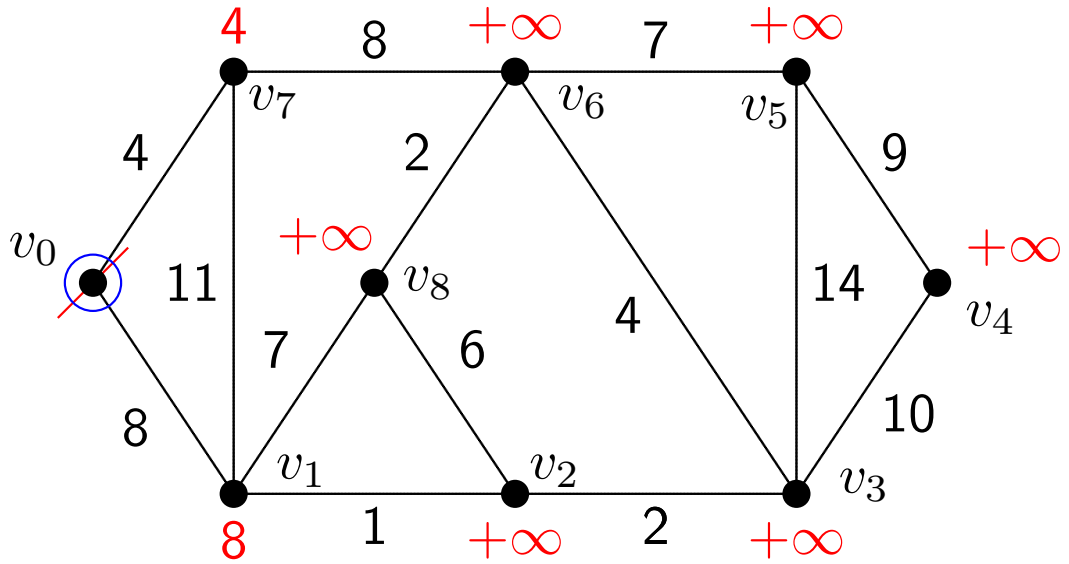


Ausgangszustand:
alle Schlüssel $= +\infty$
aktueller Knoten u: ◉
Startknoten: r $(= v_0)$

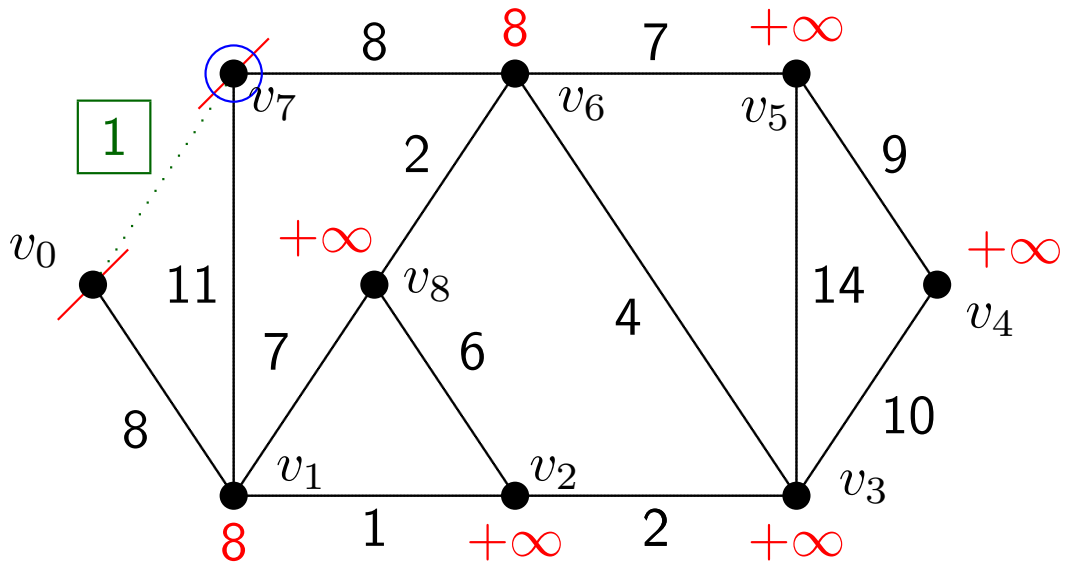


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_1 = 8, v_7 = 4)$

Beispiel 105 (Prim's Algorithmus)

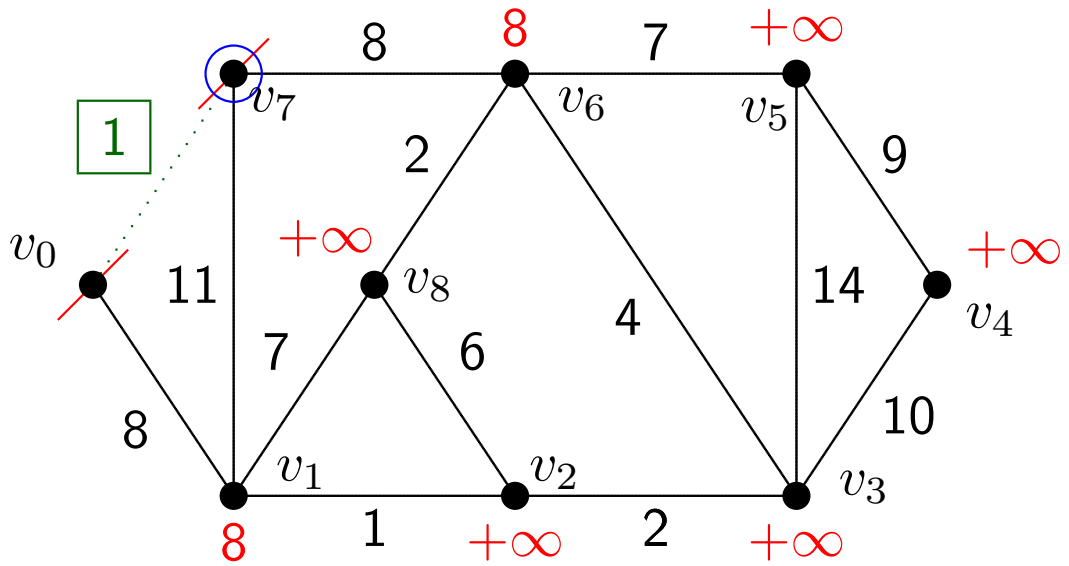


suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit $w(\{u,v\}) <$ Schlüssel$[v]$:
$(v_1 = 8, v_7 = 4)$

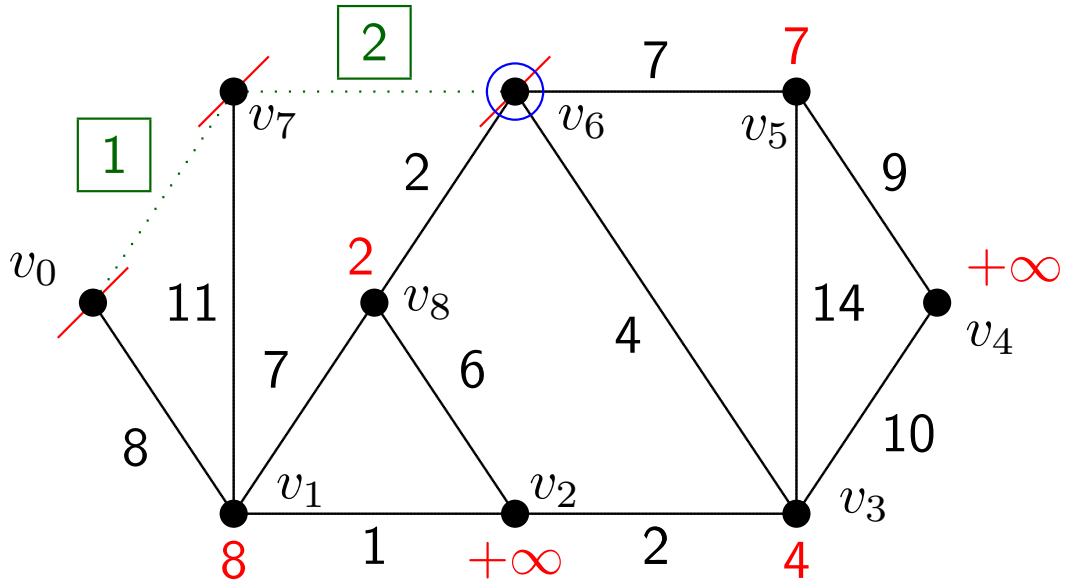


suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit $w(\{u,v\}) <$ Schlüssel$[v]$:
$(v_6 = 8)$

Beispiel 105 (Prim's Algorithmus)

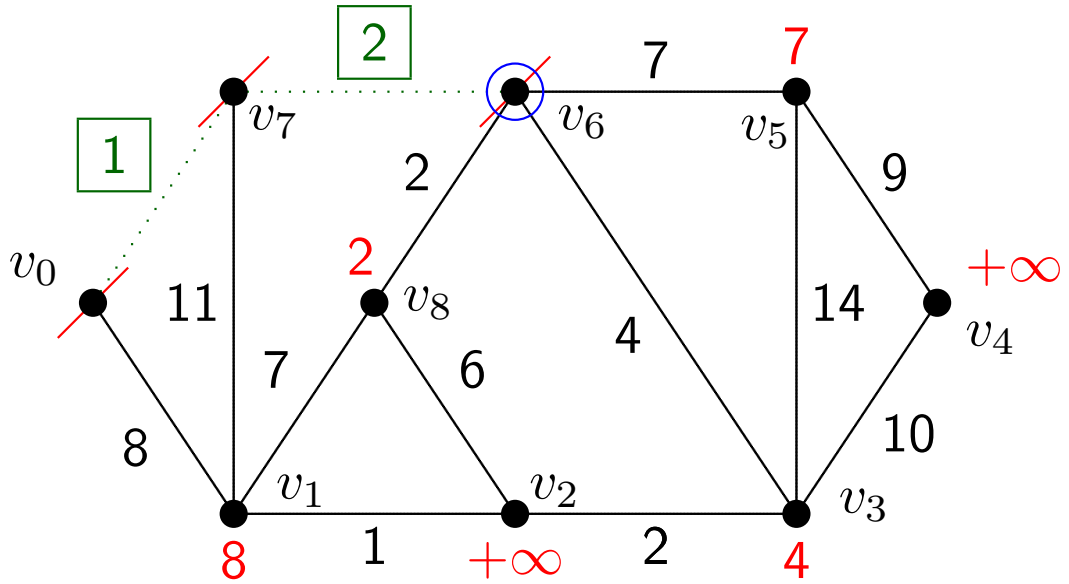


suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit $w(\{u,v\}) <$ Schlüssel$[v]$:
$(v_6 = 8)$

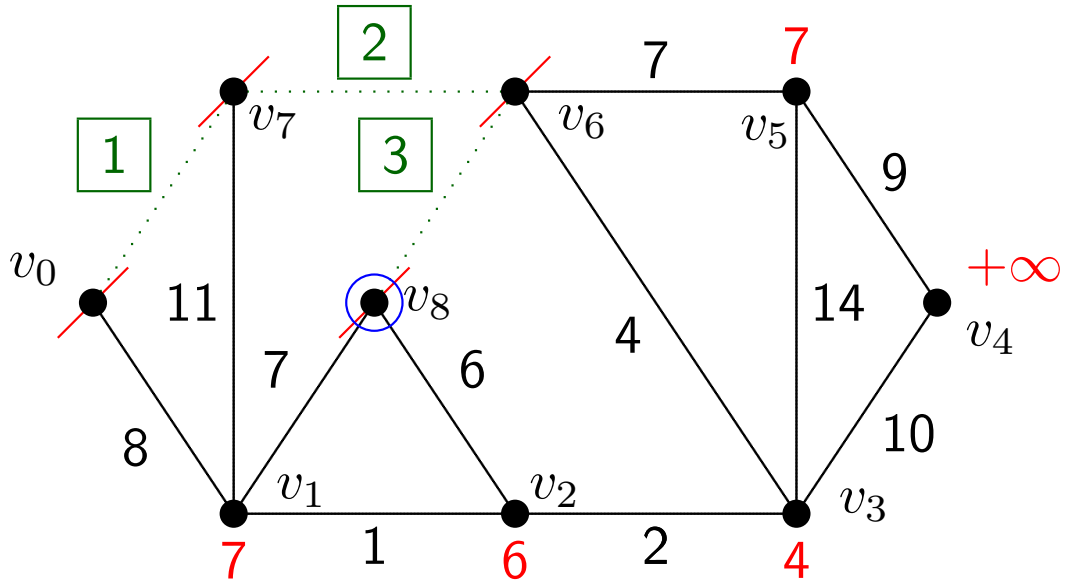


suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit $w(\{u,v\}) <$ Schlüssel$[v]$:
$(v_3 = 4, v_5 = 7, v_8 = 2)$

Beispiel 105 (Prim's Algorithmus)

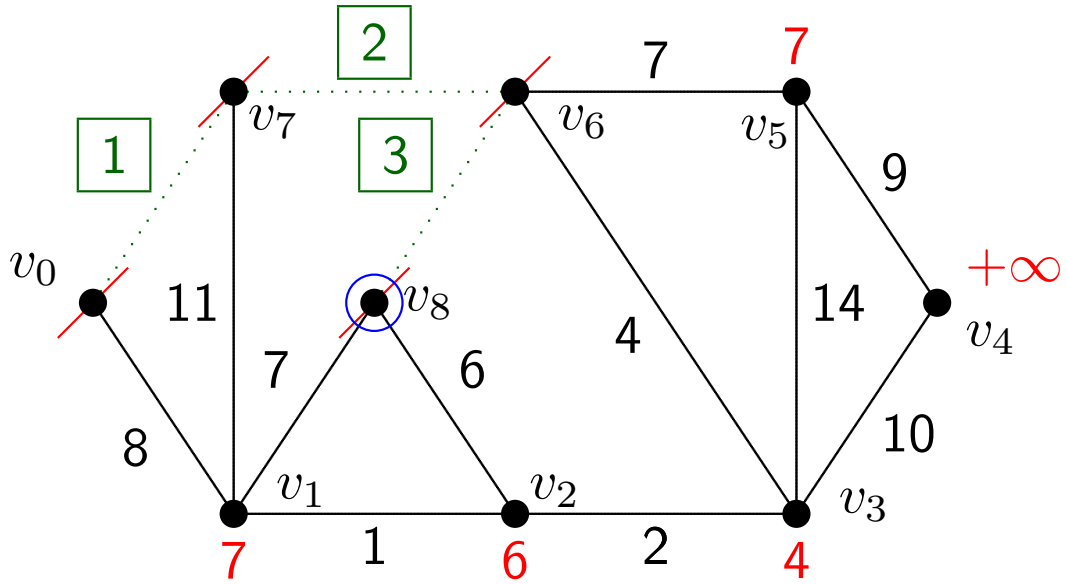


suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$:
$(v_3 = 4, v_5 = 7, v_8 = 2)$

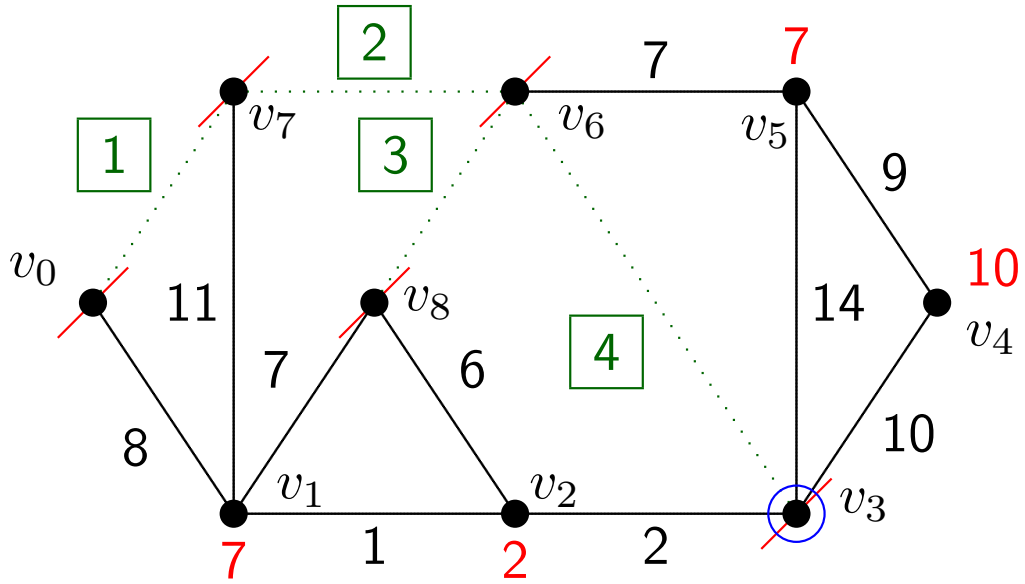


suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$:
$(v_1 = 7, v_2 = 6)$

Beispiel 105 (Prim's Algorithmus)

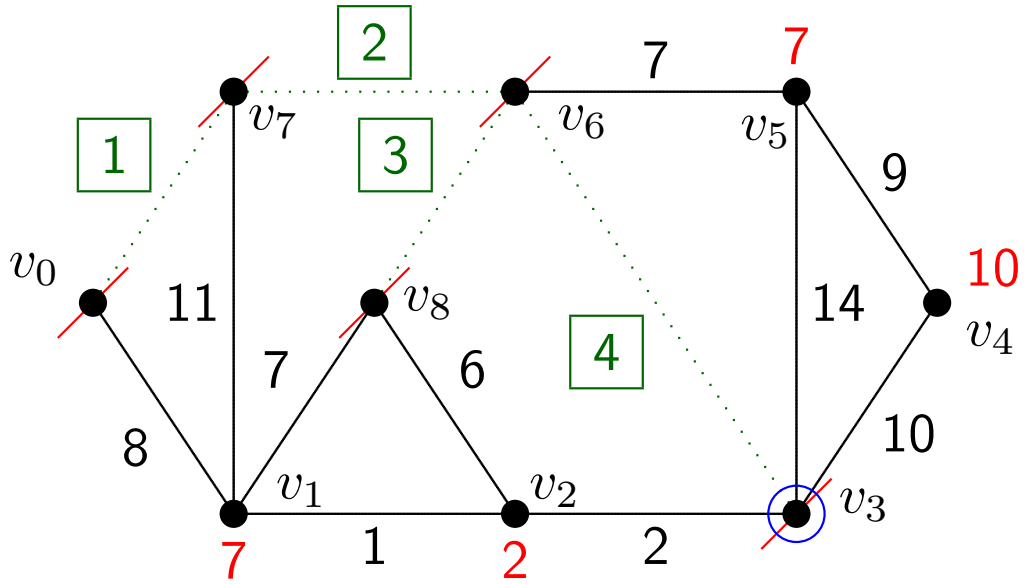


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_1 = 7, v_2 = 6)$

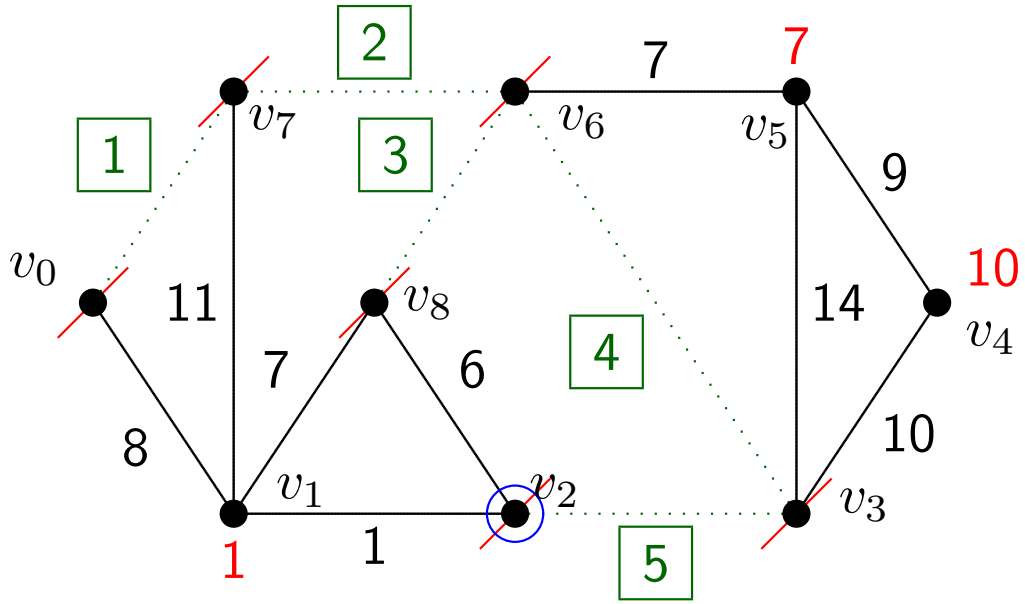


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_2 = 2, v_4 = 10)$

Beispiel 105 (Prim's Algorithmus)

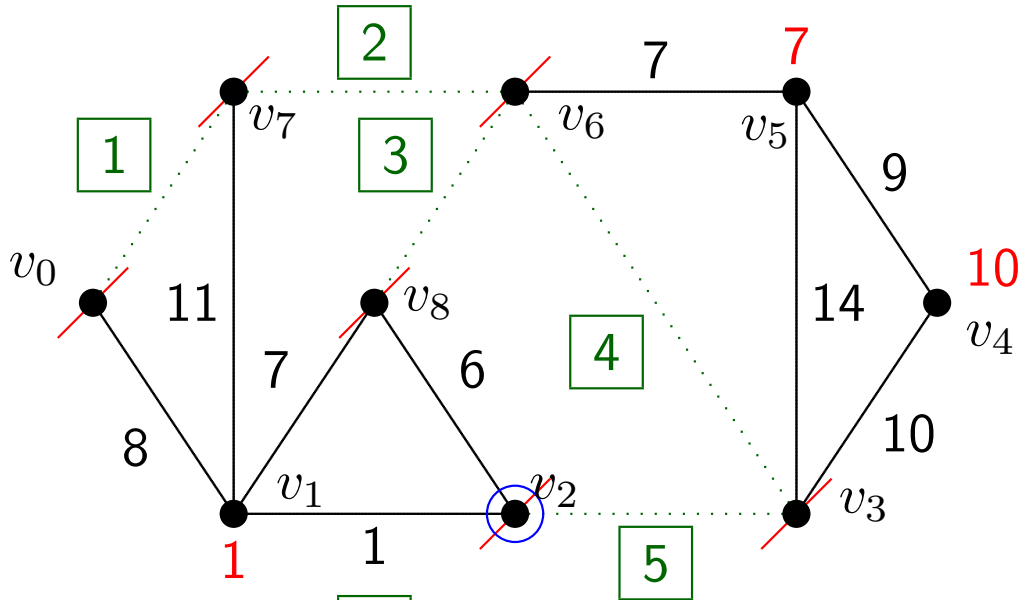


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_2 = 2, v_4 = 10)$

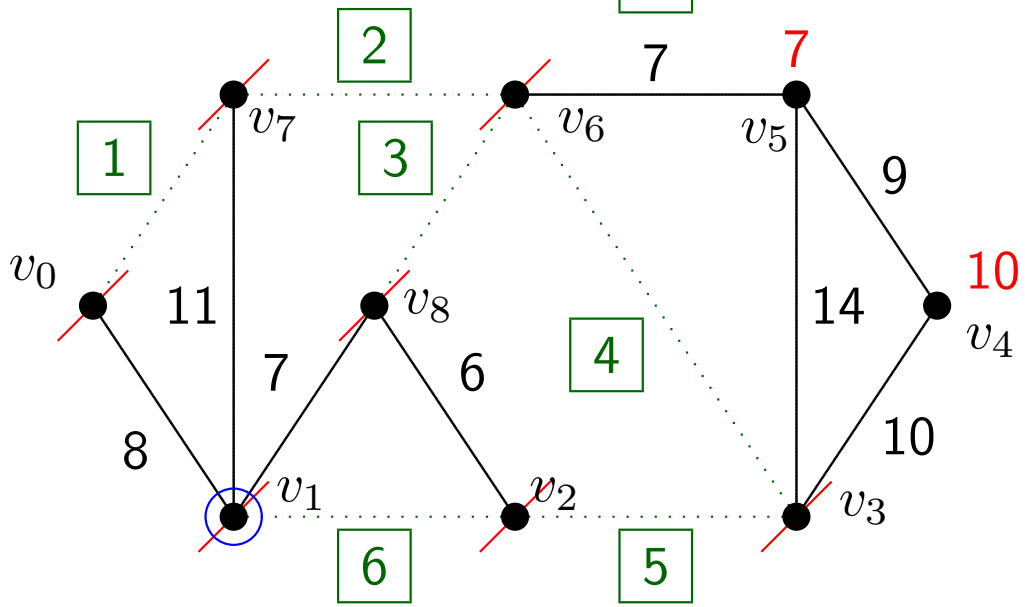


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_1 = 1)$

Beispiel 105 (Prim's Algorithmus)

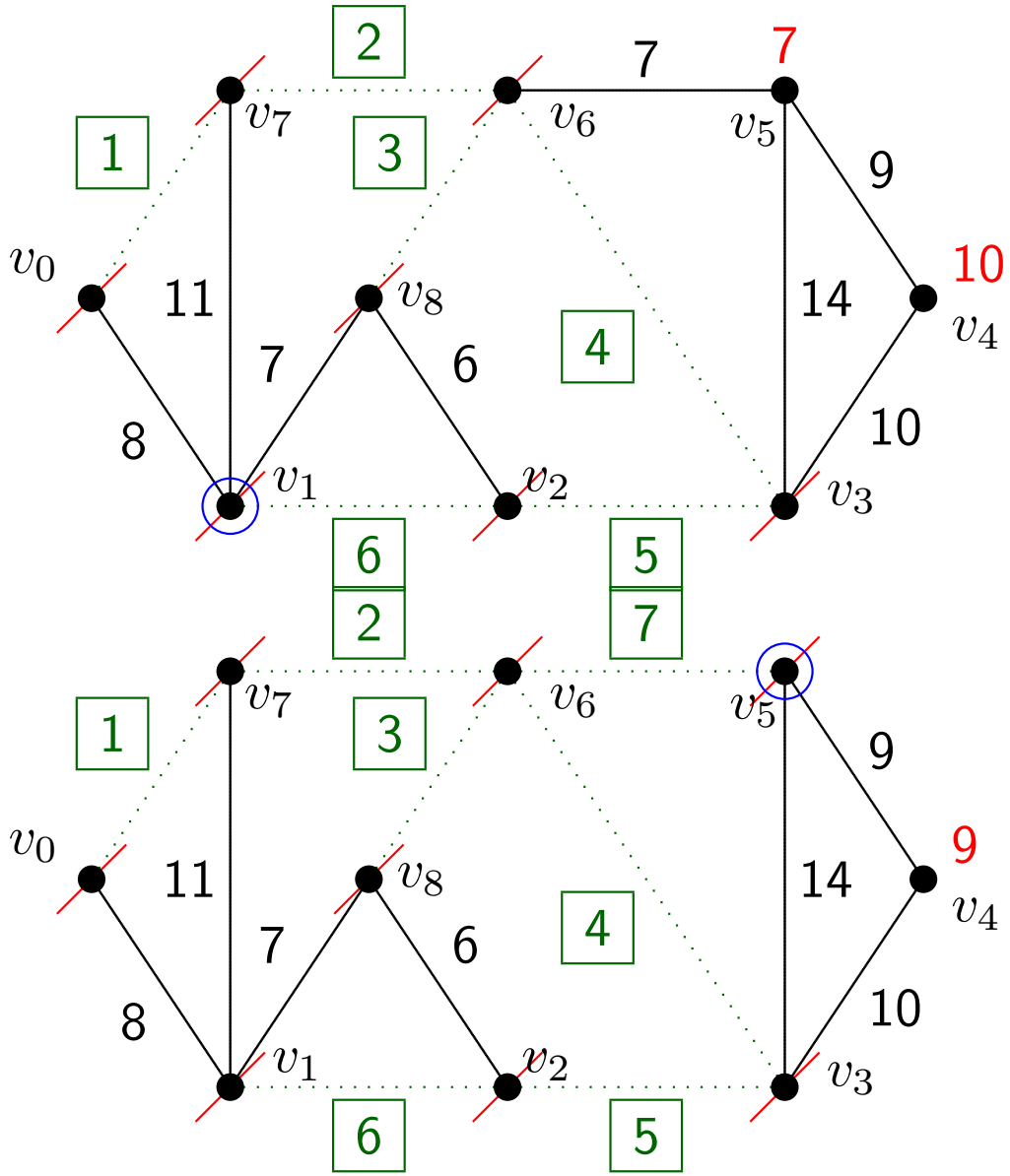


suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit $\mathrm{w}(\{u,v\}) <$ Schlüssel$[v]$:
$(v_1 = 1)$



suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit $\mathrm{w}(\{u,v\}) <$ Schlüssel$[v]$:
solche Nachbarn existieren nicht

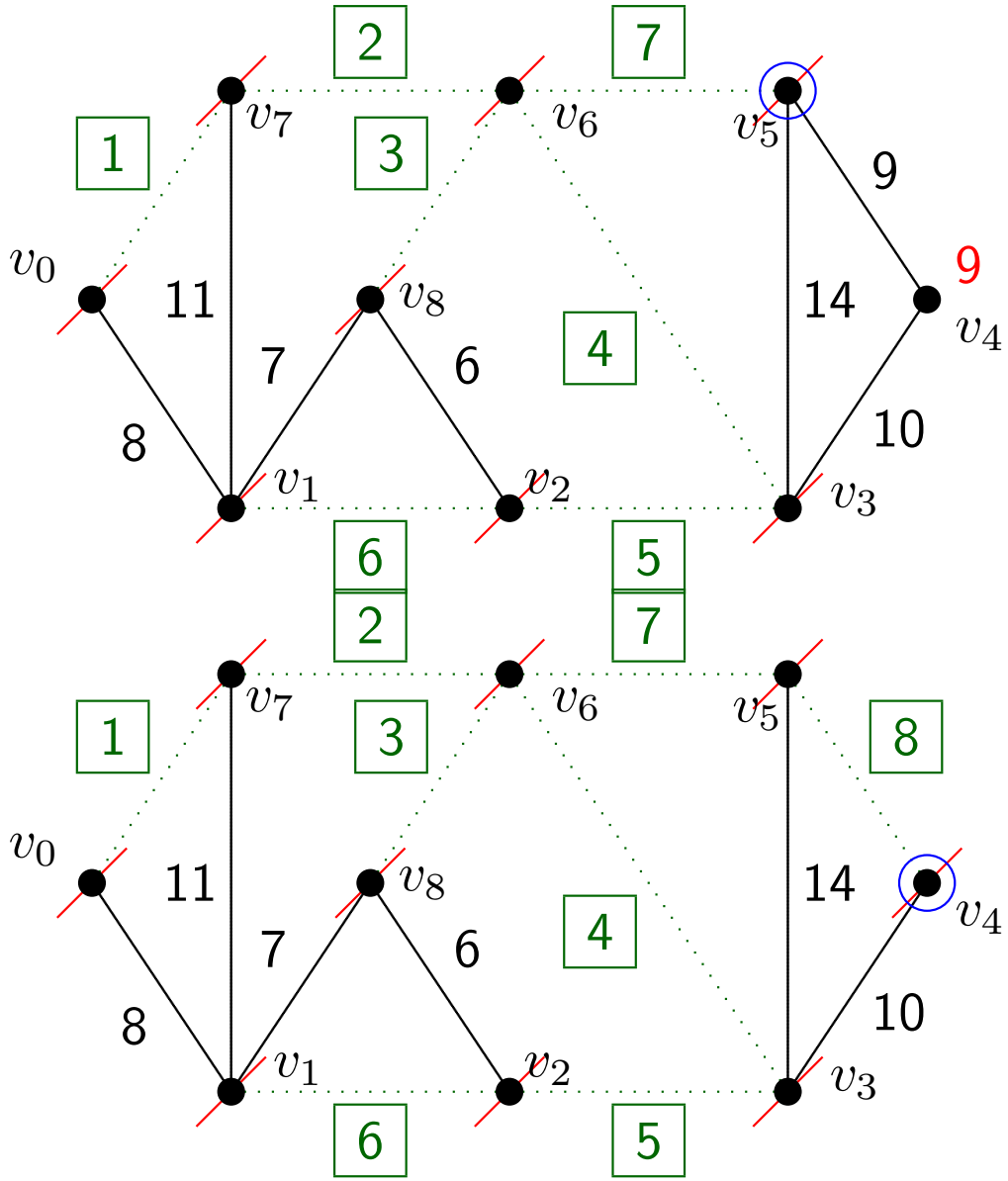
Beispiel 105 (Prim's Algorithmus)



suche $u := \text{FindMin}(PQ)$
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
$w(\{u,v\}) < \text{Schlüssel}[v]$:
solche Nachbarn existieren nicht

suche $u := \text{FindMin}(PQ)$
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
$w(\{u,v\}) < \text{Schlüssel}[v]$:
$(v_4 = 9)$

Beispiel 105 (Prim's Algorithmus)



suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_4 = 9)$

Endzustand:
suche $u :=$ FindMin(PQ)
und entferne u aus PQ,
damit ist PQ leer und der Algorithmus beendet

Korrektheit: ist klar.

Zeitkomplexität:

- n *ExtractMin*
- $\mathcal{O}(m)$ sonstige Operationen inclusive *DecreaseKey*

Implementierung der Priority Queue mittels Fibonacci-Heaps:

Initialisierung	$\mathcal{O}(n)$
ExtractMins	$\mathcal{O}(n \log n)$ $(\leq n$ Stück$)$
DecreaseKeys	$\mathcal{O}(m)$ $(\leq m$ Stück$)$
Sonstiger Overhead	$\mathcal{O}(m)$

Satz 106

Sei $G = (V, E)$ ein ungerichteter Graph (zusammenhängend, einfach) mit Kantengewichten w. Prim's Algorithmus berechnet, wenn mit Fibonacci-Heaps implementiert, einen minimalen Spannbaum von (G, w) in Zeit $\mathcal{O}(m + n \log n)$ (wobei $n = |V|,\ m = |E|$). Dies ist für $m = \Omega(n \log n)$ asymptotisch optimal.

Beweis:

s.o. □

3.5 Prim's Algorithmus, zweite Variante

Die Idee der folgenden Variante von Prim's Algorithmus ist:

Lasse die Priority Queues nicht zu groß werden.

Seien dazu $G = (V, E)$, $|V| = n$, $|E| = m$, w Gewichtsfunktion, und k ein Parameter, dessen Wert wir erst später festlegen werden.

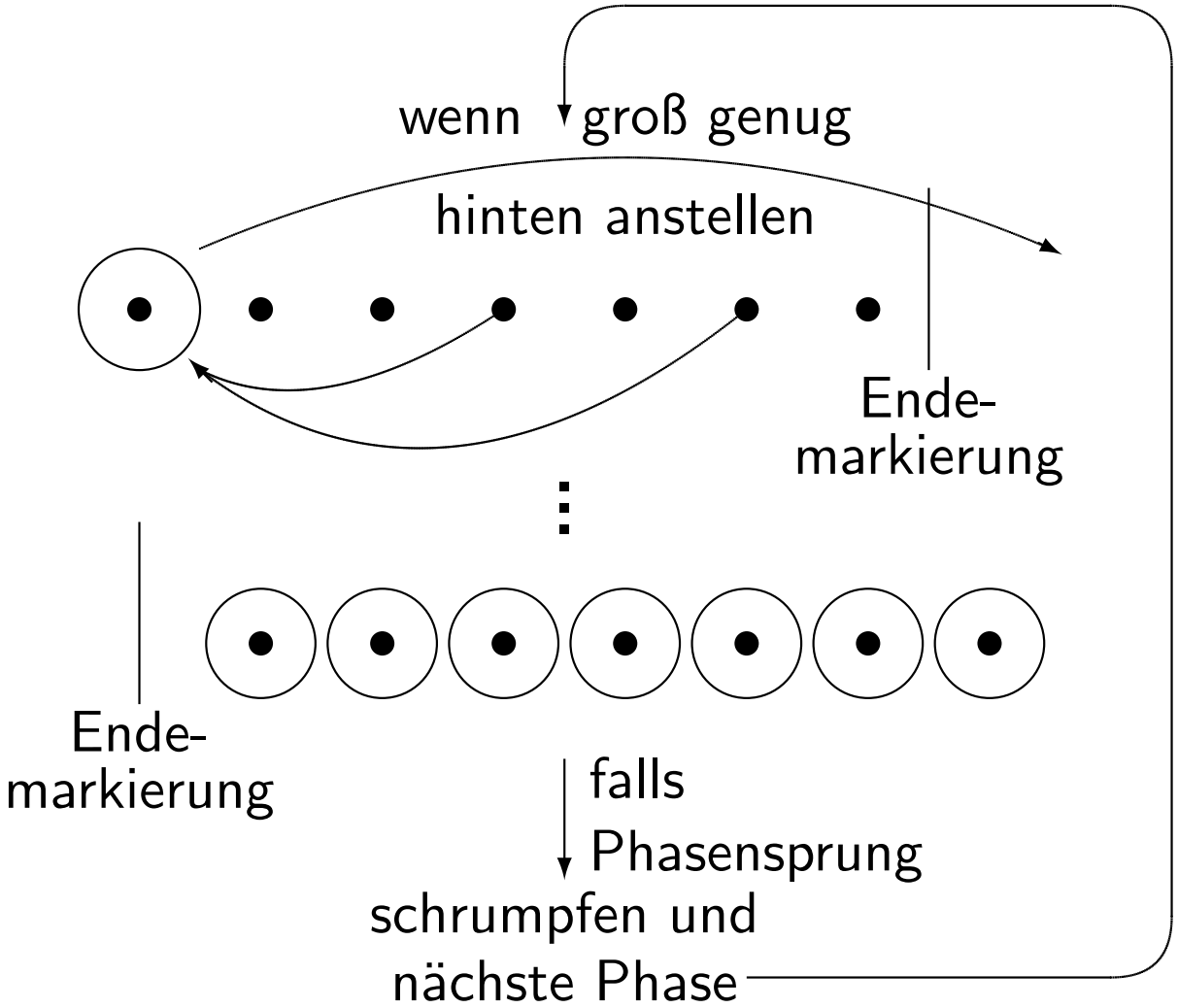
Der Algorithmus arbeitet nun in Phasen wie folgt:

1. Initialisiere eine Schlange von Bäumen, jeder Baum anfangs ein (Super-) Knoten. Zu jedem Baum initialisiere eine Priority Queue (Fibonacci-Heap) mit den Nachbarn der Knoten im Baum, die selbst nicht im Baum sind, als Elementen und jeweils dem Gewicht einer leichtesten Kante zu einem Knoten im Baum als Schlüssel.
2. Markiere jeden Baum in der Schlange mit der Nummer der laufenden Phase.

3. Bestimme k für die Phase
4.

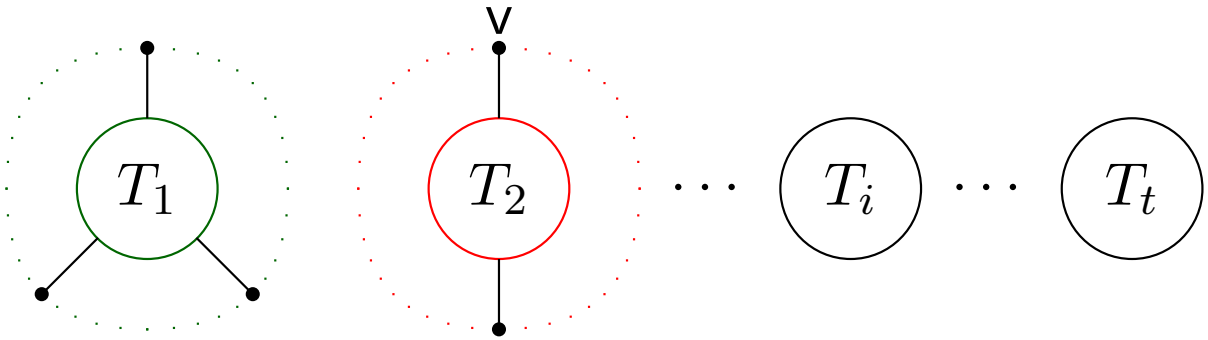
while vorderster Baum in der Schlange hat laufende Phasennummer **do**
    lasse ihn wachsen, solange seine Priority Queue höchstens k Elemente enthält (und noch etwas zu tun ist)
    if Priority Queue zu groß (mehr als k Elemente) **then**
        füge Baum mit inkrementierter Phasennummer am Ende der Schlange ein
    fi
od

5. Falls „Phasensprung“: Schrumpfe alle Bäume zu Superknoten, reduziere Kantenmenge (d.h., behalte zwischen zwei Knoten jeweils nur die leichteste Kante)
6. Beginne nächste Phase mit Schritt 1.!


wenn groß genug
hinten anstellen
Ende-
markierung
Ende-
markierung
falls
Phasensprung
schrumpfen und
nächste Phase

Analyse des Zeitbedarfs:

Sei t die Anzahl der Bäume in der Schlange zu Beginn einer Phase.
Betrachte die Schlange von Bäumen:

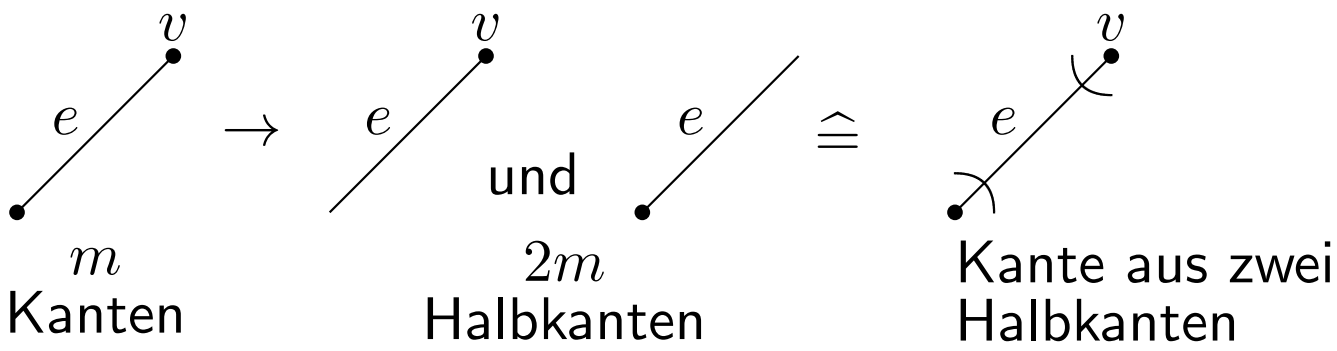


Abgesehen von den Operationen, die bei der Vereinigung von Superknoten anfallen, beträgt der Zeitaufwand pro Phase $\mathcal{O}(m)$.

Vereinigung zweier Superknoten, z.B. T_1 und T_2:
Für jeden (Super-)Knoten v in T_2's Priority Queue:

1. $v \in T_1$: $\surd$ (nichts zu tun)
2. v in Priority Queue von T_1: *DecreaseKey*. Hilfsdatenstruktur: für alle Knoten in den Priority Queues ein Zeiger auf den Superknoten, in dessen Priority Queue der Knoten letztmals am Anfang der Schlange stand.
3. sonst: Einfügen

Betrachte Knoten mit „Halbkanten“: jede Halbkante kommt nur 1x in allen Bäumen der Queue vor. Mit m Kanten ergeben sich $2m$ Halbkanten.



Zeitaufwand pro Phase (mit Bildung der Superknoten zu Beginn):

- Initialisierung: $\mathcal{O}(m)$
- *ExtractMin*: $< t$ Operationen
- sonstige Priority Queue-Operationen, Overhead: Zeit $\mathcal{O}(m)$

Da die Priority Queues höchstens k Elemente enthalten, wenn darauf eine „teure“ Priority Queue-Operation durchgeführt wird, sind die Kosten pro Phase

$$\mathcal{O}(t \log k + m) .$$

Setze $k = 2^{\frac{2m}{t}}$ (damit $t \log k = 2m$). Damit sind die Kosten pro Phase $\mathcal{O}(m)$.

Wieviele Phasen führt der Algorithmus aus?
t ist die Anzahl der Superknoten am Anfang einer Phase, t' sei diese Zahl zu Beginn der nächsten Phase. Sei a die durchschnittliche Anzahl ursprünglicher Knoten in jeder der t Priority Queues zu Anfang der Phase, a' entsprechend zu Beginn der nächsten Phase.

Wir haben:

1. $a = \frac{2m}{t}$
2. $t' \leq \frac{2m}{k}$ (mit Ausnahme ev. der letzten Phase)

Also:

$$a' = \frac{2m}{t'} \geq k = 2^{\frac{2m}{t}} = 2^a$$

Für die erste Phase ist $a = \frac{2m}{n}$, für jede Phase ist $a \leq n - 1$. Also ist die Anzahl der Phasen

$$\leq 1 + \min\left\{i; \log_2^{(i)}(n-1) \leq \frac{2m}{n}\right\} .$$

Setzen wir $\beta(m,n) := \min\left\{i; \log_2^{(i)} n \leq \frac{m}{n}\right\}$, dann gilt

$$\beta(m,n) \leq \log^* n \text{ für } n \leq m \leq \binom{n}{2} .$$

Satz 107

Für gewichtete, zusammenhängende (ungerichtete) Graphen mit n Knoten, m Kanten kann ein minimaler Spannbaum in Zeit $\mathcal{O}(\min\{m \cdot \beta(m,n), m + n \log n\})$ bestimmt werden.

3.6 Erweiterungen

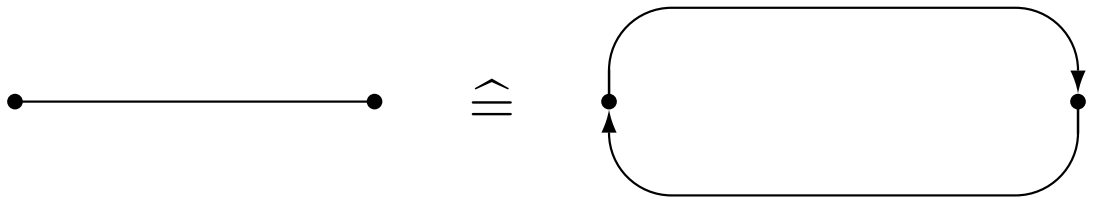
Euklidische minimale Spannbäume stellen ein Problem dar, für das es speziellere Algorithmen gibt. Literatur hierzu:

Andrew Chih-Chi Yao:
On constructing minimum spanning trees in k-dimensional spaces and related problems
SIAM J. Comput. **11**(4), pp. 721–736 (1982)

Kapitel V Kürzeste Pfade

1. Grundlegende Begriffe

Betrachte Digraph $G = (V, A)$ oder Graph $G = (V, E)$.



Kante im ungerichteten Graph

entgegengesetzte (antiparallele) Kanten

Distanzfunktion: $\mathrm{d} : A \longrightarrow \mathbb{R}^+$ (bzw. $\longrightarrow \mathbb{R}$)
O.B.d.A.: $A = V \times V$, $\mathrm{d}(x, y) = +\infty$ für Kanten, die eigentlich nicht vorhanden sind.
$\mathrm{dis}(v, w) :=$ Länge eines kürzesten Pfades von v nach w
$\in \mathbb{R}^+ \cup \{+\infty\}$.

Arten von Kürzeste-Pfade-Problemen:

1. single-pair-shortest-path (spsp). Beispiel: Kürzeste Entfernung von München nach Frankfurt.
2. single-source-shortest-path: gegeben G, d und $s \in V$, bestimme für alle $v \in V$ die Länge eines kürzesten Pfades von s nach v (bzw. einen kürzesten Pfad von s nach v) (sssp). Beispiel: Kürzeste Entfernung von München nach allen anderen Großstädten.
3. all-pairs-shortest-path (apsp). Beispiel: Kürzeste Entfernung zwischen allen Großstädten.

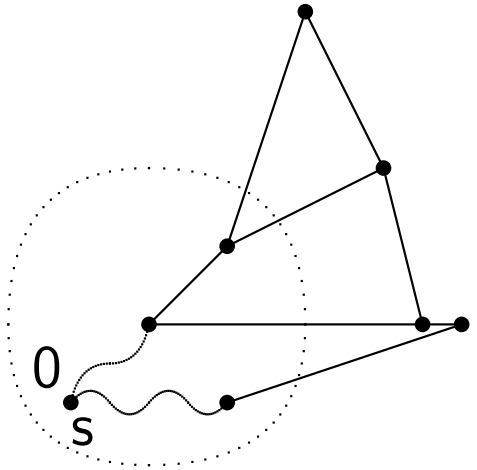
Bemerkung: Wir kennen keinen Algorithmus, der das single-pair-shortest-path berechnet, ohne nicht gleichzeitig (im worst-case) das single-source-shortest-path-Problem zu lösen.

2. Das single-source-shortest-path-Problem

Zunächst nehmen wir an, dass $\mathrm{d} \geq 0$ ist. Alle kürzesten Pfade von a nach b sind o.B.d.A. einfache Pfade.

2.1 Dijkstra's Algorithmus

Gegeben: $G = (V, A)$, $(A = V \times V)$, Distanzfunktion $\mathrm{d} : A \to \mathbb{R}^+ \cup \{+\infty\}$, Startknoten s, G durch Adjazenzlisten dargestellt.



algorithm sssp:=

$S := \{s\}$; $\mathrm{dis}[s] := 0$; initialisiere eine Priority Queue PQ, die alle Knoten $v \in V \setminus \{s\}$ enthält mit Schlüssel $\mathrm{dis}[v] := \mathrm{d}(s, v)$

for alle $v \in V - \{s\}$ **do** from$[v] := s$ **od**

while $S \neq V$ **do**

$v :=$ *ExtractMin*(PQ)

$S := S \cup \{v\}$

for alle $w \in V \setminus S$, $\mathrm{d}(v, w) < \infty$ **do**

if $\mathrm{dis}[v] + \mathrm{d}(v, w) < \mathrm{dis}[w]$ **then**

DecreaseKey$(w, \mathrm{dis}[v] + \mathrm{d}(v, w))$

co *DecreaseKey* aktualisiert $\mathrm{dis}[w]$ **oc**

from$[w] := v$

fi

od

od

Seien $n = |V|$ und $m =$ die Anzahl der wirklichen Kanten in G.

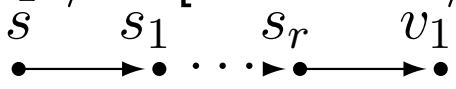
Laufzeit (mit Fibonacci-Heaps):

Initialisierung:	$\mathcal{O}(n)$
ExtractMin:	$n \cdot \mathcal{O}(\log n)$
Sonstiger Aufwand:	$m \cdot \mathcal{O}(1)$ (z.B. *DecreaseKey*)

$\Rightarrow$ Zeitbedarf also: $\boxed{\mathcal{O}(m + n \log n)}$

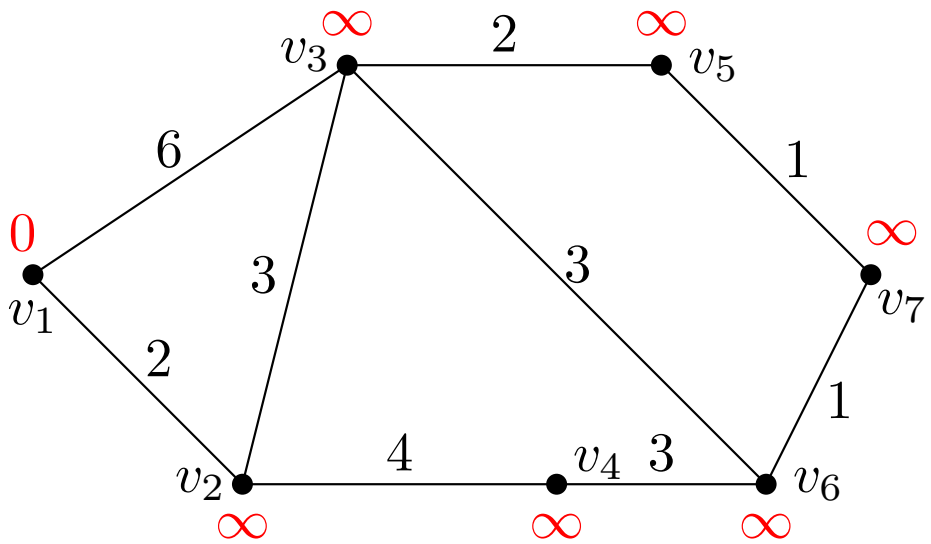
Korrektheit: Wir behaupten, dass in dem Moment, in dem ein $v \in V \setminus \{s\}$ Ergebnis der *ExtractMin* Operation ist, der Wert $\mathrm{dis}[v]$ des Schlüssels von v gleich der Länge eines kürzesten Pfades von s nach v ist.

Beweis:
[durch Widerspruch] Sei $v \in V \setminus \{s\}$ der erste Knoten, für den diese Behauptung nicht stimmt, und sei

ein kürzester Pfad von s nach v, mit einer Länge $< \mathrm{dis}[v]$. Dabei sind $s_1, \ldots, s_r \in S$, $v_1 \notin S$ [$r = 0$ und/oder $q = 0$ ist möglich]. Betrachte den Pfad 

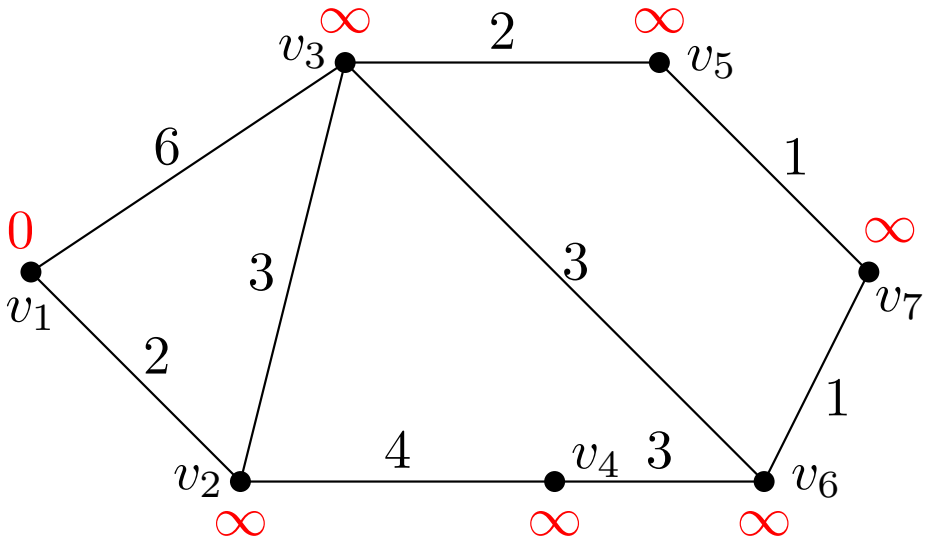
; seine Länge ist $< \mathrm{dis}[v]$, für $q \geq 1$ (ebenso für $q = 0$) ist also $\mathrm{dis}[v_1] < \mathrm{dis}[v]$, im Widerspruch zur Wahl von v. □

Beispiel 108 (Dijkstra's Algorithmus)

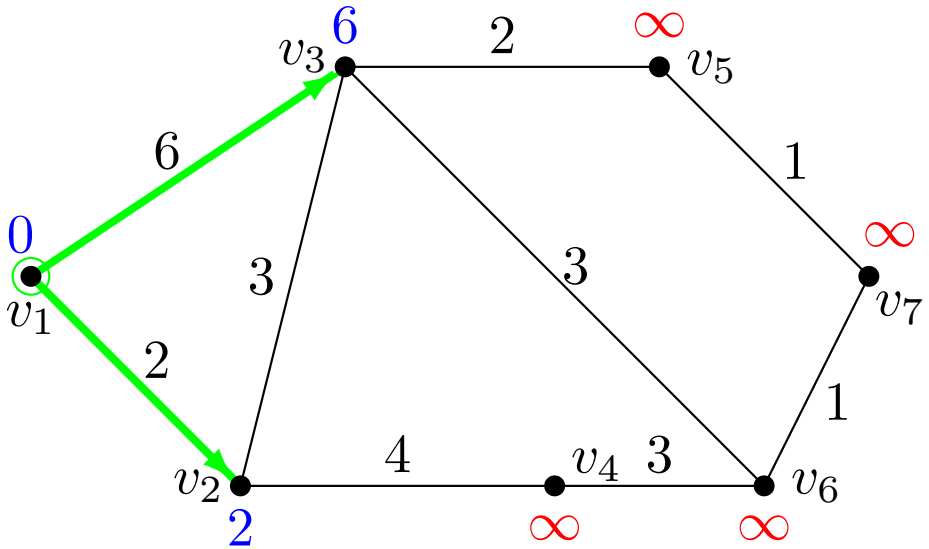


gegeben Graph G;
v_1 ist der Startknoten;
setze v_1 als Bezugsknoten;
setze $\mathrm{dis}[v_1] = 0$;
setze $\mathrm{dis}[\text{Rest}] = +\infty$;

Beispiel 108 (Dijkstra's Algorithmus)

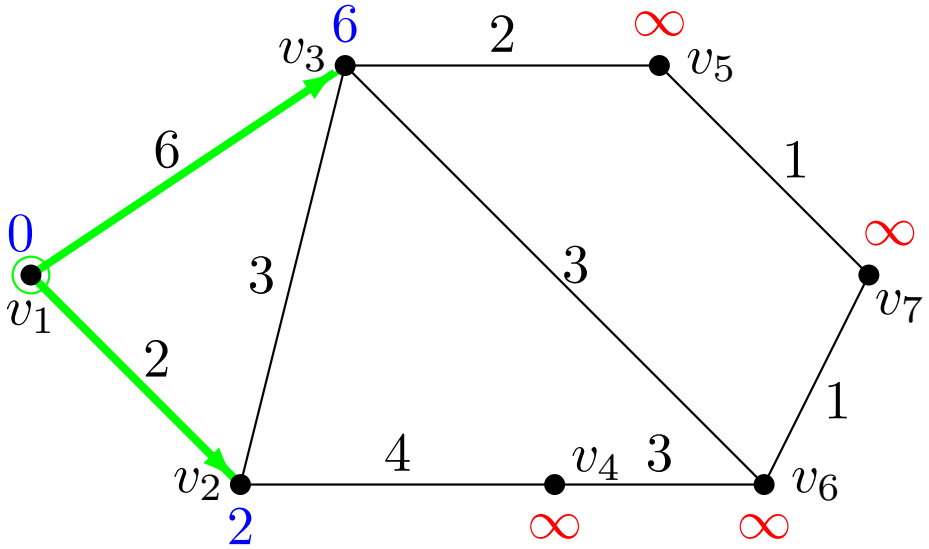


gegeben Graph G;
v_1 ist der Startknoten;
setze v_1 als Bezugsknoten;
setze $\mathrm{dis}[v_1] = 0$;
setze $\mathrm{dis}[\text{Rest}] = +\infty$;



setze $\mathrm{dis}[v_2] = 2$;
markiere (v_1, v_2);
setze $\mathrm{dis}[v_3] = 6$;
markiere (v_1, v_3);
setze v_2 als Bezugsknoten,
da $\mathrm{dis}[v_2]$ minimal;

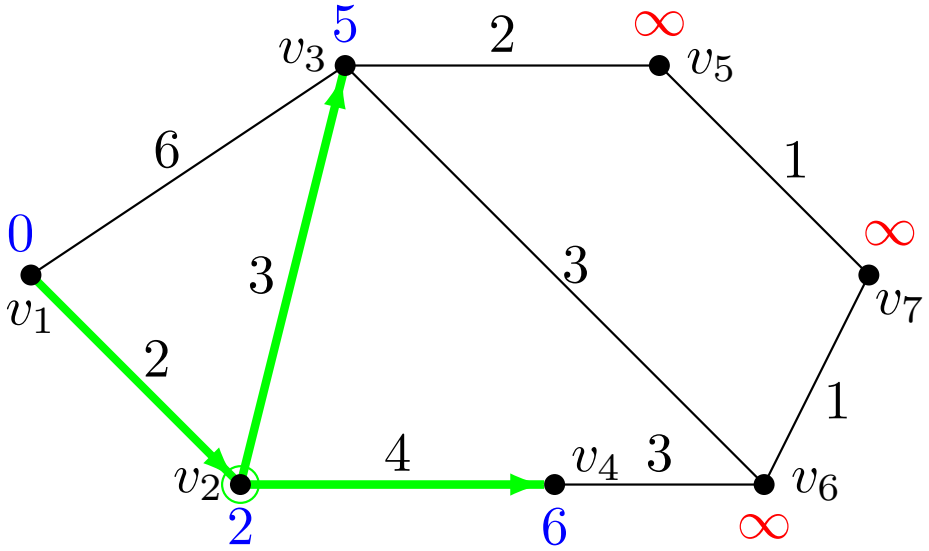
Beispiel 108 (Dijkstra's Algorithmus)



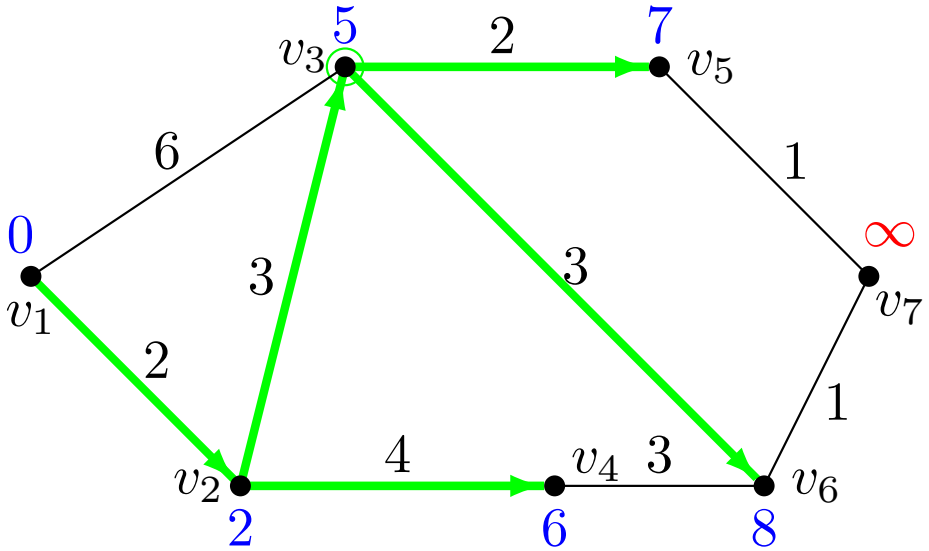
setze $\mathrm{dis}[v_2] = 2$;
markiere (v_1, v_2);
setze $\mathrm{dis}[v_3] = 6$;
markiere (v_1, v_3);
setze v_2 als Bezugsknoten,
da $\mathrm{dis}[v_2]$ minimal;

setze $\mathrm{dis}[v_3] = 2 + 3 = 5$;
markiere (v_2, v_3);
unmarkiere (v_1, v_3);
setze $\mathrm{dis}[v_4] = 2 + 4 = 6$;
markiere (v_2, v_4);
setze v_3 als Bezugsknoten,
da $\mathrm{dis}[v_3]$ minimal;

Beispiel 108 (Dijkstra's Algorithmus)

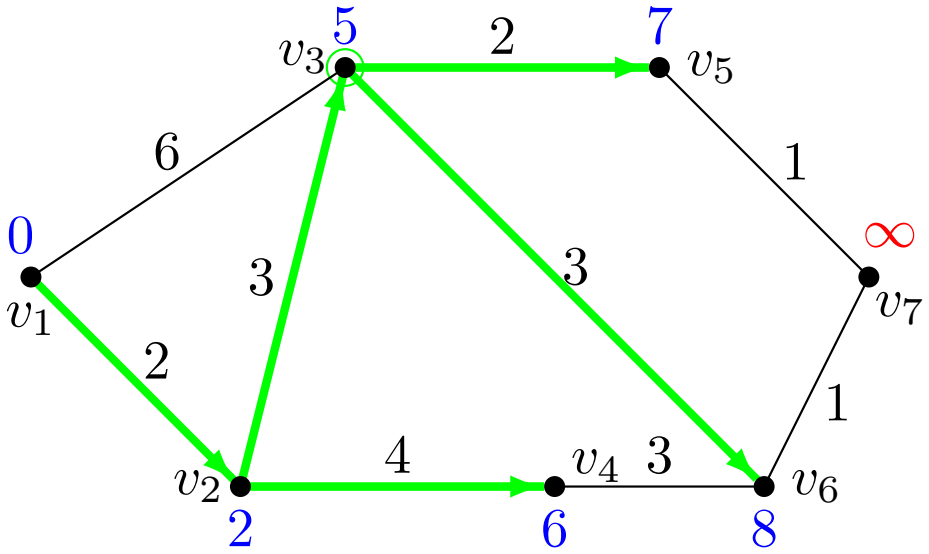


setze $\mathrm{dis}[v_3] = 2 + 3 = 5$;
markiere (v_2, v_3);
unmarkiere (v_1, v_3);
setze $\mathrm{dis}[v_4] = 2 + 4 = 6$;
markiere (v_2, v_4);
setze v_3 als Bezugsknoten,
da $\mathrm{dis}[v_3]$ minimal;

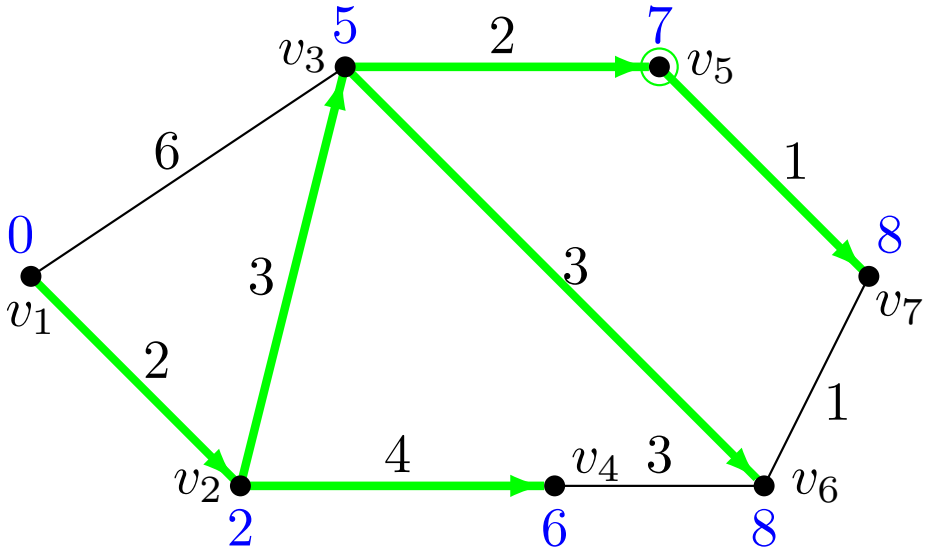


setze $\mathrm{dis}[v_5] = 5 + 2 = 7$;
markiere (v_3, v_5);
setze $\mathrm{dis}[v_6] = 5 + 3 = 8$;
markiere (v_3, v_6);
setze v_4, dann v_5
als Bezugsknoten;

Beispiel 108 (Dijkstra's Algorithmus)



setze $\mathrm{dis}[v_5] = 5 + 2 = 7$;
markiere (v_3, v_5);
setze $\mathrm{dis}[v_6] = 5 + 3 = 8$;
markiere (v_3, v_6);
setze v_4, dann v_5
als Bezugsknoten;



setze $\mathrm{dis}[v_7] := 7 + 1 = 8$;
markiere (v_5, v_7);
alle Knoten wurden erreicht:
$\Rightarrow$ Algorithmus zu Ende

Beobachtung:

- *ExtractMin* liefert eine (schwach) monoton steigende Folge von Schlüsseln $\mathrm{dis}[\cdot]$;
- Die Schlüssel $\neq \infty$ in PQ sind stets $\leq \mathrm{dis}[v] + C$, wobei v das Ergebnis der vorangehenden *ExtractMin*-Operation (bzw. s zu Beginn) und $C := \max_{(u,w)\in A}\{\mathrm{dis}(u,w)\}$ ist.

Satz 109

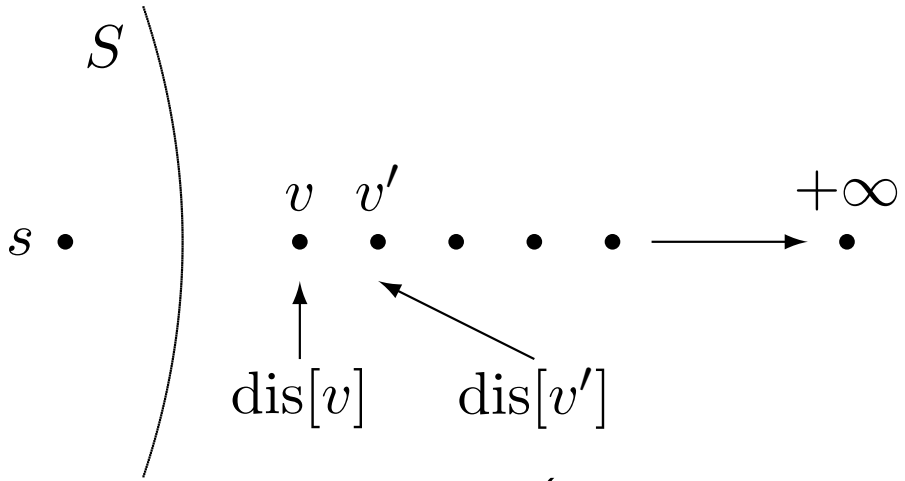
Dijkstra's Algorithmus (mit Fibonacci-Heaps) löst das single-source-shortest-path-Problem in Zeit $\mathcal{O}(m + n \log n)$.

Bemerkungen:

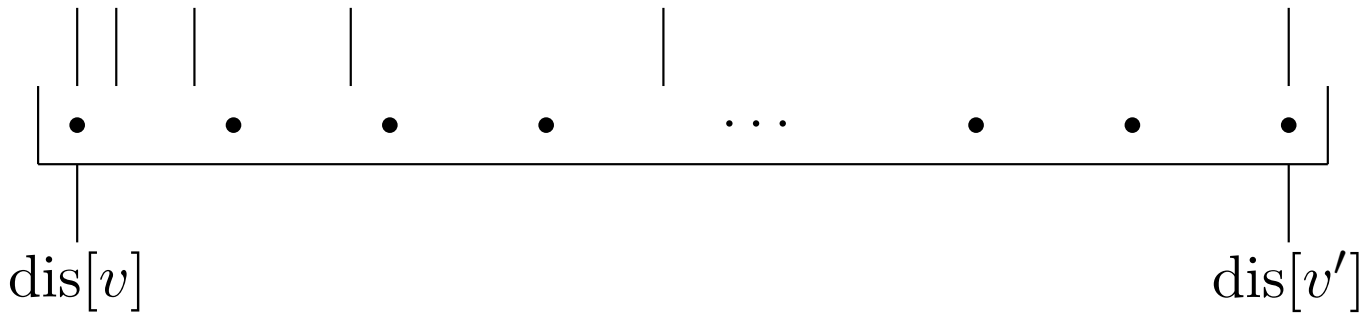
1. Verwendet man Dijkstra's Algorithmus mit d-Heaps, so erhält man Laufzeit
$$\mathcal{O}(m \log_{2+\frac{m}{n}} n)$$

2.  Mikkel Thorup:
Undirected single-source shortest paths with positive integer weights in linear time
J. ACM **46**(3), pp. 362–394 (1999)

2.2 Dijkstra's Algorithmus mit Radix-Heaps



$\mathrm{d}(v) \in [0, \ldots, C]$ (alle Werte sind zwischen $\mathrm{dis}[v]$ und nC)



Wir verwenden Radix-Heaps (siehe dort), in der einstufigen Variante wie vorgestellt.

Satz 110

Dijkstra's Algorithmus mit einstufigen Radix-Heaps hat Zeitkomplexität $\mathcal{O}(m + n \log C)$.

Beweis:

siehe Satz 72. □

Verbesserungen:

1. zweistufige Heaps: $\boxed{\mathcal{O}(m + n \cdot \frac{\log C}{\log \log C})}$
2. mehrstufige Heaps (+Fibonacci-Heaps): $\boxed{\mathcal{O}(m + n\sqrt{\log C})}$

R.K. Ahuja, Kurt Mehlhorn, J.B. Orlin, R.E. Tarjan:
Faster Algorithms for the Shortest Path Problem
J.ACM **37**, pp. 213–223 (1990)

2.3 Bellman-Ford-Algorithmus

Wir setzen (zunächst) wiederum voraus:

$$\mathrm{d} \geq 0 \; .$$

Dieser Algorithmus ist ein Beispiel für dynamische Programmierung.

Sei $B_k[i] :=$ Länge eines kürzesten Pfades von s zum Knoten i, wobei der Pfad höchstens k Kanten enthält.
Gesucht ist $B_{n-1}[i]$ für $i = 1, \ldots, n$ (o.B.d.A. $V = \{1, \ldots, n\}$).

Initialisierung:

$$B_1[i] := \begin{cases} \mathrm{d}(s,i) & , \text{ falls } \mathrm{d}(s,i) < \infty, i \neq s \\ 0 & , \text{ falls } i = s \\ +\infty & , \text{ sonst} \end{cases}$$

Iteration:

for $k := 2$ **to** $n-1$ **do**
  for $i := 1$ **to** n **do**

$$B_k[i] := \begin{cases} 0 & , \text{falls } i = s \\ \min_{j \in N^{-1}(i)} \{B_{k-1}[i], B_{k-1}[j] + \mathrm{d}(j,i)\} & , \text{sonst} \end{cases}$$

  od
od

Bemerkung: $N^{-1}(i)$ ist die Menge der Knoten, von denen aus eine Kante zu Knoten i führt.

Korrekheit:
klar (Beweis durch vollständige Induktion)

Zeitbedarf:
Man beachte, dass in jedem Durchlauf der äußeren Schleife jede Halbkante einmal berührt wird.

Satz 111

Der Zeitbedarf des Bellman-Ford-Algorithmus ist $\mathcal{O}(n \cdot m)$.

Beweis:

s.o. □

3. Floyd's Algorithmus für das all-pairs-shortest-path-Problem

Dieser Algorithmus wird auch als „Kleene's Algorithmus" bezeichnet. Er ist ein weiteres Beispiel für dynamische Programmierung.

Sei $G = (V, A)$ mit Distanzfunktion $\mathrm{d} : A \to \mathbb{R}_0^+$ gegeben. Sei o.B.d.A. $V = \{v_1, \ldots, v_n\}$.

Wir setzen nun

$c_{ij}^k :=$ *Länge eines kürzesten Pfades von* v_i *nach* v_j*, der als innere Knoten (alle bis auf ersten und letzten Knoten) nur Knoten aus* $\{v_1, \ldots, v_k\}$ *enthält.*

algorithm floyd :=

for alle (i,j) **do** $c_{ij}^{(0)} := \mathrm{d}(i,j)$ **od** **co** $1 \leq i,j \leq n$ **oc**

for $k := 1$ **to** n **do**

for alle (i,j), $1 \leq i,j \leq n$ **do**

$$c_{ij}^{(k)} := \min\left\{c_{ij}^{(k-1)},\ c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\right\}$$

od

od

Laufzeit: $\mathcal{O}(n^3)$

Korrektheit:

Zu zeigen: $c_{ij}^{(k)}$ des Algorithmus $= c_{ij}^{k}$ (damit sind die Längen der kürzesten Pfade durch $c_{ij}^{(n)}$ gegeben).

Beweis:

Richtig für $k = 0$. Induktionsschluss: Ein kürzester Pfad von v_i nach v_j mit inneren Knoten $\in \{v_1, \ldots, v_{k+1}\}$ enthält entweder v_{k+1} gar nicht als inneren Knoten, oder er enthält v_{k+1} genau einmal als inneren Knoten. Im ersten Fall wurde dieser Pfad also bereits für $c_{ij}^{(k)}$ betrachtet, hat also Länge $= c_{ij}^{(k)}$. Im zweiten Fall setzt er sich aus einem kürzesten Pfad P_1 von v_i nach v_{k+1} und einem kürzesten Pfad P_2 von v_{k+1} nach v_j zusammen, wobei alle inneren Knoten von P_1 und $P_2 \in \{v_1, \ldots, v_k\}$ sind. Also ist die Länge des Pfades $= c_{i,k+1}^{(k)} + c_{k+1,j}^{(k)}$. □

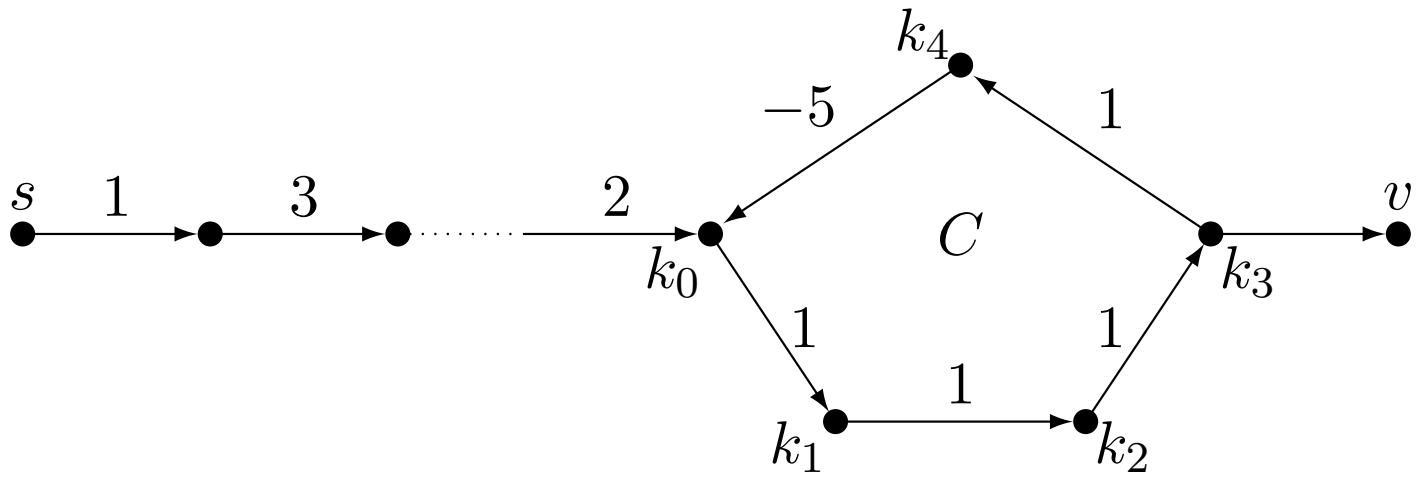
Satz 112

Floyd's Algorithmus für das all-pairs-shortest-path-Problem hat Zeitkomplexität $\mathcal{O}(n^3)$.

4. Digraphen mit negativen Kantengewichten

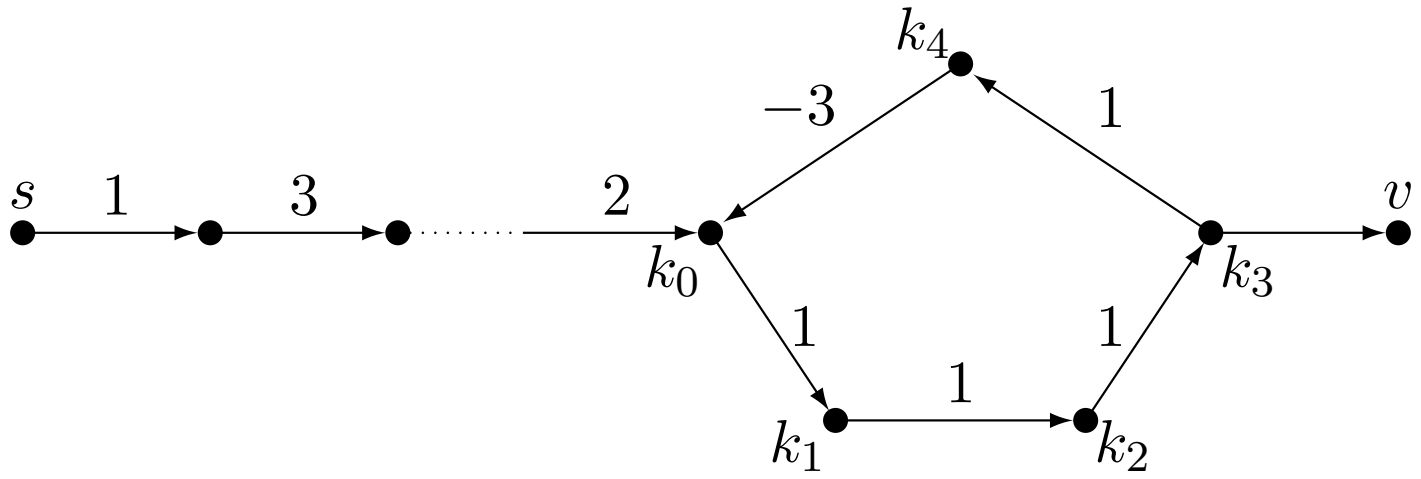
4.1 Grundsätzliches

Betrachte Startknoten s und einen Kreis C mit Gesamtlänge < 0.



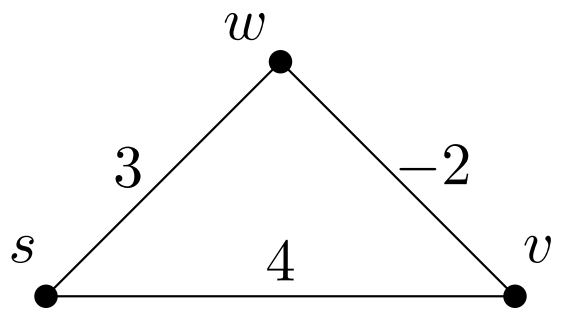
Sollte ein Pfad von s nach C und von C nach v existieren, so ist ein kürzester Pfad von s nach v nicht definiert.

Falls aber die Gesamtlänge des Kreises $C \geq 0$ ist,



dann ist der kürzeste Pfad wohldefiniert. Probleme gibt es also nur dann, wenn G einen Zyklus negativer Länge enthält.

Dijkstra's Algorithmus funktioniert bei negativen Kantenlängen nicht:



Bei diesem Beispielgraphen (der nicht einmal einen negativen Kreis enthält) berechnet der Dijkstra-Algorithmus die minimale Entfernung von s nach w fälschlicherweise als 3 (statt 2).

4.2 Modifikation des Bellman-Ford-Algorithmus

$B_k[i]$ gibt die Länge eines kürzesten gerichteten s-i-Pfades an, der aus höchstens k Kanten besteht. Jeder Pfad, der keinen Kreis enthält, besteht aus maximal $n-1$ Kanten. In einem Graphen ohne negative Kreise gilt daher:

$$\forall i \in V : B_n[i] \geq B_{n-1}[i]$$

Gibt es hingegen einen (von s aus erreichbaren) Kreis negativer Länge, so gibt es einen Knoten $i \in V$, bei dem ein Pfad aus n Kanten mit der Länge $B_n[i]$ diesen Kreis häufiger durchläuft als jeder Pfad aus maximal $n-1$ Kanten der Länge $B_{n-1}[i]$. Demnach gilt in diesem Fall:

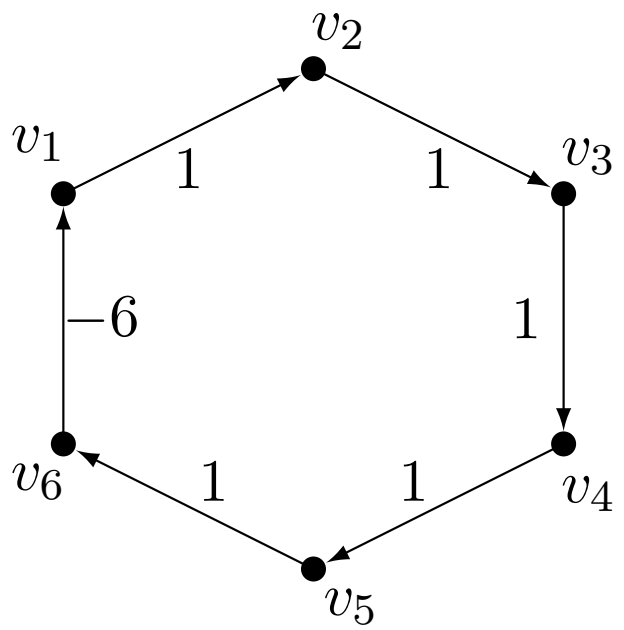
$$B_n[i] < B_{n-1}[i]$$

Man kann also in den Algorithmus von Bellman-Ford einen Test auf negative Kreise einbauen, indem man auch für alle $i \in V$ $B_n[i]$ berechnet und am Ende den folgenden Befehl einfügt:

for $i := 1$ **to** n **do**
  if $B_n[i] < B_{n-1}[i]$ **then** stop „Negativer Kreis“ **fi**

4.3 Modifikation des Floyd-Algorithmus

Falls kein negativer Kreis existiert, funktioniert der Algorithmus weiterhin korrekt.



$$c_{16}^6 = 5 = c_{16}^5$$

$$c_{61}^6 = -6 = c_{61}^5$$

$$c_{11}^6 = \min\{c_{11}^5, c_{16}^5 + c_{61}^5\} = -1$$

$\Rightarrow$ der Graph enthält einen negativen Kreis, gdw ein $c_{ii}^n < 0$ existiert.

Man kann also in den Algorithmus von Floyd einen Test auf negative Kreise einbauen, indem man am Ende den folgenden Befehl einfügt:

for $i := 1$ **to** n **do**
    if $c_{ii}^n < 0$ **then** stop „Negativer Kreis“ **fi**

4.4 Der Algorithmus von Johnson

Definition 113
Sei $\mathrm{d} : A \to \mathbb{R}$ eine Distanzfunktion. Eine Abbildung

$$r : V \to \mathbb{R}$$

heißt Rekalibrierung, falls gilt:

$$(\forall (u,v) \in A)[r(u) + \mathrm{d}(u,v) \geq r(v)]$$

Beobachtung: Sei r eine Rekalibrierung (für d). Setze $\mathrm{d}'(u,v) := \mathrm{d}(u,v) + r(u) - r(v)$. Dann gilt:

$$\mathrm{d}'(u,v) \geq 0$$

Sei $u = v_0 \to \cdots \to v_k = v$ ein Pfad. Dann ist:

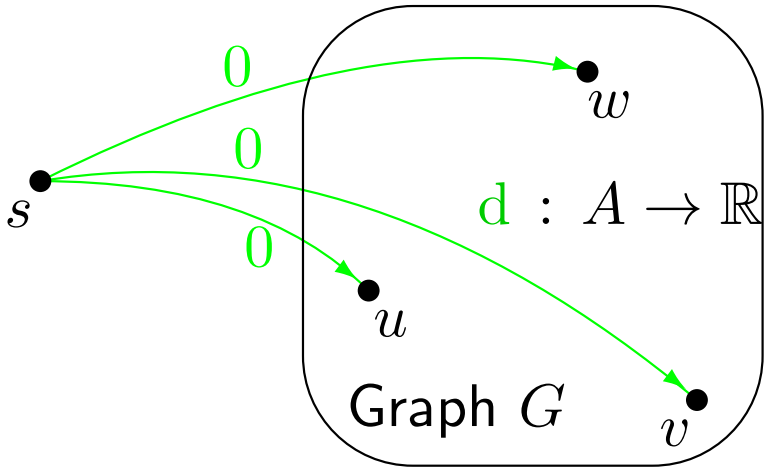
$$\text{d-Länge} := \sum_{i=0}^{k-1} \mathrm{d}(v_i, v_{i+1})$$

Demnach ist:

$$\begin{aligned}
\text{d}'\text{-Länge} &= \sum_{i=0}^{k-1} \mathrm{d}'(v_i, v_{i+1}) \\
&= \sum_{i=0}^{k-1} (\mathrm{d}(v_i, v_{i+1}) + r(v_i) - r(v_{i+1})) \\
&= \sum_{i=0}^{k-1} \mathrm{d}(v_i, v_{i+1}) + r(v_0) - r(v_k)
\end{aligned}$$

Also ist ein d-kürzester Pfad von u $(= v_0)$ nach v $(= v_k)$ auch ein d'-kürzester Pfad und umgekehrt. Nach einer Rekalibrierung kann man also auch die Algorithmen anwenden, die eine nichtnegative Distanzfunktion d voraussetzen (z.B. Dijkstra).

Berechnung einer Rekalibrierung:



Füge einen neuen Knoten s hinzu und verbinde s mit jedem anderen Knoten $v \in V$ durch eine Kante der Länge 0.

Berechne sssp von s nach allen anderen Knoten $v \in V$ (z.B. mit Bellman-Ford). Sei $r(v)$ die dadurch berechnete Entfernung von s zu $v \in V$. Dann ist r eine Rekalibrierung, denn es gilt:

$$r(u) + \mathrm{d}(u, v) \geq r(v) \,.$$

5. Zusammenfassung

	$d \geq 0$	d allgemein
sssp	D (Fibonacci): $\mathcal{O}(m + n \cdot \log n)$ D (Radix): $\mathcal{O}(m + n\sqrt{\log C})$	B-F: $\mathcal{O}(n \cdot m)$
apsp	D: $\mathcal{O}(n \cdot m + n^2 \min\{\log n, \sqrt{\log C}\})$ F: $\mathcal{O}(n^3)^{(*)}$	J: $\mathcal{O}(n \cdot m + n^2 \log n)$ F: $\mathcal{O}(n^3)$

Bemerkung$^{(*)}$: In der Praxis ist der Floyd-Algorithmus für kleine n besser als Dijkstra's Algorithmus.

6. Transitive Hülle

6.1 Min-Plus-Matrix-Produkt und Min-Plus-Transitive Hülle

Ring $\mathbb{Z}(+, \times)$ — Gruppe, Halbgruppe

Semiring $\mathbb{N}(+, \times)$ — Halbgruppe, Halbgruppe

Wir betrachten den (kommutativen) Semiring über $\mathbb{R} \cup \{\infty\}$ mit den Operationen $\min$ und $+$. Für jede der beiden Operationen haben wir ein Monoid. Es gilt das Distributivgesetz $a + \min\{b, c\} = \min\{a + b, a + c\}$.

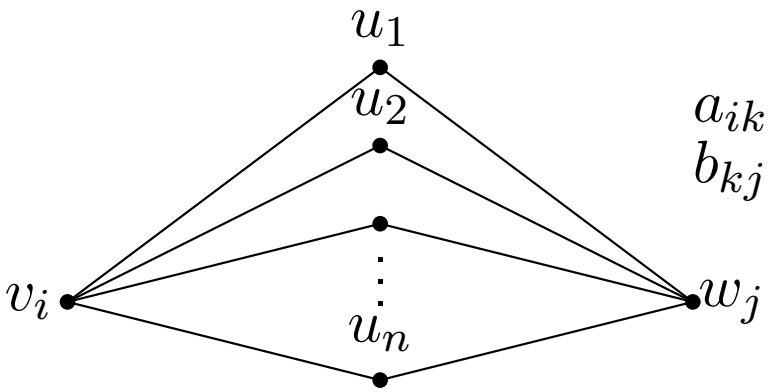
Normale Matrixmultiplikation:

$$A = (a_{ij})_{1 \le i,j \le n}, \quad B = (b_{ij})_{1 \le i,j \le n}, \quad I = (\delta_{ij})_{1 \le i,j \le n}$$

$$C = A \cdot B = (c_{ij})_{1 \le i,j \le n}, \quad c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Entsprechend für Min-Plus:

$$c_{ij} = \min\{a_{ik} + b_{kj};\ 1 \le k \le n\}$$



$$a_{ik} = \mathrm{d}(v_i, u_k)$$
$$b_{kj} = \mathrm{d}(u_k, w_j)$$

Anwendung:
kürzeste Wege von v_i nach w_j in einem Graph ($A = B$); dabei ist

$$I_{\min,+} = \begin{pmatrix} 0 & & \infty \\ & \ddots & \\ \infty & & 0 \end{pmatrix}$$

Sei A Entfernungsmatrix, $A = (a_{ij})_{1 \le i,j \le n} = (\mathrm{d}(v_i, v_j))_{1 \le i,j \le n}$.
Setze $a_{ii} = 0$ für $i = 1, \ldots, n$.

Betrachte A^2 mit dem Min-Plus-Produkt, $A^2 =: (a_{ij}^{(2)})_{1 \le i,j \le n}$.

Dann ist $a_{ij}^{(2)}$ die Länge eines kürzesten Pfades von v_i nach v_j, der höchstens zwei Kanten enthält. Induktion ergibt: $a_{ij}^{(k)}$ ist die Länge eines kürzesten Pfades von v_i nach v_j mit höchstens k Kanten.
Falls die $\mathrm{d}(v_i, v_j)$ alle ≥ 0 sind, gibt es immer kürzeste Pfade, die höchstens $n - 1$ Kanten enthalten.

Damit ergibt sich folgende alternative Lösung des all-pairs-shortest-path-Problems:

Berechne A^{n-1} (Min-Plus)!

Es genügt auch, $A^{2^{\lceil \log(n-1) \rceil}}$ durch wiederholtes Quadrieren zu berechnen (nicht $A^2, A^3, A^4, \ldots$).

Definition 114
$A^* := \min_{i \geq 0}\{A^i\}$ heißt Min-Plus-Transitive Hülle.

Bemerkung: $\min$ wird komponentenweise gebildet. Wenn $\mathrm{d} \geq 0$, dann $A^* = A^{n-1}$.

6.2 Boolesche Matrixmultiplikation und Transitive Hülle

Wir ersetzen nun im vorhergehenden Abschnitt die Distanzmatrix durch die (boolesche) Adjazenzmatrix und $(\min, +)$ durch $(\vee, \wedge)$, d.h.:

$$C = A \cdot B; \quad c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$$

Wenn wir zudem $a_{ii} = 1$ für $1 \leq i \leq n$ setzen, dann gilt für A^k (boolesches Produkt, $A^0 = I$)

$$a_{ij} = \begin{cases} 1 & \text{falls es im Graphen einen Pfad von } v_i \text{ nach } v_j, \\ & \text{bestehend aus } \leq k \text{ Kanten, gibt} \\ 0 & \text{falls es im Graphen keinen Pfad von } v_i \text{ nach } v_j, \\ & \text{bestehend aus } \leq k \text{ Kanten, gibt} \end{cases}$$

Transitive Hülle:

$$A^* := \bigvee_{i \geq 0} A^i \qquad (= A^{n-1})$$

ist damit die Adjazenzmatrix der transitiven Hülle des zugrunde liegenden Digraphen.

Satz 115

Sei $M(n)$ die Zeitkomplexität für das boolesche Produkt zweier $n \times n$-Matrizen, $T(n)$ die Zeitkomplexität für die transitive Hülle einer $n \times n$ booleschen Matrix.

Falls $\underbrace{T(3n) \leq cT(n)}_{\text{sicher erfüllt, falls } T \text{ polynomiell}}$ *und* $\underbrace{M(2n) \geq 4M(n)}_{\text{sicher erfüllt, falls } M(n) \geq n^2}$*, dann gilt:*

$$T(n) = \Theta(M(n))\,.$$

Beweis:

(1) Matrixmultiplikation $\prec$ transitive Hülle:

Seien boolesche Matrizen A, B gegeben und ihr boolesches Produkt $C = A \cdot B$ gesucht.

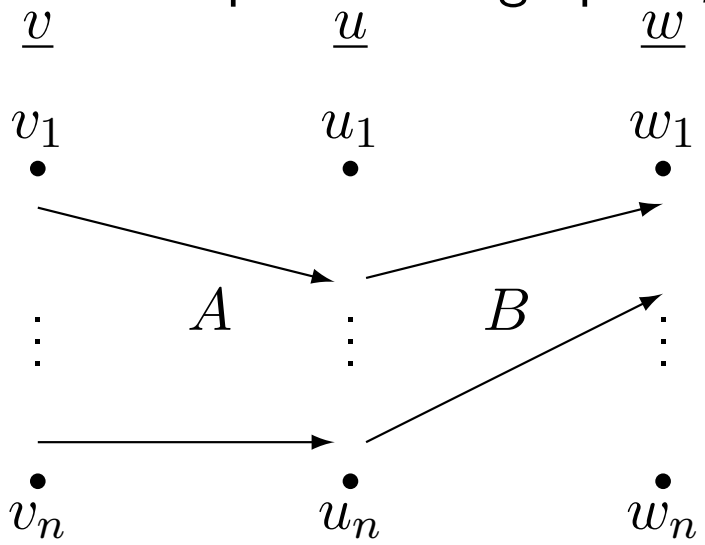
Setze:

$$L = \underbrace{\left.\begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}\right\} 3n}_{3n}$$

Beweis (Forts.):

L ist die Adjazenzmatrix eines tripartiten Digraphen, denn:

	$\underline{v}$	$\underline{u}$	$\underline{w}$
$\underline{v}$	0	A	0
$\underline{u}$	0	0	B
$\underline{w}$	0	0	0



Daher kann L^* leicht bestimmt werden:

$$L^* = \begin{pmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix} \qquad (= I \vee L \vee L^2)$$

Also gilt: $M(n) \leq T(3n) = \mathcal{O}(T(n))$.

Beweis (Forts.):

(2) Transitive Hülle $\prec$ Matrixmultiplikation:
Gegeben: $n \times n$ boolesche Matrix L; gesucht: L^*; Annahme: n ist Zweierpotenz. Teile auf:

$$L = \underbrace{\begin{pmatrix} A & B \\ C & D \end{pmatrix}}_{\frac{n}{2}\ \ \frac{n}{2}} \begin{matrix} \} \frac{n}{2} \\ \} \frac{n}{2} \end{matrix} ; \quad L^* = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

Es gilt also:

$E =$	$(A \vee BD^*C)^*$	betrachte alle Pfade von der ersten Hälfte der Knoten zur ersten Hälfte
$F =$	EBD^*	analog
$G =$	D^*CE	analog
$H =$	$D^* \vee GF$	analog

Beweis (Forts.):

Um L^* zu berechnen, benötigen wir zwei Transitive-Hülle-Berechnungen und sechs Matrixprodukte für Matrizen der Dimension $\frac{n}{2} \times \frac{n}{2}$ (nämlich $M_1 = D^*C$, $M_2 = BM_1$, $M_3 = EB$, $M_4 = M_3D^*$, $M_5 = M_1E$, $M_6 = GF$), plus den Aufwand für $\vee$, der $\leq c'n^2$ ist. Wir zeigen nun durch Induktion ($n = 1\surd$), dass $T(n) \leq cM(n)$:

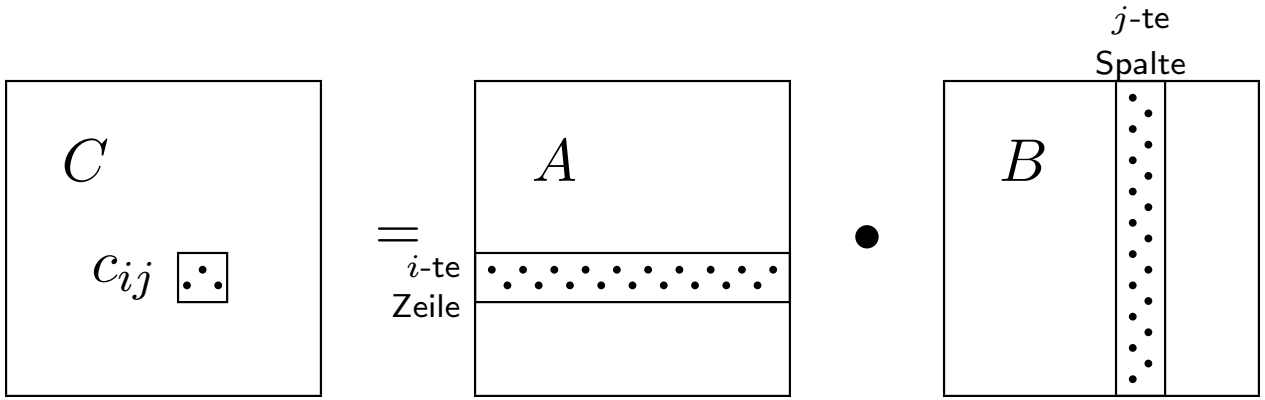
$$
\begin{array}{lll}
T(n) & \leq 2T(\frac{n}{2}) + 6M(\frac{n}{2}) + c'n^2 & \\
& \leq 2cM(\frac{n}{2}) + 6M(\frac{n}{2}) + c'n^2 & \left|\begin{array}{ll}\text{Vor.:} & M(2n) \geq 4M(n) \\ & \text{da } M(n) \geq n^2\end{array}\right. \\
& \leq \frac{1}{4}(2c + 6 + 4c')M(n) & \\
& \leq cM(n) &
\end{array}
$$

falls $c \geq \frac{1}{4}(2c + 6 + 4c')$, also falls $c \geq 3 + 2c'$.

Also $T(n) = \mathcal{O}(M(n))$. □

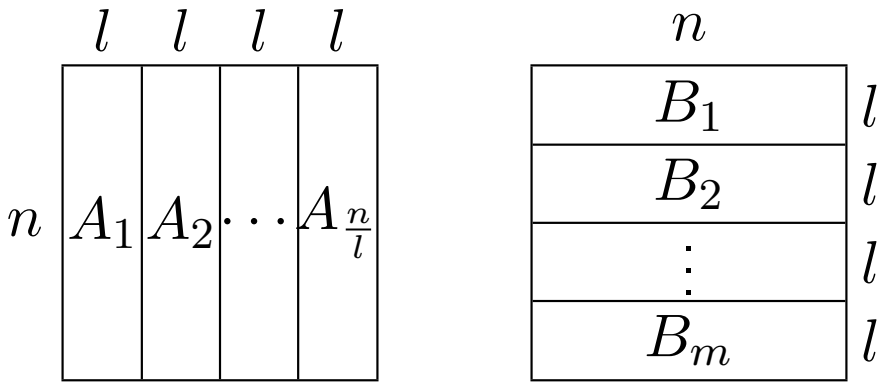
6.3 Der 4-Russen-Algorithmus für boolesche Matrixmultiplikation

Gegeben zwei boolesche $n \times n$ Matrizen A, B; gesucht $C = A \cdot B$.



Sei $l := \lfloor \log n \rfloor$, o.B.d.A. gelte $l | n$ (l teilt n).

Teile A auf (setze $m := \frac{n}{l}$):



Sei $A = A'_1 \vee A'_2 \vee \ldots \vee A'_m$, $B = B'_1 \vee B'_2 \vee \ldots \vee B'_m$,
$C_i := A'_i \cdot B'_i$ für $i = 1, \ldots, m$. Dann gilt

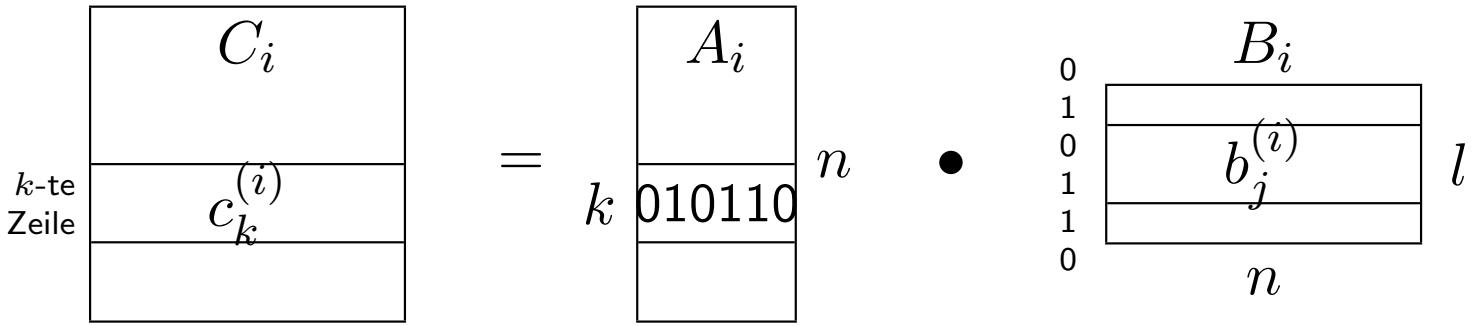
$$C = \bigvee_{i=1}^{m} C_i, \text{ da}$$

$$C = AB = \left(\bigvee_{i=1}^{m} A'_i\right)\left(\bigvee_{i=1}^{m} B'_i\right) = \bigvee_{\substack{1 \le i \le m \\ 1 \le j \le m}} A'_i B'_j = \bigvee_{i=1}^{m} A_i B_i,$$

da $A'_i B'_j = 0$ für $i \neq j$ (A'_i und B'_j sind ja $n \times n$ Matrizen mit 0 außerhalb des jeweiligen Streifens).

Gegeben die C_i's, benötigen wir Zeit $\mathcal{O}(mn^2)$.

Betrachte eine Zeile von C_i:



$$c_k^{(i)} = \bigvee_{j=1}^{l} a_k^{(i)} \cdot b_j^{(i)}$$

Der Algorithmus berechnet einfach zunächst alle booleschen Linearkombinationen der Zeilen von B_i (Prozedur bcomb) und damit $c_k^{(i)}$ für alle überhaupt möglichen $a_k^{(i)}$.
Betrachte A, B und C als Matrizen von Zeilenvektoren:

$$A = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, \quad B = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}, \quad C = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$$

proc bcomb(**int** i) =
  comb$[0] := [0, \ldots, 0]$
  for $j := 1$ **to** $2^{\lfloor \log n \rfloor} - 1$ **do**
    $p := \lfloor \log j \rfloor$ **co** p Index der vordersten 1 von j **oc**
    comb$[j]$:=comb$[j - 2^p] \vee b_{(i-1)\lfloor \log n \rfloor + 1 + p}$
  od

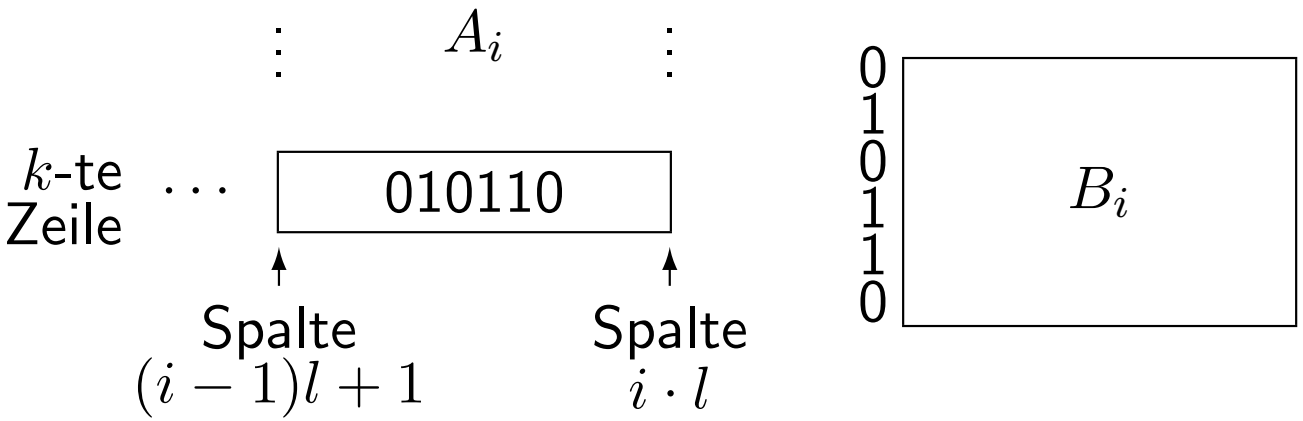
Zeitbedarf:

(a) sequentiell: $\boxed{\mathcal{O}(n^2)}$

(b) Vektoroperationen der Breite n: $\boxed{\mathcal{O}(n)}$

```
algorithm four-russians(array a, b, c) =
  co a, b, c als Vektoren von n Zeilenvektoren organisiert oc
  const l = ⌊log n⌋ co wir nehmen an l|n oc
  array comb[0..2^{l-1}] of boolean-vector; int nc
  for i := 1 to n do c[i] := [0, ..., 0] od
  for i := 1 to n/l do    co berechne die C_i's oc
      bcomb(i)
      for j := 1 to n do
          co Bitmuster in Binärzahl wandeln oc
          nc := 0
          for k := i · l downto (i − 1) · l + 1 do
              nc := nc + nc+ if a[j, k] then 1 else 0 fi
          od
          c[j] := c[j]∨comb[nc]
      od
  od
```

Beispiel 116



Zeitbedarf:

(a) sequentiell:

$$\mathcal{O}\left(\frac{n}{l}\cdot\left(n^2+n(l+n)\right)\right)=\mathcal{O}\left(\frac{n^3}{l}\right)=\boxed{\mathcal{O}\left(\frac{n^3}{\log n}\right)}$$

(b) Vektorrechner der Breite n (Interpretation eines Bitintervalls als Zahl in $\mathcal{O}(1)$ Zeit):

$$\mathcal{O}\left(\frac{n}{l}\cdot\left(n+n(1+1)\right)\right)=\boxed{\mathcal{O}\left(\frac{n^2}{\log n}\right)\text{ (Vektoroperationen)}}$$

Satz 117

Der 4-Russen-Algorithmus berechnet das Produkt zweier boolescher Matrizen sequentiell in Zeit $\mathcal{O}\left(\frac{n^3}{\log n}\right)$ *bzw. mit* $\mathcal{O}\left(\frac{n^2}{\log n}\right)$ *Bitvektoroperationen der Breite* n*.*

Beweis:

s.o. □

V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, I.A. Faradzev:
On economical construction of the transitive closure of an oriented graph
Soviet Math. Dokl. **11**, pp. 1209–1210 (1970)

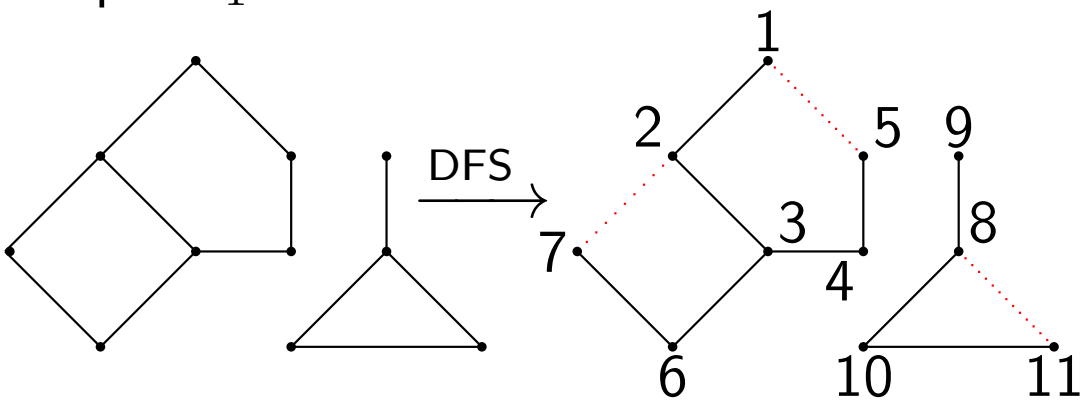
6.4 Transitive Hülle und DFS

Wir erinnern uns an den DFS-Algorithmus:

for all nodes v **do** unmark v **od**
$count := 0$
while $\exists$ unvisited v **do**
    $r :=$ pick (random) unvisited node
    push r onto stack
    while stack $\neq \emptyset$ **do**
        $v :=$ pop top element
        if v unvisited **then**
            mark v visited
            push all neighbours of v onto stack
            num[v]:= $++count$
        fi
    od
od

Beispiel 118

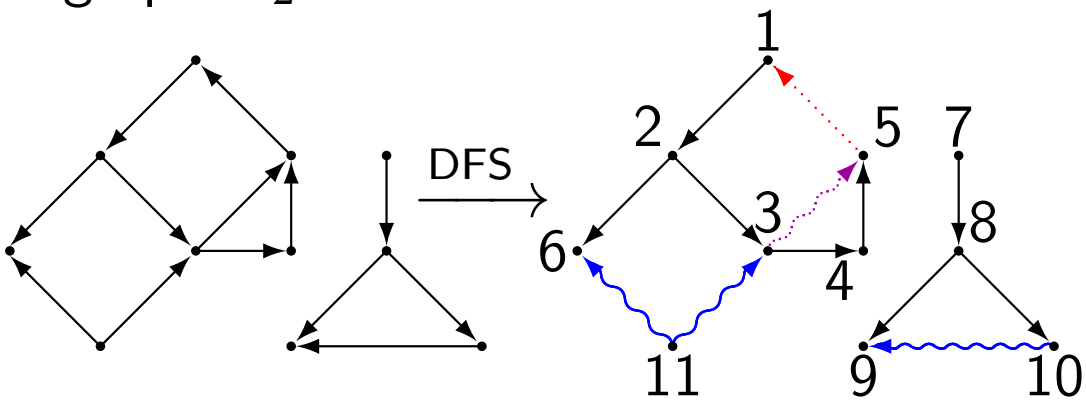
Graph G_1:



Bezeichnungen:

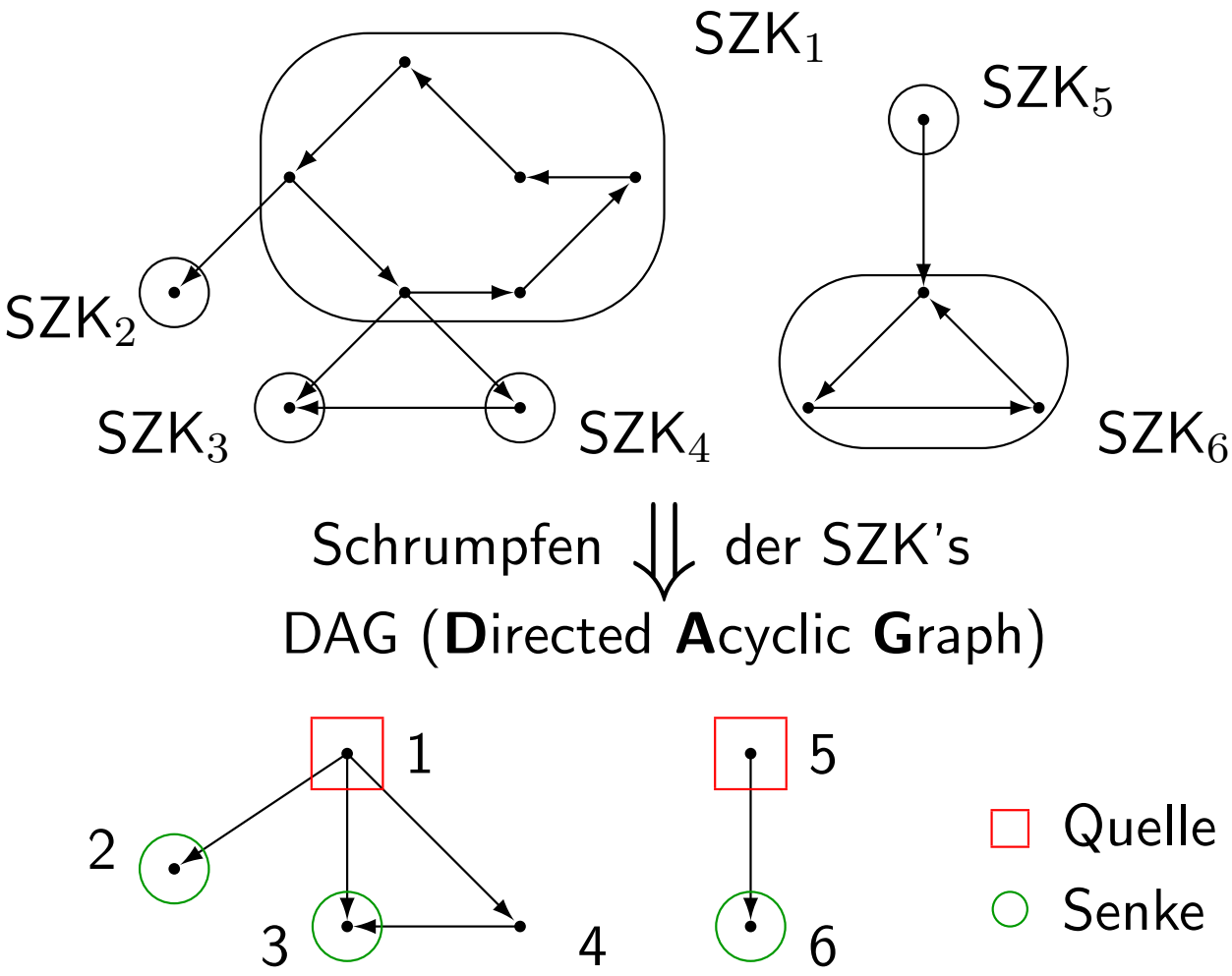
- Baumkante
- Rückwärtskante
- Querkante
- Vorwärtskante

Digraph G_2:

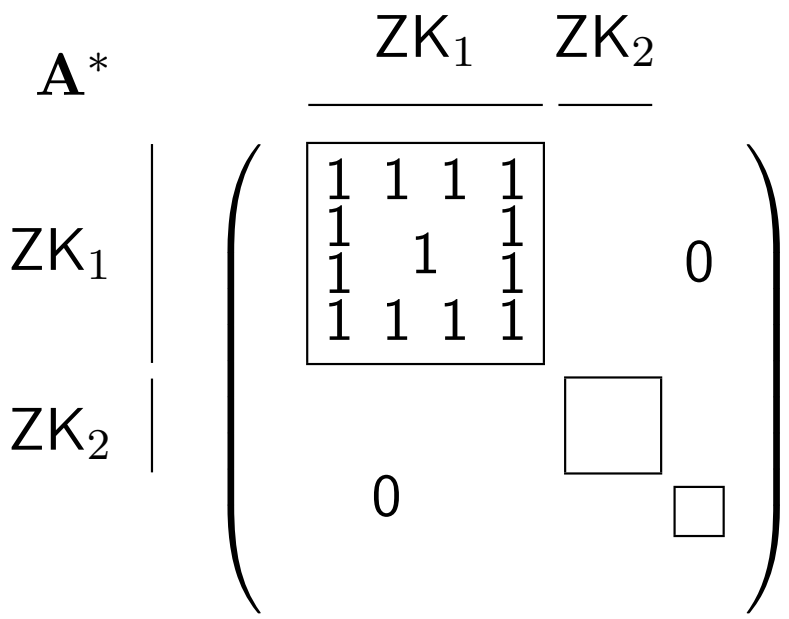


Bem.: Alle Baumkanten zusammen: DFS-Wald (Spannwald).

Beispiel 119 (Starke Zusammenhangskomponenten (in Digraphen))

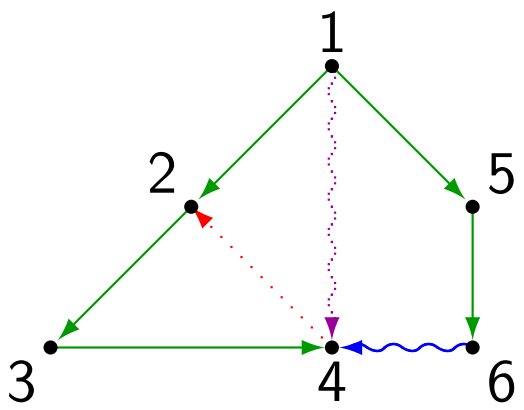


DFS in ungerichteten Graphen mit c Zusammenhangskomponenten, n Knoten, m Kanten: $n - c$ Baumkanten, $m - n + c$ Rückwärtskanten, Zeitaufwand $\mathcal{O}(n + m)$



A^* aus DFS mit Aufwand $\Theta(n^2)$

DFS in gerichteten Graphen (Digraphen) mit n Knoten, m Kanten:
Baum-, Vorwärts-, Rückwärts- und Querkanten:



Bezeichnungen:

Baumkanten
Rückwärtskanten
Querkanten
Vorwärtskanten

Zeitaufwand: $\boxed{\mathcal{O}(n+m)}$

u ist von v aus auf einem gerichteten Pfad erreichbar gdw u Knoten in dem DFS-Baum (DFS-visit(v)) mit Wurzel v ist.

Also: Transitive Hülle in Zeit $\mathcal{O}(n \cdot (m+n))$. Sehr gut für dünne Graphen.

7. Ein besserer Algorithmus für das apsd-Problem in ungewichteten Digraphen

Sei $G = (V, E)$ ein Digraph mit der Knotenmenge $\{0, \ldots, n-1\}$, dessen Kanten alle die Länge 1 haben.

Sei $D = (d_{ij})_{0 \le i,j < n}$ die zugehörige Matrix, mit Einträgen $d_{ij} \in \{0, 1, \infty\}$.

Entsprechend sei D^* die Distanzmatrix, so dass

$$d^*_{ij} = \text{Länge eines kürzesten Wegs von } i \text{ nach } j$$

Setze weiterhin

$$D^{(l)} = (d^{(l)}_{ij})_{0 \le i,j < n} \text{ mit } d^{(l)}_{ij} = \begin{cases} d^*_{ij} & \text{falls } d^*_{ij} \le l \\ \infty & \text{sonst} \end{cases}$$

Algorithmus a1

co Sei $A = (a_{ij})_{0 \le i,j < n}$ mit $a_{ij} = \begin{cases} 1 \text{ falls } d_{ij} \le 1 \\ 0 \text{ sonst} \end{cases}$ **oc**

$B := I$ **co** boolesche Einheitsmatrix **oc**

for $l := 2$ **to** $r + 1$ **do**

  $B := A \cdot B$ **co** boolesches Matrixprodukt **oc**

  for $0 \le i, j < n$ **do**

    if $b_{ij} = 1$ **then** $d_{ij}^{(l)} := l$ **else** $d_{ij}^{(l)} := \infty$ **fi**

    if $d_{ij}^{(l-1)} \le l$ **then** $d_{ij}^{(l)} := d_{ij}^{(l-1)}$ **fi**

  od

od

Dieser Algorithmus berechnet $D^{(1)} = D$, $D^{(2)}$, $D^{(3)}, \ldots, D^{(r)}$.

Sei ω eine Zahl ≥ 2, so dass die Matrixmultiplikation in Zeit $\mathcal{O}(n^\omega)$ durchgeführt werden kann (Winograd/Coppersmith: $\omega \leq 2,376$).

Zeitaufwand für Algorithmus a1: $\boxed{\mathcal{O}(rn^\omega)}$

Algorithmus apsd =

Berechne, mit Algorithmus a1, $D^{(l)}$ für $l = 1, \ldots, r$; $l := r$

for $s := 1$ **to** $\left\lceil \log_{\frac{3}{2}} \frac{n}{r} \right\rceil$ **do**

  for $i := 0$ **to** $n - 1$ **do**

    finde in Zeile i von $D^{(l)}$ das d, $\left\lceil \frac{l}{2} \right\rceil \leq d \leq l$, das in dieser Zeile am wenigsten oft vorkommt

    $S_i :=$ Menge der zugehörigen Spaltenindizes

  od

  $l_1 := \left\lceil \frac{3}{2} l \right\rceil$

  for $0 \leq i, j < n$ **do**

    $m_{ij} :=$ **if** $S_i \neq \emptyset$ **then** $\min_{k \in S_i} \{d_{ik}^{(l)} + d_{kj}^{(l)}\}$ **else** ∞ **fi**

    if $d_{ij}^{(l)} \leq l$ **then** $d_{ij}^{(l_1)} := d_{ij}^{(l)}$

    elif $m_{ij} \leq l_1$ **then** $d_{ij}^{(l_1)} = m_{ij}$ **else** $d_{ij}^{(l_1)} := \infty$ **fi**

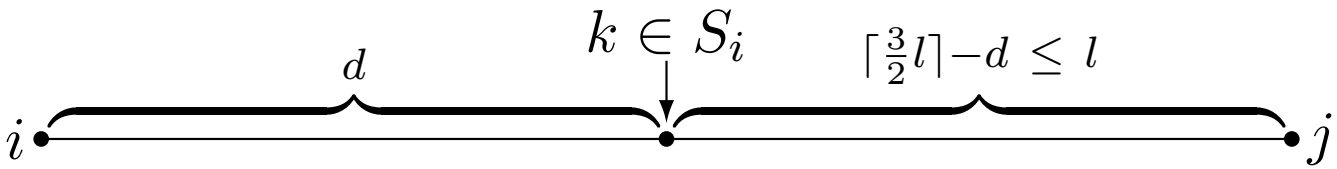
  od

  $l := l_1$

od

apsd berechnet
zunächst $D^{(1)}, \ldots, D^{(r)}$, dann $D^{(l)}$ für

$$l = r, \left\lceil \frac{3}{2} r \right\rceil, \left\lceil \frac{3}{2} \left\lceil \frac{3}{2} r \right\rceil \right\rceil, \ldots, \underbrace{n'}_{\geq n}$$



Da für d $\frac{l}{2}$ Werte zur Auswahl stehen, gilt:

$$|S_i| \leq \frac{2n}{l}$$

Damit ist die Laufzeit

$$\mathcal{O}\left(rn^{\omega}+\sum_{s=1}^{\left\lceil \log_{\frac{3}{2}} \frac{n}{r}\right\rceil}\frac{n^3}{\left(\frac{3}{2}\right)^s\cdot r}\right)=\mathcal{O}\left(rn^{\omega}+\frac{n^3}{r}\right)$$

Setze r so, dass die beiden Summanden gleich sind, also $r=n^{\frac{3-\omega}{2}}$. Damit ergibt sich für apsd die Laufzeit $\mathcal{O}\left(n^{\frac{3+\omega}{2}}\right)$.

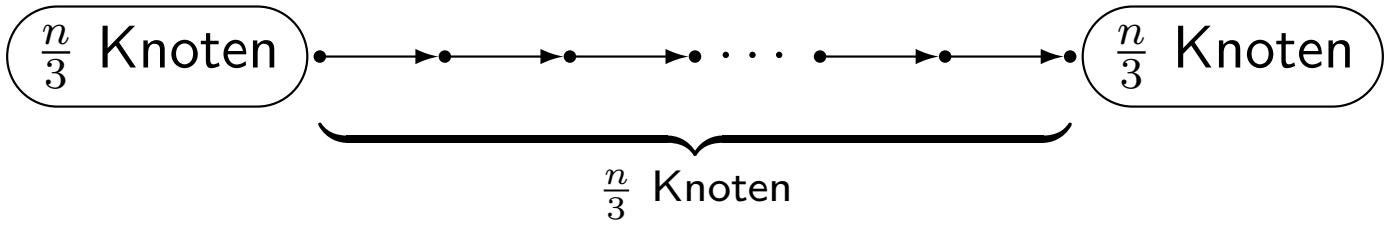
Satz 120

Das all-pairs-shortest-distance-Problem für Digraphen mit Kantenlänge $=1$ lässt sich in Zeit $\mathcal{O}(n^{\frac{3+\omega}{2}})$ lösen (ω Exponent für Matrixmultiplikation).

Beweis:

$\surd$ □

Bemerkung: Beim apsp kann die Größe der Ausgabe $\Omega(n^3)$ sein:

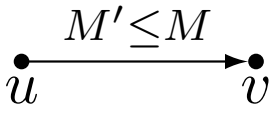


Verwende stattdessen die kürzeste-Pfade-Nachfolger-Matrix $N \in [0, \ldots, n-1]^{n \times n}$, wo n_{ij} die Nummer des zweiten Knoten auf „dem“ kürzesten Pfad von Knoten i zu Knoten j ist.
Eine solche Matrixdarstellung für die kürzesten Pfade zwischen allen Knotenpaaren kann in Zeit $\mathcal{O}(n^{\frac{3+\omega}{2}} \cdot \log^c n)$ (für ein geeignetes $c > 0$) bestimmt werden.

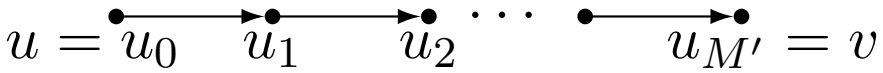
Satz 121

Für Digraphen mit Kantenlängen $\in \{1, 2, \ldots, M\}, M \in \mathbb{N}$, *kann APSD in Zeit* $\mathcal{O}((Mn)^{\frac{3+\omega}{2}})$ *gelöst werden.*

Beweis:

Idee: Ersetze 

durch:



□

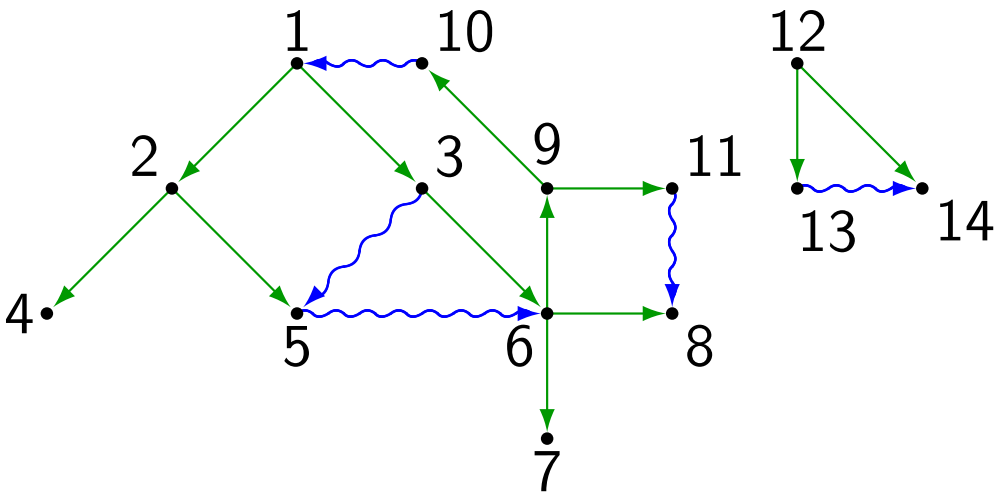
8. Ein Algorithmus für die transitive Hülle in Digraphen mit linearer erwarteter Laufzeit

Wir nehmen an, dass die Wahrscheinlichkeit eines Digraphen mit n Knoten und m Kanten eine Funktion (nur) von n und m ist.
Daraus folgt (wir lassen der Einfachheit halber Schleifen ($=$ Kanten $v \to v$) zu), dass jede Kante (u, v) mit Wahrscheinlichkeit $\frac{m}{n^2}$ vorhanden ist, falls wir Digraphen mit n Knoten und m Kanten betrachten.

Erinnerung: Breitensuche (BFS): Schlange, queue, FIFO:

Durchlaufe Graphen, indem wir, solange möglich, den vordersten Knoten v aus der Queue nehmen, ihn behandeln und die Liste seiner noch nicht behandelten Nachbarn in die Queue hinten einfügen.

Beispiel 122



Bezeichnungen:

Baumkanten

Querkanten

algorithm exp-lin-transitive-closure

0. Konstruiere die linear geordneten Adjazenzlisten L_i^r, $i = 1, \ldots, n$ des Graphen G^r (entsteht aus G durch Umkehrung aller Kanten).
   Beispiel:

$$\begin{array}{llllll|llllll} L_1 & = & 4 & 1 & 2 & & L_1^r & = & 1 & 2 & 4 & \\ L_2 & = & 1 & 4 & 3 & & L_2^r & = & 1 & 3 & 4 & \\ L_3 & = & 3 & 2 & & & L_3^r & = & 2 & 3 & & \\ L_4 & = & 1 & 4 & 2 & & L_4^r & = & 1 & 2 & 4 & \end{array}$$

   Ersetze ebenfalls alle L_i durch L_i^{rr} ($\rightarrow$ sortierte Adjazenzlisten)

1. Berechne für jeden Knoten i in BFS-Art eine Liste S_i von von i aus erreichbaren Knoten, so dass (i) oder (ii) gilt:
   (i) $|S_i| < \lfloor \frac{n}{2} \rfloor + 1$ und S_i enthält alle von i aus erreichbaren Knoten
   (ii) $|S_i| = \lfloor \frac{n}{2} \rfloor + 1$
2. Entsprechende Listen P_i von Vorgängern von i
3. **for** $i := 1$ **to** n **do**
   $$L_i^* := S_i \cup \{j; i \in P_j\} \cup \{j;\ |S_i| = |P_j| = \lfloor \tfrac{n}{2} \rfloor + 1\}$$
   od
   bilde $(L_i^*)^{rr}$ für $i = 1, \ldots, n$

Korrektheit: $j \in L_i^*$ gdw (i,j) in transitiver Hülle.

Satz 123

Sei die Wahrscheinlichkeit eines Graphen (lediglich) eine Funktion der Anzahl n seiner Knoten und der Anzahl m seiner Kanten. Dann ist der Erwartungswert für die Laufzeit des obigen Algorithmus $\mathcal{O}(n + m^*)$. Dabei ist m^* der Erwartungswert für die Anzahl der Kanten in der transitiven Hülle.

Beweis:

Das Durchlaufen des Graphen (von v_1 aus für die Konstruktion von S_1) in BFS-Manier liefert eine Folge $(a_t)_{t\geq 1}$ von Knoten. Sei $L_{\sigma(\nu)}$ die ν-te Adjazenzliste, die in der BFS erkundet wird,

$$L_{\sigma(\nu)} = (a_t;\ h(\nu) < t \leq h(\nu+1))$$

Sei $\Delta L_{\sigma(\nu)}$ die Menge der a_t in $L_{\sigma(\nu)}$, und sei (für $i = 1$)

$$S_1(t) := \{a_1, a_2, \ldots, a_t\}.$$

Wie bereits gezeigt, ist

$$\Pr[j \in L_i] = \frac{m}{n^2}, \quad \forall i, j$$

Beweis (Forts.):

Alle n-Tupel von geordneten Listen $L_1, L_2, \ldots, L_n$ mit

$$m = \sum_{i=1}^{n} |L_i|$$

sind aufgrund der Voraussetzungen über die Wahrscheinlichkeitsverteilung gleich wahrscheinlich. Also:
Beobachtung 1: Die Mengen $\Delta L_{\sigma(\nu)}$, $\nu = 1, 2, \ldots$ sind (paarweise) unabhängig.
Beobachtung 2: Die $\Delta L_{\sigma(\nu)}$ sind gleichverteilt, d.h. für alle $A \subseteq \{1, \ldots, n\}$ gilt:

$$\Pr[\Delta L_{\sigma(\nu)} = A] = (\frac{m}{n^2})^{|A|}(1 - \frac{m}{n^2})^{n-|A|}$$

Beweis (Forts.):

Lemma 124

Sei $A \subseteq \{1, \ldots, n\}$ mit $|A| = k$. Sei $(\bar{a}_t)_{t \geq 1}$ eine Folge von unabhängigen gleichverteilten Zufallsvariablen mit Wertemenge $\{1, \ldots, n\}$. Dann gilt:

$$\min\{t; \bar{a}_t \notin A\} \text{ hat Erwartungswert } 1 + \frac{k}{n-k}.$$

Beweis:

[des Lemmas] $\Pr[\bar{a}_1, \ldots, \bar{a}_r \in A \text{ und } \bar{a}_{r+1} \notin A] = \left(\frac{k}{n}\right)^r \cdot \left(1 - \frac{k}{n}\right)$.
Also ist der Erwartungswert für $\min\{t; \bar{a}_t \notin A\}$:

$$1 + \sum_{r=0}^{\infty} r \left(\frac{k}{n}\right)^r \left(1 - \frac{k}{n}\right) = 1 + \frac{k}{n}\left(1 - \frac{k}{n}\right) \sum_{r \geq 1} r \left(\frac{k}{n}\right)^{r-1}$$

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

$$= 1 + \frac{\frac{k}{n}}{1 - \frac{k}{n}} = 1 + \frac{k}{n-k}\,.$$

□

Anmerkung:

$$\sum_{r=0}^{\infty} r z^{r-1} = \frac{d}{dz} \frac{1}{1-z} = \frac{1}{(1-z)^2}$$

Lemma 125

Sei $(\bar{a}_t)_{t\geq 1}$ *wie oben,* $k \leq \frac{n}{2}$*. Dann hat* $\min\{t; |\{\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_t\}| > k\}$ *Erwartungswert* $\leq \frac{3}{2}(k+1)$*.*

Beweis:

[des Lemmas] Wegen der Additivität des Erwartungswertes gilt: Der gesuchte Erwartungswert ist

$$\leq \sum_{\nu=0}^{k} \left(1 + \frac{\nu}{n-\nu}\right)$$

$$\leq k + 1 + \sum_{\nu=1}^{k} \frac{\nu}{\frac{n}{2}}$$

$$\leq k + 1 + \frac{k(k+1)}{n} \leq \frac{3}{2}(k+1)\,.$$

□

Beweis (Forts.):

Wenn wir jedes $L_{\sigma(\nu)}$ in sich zufällig permutieren, erhalten wir eine Folge $(\bar{a}'_t)_{t\geq 1}$ von Zufallsvariablen, so dass

$$|\{\bar{a}_1,\ldots,\bar{a}_t\}| = |\{\bar{a}'_1,\ldots,\bar{a}'_t\}| \text{ für } t = h(\nu), \quad \forall \nu$$

Da die $\bar{a}'_t$ innerhalb eines jeden Intervalls $h(\nu) < t \leq h(\nu+1)$ paarweise verschieden sind, gilt für sie die vorhergehende Abschätzung erst recht. Wir betrachten nun aus Symmetriegründen o.B.d.A. lediglich den Knoten $i = 1$. Dann sind $(|S_1(t)|)_{t\geq 1}$ und $(|S_1(t)'|)_{t\geq 1}$ gleichverteilte Folgen von Zufallsvariablen, denn dies gilt zunächst für alle Zeitpunkte der Form $t = h(y)$; da aber für diese Zeitpunkte $S_1(t) = S'_1(t)$ ist und $h(\ )$ zufällig verteilt ist, ist auch die Verteilung der beiden Folgen zwischen diesen Intervallgrenzen identisch.

Beweis (Forts.):

Sei s der Erwartungswert und $|S_i|$ die tatsächliche Größe von S_i nach Phase 1. Dann

$$s = \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor + 1} k \cdot \Pr[|S_i| = k] \,.$$

Die erwartete Anzahl der Schritte (und damit die Kosten) in der BFS sei q. Dann gilt:

$$q \leq \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor + 1} \Pr[|S_i| = k] \frac{3}{2} k = \frac{3}{2} s.$$

Also ist der Aufwand des Algorithmus für die Phasen 1 und 2 im Erwartungswert $\leq 3sn$. Da $ns \leq m^*$ und die Kosten der anderen Phasen offensichtlich durch $\mathcal{O}(n + m + m^*)$ beschränkt sind, folgt die Behauptung. □

C.P. Schnorr:

An algorithm for transitive closure with linear expected time

SIAM J. Comput. **7**(2), pp. 127–133 (1978)

9. Matrixmultiplikation à la Strassen

Betrachte das Produkt zweier 2×2-Matrizen:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}; \quad c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}$$

Man beachte, dass diese und die folgenden Formeln nicht die Kommutativität der Multiplikation voraussetzen. Sie gelten also auch, falls die a_{ij}, b_{ij}, c_{ij} eigentlich $n \times n$-Matrizen A_{ij}, B_{ij}, C_{ij} sind (jeweils $i, j \in \{1, 2\}$).

Bilde:

$$
\begin{aligned}
m_1 &:= (a_{12} - a_{22})(b_{21} + b_{22}) \\
m_2 &:= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_3 &:= (a_{11} - a_{21})(b_{11} + b_{12}) \\
m_4 &:= (a_{11} + a_{12})b_{22} \\
m_5 &:= a_{11}(b_{12} - b_{22}) \\
m_6 &:= a_{22}(b_{21} - b_{11}) \\
m_7 &:= (a_{21} + a_{22})b_{11}
\end{aligned}
$$

Dann gilt:

$$
\begin{aligned}
c_{11} &:= m_1 + m_2 - m_4 + m_6 \\
c_{12} &:= m_4 + m_5 \\
c_{21} &:= m_6 + m_7 \\
c_{22} &:= m_2 - m_3 + m_5 - m_7
\end{aligned}
$$

Sei $n = 2^k$ und $M(n)$ die Anzahl arithmetischer Operationen (Mult, Add, Sub), die Strassen's Algorithmus bei $n \times n$ Matrizen benötigt:

$$M(1) = 1$$
$$M(n) = 7M\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$$

Expansion dieser Rekursionsgleichung $\Rightarrow$

$$\begin{aligned} M(n) &= 7^{k+1} - 6n^2 = 7 \cdot 2^{k \log_2 7} - 6n^2 \\ &= 7n^{\log_2 7} - 6n^2 \\ &< 7n^{2,80736} - 6n^2 \\ &= \mathcal{O}(n^{2,81})\,. \end{aligned}$$

Der Exponent ω für die Matrixmultiplikation ist also $< 2,81$.

proc $\textsc{matmult}(A, B, n) =$

co A und B sind $n \times n$ Matrizen **oc**

if $n < 16$ **then** berechne $A \cdot B$ mit klassischer Methode

elif n gerade **then**

spalte A und B in je 4 $\frac{n}{2} \times \frac{n}{2}$ Matrizen auf und wende Strassen's Formeln an. Führe die Multiplikationen rekursiv mit $\textsc{matmult}$ aus.

else

spalte A und B in eine $(n-1) \times (n-1)$ Matrix (A_{11}, B_{11}) und drei verbleibende Blöcke auf. Wende $\textsc{matmult}$ rekursiv auf A_{11}, B_{11} an und berechne die anderen Produkte mit der klassischen Methode.

fi

Satz 126

Der MATMULT*-Algorithmus hat folgende Eigenschaften:*

1. *Für* $n < 16$ *ist die Anzahl der arithmetischen Operationen genauso hoch wie bei der klassischen Methode.*
2. *Für* $n \geq 16$ *ist die Anzahl der arithmetischen Operationen echt kleiner als bei der klassischen Methode.*
3. MATMULT *braucht nie mehr als* $4,91n^{\log_2 7}$ *arithmetische Operationen.*

Beweis:

1. $\surd$

Beweis (Forts.):

2. Sei n gerade und seien A, B zwei $n \times n$ Matrizen. Wir wenden Strassen's Formeln an, führen aber die 7 Multiplikationen der $\frac{n}{2} \times \frac{n}{2}$ Matrizen mit der klassischen Methode aus.
Gesamtzahl arithmetischer Operationen:

$$7\left(2\left(\frac{n}{2}\right)^3 - \left(\frac{n}{2}\right)^2\right) + 18\left(\frac{n}{2}\right)^2 = \frac{7}{4}n^3 + \frac{11}{4}n^2$$

Dies ist besser als die klassische Methode, falls:

$$\frac{7}{4}n^3 + \frac{11}{4}n^2 < 2n^3 - n^2 \Leftrightarrow \frac{15}{4}n^2 < \frac{1}{4}n^3 \Leftrightarrow 15 < n \Leftrightarrow n \geq 16$$

Sei n ungerade. Multiplizieren wir $A \cdot B$ mit der klassischen Methode, so auch $A_{11} \cdot B_{11}$. Also brauchen wir durch Anwendung von Strassen's Formeln auf die $(n-1) \times (n-1)$ Matrizen ($n-1$ gerade) weniger Operationen, wenn $n - 1 \geq 16$ ist.

Beweis (Forts.):

Sei $M'(n)$ die Anzahl arithmetischer Operationen in MATMULT.
Dann ist:

$$M'(n) = \begin{cases} 2n^3 - n^2 & \text{falls } n < 16 \\ 7M'\left(\frac{n}{2}\right) + \frac{18}{4}n^2 & \text{falls } n \geq 16 \text{ gerade} \\ 7M'\left(\frac{n-1}{2}\right) + \frac{42}{4}n^2 - 17n + \frac{15}{2} & \text{falls } n \geq 16 \text{ ungerade} \end{cases}$$

Wir definieren für $x \in \mathbb{R}^+$:

$$\bar{M}(x) = \begin{cases} 2x^3 - x^2 & \text{falls } x < 32 \\ 7\bar{M}\left(\frac{x}{2}\right) + \frac{42}{4}x^2 & \text{falls } x \geq 32 \end{cases}$$

Dann gilt:

$$\bar{M}(n) \geq M'(n) \text{ für alle } n \in \mathbb{N}\,.$$

Beweis (Forts.):

Es ist

$$\bar{M}(x) = \sum_{i=0}^{k-1} \left[7^i \cdot \frac{42}{4} \left(\frac{x}{2^i} \right)^2 \right] + 7^k \left(2 \cdot \left(\frac{x}{2^k} \right)^3 - \left(\frac{x}{2^k} \right)^2 \right)$$

für $x \geq 32$, wobei $k := \min \left\{ l \,\middle|\, \frac{x}{2^l} < 32 \right\}$.

Mit $k = \lfloor \log_2 x \rfloor - 4 =: \log_2 x - t$ erhalten wir $t \in [4, 5[$ und

$$\bar{M}(x) \leq 7^{\log_2 x} \left(13 \left(\frac{4}{7} \right)^t + 2 \left(\frac{8}{7} \right)^t \right) \leq 4,91 \cdot 7^{\log_2 x} \text{ für } x \geq 32.$$

Für $x < 32$ direkt nachrechnen. □


Volker Strassen:
Gaussian elimination is not optimal
Numer. Math. **13**, pp. 354–356 (1969)


Victor Ya. Pan:
New fast algorithms for matrix operations
SIAM J. Comput. **9**(2), pp. 321–342 (1980)

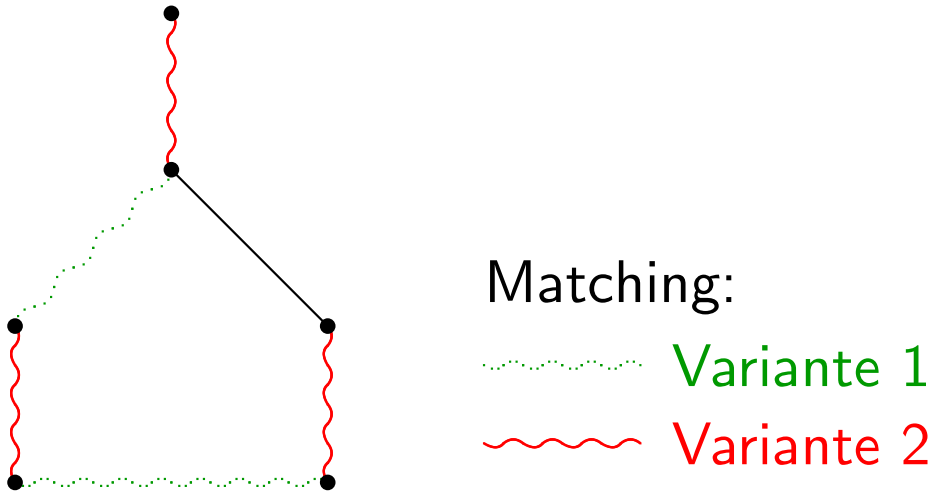

Don Coppersmith, Shmuel Winograd:
Matrix multiplication via arithmetic progressions
J. Symbolic Computation **9**(3), pp. 251–280 (1990)

Kapitel VI Matchings in Graphen

1. Grundlagen

Definition 127

Sei $G = (V, E)$ ein ungerichteter, schlichter Graph. Ein Matching M in G ist eine Teilmenge von E, so dass keine zwei Kanten aus M einen Endpunkt gemeinsam haben.



Definition 128

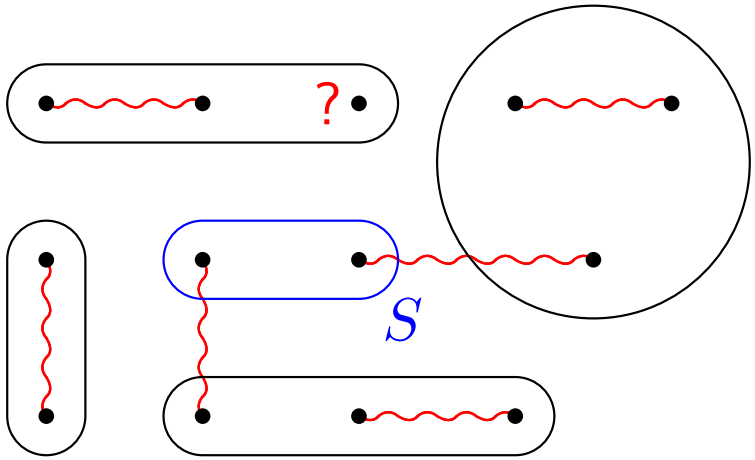
1. Ein Matching M in $G = (V, E)$ heißt perfekt (oder vollkommen), falls
$$|M| = \frac{|V|}{2}$$
2. Ein Matching M heißt „Matching maximaler Kardinalität“ (engl. maximum matching) in G, falls es in G kein Matching M' mit $|M'| > |M|$ gibt (durchgehend geringelt/rot im Beispiel)
3. Ein Matching M heißt maximal in G, falls es bezüglich „$\subseteq$“ maximal ist (engl. maximal matching) (gepunktet geringelt/grün im Beispiel)

Definition 129

Sei $G = (V, E)$ ein Graph. Wenn V in zwei nichtleere Teilmengen V_1 und V_2 partitioniert werden kann ($V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$), so dass $E \subseteq V_1 \times V_2$ ist, dann heißt G bipartit ($G = (V_1, V_2, E)$).

Satz 130

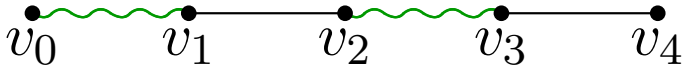
(ohne Beweis) Ein Graph $G = (V, E)$ hat ein perfektes Matching genau dann, wenn $|V|$ gerade ist und es kein $S \subseteq V$ gibt, so dass der durch $V \setminus S$ induzierte Teilgraph mehr als $|S|$ Zusammenhangskomponenten ungerader Größe enthält.



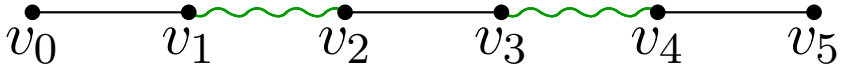
Bemerkung: Die Richtung „$\Rightarrow$“ ist klar, da in einem perfekten Matching in jeder „ungeraden“ ZHK mindestens ein Knoten mit einem Knoten in S gematcht sein muss.

Definition 131

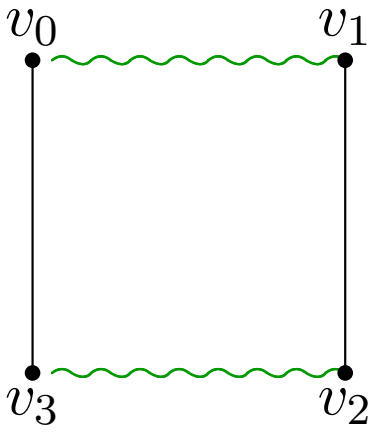
1. Ein einfacher Pfad (Kreis) $v_0, v_1, \ldots, v_r$ heißt alternierend bzgl. eines Matchings M, falls die Kanten $\{v_i, v_{i+1}\}$, $0 \leq i < r$, abwechselnd in M und nicht in M liegen.



gerade Länge

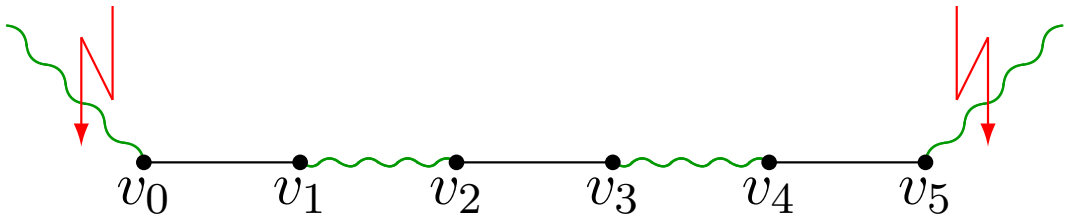


ungerade Länge



Definition 131

2. Ein alternierender Pfad bzgl. eines Matchings M heißt augmentierend, falls er bzgl. „$\subseteq$“ maximal ist und an beiden Enden ungematchte Knoten hat.



ungerade Länge

Bemerkung: Es kann keine augmentierenden Kreise geben.

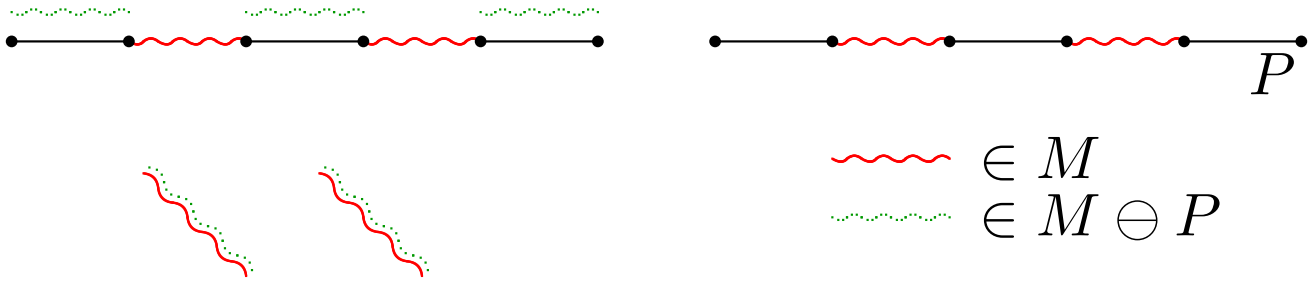
Definition 132

Seien S, T zwei Mengen, dann bezeichne $S \ominus T$ die symmetrische Differenz von S und T, d.h. $S \ominus T = (S - T) \cup (T - S)$.

Lemma 133 (Augmentierung eines Matchings)

Sei M ein Matching, P ein augmentierender Pfad bzgl. M. Dann ist auch $M \ominus P$ ein Matching, und es gilt $|M \ominus P| = |M| + 1$.

Beweis:



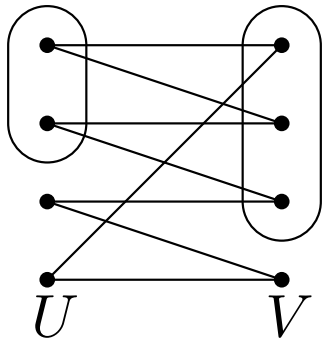
□

Satz 134 (Heiratssatz [Frobenius, Hall, Rado, König])

Sei $G = (U, V, E)$ ein bipartiter Graph. G enthält ein Matching der Kardinalität $|U|$ genau dann, wenn gilt:

$$\forall A \subseteq U : |N(A)| \geq |A|,$$

wobei $N(A)$ die Nachbarschaft von A (in V) bezeichnet.



Beweis:

Die Richtung („$\Rightarrow$“) ist klar.

„$\Leftarrow$“: M sei ein maximum Matching in G. Annahme: $|M| < |U|$. Sei $A' \subseteq U$ die Teilmenge der durch M nicht gematchten Knoten in U. Seien weiter A (bzw. B) die von A' aus mittels alternierender Pfade erreichbaren Knoten in U (bzw. V). Enthält B einen ungematchten Knoten, dann ist ein dazugehöriger alternierender Pfad P augmentierend und $|M \ominus P| > |M|$, im Widerspruch zur Voraussetzung.

Andernfalls ist $|A| > |B|$ (da A zu jedem Knoten in B seinen gematchten Partner enthält), aber auch, im Widerspruch zur Voraussetzung, $N(A) \subseteq B$, also $|N(A)| < |A|$. $\square$

Alternativer Beweis:

Die Richtung („⇒“) ist (noch immer) klar.

„⇐“: Sei M ein Matching in G, mit $|M| < |U|$, und sei $u_0 \in U$ ein in M ungematchter Knoten. Da $|N(\{u_0\})| \geq 1$, hat u_0 einen Nachbarn $v_1 \in V$. Falls v_1 ungematcht ist, sind wir fertig, da wir einen augmentierenden Pfad gefunden haben. Andernfalls sei $u_1 \in U$ der mit v_1 gematchte Knoten. Dann ist $u_1 \neq u_0$ und $|N(\{u_0, u_1\})| \geq 2$, und es gibt einen Knoten $v_2 \in V$, der zu u_0 oder u_1 adjazent ist.

Falls v_2 in M nicht gematcht ist, beenden wir die Konstruktion, andernfalls fahren wir in obiger Weise fort und erreichen, wegen $|V| < \infty$, schließlich einen ungematchten Knoten $v_r \in V$. Damit haben wir aber wiederum einen augmentierenden Pfad $P = (v_r, u_{i_1}, v_{i_1}, \ldots, u_{i_k}, v_{i_k}, u_0)$ (mit $i_1 > \cdots > i_k$) gefunden! □

Definition 135

Sei $G = (V, E)$ ein (ungerichteter) Graph. Eine Teilmenge $D \subseteq V$ heißt Träger von G (engl. vertex cover), falls D mit jeder Kante in E mindestens einen Knoten gemeinsam hat (also „jede Kante bedeckt“).

Beobachtung: Sei D ein Träger von G und M ein Matching in G. Dann gilt offensichtlich

$$|D| \geq |M| \,,$$

da der Träger D jede Kante im Matching M treffen muss und die Kanten in M alle paarweise disjunkt sind.

Korollar 136

Seien D und M wie oben. Dann gilt

$$\min_{D}\{|D|\} \geq \max_{M}\{|M|\} \,.$$

Satz 137

Sei $G = (U, V, E)$ ein (ungerichteter) bipartiter Graph. Dann gilt

$$\min\{|D|;\ D \text{ Träger von } G\} = \max\{|M|;\ M \text{ Matching in } G\}$$

Beweis:

Wir nehmen o.B.d.A. an, dass alle Knoten in G Grad ≥ 1 haben. Sei M ein maximum Matching in G, sei $A \subseteq U$ die von M gematchte Teilmenge von U, $B \subseteq V$ die von V.

Sei nun B' die Menge der Knoten in V, die von $U \setminus A$ aus mittels eines alternierenden Pfades erreichbar sind. Dann ist $B' \subseteq B$, da wir andernfalls einen augmentierenden Pfad gefunden haben. Sei A' die Menge der Knoten in A, die durch M mit Knoten in B' gematcht sind. Dann gibt es keine Kante zwischen A' und $V \setminus B$, da jede solche Kante wiederum zu einem augmentierenden Pfad führen würde.

Es gilt (i) $N(U \setminus A) \subseteq B'$, (ii) $N(A') = B'$, (sowie (iii) $N(V \setminus B) \subseteq A \setminus A'$). Damit ist $B' \cup (A \setminus A')$ ein Träger von G. Da $|A| = |M|$ und $|A'| = |B'|$, folgt die Behauptung. □

Satz 138 (Berge (1957))

Ein Matching hat maximale Kardinalität genau dann, wenn es keinen augmentierenden Pfad dafür gibt.

Beweis:

s.u. □

Lemma 139

Seien M, N Matchings in G, und sei $|N| > |M|$. Dann enthält $N \ominus M$ mindestens $|N| - |M|$ knotendisjunkte augmentierende Pfade bzgl. M.

Beweis:

Der Grad eines Knotens in $(V, N \ominus M)$ ist ≤ 2. Die Zusammenhangskomponenten von $(V, N \ominus M)$ sind also

1. isolierte Knoten
2. einfache Kreise (gerader Länge)
3. alternierende Pfade

Seien $C_1, \ldots, C_r$ die Zusammenhangskomponenten in $(V, N \ominus M)$, dann gilt:

$$M \ominus \underbrace{C_1 \ominus C_2 \ominus \cdots \ominus C_r}_{N \ominus M} = N$$

Nur die C_i's, die augmentierende Pfade bzgl. M sind, vergrößern die Kardinalität des Matchings, und zwar jeweils um genau 1. Also muss es mindestens $|N| - |M|$ C_i's geben, die augmentierende Pfade bzgl. M sind (und knotendisjunkt, da ZHKs).

Korollar 140

Satz von Berge

2. Kürzeste augmentierende Pfade

Lemma 141

Sei M ein Matching der Kardinalität r, und sei s die maximale Kardinalität eines Matchings in $G = (V, E)$, $s > r$. Dann gibt es einen augmentierenden Pfad bzgl. M der Länge $\leq 2 \left\lfloor \frac{r}{s-r} \right\rfloor + 1$.

Beweis:

Sei N ein Matching maximaler Kardinalität in G, $|N| = s$. $N \ominus M$ enthält $\geq s - r$ augmentierende Pfade bzgl. M, die alle knotendisjunkt und damit auch kantendisjunkt sind. Falls einer dieser Pfade Länge 1 hat, sind wir fertig. Ansonsten enthält mindestens einer dieser augmentierenden Pfade $\leq \left\lfloor \frac{r}{s-r} \right\rfloor$ Kanten aus M. □

Lemma 142

Sei P ein kürzester augmentierender Pfad bzgl. M, und sei P' ein augmentierender Pfad bzgl. des neuen Matchings $M \ominus P$. Dann gilt:

$$|P'| \geq |P| + 2|P \cap P'|$$

Beweis:

$N = M \ominus P \ominus P'$, also $|N| = |M| + 2$. Also enthält $M \ominus N$ mindestens 2 knotendisjunkte augmentierende Pfade bzgl. M, etwa P_1 und P_2. Es gilt:

$$\begin{aligned} |N \ominus M| &= |P \ominus P'| \\ &= |(P - P') \cup (P' - P)| \\ &= |P| + |P'| - 2|P \cap P'| \\ &\geq |P_1| + |P_2| \geq 2|P|, \end{aligned}$$

also

$$|P| + |P'| - 2|P \cap P'| \geq 2|P|,$$

also

$$|P'| \geq \underbrace{2|P| - |P|}_{|P|} + 2|P \cap P'|$$

□

Schema für Matching-Algorithmus:

1. Beginne mit Matching $M_0 := \emptyset$.
2. Berechne Folge $M_0,\ P_0,\ M_1,\ P_1, \ldots,\ M_i,\ P_i, \ldots$, wobei jeweils P_i ein kürzester augmentierender Pfad bzgl. M_i ist und

$$M_{i+1} := M_i \ominus P_i\,.$$

Im obigen Schema gilt

$$|P_{i+1}| \geq |P_i| \text{ für alle } i\,.$$

Lemma 143

Seien P_i und P_j in obiger Folge zwei augmentierende Pfade gleicher Länge. Dann sind P_i und P_j knotendisjunkt.

Beweis:

Annahme: es gibt eine Folge $(P_k)_{k\geq 0}$ mit $|P_i| = |P_j|$, $j > i$, $P_i \cap P_j \neq 0$, $j - i$ minimal gewählt. Durch die Wahl von j sind die Pfade $P_{i+1}, \ldots, P_j$ knotendisjunkt. Also ist P_j auch ein augmentierender Pfad bzgl. $M' := M_{i+1}$, dem Matching nach den Augmentierungen mit $P_0,\ P_1, \ldots, P_i$. Aus dem vorhergehenden Lemma folgt $|P_j| \geq |P_i| + 2|P_i \cap P_j|$, also P_i, P_j kantendisjunkt, da $|P_i| = |P_j|$. Da in M' jeder Knoten in P_i gematcht ist (und dies dann auch in $M' \ominus P_{i+1} \ominus P_{i+2} \ominus \cdots \ominus P_{j-1}$ gilt), können auch P_i und P_j keinen Knoten gemeinsam haben. □

Satz 144

Sei s die maximale Kardinalität eines Matchings in $G = (V, E)$. Dann enthält die Folge $|P_0|, |P_1|, \ldots$ höchstens $\lfloor 2\sqrt{s} + 1 \rfloor$ verschiedene Werte.

Beweis:

Sei $r := \lfloor s - \sqrt{s} \rfloor$ $(= s - \lceil \sqrt{s} \rceil)$. Per Konstruktion ist $|M_i| = i$, also $|M_r| = r$. Mit Lemma 141 folgt

$$|P_r| \leq 2 \left\lfloor \frac{\lfloor s - \sqrt{s} \rfloor}{s - \lfloor s - \sqrt{s} \rfloor} \right\rfloor + 1 \leq 2 \left\lfloor \frac{s}{\sqrt{s}} \right\rfloor + 1 = 2 \lfloor \sqrt{s} \rfloor + 1.$$

Für $i \leq r$ ist also $|P_i|$ eine der ungeraden Zahlen in $[1, 2\sqrt{s} + 1]$, also eine von $\lfloor \sqrt{s} \rfloor + 1$ ungeraden Zahlen. $P_{r+1}, \ldots, P_{s-1}$ tragen höchstens $s - r - 1 < \sqrt{s}$ zusätzliche Längen bei. □

Verfeinertes Schema:

$M := \emptyset$
while $\exists$ augmentierender Pfad bzgl. M **do**
$l :=$ Länge eines kürzesten augmentierenden Pfades bzgl. M
bestimme eine bzgl. „$\subseteq$“ maximale Menge $\{Q_1, \ldots, Q_k\}$ augmentierender Pfade bzgl. M, die alle Länge l haben und knotendisjunkt sind
$M := M \ominus Q_1 \ominus \cdots \ominus Q_k$
od

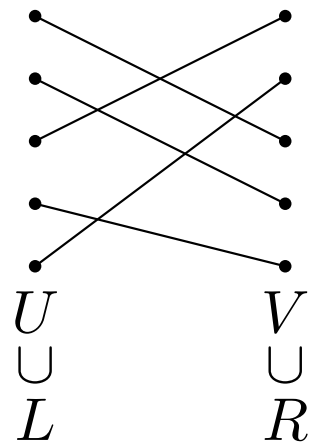
Korollar 145
Die obige $while$-Schleife wird höchstens $\mathcal{O}\left(|V|^{\frac{1}{2}}\right)$ mal durchlaufen.

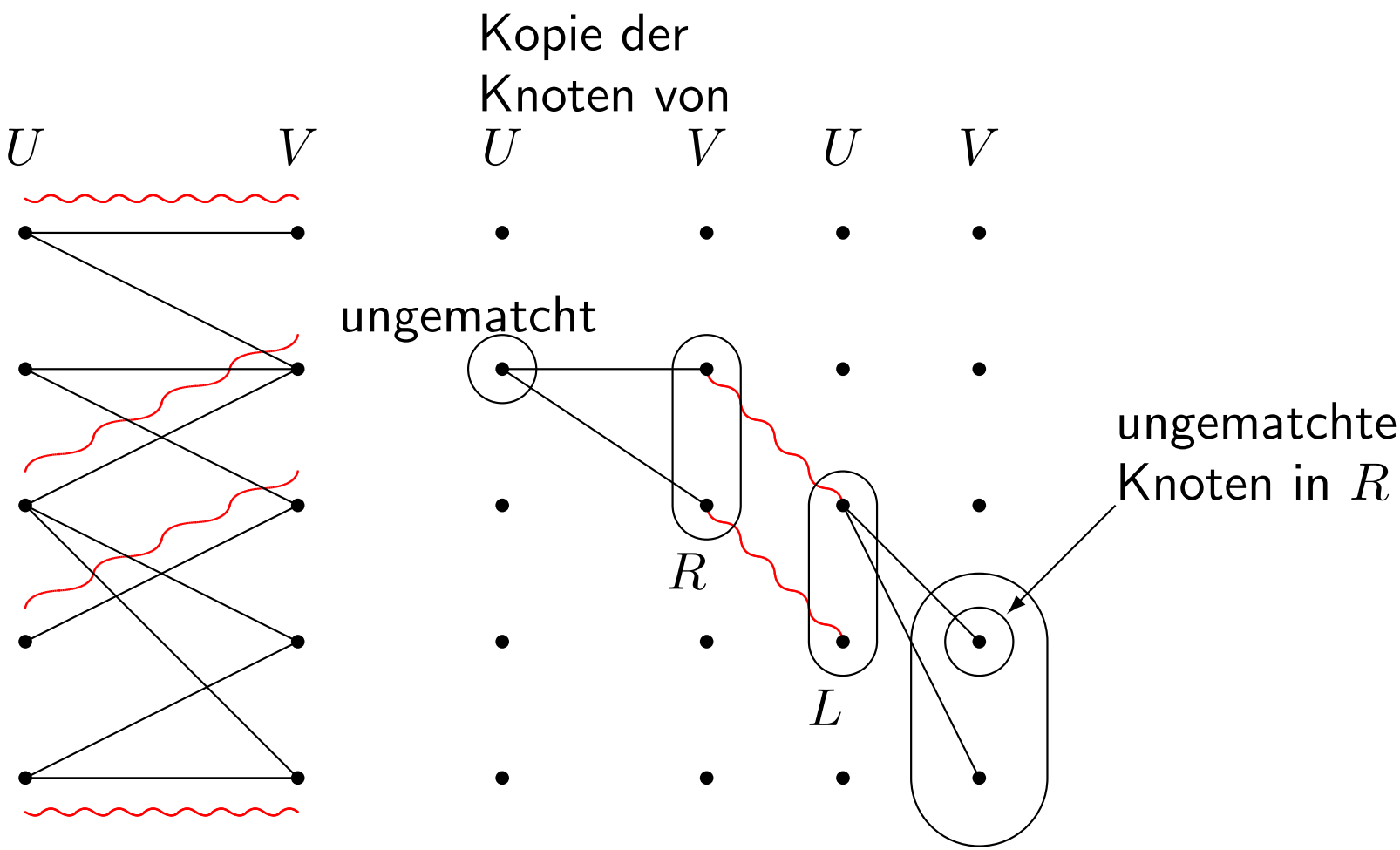
3. Maximum Matchings in bipartiten Graphen

Sei $G = (U, V, E)$ ein bipartiter Graph, M ein Matching in G.

Zur Bestimmung der Länge eines kürzesten augmentierten Pfades bzgl. M führen wir eine simultane alternierende BFS durch, die von allen in M ungematchten Knoten in U aus startet.

for alle $v \in U \cup V$ **do** $label[v] := 0$ **od**
$R := \emptyset$; $l := 1$
for alle ungematchten Knoten $v \in U$ **do**
    for alle $\{v, w\} \in E$ **do** $label[w] := 1$; $R := R \cup \{w\}$ **od**
od
while $R \neq \emptyset$ **and** R enthält keinen ungematchten Knoten **do**
    $L := \emptyset$; $l++$
    for $w \in R, \{v, w\} \in M$ **do** $L := L \cup \{v\}$; $label[v] := l$ **od**
    $R := \emptyset$; $l++$
    for alle $v \in L, \{v, w\} \in E \setminus M$ **do**
        if $label[w] = 0$ **then**
            $R := R \cup \{w\}$; $label[w] := l$
        fi
    od
od
$R :=$ Menge der ungematchten Knoten in R




Kopie der
Knoten von
U
V
U
V
U
V
ungematcht
R
L
ungematchte
Knoten in R

Nachdem wir die Länge l eines kürzesten augmentierenden Pfades bzgl. M ermittelt haben, führen wir nacheinander von jedem ungematchten Knoten in U aus eine (zwischen ungematchten und gematchten Kanten) alternierende DFS bis zur Tiefe l aus, wobei wir

1. wenn wir einen ungematchten Knoten (in Tiefe l) erreichen, einen kürzesten augmentierenden Pfad Q_i gefunden haben; für den weiteren Verlauf der DFSs markieren wir Q_i als gelöscht;
2. jede Kante, über die die DFS zurücksetzt, ebenfalls als gelöscht markieren.

Der Zeitaufwand für diese DFSs beträgt $\mathcal{O}(n+m)$, da wir jede Kante höchstens zweimal (einmal in der DFS vorwärts, einmal rückwärts) besuchen.

Lemma 146

Gegeben die Länge eines kürzesten augmentierenden Pfades, kann eine bzgl. „$\subseteq$“ maximale Menge kürzester augmentierender Pfade in Zeit $\mathcal{O}(n+m)$ gefunden werden.

Satz 147

In bipartiten Graphen kann ein Matching maximaler Kardinalität in Zeit

$$\mathcal{O}\left(n^{\frac{1}{2}}(n+m)\right)$$

gefunden werden.

Beweis:

Gemäß Korollar 145 genügen $\mathcal{O}(n^{\frac{1}{2}})$ Phasen, in denen jeweils mittels einer simultanen BFS und einer sequentiellen DFS (beide in Zeit $\mathcal{O}(n+m)$) eine maximale Menge kürzester augmentierender Pfade bestimmt wird. □

John Hopcroft, Richard Karp:
An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs
SIAM J. Comput. **2**(4), pp. 225–231 (1973)

4. Maximum Matchings in allgemeinen Graphen

In einem allgemeinen (nicht unbedingt bipartiten) Graphen können wir in Zeit $\mathcal{O}(n+m)$ jeweils einen kürzesten augmentierenden Pfad finden und erhalten damit

Satz 148

In einem Graph $G = (V, E)$ kann ein Matching maximaler Kardinalität in Zeit

$$\mathcal{O}(n \cdot m)$$

gefunden werden.

Für Maximum-Matching-Algorithmen in allgemeinen Graphen mit Laufzeit $\mathcal{O}(n^{\frac{1}{2}}(n+m))$ verweisen wir auf die Literatur:

Silvio Micali, Vijay V. Vazirani:

An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs

Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science, FOCS'80 (Syracuse, NY, October 13–15, 1980), pp. 17–27 (1980)

Vijay V. Vazirani:

A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm

Combinatorica **14**(1), pp. 71–109 (1994)

Norbert Blum:

A new approach to maximum matching in general graphs

Proceedings of the 17th International Colloquium on Automata, Languages and Programming, ICALP'90 (Warwick University, England, July 16–20, 1990), LNCS **443**, pp. 586–597 (1990)

5. Matchings in gewichteten bipartiten Graphen

5.1 Zerlegung doppelt stochastischer Matrizen

Sei M eine $n \times n$-Matrix mit reellen Einträgen $m_{ij} \geq 0$, so dass alle Zeilen- und alle Spaltensummen $= r > 0$ sind. Wir assoziieren mit M den bipartiten Graphen $G = G_M = (U, V, E)$, wobei $U = \{u_1, \ldots, u_n\}$ die n Zeilen und $V = \{v_1, \ldots, v_n\}$ die n Spalten von M repräsentiert. G enthält die Kante $\{u_i, v_j\}$ gdw $m_{ij} > 0$.

Ein Matching in G entspricht einer Menge von Einträgen $m_{ij} > 0$, die alle in verschiedenen Zeilen und Spalten vorkommen. Wir nennen eine solche Menge von Positionen eine Diagonale der Matrix M.

Ein Träger von M ist eine Menge von Zeilen und Spalten, die zusammen alle Matrixeinträge > 0 enthalten.

Beispiel 149

$$\begin{matrix} \mathbf{2} & 0 & 1 & 3 & 0 \\ 0 & 4 & 0 & 0 & \mathbf{2} \\ 3 & \mathbf{1} & 2 & 0 & 0 \\ 1 & 1 & \mathbf{3} & 0 & 1 \\ 0 & 0 & 0 & \mathbf{3} & 3 \end{matrix}$$

Die markierten Elemente bilden eine Diagonale der Größe $n = 5$.

Annahme: M hat keine Diagonale der Größe n. Dann gibt es nach Satz 137 z Zeilen und s Spalten von M mit $z + s < n$, die alle Einträge > 0 von M bedecken. Damit wäre aber

$$r \cdot n = \sum m_{ij} \leq r \cdot (z + s) < r \cdot n$$

Widerspruch.

Also existiert eine Diagonale der Größe n und ein entsprechendes perfektes Matching M_1 von G. Die Adjazenzmatrix P_1 von $G_1 = (V, M_1)$ enthält in jeder Zeile und Spalte genau eine 1, ist also eine so genannte Permutationsmatrix (alle anderen Einträge sind 0). Sei nun

$$m_1 := \min\{m_{ij};\ \{u_i, v_j\} \in M_1\}.$$

Die Matrix $M - m_1 M_1$ hat wiederum konstante Zeilen- und Spaltensummen (nämlich $r - m_1$) und strikt mehr Einträge $= 0$ als M.

Durch Iteration ergibt sich damit

Satz 150 (Birkhoff, von Neumann)

Sei M eine doppelt-stochastische Matrix (d.h. alle Zeilen- und Spaltensummen sind $= 1$, alle Einträge sind reell und ≥ 0), dann gibt es eine Darstellung

$$M = \sum_{i=1}^{k} m_i P_i \,,$$

wobei die P_i Permutationsmatrizen und die $m_i \in \mathbb{R}$, mit $m_i > 0$ und $\sum_{i=1}^{k} m_i = 1$.

Bemerkung: Jede doppelt-stochastische Matrix ist also eine Konvexkombination von endlich vielen Permutationsmatrizen.

5.2 Matchings in knotengewichteten bipartiten Graphen

Sei $G = (U, V, E)$ ein bipartiter Graph mit einer Gewichtsfunktion $w : U \to \mathbb{R}^+$. Wir suchen ein Matching M in G, so dass die Summe der Gewichte der gematchten Knoten in U maximiert wird. Eine Teilmenge $T \subseteq U$, die durch ein Matching in G gematcht werden kann, heißt auch Transversale.

Sei $\mathcal{T}$ die Menge der Transversalen von G (beachte: $\emptyset \in \mathcal{T}$).

Satz 151

Die Menge $\mathcal{T}$ der Transversalen von G bildet ein Matroid.

Beweis:
Es ist klar, dass $\emptyset \in \mathcal{T}$ und dass $\mathcal{T}$ unter Teilmengenbildung abgeschlossen ist. Seien T und T' Transversalen $\in \mathcal{T}$ mit $|T'| = |T| + 1$, und seien M bzw. M' zugehörige Matchings. Dann gibt es bzgl. M einen augmentierenden Pfad P. Ein Endknoten von P liegt in U. Augmentieren wir M mittels P, erhalten wir also eine Transversale der Kardinalität $|T| + 1$. □

Das greedy-Paradigma lässt sich also anwenden, um in Zeit $\mathcal{O}(n \cdot m)$ eine Transversale maximalen Gewichts zu konstruieren.

Hier ist das offizielle Ende
der Vorlesung
Effiziente Algorithmen und Datenstrukturen I
im Wintersemester 2010/11

Die folgenden Folien sind zusätzliches Material aus früheren Vorlesungen.

5.3 Matchings in kantengewichteten bipartiten Graphen

Sei nun $G = (U, V, E)$ ein bipartiter Graph mit einer Gewichtsfunktion w von den Kanten in die nichtnegativen reellen Zahlen. Wir können o.B.d.A. annehmen, dass $|U| = |V|$ $(= n)$ und dass $G = K_{n,n}$, indem wir geeignet Knoten sowie Kanten mit Gewicht 0 hinzunehmen.
Damit können wir auch o.B.d.A. voraussetzen, dass jedes Matching maximalen oder minimalen Gewichts in G perfekt ist.
Indem wir $w(u_i, v_j)$ durch $w_{max} - w(u_i, v_j)$ ersetzen (wobei w_{max} das größte auftretende Gewicht ist), reduziert sich das Problem, ein Matching maximalen/minimalen Gewichts zu finden, auf das, eines minimalen/maximalen Gewichts zu finden.

Wir betrachten daher o.B.d.A. das Problem, in G ein perfektes Matching minimalen Gewichts zu finden.

Wir suchen also eine Diagonale der Größe n mit minimalem Gewicht. Sei W die Gewichtsmatrix. Verändern wir das Gewicht eines jeden Elements einer Zeile/Spalte von W um einen festen Betrag δ, so ändert sich das Gewicht einer jeden Diagonale ebenfalls um δ (da diese ja genau ein Element aus jeder Zeile bzw. aus jeder Spalte enthält), und eine Diagonale minimalen Gewichts bleibt minimal.

Durch Subtraktion geeigneter Konstanten von den Zeilen bzw. Spalten der Matrix W können wir daher eine äquivalente Gewichtsmatrix W' erhalten, die in jeder Zeile und Spalte mindestens eine 0 enthält, während alle Werte noch immer ≥ 0 sind.

Beispiel 152

$$\begin{pmatrix} 9 & 11 & 12 & 11 \\ 6 & 3 & 8 & 5 \\ 7 & 6 & 13 & 11 \\ 9 & 10 & 10 & 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 2 & 3 & 2 \\ 3 & 0 & 5 & 2 \\ 1 & 0 & 7 & 5 \\ 2 & 3 & 3 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 2 & 3 & 2 \\ 3 & 0 & 5 & 2 \\ 1 & 0 & 7 & 5 \\ 2 & 3 & 3 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} \mathbf{0} & 2 & 0 & 2 \\ 3 & 0 & 2 & 2 \\ 1 & \mathbf{0} & 4 & 5 \\ 2 & 3 & \mathbf{0} & 0 \end{pmatrix}$$

Enthält die Matrix W' eine 0-Diagonale der Größe n (also eine Diagonale, deren Elemente alle $= 0$ sind), so geben die Positionen dieser Diagonale auch die Kanten eines perfekten Matchings minimalen Gewichts im Graphen G mit der ursprünglichen Gewichtsfunktion w an, und wir sind fertig.

Andernfalls ist die maximale Länge einer 0-Diagonale in W' kleiner als n. Wir nennen eine Menge von Zeilen und Spalten einer Matrix W' eine 0-Überdeckung von W', falls diese Zeilen und Spalten alle Einträge $= 0$ der Matrix beinhalten.

Im vorhergehenden Beispiel

$$\begin{pmatrix} 0 & 2 & 0 & 2 \\ 3 & 0 & 2 & 2 \\ 1 & 0 & 4 & 5 \\ 2 & 3 & 0 & 0 \end{pmatrix}$$

bilden z.B. die Zeilen 1 und 4 zusammen mit der Spalte 2 eine 0-Überdeckung der Größe 3.

Aus Satz 137 wissen wir, dass die maximale Größe einer 0-Diagonale gleich der minimalen Größe einer 0-Überdeckung ist.

Falls W' also eine 0-Überdeckung der Größe $< n$ hat, ändern wir die Gewichtsmatrix W' zu einer Gewichtsmatrix W'' so, dass die Einträge ≥ 0 und minimale perfekte Matchings solche bleiben.

Es existiere also eine 0-Überdeckung von W', die z Zeilen und s Spalten enthalte, mit $z + s < n$. Sei w_{min} das Minimum der nicht überdeckten Einträge von W'. Also ist $w_{min} > 0$.
Um aus W' die Matrix W'' zu erhalten, verfahren wir wie folgt:

1. subtrahiere w_{min} von jedem Element der $n - z$ nicht überdeckten Zeilen (dadurch können vorübergehend negative Einträge entstehen);
2. addiere w_{min} zu allen Elementen der s überdeckten Spalten.

Damit ergibt sich für die Einträge w''_{ij} von W''

$$w''_{ij} = \begin{cases} w'_{ij} - w_{min} & \text{falls } w'_{ij} \text{ nicht überdeckt ist} \\ w'_{ij} & \text{falls } w'_{ij} \text{ entweder von einer Zeile oder von einer Spalte überdeckt ist} \\ w'_{ij} + w_{min} & \text{falls } w'_{ij} \text{ von einer Zeile und von einer Spalte überdeckt ist} \end{cases}$$

Es sind also insbesondere alle w''_{ij} wieder ≥ 0.

Für unsere Beispielsmatrix ergibt sich

$$\begin{pmatrix} 0 & 2 & 0 & 2 \\ 3 & 0 & 2 & 2 \\ 1 & 0 & 4 & 5 \\ 2 & 3 & 0 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 2 & 0 & 2 \\ 2 & -1 & 1 & 1 \\ 0 & -1 & 3 & 4 \\ 2 & 3 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 2 & 0 & 2 \\ 2 & -1 & 1 & 1 \\ 0 & -1 & 3 & 4 \\ 2 & 3 & 0 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 3 & \mathbf{0} & 2 \\ 2 & \mathbf{0} & 1 & 1 \\ \mathbf{0} & 0 & 3 & 4 \\ 2 & 4 & 0 & \mathbf{0} \end{pmatrix} =: W''$$

Die Anzahl der in W' sowohl durch eine Zeile als auch durch eine Spalte überdeckten Einträge ist $s \cdot z$, die der überhaupt nicht überdeckten Einträge $n^2 - n(s+z) + sz$.

Damit ergibt sich für die Gesamtgewichtsveränderung

$$\begin{aligned}\sum_{ij}(w''_{ij} - w'_{ij}) &= ((sz) - (n^2 - n(s+z) + sz)) \cdot w_{min} \\ &= (n(s+z) - n^2) \cdot w_{min} \\ &< 0\end{aligned}$$

Da die Kantengewichte als ganzzahlig ≥ 0 vorausgesetzt sind, kann die Transformation $W' \Rightarrow W''$ nur endlich oft wiederholt werden, und der Algorithmus terminiert.

Man kann zeigen:

Satz 153

In einem kantengewichteten bipartiten Graphen mit ganzzahligen Kantengewichten ≥ 0 kann ein gewichtsmaximales (bzw. ein gewichtsminimales perfektes) Matching in Zeit

$$\mathcal{O}(n^3)$$

bestimmt werden.

Ausblick auf Themen in EADS II

1. Flüsse in Netzwerken
2. String und Pattern Matching
3. Textkompression
4. Planare Graphen
5. Scheduling
6. Lineare Optimierung
7. Ganzzahlige Optimierung
8. Branch & Bound
9. $\mathcal{NP}$-Vollständigkeit
10. Approximationsalgorithmen