Technische Universität München
Institut für Informatik

Effiziente Algorithmen und Datenstrukturen
WS 1998/99
(WS 97/98)

Stand: 22. Juni 1999

Dozent:

Prof. Dr. Ernst W. Mayr

Mitschrift:

Bardia Kabiri
(Martin Stumpf)
(Thomas Schöpf)
(Martin Hans)
(Guido Wimmel)

Inhaltsverzeichnis

Kapitel 0

Einleitung und Auffrischung

0.0 Einleitendes Beispiel

Berechnung von F_n, die $n-te$ Fibonachi-Zahl:
$\mathrm{F}_0 = 0$; $\mathrm{F}_1 = 1$
$\mathrm{F}_n = \mathrm{F}_{n-1} + \mathrm{F}_{n-2}$ für alle $n \geq 2$

- 1.Methode: Funktionales Programmieren

$$\begin{aligned} \mathtt{f}(n) := \quad & \underline{\mathtt{if}}\ n = 0\ \underline{\mathtt{then}}\ 0 \\ & \underline{\mathtt{elseif}}\ n = 1\ \underline{\mathtt{then}}\ 1 \\ & \underline{\mathtt{else}}\ \ \mathtt{f}(n-1) + \ \mathtt{f}(n-2) \\ & \underline{\mathtt{fi}} \end{aligned}$$

  $\mathrm{T}(n)$: Zeitbedarf für die Berechnung von $\mathrm{f}(n)$:
  $\mathrm{T}(n) = \mathrm{T}(n-1) + \mathrm{T}(n-2) \quad ; n \geq 2$
  $\mathrm{T}(0) = \mathrm{T}(1) = 1$
  $\Rightarrow \mathrm{T}(n) = \mathrm{c}_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + \mathrm{c}_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$
  $\begin{cases} n = 1 : & 1 = \mathrm{c}_1(\frac{1+\sqrt{5}}{2}) + \mathrm{c}_2(\frac{1-\sqrt{5}}{2}) \\ n = 0 : & 1 = \mathrm{c}_1 + \mathrm{c}_2 \end{cases}$
  $\mathrm{c}_1 = \frac{1}{2} + \frac{\sqrt{5}}{10}$
  $\mathrm{c}_2 = \frac{1}{2} - \frac{\sqrt{5}}{10}$
  $\Rightarrow \mathrm{T}(n) = (\frac{1}{2} + \frac{\sqrt{5}}{10})(\frac{1+\sqrt{5}}{2})^n + (\frac{1}{2} - \frac{\sqrt{5}}{10})(\frac{1-\sqrt{5}}{2})^n$

- 2.Methode: Dynamische Programmierung
  array: $\mathrm{F}[0:n]$; $\mathrm{F}[0] = 0$; $\mathrm{F}[1] = 1$

$$\begin{aligned} & \underline{\text{for}}\ i = 2\ \underline{\text{to}}\ n\ \underline{\text{do}} \\ & \ \mathrm{F}[i] = \ \mathrm{F}[i-1] + \ \mathrm{F}[i-2] \end{aligned}$$

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

- 3.Methode: Direkt (mit etwas mathematischer Arbeit)

$$\mathrm{F}(n) := \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

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

0.1 Was sind „kombinatorische“ Algorithmen?

Probleme, die, könnten wir alle Fälle aufzählen, trivial wären, aber eine sehr große Anzahl von Fällen haben.
Beispiel: Hamiltonscher Kreis (NP-vollständig), Eulerscher Kreis (P)

0.2 Maschinenmodelle

- Turingmaschine (TM)
- Random Access Machine (RAM)
- Schaltkreise
- (Quantencomputer)
- (DNA-Computer)

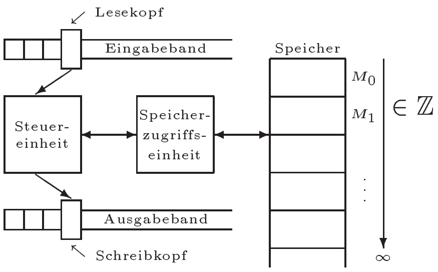


Abb. 1: **Random Access Machine**

0.3 Komplexitätsmaße

- Problem: Sprache L, Alphabet Σ, $\forall x \in \Sigma^*$:
  - Hält die Maschine nach endl. Zeit und gibt eine Antwort?

    $$\left.\begin{array}{l} x \in L : \text{Antwort "ja"} \\ x \notin L : \text{Antwort "nein"} \end{array}\right\} \text{Maschine } \underline{\text{entscheidet}}\ L$$

    $f(x)$, Maschine <u>berechnet</u> f (partielle Funktion)

  - Entweder
    - hält die Maschine nach endl. Zeit und sagt “ja“ oder
    - die Maschine hält nicht oder hält und sagt “nein“. Falls $\forall x \in L$, o.B.d.A. das nicht vorkommt, $\Rightarrow$ Maschine <u>akzeptiert</u> L

- Wie werden die Kosten berechnet? Betrachte Eingaben x der Länge n
  - worst-case Komplexität:

    $$\mathrm{C}(n) := \max\{\text{Komplexität}(x);\ |x| = n\}$$

  - durchschnittliche Komplexität (average complexity):

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

    allgemeiner: Wahrscheinlichkeitsmaß μ

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

  - amortisierte Komplexität:
    durchschnittliche Kosten der Operationen in Folgen
    worst-case über alle Folgen 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 $|x| = n$.

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

- Kosteneinheiten:
  - uniformes Kostenmodell
  - logarithmisches Kostenmodell: Kosten $\sim$ Länge der Operanden

  Beispiel:

  $r := 2$
  <u>for</u> $i = 1$ <u>to</u> n <u>do</u> $r = r^2$ <u>od</u>
  $(\Rightarrow r^{2^n})$

  Zeit:

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

  Platz:

  - uniform: 1
  - logarithmisch: 2^n

0.3.1 Probleme und Funktionen

- Probleme sind Teilmengen von Σ^*, wobei Σ endliches Alphabet ist.
- Berechnung von Funktionen:

  Definition: Seien $\Sigma,\ \Gamma$ Alphabete. Eine TM (bzw. RAM) M berechnet eine (partielle) Funktion $\mathrm{f} : \Sigma^* \to \Gamma^*$ gdw. $(\forall x \in \Sigma^*)$ [falls M mit Eingabe x hält, dann ist die Ausgabe von M gleich $\mathrm{f}(x), x \in$ Definitionsbereich von f].

0.3.2 Wachstumsverhalten von Funktionen

Definition: f, g seien Funktionen von $\mathbb{N}$ nach $\mathbb{R}$

- $\mathrm{g}(n)$ ist $\mathcal{O}(\mathrm{f}(n))$ [auch: g ist $\mathcal{O}(\mathrm{f})$, $\mathrm{g}(n) = \mathcal{O}(\mathrm{f}(n))$, $\mathrm{g}(n) \in \mathcal{O}(\mathrm{f}(n))$] gdw. $(\exists C \in \mathbb{R}, \exists n_0 \in \mathbb{N})[(\forall n \geq n_0)[|\mathrm{g}(n)| \leq C \cdot |\mathrm{f}(n)|]]$
  Beispiel: n^3 ist <u>nicht</u> $\mathcal{O}\left(\frac{n^3}{\log n}\right)$
- $\mathrm{g}(n)$ ist $\Omega(\mathrm{f}(n))$ gdw. $(\exists C > 0, \exists n_0 \in \mathbb{N})[(\forall n \geq n_0)[|\mathrm{g}(n)| \geq C \cdot |\mathrm{f}(n)|]]$

- $\mathrm{g}(n)$ ist $\Theta(\mathrm{f}(n))$ gdw. g ist $\mathcal{O}(\mathrm{f})$ und g ist $\Omega(\mathrm{f})$
- $\mathrm{g}(n)$ ist $o(\mathrm{f}(n))$ gdw. $(\forall C > 0\ \exists n_0)[(\forall n \geq n_0)[|\mathrm{g}(n)| \leq C \cdot |\mathrm{f}(n)|]]$
- $\mathrm{g}(n)$ ist $\omega(\mathrm{f}(n))$ gdw. $(\forall C > 0\ \exists n_0)[(\forall n \geq n_0)[|\mathrm{g}(n) \geq C \cdot |\mathrm{f}(n)|]]$
- $\mathrm{g}(n)$ ist $\Omega_\infty(\mathrm{f}(n))$ gdw. $(\exists C > 0)[|\mathrm{g}(n)| \geq C \cdot |\mathrm{f}(n)|$ für unendlich viele $n]$
  Beispiel: $n^2 \sin \frac{\pi}{2} n = \Omega_\infty(n)$
- $\mathrm{g}(n)$ ist $\omega_\infty(\mathrm{f}(n))$ gdw. $(\forall C > 0)[|\mathrm{g}(n)| \geq C \cdot |\mathrm{f}(n)|]$ für unendlich viele $n]$

0.3.3 Rekursionsgleichungen

Mergesort:

$$
\begin{aligned}
\mathrm{T}(n) &= 2\mathrm{T}\left(\frac{n}{2}\right) + cn \\
&= cn + 2\mathrm{T}\left(\frac{n}{2}\right) \\
&= cn + 2\left(c\frac{n}{2} + 2\mathrm{T}\left(\frac{n}{4}\right)\right) \\
&= cn + cn + 4\mathrm{T}\left(\frac{n}{4}\right) \\
&\approx cn \log_2 n \text{ (nur genau für Zweierpotenzen, sonst} \pm 1)
\end{aligned}
$$

Allgemein: $\mathrm{C}(n) = \mathrm{F}_1(\mathrm{C}(n^{(1)}), \mathrm{C}(n^{(2)}), \ldots, \mathrm{C}(n^{(r)})) + \mathrm{F}_2(n)$; $n^{(i)} < n$ für alle i

0.3.3.1 Multiplikatoren

$\mathrm{f}_1 = 1;\ \mathrm{f}_n = 2\mathrm{f}_{n-1} + n,\ n \geq 2;$

$$
\left.
\begin{array}{rll}
\mathrm{f}_n &= 2\mathrm{f}_{n-1} + n & | \cdot 1 \\
\mathrm{f}_{n-1} &= 2\mathrm{f}_{n-2} + n - 1 & | \cdot 2 \\
\vdots & & \vdots \\
\mathrm{f}_2 &= 2\mathrm{f}_1 + 2 & | \cdot 2^{n-2} \\
\mathrm{f}_1 &= 1 & | \cdot 2^{n-1}
\end{array}
\right\} \mathrm{f}_n = 2^{n-1}\mathrm{f}_1 + \sum_{i=0}^{n-2} 2^i(n-i) = 2^{n+1} - n - 2
$$

Addieren aller Gleichungen:

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

(2) (Geometrische Reihe): $n \sum\limits_{i=0}^{n-2} 2^i = n(2^{n-1} - 1)$

(3) (Substitution: $n-2=k$):

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

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

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

$(2)-(3)$:

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

$(1)+(2)-(3)$:

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

0.3.3.2 Charakteristisches Polynom

Sei

$$(*)\ \mathrm{f}_n = \mathrm{f}_{n-1} + \mathrm{f}_{n-2} \text{ für } n \geq 2,\ \mathrm{f}_1 = 1,\ \mathrm{f}_0 = 0$$

(lineare homogene Rekursionsgleichung zweiter Ordnung)
Setze $\mathrm{f}_n = a^n$ für ein unbekanntes a.
$a^n - a^{n-1} - a^{n-2} = 0$; da hier $a \neq 0$: $a^2 - a - 1 = 0$; $a_{1/2} = \frac{1\pm\sqrt{5}}{2}$
a_1 und a_2 Lösungen von $(*)$, dann auch $\mathrm{f}_n = c_1 a_1^n + c_2 a_2^n$ eine Lösung für $(*)$.
$\mathrm{f}_1 = 1$ und $\mathrm{f}_0 = 0$ liefern zwei Gleichungen für c_1 und c_2.
Lösung: $\mathrm{f}_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$

Satz: Sei $\mathrm{p}(x)$ das charakteristische Polynom der Rekursionsgleichung

$$(**)\ p_0 \mathrm{f}_n + p_1 \mathrm{f}_{n-1} + \ldots + p_k \mathrm{f}_{n-k} = 0$$

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

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

mit Konstanten c_{ij}.

0.3.3.3 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 definierte Funktion sucht.

Definition: Für eine Folge $(f_n)_{n \geq 0}$ ist die "erzeugende Funktion" definiert als:

$$\mathrm{F}(z) = \sum_{n=0}^{\infty} \mathrm{f}_n z^n; \quad z \in \mathbb{C}$$

Exponentielle Erzeugendenfunktion:

$$\mathrm{F}(z) = \sum_{n \geq 0} \frac{\mathrm{f}_n}{n!} z^n$$

auch Notation:

$$\mathrm{F}(z) = \sum_{n>0} \mathrm{f}_n z^n \Rightarrow \mathrm{f}_n = [z^n]\mathrm{F}(z)$$

Sei $\mathrm{F}(z) = \sum_{n>0} \mathrm{f}_n z^n$ und $\mathrm{G}(z) = \sum_{n \geq 0} \mathrm{g}_n z^n$

Erz. Fkt.	n-tes Folgenglied	Anmerkungen:
$c\mathrm{F}$	$c\mathrm{f}_n$	
$\mathrm{F} + \mathrm{G}$	$\mathrm{f}_n + \mathrm{g}_n$	
$\mathrm{F} \cdot \mathrm{G}$	$\mathrm{h}_n := \sum_{i=0}^{n} \mathrm{f}_i \mathrm{g}_{n-i}$ (Konvolution)	$\left(\sum_{n \geq 0} \mathrm{f}_n z^n\right)\left(\sum_{n \geq 0} \mathrm{g}_n z^n\right) = \sum_{n \geq 0} \mathrm{h}_n z^n$ (mit $\mathrm{h}_n = \sum_{i=0}^{n} \mathrm{f}_i \mathrm{g}_{n-i}$)
$z^k\mathrm{F}$	<u>if</u> $n < k$ <u>then</u> 0<u>else</u> f_{n-k} <u>fi</u>	
$\frac{\mathrm{F}(z)}{1-z}$	$\sum_{i=0}^{n} \mathrm{f}_i$	$\frac{1}{1-z} = \sum_{n \geq 0} z^n$
$z\frac{d\mathrm{F}(z)}{dz}$	$n\mathrm{f}_n$	
$\int_0^x \mathrm{F}(t)dt$	<u>if</u> $n = 0$ <u>then</u> 0 <u>else</u> $\frac{\mathrm{f}_{n-1}}{n}$ <u>fi</u>	$\mathrm{f}_n z^n$ („geht über auf") $\mathrm{f}_n \frac{z^{n+1}}{n+1}$
$\mathrm{F}(cz)$	$c^n \mathrm{f}_n$	$\mathrm{F}(cz) = \sum_{n \geq 0} \mathrm{f}_n (cz)^n = \sum_{n \geq 0} \mathrm{f}_n c^n z^n$

Beispiel:

$$\begin{aligned}
\mathrm{F}(z) &:= \sum_{n\geq 0} 2^n z^n = \frac{1}{1-2z}\\
\mathrm{G}(z) &:= \sum_{n\geq 0} n z^n = \frac{z}{(1-z)^2}\\
\Rightarrow \mathrm{F}(z)\mathrm{G}(z) &= \frac{z}{(1-z)^2(1-2z)} = \sum_{n\geq 0}\sum_{i=0}^{n}(n-i)2^i z^n
\end{aligned}$$

Partialbruchzerlegung:

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

Also:

$$\begin{aligned}
\sum_{i=0}^{n} 2^i (n-i) &= [z^n](\mathrm{FG})(z)\\
&= [z^n]\left(\sum_{n\geq 0}(2^{n+1}-n-2)z^n\right)\\
&= 2^{n+1}-n-2;
\end{aligned}$$

0.3.3.4 Transformation des Definitions- bzw. Wertebereichs

Beispiel 1:

$\mathrm{f}_n = \mathrm{f}_{n-1}\cdot \mathrm{f}_{n-2}$ für $n \geq 2$;
$\mathrm{f}_1 = 2;\ \mathrm{f}_0 = 1$

Setze: $\mathrm{g}_n = \log \mathrm{f}_n$

$\Rightarrow \mathrm{g}_n = \mathrm{g}_{n-1} + \mathrm{g}_{n-2}$ für $n \geq 2$

$\mathrm{g}_1 = \log 2 = 1,\ \mathrm{g}_0 = 0$ für $\log = \log_2$

$\Rightarrow \mathrm{g}_n = \mathrm{F}_n$ (n-te Fibonacci-Zahl)

$\Rightarrow \mathrm{f}_n = 2^{\mathrm{F}_n}$

Beispiel 2:

$\mathrm{f}_n = 3\mathrm{f}_{\frac{n}{2}} + n$ für $n = 2^k$;

$\mathrm{f}_1 = 1$

$$\begin{aligned}
\text{Setze: } \mathrm{g}_k &= \mathrm{f}_{2^k} \\
\text{Dann: } \mathrm{g}_k &= 3\mathrm{g}_{k-1} + 2^k,\ k \geq 1 \\
\mathrm{g}_0 &= 1 \\
\text{Es ergibt sich: } \mathrm{g}_k &= 3^{k+1} - 2^{k+1}, \\
\text{also: } \mathrm{f}_n &= 3 \cdot 3^k - 2 \cdot 2^k \\
&= 3(2^{\log 3})^k - 2 \cdot 2^k \\
&= 3(2^k)^{\log 3} - 2 \cdot 2^k \\
&= 3n^{\log 3} - 2n
\end{aligned}$$

Kapitel 1

Höhere Datenstrukturen

1.1 Grundlegende Operationen

Es sei U das Universum von Schlüsseln mit der Ordnung $\leq$. Menge $S \subset U$. Gegeben seien eine Menge von Datensätzen $x_1, \cdots, x_n$, wobei jeder Datensatz x durch einen Schlüssel Key$(x) = K \subset S$ gekennzeichnet ist. $x = (K,\mathrm{Val}(K))$.

IsElement(K, S):	ist $K \in S$, wenn ja, return Val(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):	$S := S \setminus \min S$
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 $(K,\mathrm{Val}(K))$
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):	$S_2 = \{K' \in S_2 \mid K' \geq K\}$ $S_1 = \{K' \in S_1 \mid K' < K\}$
Concatenate(S_1, S_2):	$S_1 := S_1 \cup S_2$; Voraussetzung: *FindMax*$(S_1) \leq$ *FindMax*(S_2)

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

1.2 Suchbäume, balancierte Bäume

1.2.1 Höhenbalancierte Bäume

Definition: Bei einem externen Suchbaum werden die Datenelemente nur an den Blättern gespeichert, und die inneren Knoten enthalten Verwaltungsinformationen. Bei internen Suchbäumen liegen die Datenelemente an den internen Knoten. Zeiger auf Blätter sind NIL-Pointer. Daher kann man sagen: An allen Knoten sind Daten gespeichert.

1.2.1.1 (a,b)-Bäume

Definition: Ein (a,b)-Baum ist ein externer Suchbaum, alle Blätter haben die gleiche Tiefe, die Anzahl der Kinder eines jeden internen Knoten ist $\leq$ b und $\geq$ a (Für die Wurzel ≥ 2), und es gilt $b \geq 2a-1$. Spezialfall: $b = 2a - 1$ (B-Bäume).

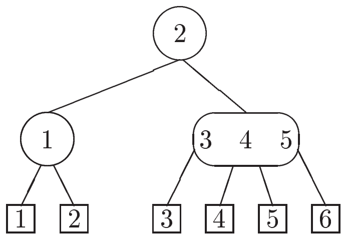
Mit jedem internen Knoten V mit d =Grad(V) werden $(d+1)$ Schlüsselwerte $K_0, \ldots, K_d$ gespeichert.
$K_0 = -\infty$, $K_d = +\infty$, $K_{i-1} <$ (alle Schlüssel im i Unterbaum von V) $\leq K_d$
etwa (*): k_i = max(Schlüssel im i-ten UB von V)

Sortierung im (a,b)-Baum:

$$\text{(Schlüssel im } i\text{-ten UB)} \leq \text{(Schlüssel im } (i+1)\text{-ten UB)}$$

Beispiel:

Wir betrachten im folgenden (2,4)-Bäume.



Lemma: Sei T ein (a,b)-Baum mit n Blättern und der Höhe h. Dann gilt:

(1) $2a^{h-1} \leq n \leq b^h$

(2) $\log_b n = \dfrac{\log n}{\log b} \leq h \leq 1 + \log_a \dfrac{n}{2}$

Beweis:

(1) $n \leq b^h$: Maximal für vollständigen Baum-Verzweigungsgrad b. Minimal für Baum mit Grad der Wurzel = 2, alle anderen internen Knoten = a.

(2) aus (1).

◇

Operationen:

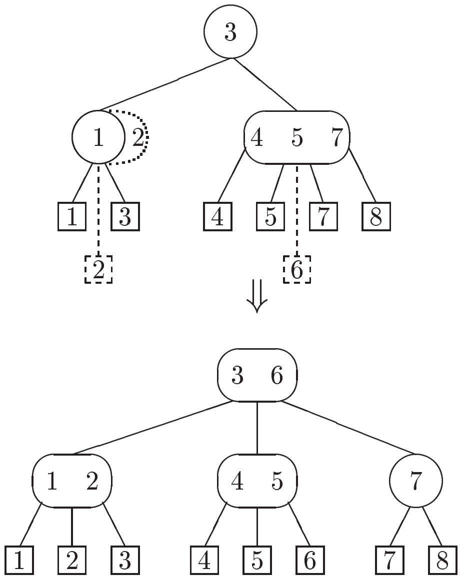
1. *IsElement*(K, S)

```
v := Wurzel von T;
while v ≠ Blatt do
    i = min{s|1 ≤ s ≤ Grad v and K ≤ K_S};
    v := i-tes Kind von v;
od;
if K =Key(v) then return Val(k)
else return False
fi;
```

$\boxed{\text{Zeitbedarf: } \theta(h) = \theta(\log n)}$

2. *Insert*(K, S)

Führe *IsElement*(K,T) aus. $\rightsquigarrow$ Blatt w;
Sei Key$(w) \neq K$, w enthält unter (*) den kleinsten Schlüssel $> K$;
Füge K links von w ein;
v =Vater von w;
Falls Grad $v = b + 1$: Rebalancierung (v);



Rebalancierung(v):

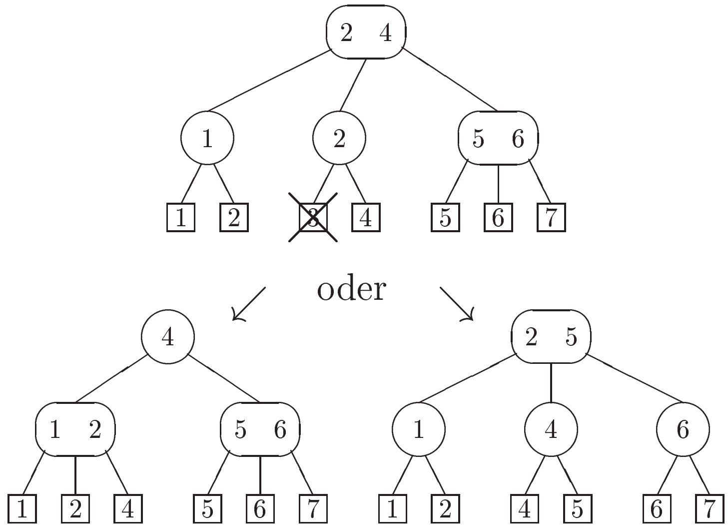
a) Falls ein unmittelbarer Bruder v' von v Grad $< b$ hat, v' adoptiert äußerstes links/rechts Kind von v.

b) Sonst: Spalte Menge C aller Kinder von v in C_1, C_2 ($\sim$gleichgroß). Spalte Knoten v in v_1, v_2,v_1 übernimmt C_1, v_2 übernimmt C_2;
$a \leq \lfloor \frac{b+1}{2} \rfloor \leq \lceil \frac{b+1}{2} \rceil \leq b$ wegen $b \geq 2a - 1$

$\boxed{\text{Kosten } \mathcal{O}(\log N)}$

3. *Delete(x, S)*:

   Führe *IsElement(K,T)* aus. $\rightsquigarrow$ Blatt w. Sei Key$(w) = K$;
   Lösche w; $v :=$ Vater w;
   Falls Grad $v = a - 1$: Rebalancierung$'(v)$;



   Rebalancierung$'(v)$:

   a) Falls ein unmittelbarer Bruder v' von v Grad $> a$ hat: v adoptiert Kind von v'
   b) Sonst verschmelze v mit einem unmittelbaren Bruder

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

Spezialfälle von (a, b)-Bäume:

- $(2, 3)$-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).
3. 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

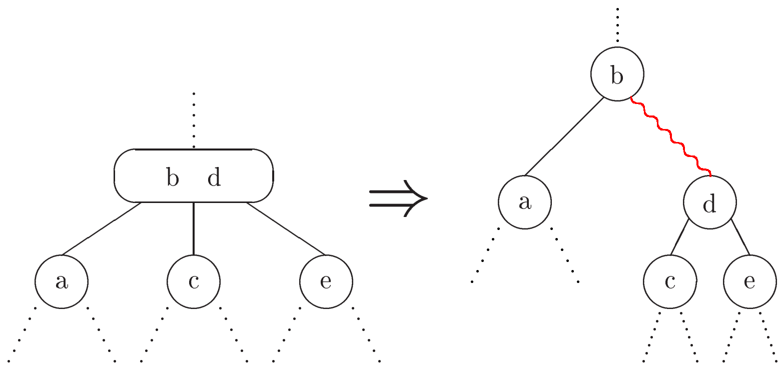
spalten wir ihn sofort auf. Weil der Vater kein b-Knoten ist (das haben wir ja bereits sichergestellt) pflanzt sich der Aufspaltungsprozeß nicht nach oben hin fort. Insbesondere kann das neue Element ohne Probleme eingefügt werden, wenn die Suche das Blattniveau erreicht hat. Damit diese Strategie möglich ist, muß b mindestens $2a$ sein (und nicht $2a-1$), 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. Sind beide Nachbarknoten vom Grad a, 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 größer 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 entgegengesetzten Richtungen 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.

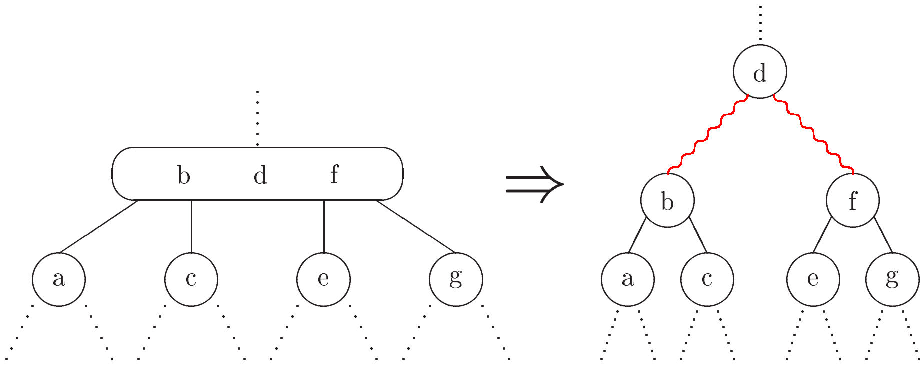
1.2.1.2 Rot-Schwarz-Bäume

Definition: 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 <u>schwarzen</u> Kanten (durchgezogene Linien)
2. alle Blätter haben die gleiche „Schwarztiefe“
3. kein Pfad von der Wurzel zu einem Blatt enthält aufeinanderfolgende <u>rote</u> 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 (2,4)-Bäumen dienen.
Vorteil: Interne Knoten haben Verzweigungsgrad = 2.

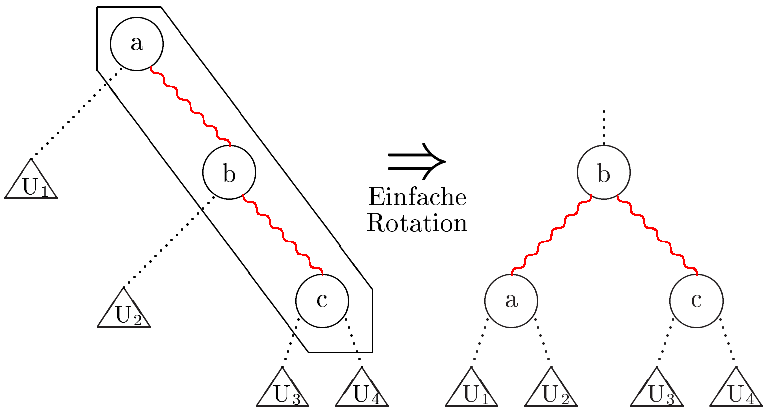


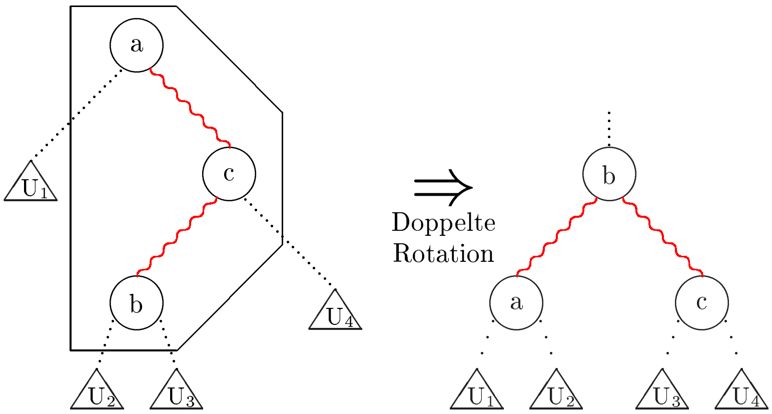


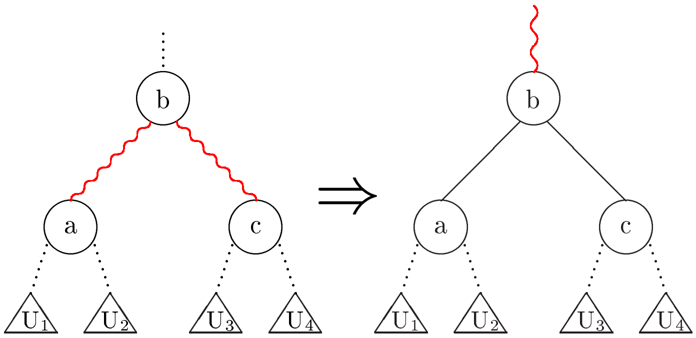
Operationen auf Rot-Schwarz implementierten (a, b)-Bäume:

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 noch nicht das Element v gespeichert. Ersetze w durch interne Knoten w' mit Kindern v, w, wobei v ein neues Blatt mit Schlüssel K sei. w' behält rote Eingangskante. Falls nun zwei aufeinanderfolgende rote Kanten an w' vorliegen, Rotation zur Rebalancierung:

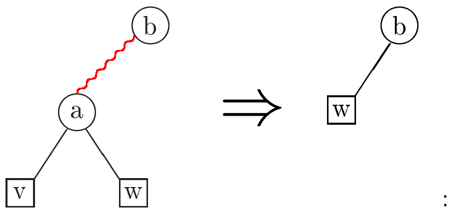






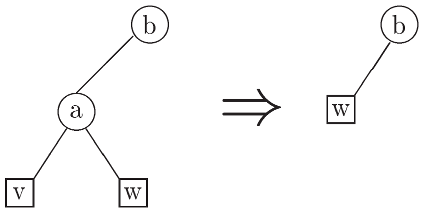
3. *Delete(K,T)*: Das Löschen des Blattes v macht es erforderlich, daß der Vater des Blattes durch den Bruder des Blattes ersetzt wird, damit der Binärbaumstruktur erhalten bleibt. Der Bruder von v ist natürlich eindeutig bestimmt. Dabei treten zwei Fälle auf:

   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. Man kann sie nicht wieder erhöhen, ohne dabei auch die Schwarztiefe anderer Blätter zu vergrößern. Daher muß sie für alle anderen Blätter des Baumes um 1 verkleinert werden. Man betrachte dazu den Pfad von w zur Wurzel als unveränderlich und färbt Kanten so um, daß auf allen anderen Pfaden jeweils genau eine Kante von schwarz zu rot wird. Ferner darf dabei nicht die Situation entstehen, daß zwei aufeinanderfolgende Kanten rot werden. Ist dies nicht vermeidbar, so muß gemäß obigem Schema rebalanciert werden.

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

Beweis: Siehe Übungsblatt. ◇

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

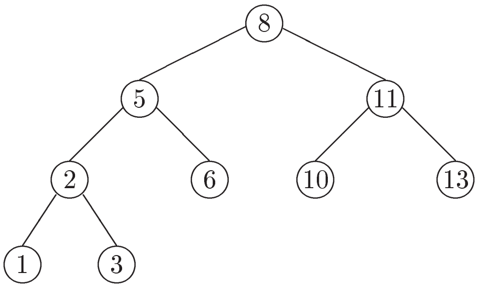
1.3 Binäre Suchbäume

1.3.1 Natürliche binäre Suchbäume

Definition: Ein natürlicher binärer Suchbaum über einem durch $\leq$ geordneten Universum U ist ein interner Binärbaum (Schlüssel an den internen Knoten).

Sortierungsbedingung:

(Alle Schlüssel im l. UB von v) $\leq$ Key(v) $\leq$ (Alle Schlüssel im r. UB von v)



Lemma: Die In-Order-Linialisierung eines binären Suchbaumes mit obigen Suchwegbedingungen ergibt eine aufsteigende Folge von Schlüsselwertern.

Operationen:

- *IsElement*(K,T):

```
v ist Wurzel;
while (v ≠ Blatt) do
    if Key(v) = K then return (Val(K));
    elseif Key(v) ≤ K then v :=rechtes Kind von v;
    else v :=linkes Kind von v;
    fi
od
```

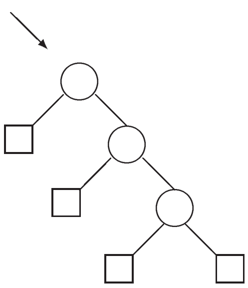
- *Insert*(K,T):

  Führe *IsElement*(K,T) aus;
  Sei K nicht enthalten, dann führt *IsElement* zu Blatt w (NIL-Pointer);
  Füge man Knoten v mit Key(v) = K an Blatt w ein;

- *Delete*(K,T):

  Führe *IsElement*(K,T) aus.
  Sei K enthalten, dann führt *IsElement* zu Knoten w (i.a. interne Knoten).
  Ersetze w durch seinen in-order-Vorgänger oder -Nachfolger.

Problem: Lineare Entartung von natürlichen binären Suchbäumen bei Einfügen in eine monotone Reihenfolge $\Rightarrow$ *Insert*, *Delete*, *IsElement* in worst-case $\mathcal{O}(n)$. Falls aber die Sclüssel in zufälliger Reihenfolge eingefügt werden $\Rightarrow$ | E(h) - $\mathcal{O}(\log n)$|. Falls alle topologische (von der Form her gleichaussehende) Bäume gleichwahrscheinlich sind $\Rightarrow$ | E(h) - $\theta(\sqrt{n})$|.



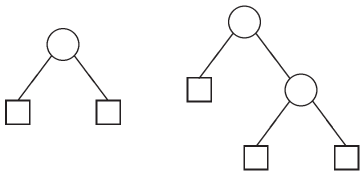
1.3.2 Höhenbalancierte binäre Suchbäume (AVL-Bäume)

Definition: Ein AVL-Baum [1] ist ein spezieller binärer Suchbaum, der neben den Sortierungsbedingungen auch die folgende Höhenbalancierungen erfüllt. Für den Knoten v gilt:
$|$ Höhe(lUB^v) - Höhe(rUB^v) $| \leq 1$

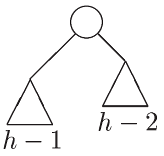
Lemma: Ein AVL-Baum der Höhe h hat mindestens $n_h = \text{Fib}(h+2)$ Blätter.

Beweis: Induktion nach h:

I) $h = 1; \Rightarrow n_h = 2 = \text{Fib}(3)$;
$h = 2 \Rightarrow n_h = 3 = \text{Fib}(4)$;



II) $h \geq 3; \Rightarrow n_h \geq n_{h-1} + n_{h-2}$
Bei einem bzgl. der Blätterzahl minimalen AVL-Baum der Höhe h darf nur einer der Unterbäume der Höhe $h-1$ haben. Der andere muß kürzer sein, aber wegen der Balancierungsbedingung mindestens Höhe $h-2$ haben. Beide sind natürlich minimale AVL-Bäume. Also gilt $n_h = n_{h-1} + n_{h-2}$. Dies ist genau die Bestimmungsgleichung für $\text{Fib}(h+1)$.



$\diamond$

Folgerung:

- AVL-Bäume der Höhe h haben mindestens $\text{Fib}(h+2)$ Blätter, also $\text{Fib}(h+2)-1$ interne Knoten.
- Da anderseits gilt: $\text{Fib}(n) \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$, hat ein AVL-Baum mit n-Blättern (oder interne Knoten) die Höhe $\mathcal{O}(\log n)$. Dies ist ein bedeutender Fortschritt gegenüber der bisweilen bei natürlichen Suchbäumen entstehenden Zeitkomplexität von $\Omega(n)$ für die Suchoperation *IsElement*.

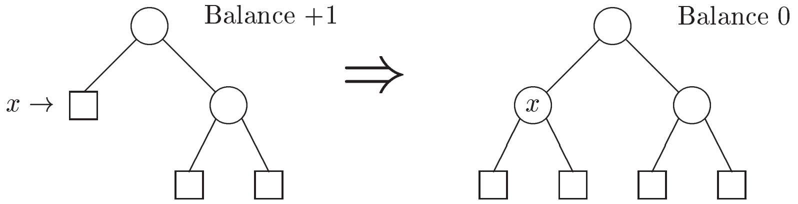
Operationen auf AVL-Bäumen:

1. *IsElement*: Diese Operation wird wie bei den natürlichen Suchbäumen implementiert. Wie oben ausgeführt, haben aber AVL-Bäume mit n Datenelementen 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

[1] Benannt nach Adelson, Velskii und Landis

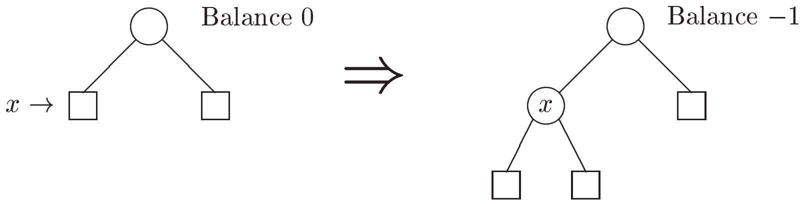
2 Fälle:

1. Fall:



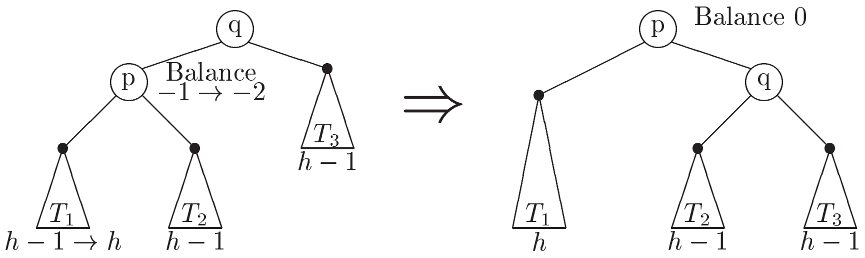
(Balance $-1 \rightarrow 0$: symmetrisch)

2. Fall:

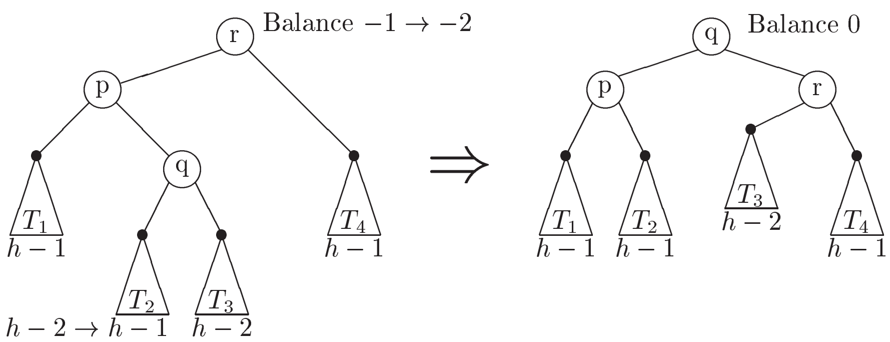


Hier ist eventuell eine Rebalancierung auf dem Pfad zur Wurzel notwendig, denn die Höhe eines Unterbaums ändert sich derart, daß die Balance nicht zu 0 wird, sondern zu +1 oder −1. Zur Rebalancierung ($T_1 \leq T_2 \leq T_3 \leq T_4$):

Fall 2-a)



Fall 2-b)



3. *Delete*: Auch diese Operation wird wie bei natürlichen Suchbäumen implementiert. Doch am Ende hat ggf. noch eine Rebalancierung auf dem Pfad

vom gelöschten Blatt (das an die Stelle des zu löschenden Elements geschrieben wird) zur Wurzel zu erfolgen.

Satz: 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 Datenelemente (oder die Anzahl der Blätter) ist.

Siehe auch:

- Aho, Hopcroft, Ullmann
- Sedgewick/Flajolet
- Info IV Skript SS94, Prof. Mayr

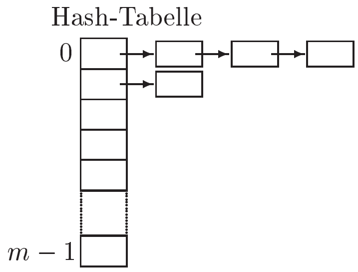
In der Theorie: Sehr schöne Eigenschaften.
In der Praxis: Sehr aufwendig zu implementieren.

1.4 Hashing

1.4.1 Grundlagen

Zutaten:

- Universum U von Schlüsseln
- Schlüsselmenge $K \subset M$
- array $T[0..m-1]$ Hashtabelle
- Hashfunktion $h : U \to [0..m-1]$



Problemstellung: Gegeben sei eine Menge M von Elementen, von denen jedes durch einen Schlüssel k aus der Menge K bezeichnet sei. Die Problemstellung ist nicht neu: Wie muß die Speicherung der Elemente aus M organisiert werden, damit jedes anhand eines Schlüssels möglich schnell lokalisiert werden kann? Gesucht ist also eine Abbildung $h : K \to T$ von der Menge aller Schlüssel in Adressraum T der Maschine. Hierbei soll jedoch eine bisher nicht beachtete Schwierigkeit berücksichtigt werden: Die Menge K der möglischen Schlüsselwerte ist wesentlisch größer als der Adreßraum. Folglich kann die Abbildung h nicht injektiv sein, es gibt Schlüssel $k_1, k_2, ..$ 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 (Kollision).

Zwei Methoden zur Kollisionsauflösung:

- Chaining (Verkettung)
- Offene Adressierung

1.4.2 Chainingverfahren

Die Hashtabelle ist eine array von m linearen Listen, wobei die i-te Liste alle k Schlüssel beinhaltet, für die gilt:

$$h(k) = i$$

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

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

$$\delta_h(k_1, S) := \sum_{k_2 \in S} \delta_h(k_1, k_2), \text{ Anzahl Kollisionen von } k_1 \text{ mit } S$$

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

worst-case: $h|_s = \text{const} \Rightarrow$ Entartung des Hashings zu linearer Suche $\mathcal{O}(|S|)$

average-case :

- Möge h das Universum U über $[0..m-1]$ gleichverteilen
- Seien alle Werte in U gleichwahrscheinlich als Argument von *Access()*, *Insert()*, *Delete()*
- Sei $\beta = \frac{n}{m}$ der maximale Füllgrad von T, $n =$ Anzahl Operationen

Dann: Die erwartete Laufzeit für n-Operationen $= \boxed{\mathcal{O}\left((1 + \frac{\beta}{2})n\right)}$

Beweis: Durch die Gleichverteilung für den Wert $h(k_i)$ bei der i-ten Operation

$$P(h(k) = j) = \frac{1}{m} \quad (0 \leq j \leq m-1)$$

$\Rightarrow \mathbb{E}$ (Listenlänge) $\leq \frac{r}{m}$ nach der r-ten Operation
$\Rightarrow \mathbb{E}$ (Kosten von $(n+1)$-ten Operationen) $= \mathcal{O}(1 + \frac{i}{m})$
$\Rightarrow \mathbb{E}$ (Kosten von n Operationen) $= \mathcal{O}((1 + \frac{1}{2}\frac{n}{m})n)$ ◇

1.4.3 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), in der alle slots in $T[\,]$ betrachtet werden, um k zu speichern bzw. zu finden, festgelegt. Dieser Vorgang heißt sondieren.

Sondieren:

Sei $s(j,k) : [0..m-1] \times U \to [0..m-1]$;
Definiere $h(j,k) = (h(k) - s(j,k)) \underline{\text{mod}}\ m$ $\quad ;(0 \leq j \leq m-1)$;
Starte mit $h(k)$, dann (falls $(h(k))$ belegt) Index $(h(k) - s(1,k)) \underline{\text{mod}}\ m$;

Problem: Sei $h(k) = h(k')$ für 2 Schlüssel $k, k' \in S \subset U$. 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! Wenn Speicher gebraucht wird, k überschreiben. Alternativ: Hashtabelle neu aufbauen/überarbeiten.

Beispiele für Sondierungen:

- Lineares Sondieren: Setze $s(j,k) = j$ d.h. sondiere gemäß $h(k), h(k)-1, \ldots, 0, m-1, .., h(k)+1$
  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: Die durchschnittliche Anzahl der Sondierungen ist :

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

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

- 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 $[0..m-1]$? Ist $s(j,k)$ geeignet alle slots zu erreichen?
  Man kann zeigen, daß für ein primes m von der Form $4i+3$ die Sondierungsgröße $(h(k) - s(j,k)) \underline{\text{mod}}\ m$ Permutation von $[0..m-1]$ ist.

Satz: Die durchschnittliche Anzahl der Sondierungen ist :

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

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

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

Satz: Die durchschnittliche Anzahl der Sondierungen ist :

$$\mathbb{E}(\#\text{ Sondierungsschritte}) = \begin{cases} \frac{1}{\beta} m \frac{1}{1-\beta} & \text{erfolgreich} \\ \frac{1}{1-\beta} & \text{erfolglos} \end{cases}$$

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

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

1.4.4 Universelles Hashing

Wir definieren eine Klasse H von geeigneten Hashfunktionen und wählen eine spezielle Hashfunktin zufällig aus H aus. Dabei soll für jedes $S \subset U$ "fast jedes" $h \in H$ die Schlüsselmenge S "fast" gleichverteilen über $[0..m-1]$. Formal:

Definition: H heißt <u>c-universell</u> ($c \in \mathbb{R}$) g.d.w. $\forall k_1, k_2 \in U$ gilt:

$$\frac{|\{\ h \in H |\ \delta_h(k_1,k_2) = 1\}|}{|H|} \leq \frac{c}{m}$$

Satz: Sei H eine c-universelle Klasse von Hash-Funktionen, und sei $h \in H$ zufällig und gleichverteilt gewählt. Wird Chaining verwendet, dann ist die erwartete Zeit für *Access()*, *Insert()* und *Delete()* gleich $\mathcal{O}(1 + c\beta)$

Beispiel einer 4-universellen Hashtabelle:
Seien $N \in \mathbb{N}$ prim, $m \in \mathbb{N}$, und $\forall a, b \in [0..N-1]$:

$$h_{a,b}(k) = ((ak + b) \underline{\bmod} N) \ \underline{\bmod} m \ \forall k \in U \subset [0..N-1]$$

Dann ist $H = \{h_{a,b} \mid 0 \leq a, b \leq N-1\}$ 4-universell.

1.4.5 Perfektes Hashing

Für das statische dictionary problem (nur *Access()*) können wir definieren:

- Eine Hash-Funktion h ist "perfekt" für $S \subset U$ gdw. für alle Paare von verschiedenen Schlüssel, also $\forall k_1, k_2 \in S$ gilt: $\delta_h(k_1,k_2) = 0$, also $h(k_1) \neq h(k_2)$.
- Eine <u>minimale</u> perfekte Hash-Funktion braucht nur den Platz $m = |S| = n$.
- Für $m \geq 3n$ kann man in Zeit $\mathcal{O}(nN)$ eine perfekte Hash-Funktion für $S \subset U$ finden, die in $\mathcal{O}(1)$ Zeit auszuwerten ist (durch ein Programm der Größe $\mathcal{O}(n \log N)$) Bits.

  Bisher bestes Resultat:
  Programmgröße $\mathcal{O}(n \log\log n + \log\log N)$
  (untere Schranke: $\Omega\left(\frac{n^2}{m} + \log\log N\right)$)

1.5 Priority Queues

Operationen:

Insert(), *Delete()*, *DeleteMin()*, *FindMin()*, *DecreaseKey()*, *Merge()*

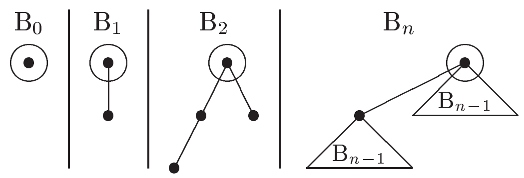
Implementierungen:

- Binomial Queues
- Fibonacci Heaps

1.5.1 Binomial Queues (binomial heaps)

Um binomial Queues betrachten zu können, definieren wir zunächst ihre nichttrivialen Bestandteile: Binomialbäume

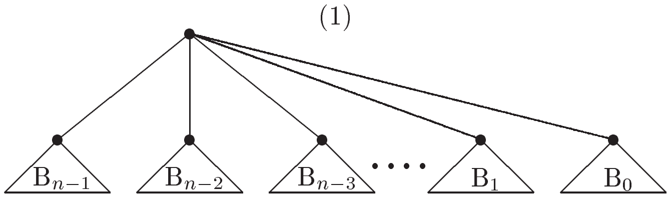
Definition: Die Binomialbäume sind rekursive wie folgt definiert:



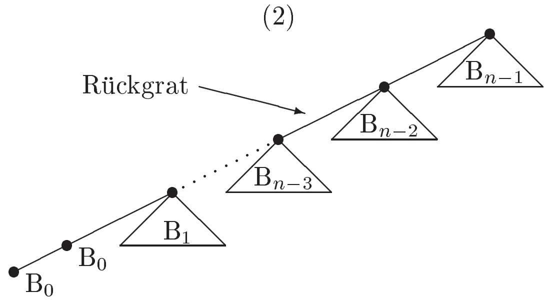
Achtung: Binomialbäume sind keine Binärbäume! Wie in folgender Darstellung dieser Bäume deutlich wird, hat B_n an der Wurzel Grad n.
Binomialbäume haben folgende Zerlegungseigenschaften:

Lemma: Ein B_n läßt sich wie folgt zerlegen:

(1)



(2)



Satz: Es gilt:

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

Beweis:

- zu 1: Induktion unter Verwendung der Definition von Binomialbäumen
- zu 2: Siehe Zerlegung (1) für Binomialbäume
- zu 3: Siehe Zerlegung (2) für Binomialbäume
- 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 in B_n, so gilt unter der entsprechenden Induktionsannahme für B_n, daß $N_i^{n+1} = N_i^n + N_{i-1}^n = \binom{n}{i} + \binom{n}{i-1}$

$\diamond$

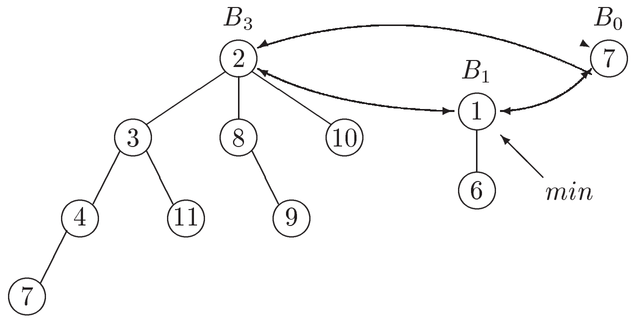
Bemerkung: Eine mögliche Implementierung von Binomialbäumen ist das Schema "Pointer zu Kind 1. und zum nächsten Geschwister".

Definition: Eine Binomial Queue mit N Elementen wird wie folgt aufgebaut:

1. Betrachte die Binärdarstellung bin(N) von N
2. Für jedes gesetzte Bit ("1") an Position i wird ein Binomialbaum vom Type B_i benötigt. Dieser hat bekanntlich 2^i Knoten
3. Verbinde die Binomialbäume in einer doppelt verketteten zirkulären Liste.
4. Beachte, daß innerhalb jedes Binomialbaums die Minimumsvariante der Heap-Bedingung, nämlich VATER$\leq$SOHN erfüllt sein muß. Dadurch ist die Wurzel eines einzelnen Binomialbaums gleichzeitig sein minimales Element.
5. Richte einen Min-Pointer auf das Element in der Wurzel-Liste mit minimalem Schlüssel. Es ist dann das kleinste Element von allen in der Binomial Queue

Beachte, daß Priority-Queues wegen $\mathcal{O}(n)$ für Suchen keine Suchstrukturen sind.

Beispiel: $N = 11 = (1011)_2$:



Operationen:

1. *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 sein muß. Tatsächlich kann ein Element in jedem der vorhandenen Binomialbäume stehen. Das Suchen ist hier nicht effizient implementierbar, denn im worst-case muß jedes Element eines Binomialbaums angeschaut werden. B_i enthält 2^i Knoten und eine Binomial Queue mit N Elementen besitzt maximal die Binomialbäume $B_0, B_1, ..B_{\log N-1}$. Also wären zum Suchen eines Elements schlimmstenfalls $2^0 + 2^1 + \cdots + 2^{\log N-1} = N - 1 = \mathcal{O}(N)$ Elemente zu betrachten. Daher wird eine gesonderte Datenstruktur, etwa ein Suchbaum, für die *IsElement*-Operation verwendet. Dort ist Suchen mit Zeitaufwand $\mathcal{O}(\log n)$ möglich.

2. *Merge*: Das Vorgehen für das Merge zweier Vorrangwarteschlangen entspricht genau der Addition zweier Binärzahlen: Ein einzelnes B_i wird übernommen, aus zwei B_is wird ein B_{i+1} konstruiert. Damit die Heap-Bedingung erhalten bleibt, wird als Wurzel des entstehenden Binomialbaums B_{i+1} die kleinere der Wurzeln der beiden B_i genommen (*). Allerdings kann ein solcher Übertrag dazu führen, daß im nächsten Verschmelzungsschritt drei B_{i+1} zu verschmelzen sind. Dann wird unter Beobachtung obiger Regel ein B_{i+2} gebildet, und der verbleibende B_{i+1} unverändert gelassen.

   Algorithmus:

   ```
   for i := 0,1,2,3,.. do
       if (∃ genau 3 B_is) then
           erzeuge/verbinde B_{i+1} nach (*) und behalte ein B_i;
       elseif (∃ genau 2 B_is) then
           erzeuge B_{i+1} nach (*) ;
       elsif (∃ ein B_i) then
           behalte es;
       fi
   od
   ```

   Komplexität: Binomial Queue Nr. 1 mit N_1 Elementen hat $\leq \log N_1$ Binomialbäume, und analog hat Binomial Queue Nr. 2 mit N_2 Elementen $\leq \log N_2$ Binomialbäume. Sei $N := N_1 + N_2$. Ein einzelner *Merge*-Vorgang, das Zusammenhängen zweier Binomialbäume erfolgt in konstanter Zeit. Natürlich muß er so oft ausgeführt werden, wie Binomialbäume in der resultierenden Binomial Queue vorliegen. Diese Anzahl wiederum ist höchstens $\log N$.

$\boxed{\text{Zeitkomplexität } \mathcal{O}(\log N) = \mathcal{O}(\log{(N_1 + N_2)})}$

3. *Insert*: Die *Insert*-Operation wird einfach durch *Merge*-Operation implementiert. Die Vorrangswarteschlange, die neu hinzukommt, ist trivial. Sie enthält nur einen Binomialbaum B_0.

$$\begin{array}{cccccccc} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ (B_7) & & & (B_4) & & (B_2) & & (B_0) \\ & & & & & & & 1 \\ \hline 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}$$

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

4. Initialisierung einer Binomial-Queue durch N sukzessive *Insert*-Operationen: Kosten mit $\mathcal{O}(n \log n)$ schlecht abgeschätzt. Dabei möge sowohl das Einfügen dieses neuen einelementigen Baum als auch jede rekursive Verschmelzungsschritt eine Zeiteinheit kosten, denn jede dieser Operationen ist von der Komplexität $\mathcal{O}(1)$. Wir sind an den gesamten Kosten zum Aufbau einer Binomial Queue mit N Elementen interessiert. Dies sind übrigens identisch zum Aufwand für das binäre Hochzählen von 0 bis N, wenn jeder Zählschritt und jeder Übertrag eine Zeiteinheit kosten:

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{0+1}_{1}+1}_{2\text{ Schritte}}+1}_{1}+1}_{3}+1}_{1}$$

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	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 die h-te Zeile von der Länge 2^k ist, und alle Elemente der Zeilen $1, 2, .., (k-1)$ enthält, wobei das letzte Element um eins erhöht ist. Daraus ergeben sich folgende Rekursionsgleichungen für a_n:

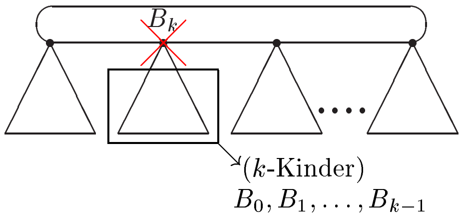
$$a_1 = 1$$
$$a_n = a_{\frac{n}{2}} + 1 \text{ für } n \in 2^{\mathbb{N}_0}$$
$$a_n = a_{n-2^{\lfloor \log n \rfloor}} \text{ sonst}$$

Diese Bestimmungsgleichung ist unmittelbar einsichtig, denn für $n \in 2^{\mathbb{N}_0}$ unterscheidet sich die Situation von $n = 2^i$ Knoten nur insofern von der Situation mit $\frac{n}{2} = 2^{i-1}$ Knoten, daß im ersten Fall noch ein Binomialbaum B_{i-1} vorliegt. Das Hinzunehmen eines weiteren Knotens löst in beiden Fällen die selbe Folge von Verschmelzungsschritten aus, mit der Einschränkung, daß im letzteren Fall der neu entstandenen Baum B_{i-1} mit dem vorhandenen zu einem B_i vereinigt wird. Das ist die "+1" in $a_n = a_{\frac{n}{2}} + 1$. Im Falle $n \notin 2^{\mathbb{N}_0}$ liegen ähnliche Verhältnisse vor. Zwei Queues mit n bzw. $(n - 2^{\lfloor \log n \rfloor})$ Knoten unterscheiden sich nur um einen Baum $B_{\lfloor \log n \rfloor}$. Gesucht ist nun $S_n := \Sigma_{k=1}^{n} a_k$. Statt obige Rekursionsgleichungen zu expandieren und eine explizite Darstellung für a_n zu suchen, die dann aufsummiert wird, betrachten wir einfach S_{2^k}. Obiger Tabelle sieht man an, daß $S_{2^k} = S_{2^{k-1}} + 1$ gilt. Substituiert man S_{2^k} durch T_k und löst man die entstehene Rekursionsgleichung $T_k = 2T_{k-1} + 1$, so ergibt sich $T_k = 2^{k+1} - 1$, also $S_n = 2n - 1$. Folglich ist:

$$\boxed{S_N = \mathcal{O}(N)}$$

5. *FindMin*: Diese Operation ist trivial ausführbar, denn es ist ein Pointer auf das minimale Element gegeben. Daher ist auch die Zeitkomplexität $\mathcal{O}(1)$.

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



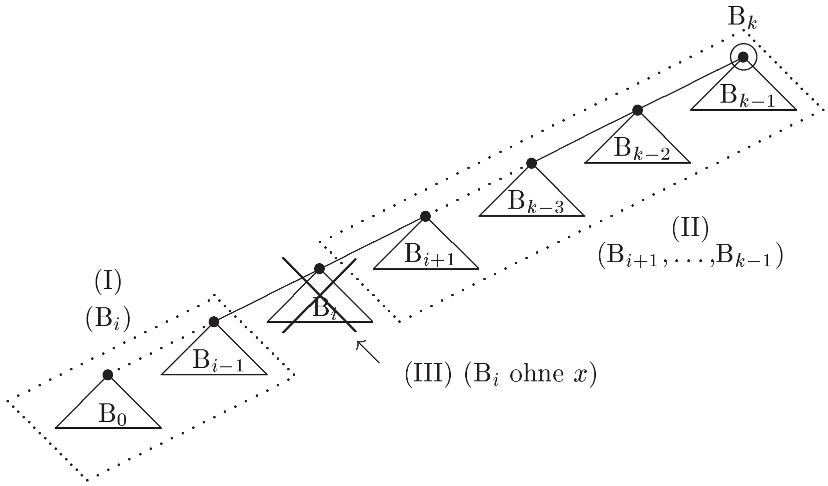
Die Teilbäume $B_0, B_1, \ldots, B_{k-1}$ sind alle zur verbleibenden Queue zu mergen. Außerdem muß der Min-Pointer aktualisiert werden. Der Zeitaufwand für *DeleteMin*-Operation ist: $\boxed{\mathcal{O}(\log N)}$

7. *Delete*: Lösche Knoten x:

1-Fall) x ist min-Wurzel: s.o.

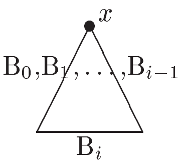
2-Fall) x ist eine andere Wurzel in der Wurzelliste. Analog zu oben, ohne den Min-Pointer zu aktualisieren.

3-Fall) x sei nicht Wurzel eines Baumes:
Angenommen, x ist in Baum B_i enthalten:



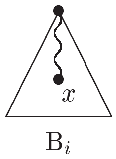
Im Fall-3 muß unterschieden werden:

3a) x ist Wurzel:



Siehe (2-Fall)

3b) x ist in Wurzel des B_i:



So lange 3-te Fall wiederholen (Rekursiv), bis Fall-3a) eintritt

Insgesamt muß die Binomial-Queue ohne B_i mit einer aus $B_0, \cdots, B_{N-1}$ bestehenden Binomial-Queue vereinigt werden.
$\boxed{\text{Zeit } \mathcal{O}(\log N)}$

8. *DecreaseKey*: Verkleinere Key(x)

1-Fall) x ist die min-Wurzel: keine Strukturänderung nötig

2-Fall) x ist eine andere Wurzel: keine Strukturänderung nötig

3-Fall) Sonst wie *Delete(x)*, aber einfügen des Baumes mit Wurzel x und den neuen Key(x) in die Wurzelliste.

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

Satz: Binomial-Queues haben für die Operationen *Insert*, *Delete*, *DeleteMin*, *Find-Min*, *Merge* und *Initialize* worst-case-Kosten von jeweils $\mathcal{O}(\log n)$.

1.5.2 Fibonacci-Heaps

Vorbemerkungen:

- Fibonacci-Heaps stellen eine Erweiterung der Binomial-Queues dar und eine weitere Möglichkeit zur Implementierung von Heaps. Die amortisierten Kosten für die Operationen *Delete()* und *DeleteMin()* betragen hierbei $\mathcal{O}(\log n)$, die für alle anderen Heap-Operationen lediglich $\mathcal{O}(1)$. Natürlich können die worst-case-Kosten für ($n\times$ *Insert*) + ($n\times$ *DeleteMin*) nicht unter $\Omega(n \log n)$ liegen, denn diese Operationen zusammengenommen stellen einen Sortieralgorithmus dar mit unterer Schranke $\Omega(n \log n)$.
- Die Verwendung von Fibonacci-Heaps erlaubt eine Verbesserung der Komplexität der Algorithmen für die 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 *DeleteMin*-Operationen nur einen geringen Anteil der Gesamtoperationen darstellen, sind Fibonacci-Heaps asymptotisch schneller als die Binomial-Queues.

Bemerkungen:

- Priority-Queues eignen sich grundsätzlich nicht für Suchoperationen, unabhängig von ihrer Implementierung. Es ist daher immer eine zusätzliche Datenstruktur, etwa ein Suchbaum, zu verwenden.
- Jedes Element besteht aus einem Schlüssel (key) und einem Wert. Der Schlüssel darf in einer Priority-Queue <u>nur einmal</u> vorkommen. Es ist jedoch zulässig, daß derselbe Datenwert mehreren verschiedenen Schlüsseln zugeordnet ist. Für die folgenden Betrachtungen sind gespeicherte Werte aber uninteressant, es werden nur die Schlüssel betrachtet.

1.5.2.1 Die Datenstruktur

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

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

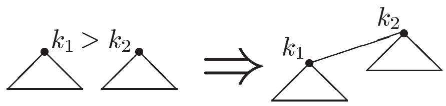
Binomial-Queue vs. Fibonacci-Heap:

- Fibonacci-Heaps: Wald von Bäumen, innerhalb derer die Heap-Bedingungen erfüllt sind.
- A-Priori: Keine Einschränkung für die Topologie der Bäume, aber ohne *Delete-* oder DekreaseKey-Operation bleibt ein Binomialwald ein Binomialwald.
- keine Invariante der Form "Nur Bäume verschiedenen Wurzel-Rangs".

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	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
DeleteMin	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

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 eins.



Dies ist gerade der konstruktive Schritt beim Aufbau von Binomialbäumen. Zur Verbesserung der amortisierten Zeitschranken sind zusätzlich folgende Regel Formuliert:

Kaskadiertes 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.
- Wenn ein Knoten ab seinem Anfangszustand zum zweitenmal einen Sohn verliert, wird er in die Wurzelliste eingefügt.
- Der kritische Zustand (ein Kind verloren) wird durch das Markierungsbit angezeigt.

Algorithmus:

```
x verliert Kind;
y = P(x) : (Vater von x)
while (y markiert) do
    entferne y;
    füge y unmarkiert in die Liste ein;
    if (P(y)=NIL) then exit fi
    y = P(y);
od
markiere y;
```

Operationen:

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

   $\boxed{\text{Kosten } \mathcal{O}(1)}$

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

   $\boxed{\text{Kosten } \mathcal{O}(1)}$

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

   $\boxed{\text{Kosten } \mathcal{O}(1)}$

4. *Delete(x)*

   i) Falls x Min-Wurzel, *DeleteMin*-Operator benutzen

   ii) Sonst:

```
while true do
    füge Liste der Kinder in die Wurzelliste ein;
    lösche x, x =Vater(x);
    if Markierung(x)=0 then Markierung(x)=1 exit
    else
        hänge x samt Unterbaum in Wurzelliste;
        entferne Markierung von x da x Wurzel;
        x = Vater(x);
    fi
od
```

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

```
x :=↑ Min-Pointer;
entferne x aus der Liste;
Konkataniere Kinderliste vom x mit der Wurzelliste;
while (∃ ≥ 2 Bäume von selben Wurzel-Rang i) do
    erzeuge Baum vom 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 gefunden werden. Dies wird wie folgt realisiert: Wir verwenden ein array, dessen Indizies 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 aktualisiern in jedem Schritt das array. 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 die Fibonacci-Heap nur verschiedene Bäume. Liegt der kanonische Fall vor, daß ausschließlich Binomialbäume im Spiel sind, so erhalten wir durch *DeleteMin*-Operation eine Binomial-Queue.

   $\boxed{\text{Zeitkomplexität für } \textit{DeleteMin}\text{: } \mathcal{O}(\max \text{ Rang} + \# \text{ Link-Schritte})}$

6. *DecreaseKey*(K, Δ):

```
entferne x;
y = P(x);
füge x unmittelbar in die Wurzelliste ein;
Key(x) := Key(x) - Δ;
while (y markiert) do
    entferne y;
    füge y unmarkiert in die Wurzelliste ein;
    if (P(y) =NILL) then exit fi
    y = P(y);
od
markiere y;
```

$\boxed{\text{Zeitkomplexität: } \mathcal{O}(\text{max Höhe})}$

Bemerkung: Wird mit einem leeren oder gar keinem Fibonacci-Heap gestartet, und werden ausschließlich die aufbauenden Operationen *Insert*, *Merge* und *FindMin* angewendet, so können nur Binomialbäume enstehen. In diesem natürlichen Fall also liegt stets ein Binomialwald vor, der jedoch i.a. nicht aufgeräumt ist. Das heißt, es existieren mehrere Binomialbäume B_i desselben Typs, die nicht paarweise zu B_{i+1}-Bäumen verschmolzen sind. Dies geschieht erst bei der Operation *DeleteMin*. Man beachte, daß nur nach einer solchen Operation momentan ein Binomial Heap vorliegt, ansonsten nur ein Binomialwald.

1.5.2.2 Amortisierte Kostenanalyse für Fibonacci-Heaps

Kostenanalyse für Operation Sequenzen:

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

ii) average-case:

  - Aufwand für Analyse zu hoch
  - realistische Verteilung für Eingabe?

iii) amortisierte Kostenanalyse: average-case-Analyse über worst-case-Sequenzen von Operationen.

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

t_i:= worst-case-Kosten der i-ten Operation
$\Delta bal_i = bal_i - bal_{i-1}$: Potentialveränderung 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):

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

Amortisierten Kosten einer Sequenz sind obere Schranke für die tatsächlicher Kosten. Anwendung auf Fibonacci-Heaps:

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

Lemma: Sei x ein Knoten im Fibonacci-Heap mit $\text{Rang}(x) = k$. 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 der $\text{Rang}(x) = i - 1$ (Das i-tes Kind hat zu dieser Zeit auch Rang $(i - 1)$). Danach kann das i-te Kind höchstens einen Sohn verloren haben $\Rightarrow$ Rang $\geq i - 2$. $\diamond$

Satz: Sei x Wurzel eines Baums im Fibonacci-Heap, $\text{Rang}(x) = k$. Dann enthält der Baum mit Wurzel x mind. F_{k+2} Elemente ($F_{k+2} = \sum_{i=2}^{k} F_i + 2$).
(D.h., die Wurzelränge sind logarithmisch beschränkt).

Beweis: Sei f_k die minimale Anzahl der Elemente im Baum, dessen Wurzel Rang k hat.
$\Rightarrow f_k \geq f_{k-2} + f_{k-3} + \ldots + f_0 + 1 + 1$
Setze: $g_k := g_{k-2} + g_{k-3} + \ldots + g_0 + 2$, d. h. $f_k \geq g_k$; $g_0 := 1$

$$\begin{array}{lllllllll}
(1) & g_k & = & g_{k-2} & + & g_{k-3} & + & \ldots & \\
(2) & g_{k-1} & = & & & g_{k-3} & + & g_{k-4} & + \ \ldots \\
\hline
(1)\text{ - }(2) & g_k - g_{k-1} & = & g_{k-2} & & & & & \\
 & \multicolumn{8}{l}{\Rightarrow g_k = g_{k-1} + g_{k-2} \Rightarrow f_k \geq g_k = F_{k+2}}
\end{array}$$

$\diamond$

Amortisierte Kosten der FH-Operationen ($a_i = t_i + \Delta bal_i$):

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

$$\left.\begin{array}{l}
\text{Einfügen der Kinder von } x \text{ in Wurzelliste:} \\
\Delta bal = \text{Rang}(x) \\
\text{Jeder Link-Schritt verkleinert \# Bäume um 1} \\
\Delta bal = -1 \\
\Rightarrow \text{Jeder Link-Schritt wird von Bankkonto} \\
\text{bezahlt und verschwindet amortisiert}
\end{array}\right\} \Rightarrow a = \mathcal{O}(\log N)$$

- *DecreaseKey*: Es ist $\Delta bal = 3 - (\#\text{ kaskadierte Schritte})$

$$\left.\begin{array}{l}
\text{Jeder Schritt erhöht die \# der Bäume um 1} \\
bal+ = 1 \\
\text{Jeder Schritt vernichtet eine Markierung} \\
bal- = 2 \\
\text{Der letzte Schritt erzeugt eine Markierung} \\
bal+ = 2
\end{array}\right\} \Rightarrow a = \mathcal{O}(1)$$

Literatur:

Fredman, Tergen: Fibonacci-Heaps and their uses in improved network optimization algorithmus, Procceedings of the 25^{th}., FOCS, 1984, S. 338-346

1.6 Sich selbst organisierende Datenstrukturen

1.6.1 Motivation

- einfach, wenig Verwaltungsoverhead
- effizient im amortisierten Sinn

1.6.2 Sich selbst organisierende lineare Listen

Beim sog. "Wörterbuchproblem" ist auf möglichst effiziente Weise eine Menge von bis zu n Elementen zu organisieren:

- Einfügen
- Löschen
- Reihenfolge ändern

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

Operationen:

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

Nach Abschluß einer jeden Operation können gewisse Umordnungsschritte stattfinden, die spätere Operationen 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 Elementenvertauschung 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 an die Liste an 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:

- Die MFR erlaubt die Ausführung einer Sequenz von Operationen *Access*, *Insert* und *Delete* mit einer amortisierten Laufzeit, die um höchstens den konstanten Faktor 2 schlechter ist als ein optimaler Listenalgorithmus.
- Für die 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. Kosten für *Access*(x_i) und *Delete*(x_i) sind gleich der Position von x_i in der Liste (von Anfang an gezählt).
2. Kosten für *Insert*(x) (x nicht in Liste) ist der Länge der Liste nach der *Insert*-Operation (d.h. neue Elemente werden zunächst an 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 Transpose) benutzen nur kostenlose Vertauschungen. 3a ist pessimistisch für MFR.

Satz: 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 $\mathrm{C_A}(s)$ der Zeitaufwand von A auf s, <u>ohne</u> bezahlte Vertauschungen. Sei $\mathrm{X_A}(s)$ die Anzahl der bezahlten Vertauschungen. Sei $\mathrm{C_{MFR}}(s)$ der Gesamtaufwand der Move_to_front-Heuristik auf s. Sei $\mathrm{F_A}(s)$ die Anzahl der kostenlosen Vertauschungen, die A auf s ausführt. Sei $m := |s|$ die Länge von s. Dann

$$\mathrm{C_{MFR}}(s) \leq 2 \cdot \mathrm{C_A}(s) + \mathrm{X_A}(s) - \mathrm{F_A}(s) - m$$

Beweis: Potentialfunktion: L_A bzw. L_MFR seien die Listen, A bzw. MFR (führt bei Eingabesequenz s) sind die Algorithmen.
Potential := Anzahl der Inversionen in L_MFR im Vergleich zu L_A
D.h. sei o.B.d.A. $L_\mathrm{A} = (x_1, x_2, \ldots, x_n)$.
Dann ist die Potential:

$$\mathit{bal}(L_\mathrm{MFR}, L_\mathrm{A}) = \sum_{j=1}^{n} |\{\ i\ ,\ j > i \text{ und } x_j \text{ ist in } L_\mathrm{MFR} \text{ vor } x_i\}|$$

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 i-Einheiten.

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

$$\begin{array}{ll} \mathrm{L_A}: & x_1\ x_2\ .\ x_i\ \ldots\ x_n \\ \mathrm{L_{MFR}}: & \ldots\ldots . x_j\ \ldots\ \underset{\substack{\uparrow \\ t}}{x_i}\ \ldots \end{array}$$

Sei k die Anzahl der Elemente (wie x_j), $j > i$, die im L_{MFR} vor x_i stehen.
t ist die Position x_i im L_{MFR}:

$t \leq i + k$; $\Rightarrow k \geq t - i$
$\Delta bal \leq -k + (i-1) \leq i - t + i - 1 \leq 2i - 1 - t$ (bei *Access*)
$\Delta bal = -k \leq i - t$ (bei *Delete*)

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

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

$$\begin{array}{lll} \mathrm{L_A}: & x_1\ x_2 \ldots x_{i-1}\ x_i \ldots x_n & \text{eine Inversion weniger} \\ \mathrm{L_{MFR}}: & \ldots x_i \ldots x_{i-1} \ldots\ldots & \end{array}$$

iv) Die amortisierten Kosten einer bezahlten Vertauschung durch Algorithmus A sind ≤ 1

$$\begin{array}{lll} \mathrm{L_A}: & x_1\ x_2 \ldots x_{i-1}\ x_i \ldots x_n & \text{plus eine Inversion} \\ \mathrm{L_{MFR}}: & \ldots x_{i-1} \ldots x_i \ldots\ldots & \end{array}$$

Anfangspotential : 0
$\Rightarrow \mathrm{C_{MFR}}(s) \leq$ amort. Kosten MFR $\leq 2 \cdot \mathrm{C_A}(s) + \mathrm{X_A}(s) - \mathrm{F_A}(s) - m$

$\diamond$

Anmerkung: Wenn wir nicht mit einer leeren Liste (oder zwei identischen Listen, d.h. $\mathrm{L_A} = \mathrm{L_{MFR}}$) starten, ergibt sich:
$\mathrm{C_{MFR}} \leq 2\mathrm{C_A}(s) + \mathrm{X_A}(s) - \mathrm{F_A}(s) - m + n^2$

Literatur:
Sleator, D.R., Tarjan: Amortized efficiency of list update and paging rules. Comm. ACM 28(2), p. 202-208 (1985).

1.6.2.1 Sich selbst organisierende Binary Keys

- Priority Queue
- Variante von Leftist-Bäumen [s.Knuth]: Ein Leftist-Baum ist ein (interner) binärer Suchbaum, so daß für jeden der Knoten gilt, daß ein kürzester Weg zu einem Blatt über das rechte Kind führt.

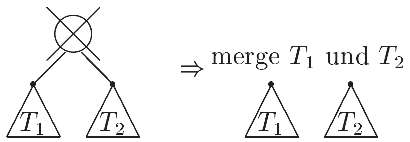
Wir wollen folgende Operationen unterstützen:

- *Insert*
- *FindMin*
- *DeleteMin*
- *Delete*
- *Merge*

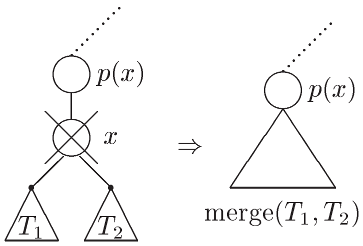
alle Operationen lassen sich auf *Merge* zurückführen

Operationen:

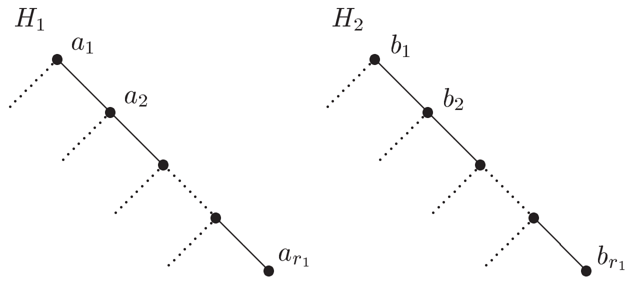
1. *Insert(T, x)*: neuer Baum mit x als Element, merge diesen mit T
2. *FindMin*: $\surd$ (Minimum an der Wurzel, wegen Heap-Bedingung)
3. *DeleteMin*:



4. *Delete*:

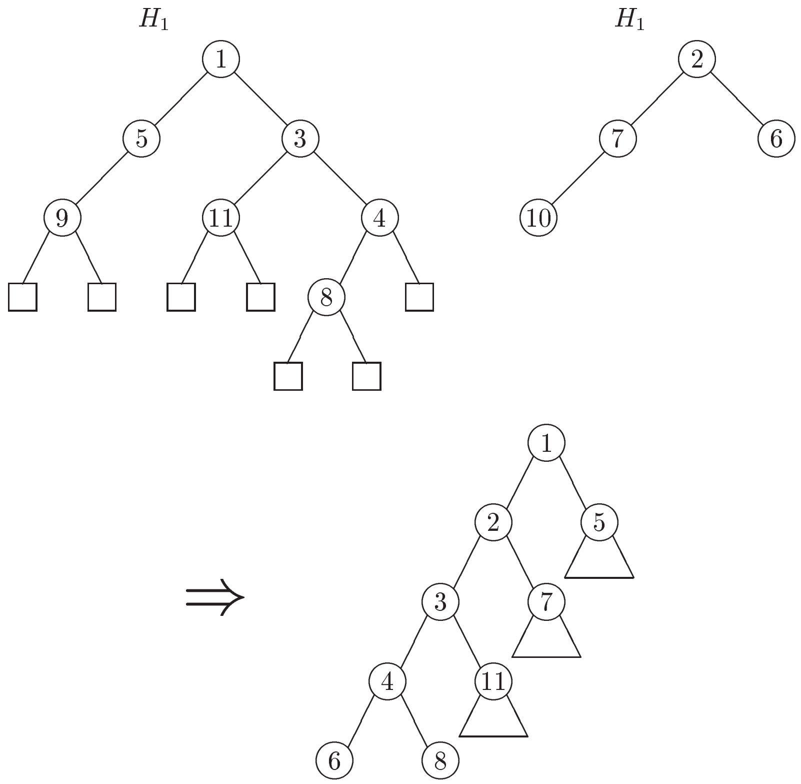


5. *Merge*: Seien zwei Heaps H_1 und H_2 gegeben, seien $a_1, .., a_{r_1}$ die Folge der Knoten in H_1 von der Wurzel zum rechtesten Blatt, sei $b_1, .., b_{r_2}$ die entsprechende Folge in H_2.



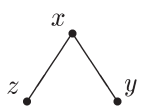
Sei $c_1, c_2, .., c_{r_1+r_2}$ die durch Merging aus $(a_1, .., a_{r_1})$ und $(b_1, .., b_{r_2})$ entstehende (sortierte) Folge. Mache $(c_1, .., c_{r_1+r_2})$ zum linkesten Pfad im neuen Baum und hänge den jeweiligen anderen (d.h. linken) UB a_i bzw. b_j als <u>rechten</u> UB des entsprechenden c_i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

Beispiel:



Amortisierte Kostenanalyse der Merge-Operation:

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



Kante(x, y) heißt leicht, falls $2.w(y) \leq w(x)$
Kante(x, y) heißt schwer, sonst

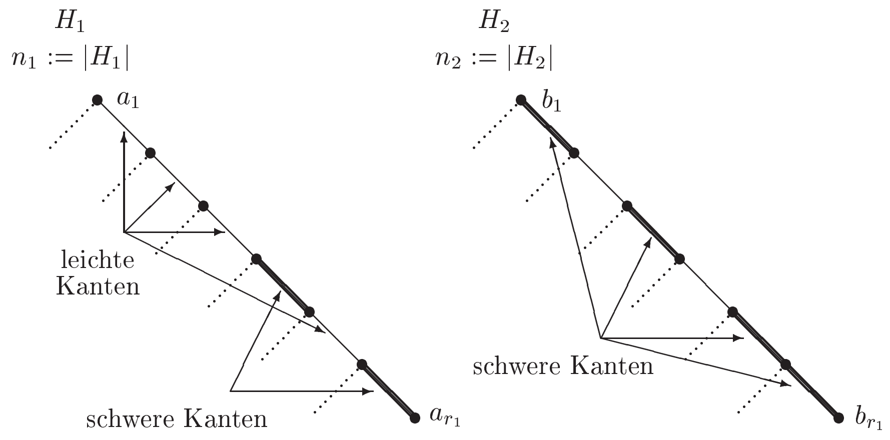
(x, y) schwer $\Rightarrow$ (x, z) leicht

Es gilt: Es ist immer ein Kind über leichte, eins über schwere Kante erreichbar.

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 $\log n$ leichte Kanten enthalten.

Potential: # schwere Kanten zu rechten Kindern



$s_1 :=$ # schwere Kanten auf den rechten Pfad. $s_2 :=$ Analog.

$r_1 \le s_1 + \log n_1$
$r_2 \le s_2 + \log n_2$

Amortisierte Kosten für *Merge:*

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

Satz: 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.

1.6.2.2 Splay-Trees als Suchbäume

1.6.2.2.1 Einleitung

Wie schon in früheren Kapiteln sollen binäre Suchbäume verwendet werden, um Elemente, bestehend aus einem Schlüssel und einer Information, anhand ihres Schlüssels zu lokalisieren. Das Universum ist total geordnet, und innerhalb eines Suchbaumes soll die Invariante "Knoten im lUb $\le$ Key(Wurzel) $\le$ Knoten im rUb" gelten.
Der hier zugrundeliegende Ansatz bemüht sich, innerhalb einer Folge von Operationen durch Realzeit-Umorganisation der Datenstruktur möglichst weit vom worst-case zu entfernen, statt die jeweiligen worst-case-Kosten zu minimieren. Bei jedem Zugriff auf den Baum soll eine möglichst einfache Reorganisationsregel angewendet werden, um die Laufzeit künftiger Operationen zu vermindern. Wie bei der MFR für selbst organisierende Listen soll erreicht werden, daß Elemente mit häufigem Zugriff möglichst nahe an der Wurzel liegen. Ein Element, auf das die Suchoperation angewendet wird, wird zur Wurzel befördert, in der i.a. berechtigten Hoffnung, daß weitere Zugriffe auf dieses Element folgen.

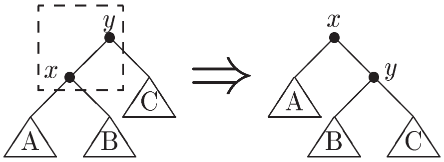
Definition: Ein Splay-Tree ist ein interner binärer Suchbaum, d.h.

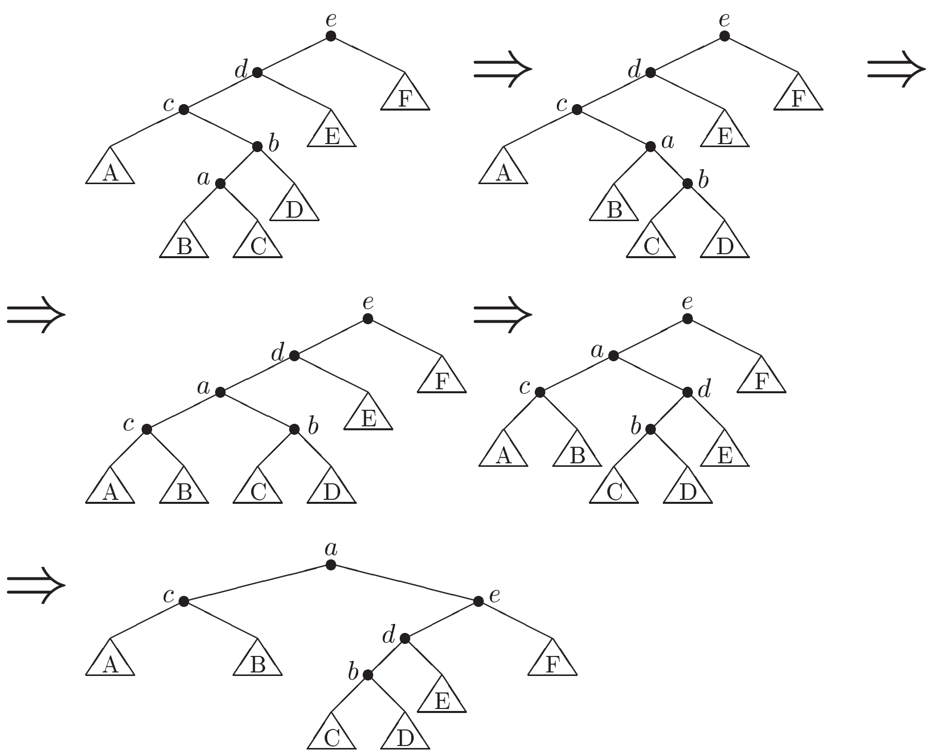
- "Knoten im lUb $\leq$ Key(Wurzel) $\leq$ Knoten im rUb"
- Alle Knoten tragen Elemente, denn Zeiger auf Blätter sind NIL-Pointer.

Rotationen, um Knoten an die Wurzel zu bewegen.
Beispiele:

i) Einfach Rotation

ii) Move to root



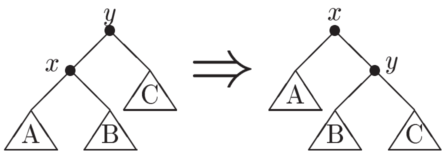


Beide Heuristiken benötigen für geeignete Folgen von Zugriffsoperationen Ω(n) amortisierte Zeit. Beim MTR (Move To Root) im probabilistischen Sinn ist dies eine gute Komplexität. Zugriffswahrscheinlichkeiten sind fest und unabhängig.

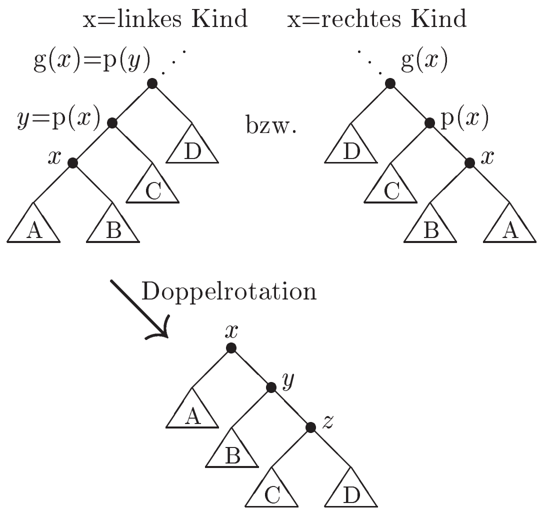
1.6.2.2.2 Die Splaying-Operation

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

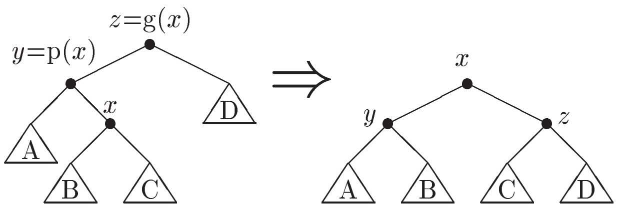
1. (zig): x ist Kind der Wurzel von T: einfache Rotation



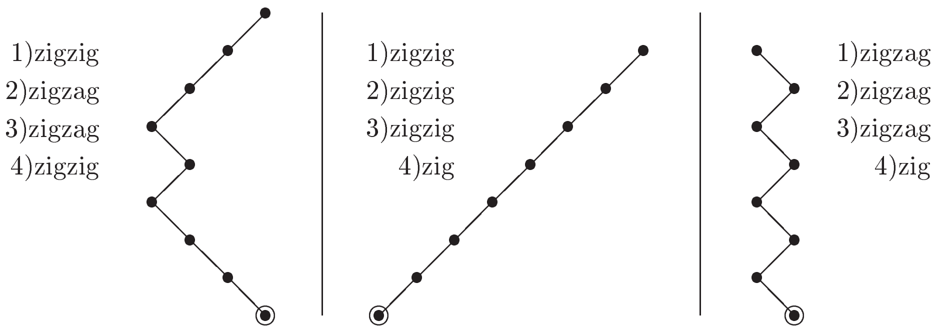
2. (zig-zig): x hat Großvater $\mathrm{g}(x)$ (und Vater $\mathrm{p}(x)$) und x und $\mathrm{p}(x)$ sind jeweils linke (bzw. rechte) Kinder ihres Vaters.



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



Übung: (Splaying-Operation mit dem eingekreisten Element durchführen)



1.6.2.2.3 Amortisierte Kostenanalyse der Splay-Operation

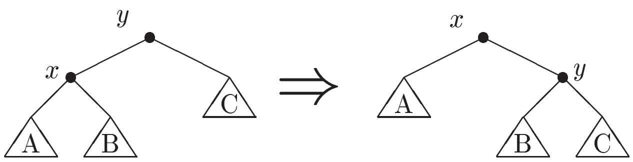
Jeder Knoten habe Gewicht $\mathrm{w}(x) > 0$. Gewicht $\mathrm{tw}(x)$ des Unterbaums mit Wurzel x ist: $\sum_y \mathrm{w}(y)$ $\qquad (\forall y$ im Unterbaum mit Wurzel $x)$

Rang $\mathrm{r}(x) = \log(\mathrm{tw}(x))$
Potential eines Baumes T: $\sum\limits_{x \in T} \mathrm{r}(x)$

Lemma: Sei T ein Splay-Tree mit Wurzel u, x ein Knoten in T. Die amortisierten Kosten für Splay(x, T) sind $\leq 1 + 3(\mathrm{r}(u) - \mathrm{r}(x)) = \mathcal{O}\left(\log \frac{\mathrm{tw}(u)}{\mathrm{tw}(x)}\right)$.

Beweis: Induktion über die Folge von (Doppel)Rotationen:
Berechne r und r′, tw und tw′, die Rang- bzw. Gewichtsfunktion vor und nach dem Rotationsschritt. Wir zeigen, daß die amortisierten Kosten im Fall 1 (zig) $\leq 1+3(\mathrm{r}'(x)-\mathrm{r}(x))$ und in den Fällen 2 und 3 (zig-zig bzw. zig-zag) $\leq 3(\mathrm{r}'(x)-\mathrm{r}(x))$ sind. y sei der Vater von x, z der Großvater (falls er existiert).

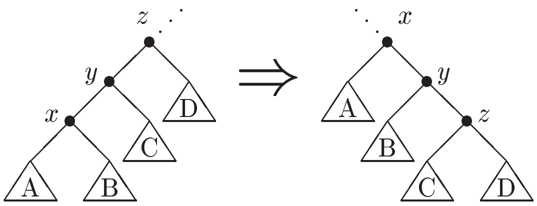
1. Fall:



amortisierte Kosten:

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

2. Fall:



amortisierte Kosten:

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

Es gilt, daß

$$2 + \mathrm{r}'(x) + \mathrm{r}'(z) - 2\mathrm{r}(x) \leq 3(\mathrm{r}'(x) - \mathrm{r}(x))$$

d.h.

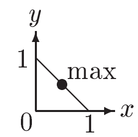
$$2\mathrm{r}'(x) - \mathrm{r}(x) - \mathrm{r}'(z) \geq 2$$

Betrachte dazu die Funktion

$$\mathrm{f}(x, y) = \log(x) + \log(y)$$

auf dem Bereich

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



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

Beweis: Da die Funktion $z \to \log(z)$ 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. 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 $x = 0.5$ (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 $g(x)$ in $x = 0.5$ sein globales Maximum 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. ◇

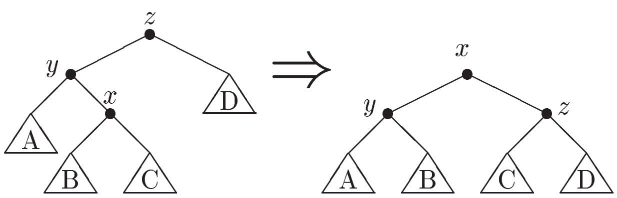
Damit gilt:

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

da

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

3. Fall:



amortisierte Kosten:

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

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

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

Sei T ein Splay-Tree mit n Knoten. Die Verringerung des Potentials durch eine Folge von Splay-Operationen $(\text{splay}(x_j, T))_{j=1..n}$, falls sich die Gewichte nicht ändern, beschränkt sich durch

$$\sum_{i=1}^{n}(\log W - \log w_i) = \sum_{i=1}^{n} \log \frac{W}{w_i}; \qquad W := \sum_{i=1}^{n} w_i \qquad w_i = \text{Gewicht von } x_i.$$

Satz: Die gesamten Kosten für die m Zugriffe sind $\mathcal{O}((m+n)\log n + m)$.

Beweis:
amortisierte Kosten = reelle + Δ_{balance}; reelle Kosten = amortisierte - Δ_{balance}. Wähle $w_i = \frac{1}{n}$ für alle Knoten. $\Rightarrow$ amortisierte Kosten für einen Zugriff $\leq 1 + 3\log n$, da $W = \sum_{i=1}^{n} w_i = 1$. Die Verringerung im Potential ist $\leq \sum_{i=1}^{n} \log \frac{W}{w_i} = \sum_{i=1}^{n} \log n = n \log n$. $\Rightarrow$ reelle Kosten $\leq m(1+3\log n) + n\log n$. ◇

Satz: 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) > 1 für alle i), dann sind die (reellen) Gesamtkosten für die Zugriffe

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

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

$$\Rightarrow \mathrm{W} = \sum_{i=1}^{n} \frac{\mathrm{q}(i)}{m} = 1 \qquad ; \text{Rest wie oben.}$$

◇

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

Beweis: Sei U die Menge, d die Tiefe eines (fest gewählten) optionalen statischen Suchbaumes, sei für $x \in U$, $d(x)$ die Tiefe von x in diesem Suchbaum. Definiere: $tw(x) := 3^{d-d(x)}$. Sei T ein beliebiges Splay-Tree für U :

$$\begin{aligned} bal(T) \leq \sum_{x\in U} r(x) = \sum_{x\in U} \log 3^{d-d(x)} &= \sum_{x\in U} (\log 3)(d - d_x) = \\ &= ((\log 3) \sum_{x\in U} (d - d(x))) = \mathcal{O}(n^2); \qquad n = |U| \end{aligned}$$

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

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

Damit ergibt sich für die amortisierte Kosten für Splay(x, T):
$\mathcal{O}(\log \frac{tw(T)}{tw(x)}) = \mathcal{O}(d(x) + 1)$.

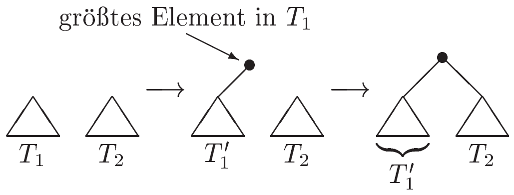
Damit sind die amortisierten Kosten $\leq c$. Zugriffskosten (# Vergleiche) im optimalen Suchbaum (wo sie d(x)+1 sind). Die gesamten amortisierten Kosten für die Zugriffsfolge sind daher $\leq c.t$.
Damit sind reelle Kosten $\leq a$.Kosten + Verringerung des Potentials = $\mathcal{O}(t + n^2)$.

$\diamond$

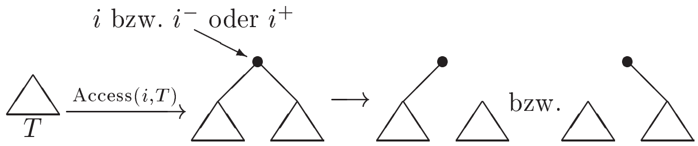
1.6.2.2.4 Wörterbuchoperationen in Splay-Trees

Betrachte (Alle diese Operationen werden mit Hilfe von Splay implementiert):

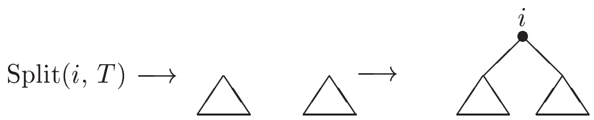
- *Access(i, T)*: $\surd$ (Haben wir bereits)
- *Join(T_1, T_2)*:

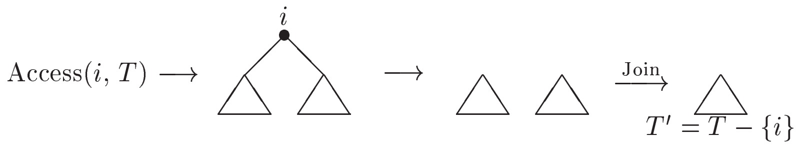
  beachte $x \in T_1,\ y \in T_2$, dann gilt $x < y$
- *Split(i, T)*:

- *Insert(i, T)*:

- *Delete(i, T)*:

  Definition: Sei $i \in U$; T ein Splay-Tree. Dann bezeichnen wir mit $i-$ (bzw. $i+$) den Vorgänger (bzw. den Nachfolger) von i in der gegebenen Ordnung in T (falls diese existieren). Falls $i-$ bzw. $i+$ undefiniert sind, so setzen wir w($i-$) = ∞ bzw. w($i+$) = ∞. Weiterhin sei W das Gesamtgewicht aller an einer Wörterbuch-Operation beteiligten Knoten.

Satz: Für die amortisierten Kosten der Wörterbuch-Operationen in Splay-Trees gelten die folgenden oberen Schranken: ($T,\ T_1,\ T_2 \neq 0$)

$$i^- := \text{ größtest Element in } T \text{ kleiner als } i$$
$$i^+ := \text{ kleinstes Element in } T \text{ größer als } i$$

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

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

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

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

$$\underset{i\in T,}{\mathsf{Delete}(i,T)}: 3\log\left(\frac{W}{\mathrm{w}(i)}\right) + 3\log\left(\frac{W-\mathrm{w}(i)}{\mathrm{w}(i^-)}\right) + \mathcal{O}(1),$$
$$\text{falls } i \text{ nicht minimal in } T$$

1.6.3 Weitere Arten von Datenstrukturen

- Dynamische Datenstrukturen:
  Referenz: Bent, S. W.: Dynamic weighted data structures, TR STAN-CS-82-916, Department of Computer Science, Stanford University, 1982
- Persistente Datenstrukturen:
  Referenz: Driscoll, J. R., Sarnak, N., Sleater, D. D., Tarjan, R. E., Making data structures persistent, Proceedings of the 18^{th} Annual ACM Symposium on Theory of Computing (STOC), 1986. p.109–121.

1.7 Radix-basierte Priority-Queue

1.7.1 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 mit den Schlüsseln key(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:

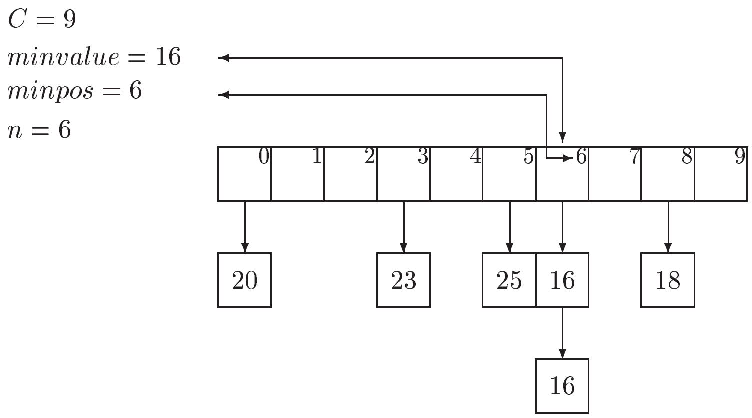
$$\boxed{\text{größter Schlüssel - kleinster Schlüssel} \leq C}$$

Diese Bedingungen sind zum Beispiel beim Algorithmus von Dijkstra erfüllt. C ist hier gleich der maximalen Kantenlänge.

1.7.1.1 1-Level-Buckets

1-Level-Buckets bestehen aus:

- einem Array $b[0 \ldots C+1]$ zur Aufnahme der Buckets. Jedes b_i enthält einen Pointer auf der Liste der Elemente im Buckets,
- 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
- einer Zahl n, die die Anzahl der gespeicherten Elementen enthält.



Wie bei jeder Vorrangswarteschlange 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, *minpos* und *minvalue* aktualisieren.
- *ExtractMin*: liefert und löscht eines der kleinsten Elemente der Vorrangswarteschlange (falls das Element das kleinste ist, muss *minpos* und *minvalue* noch aktualisiert werden).
- *DecreaseKey(x, k)*: verringert Schlüssel des Elements x auf den Wert k (falls nötig, *minpos* und *minvalue* aktualisieren).

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);

2. *Insert*: Füge x in $b[\text{Key}(x) \bmod (C+1)]$ ein

   $n := n + 1$;
   if (Key(x) $< minvalue$) then
   co x ist jetzt das Element mit dem kleinsten Schlüssel oc
       $minpos :=$ Key(x) mod (C+1);
       $minvalue :=$ Key(x);
   fi

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

   $n := n - 1$;
   if ($n > 0$) then
       co suche neues Element mit dem kleinsten Schlüssel oc
       while $b[minpos]$ ist leer do
           $minpos := (minpos + 1)$ mod $(C + 1)$ od
       $minvalue :=$ Key eines Elements in $b[minpos]$;
   else $minvalue := \infty$; fi

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

   Key(x) := k;
   if (Key(x) $< minvalue$) then
       $minpos :=$ Key(x) mod (C+1);
       $minvalue :=$ Key(x);
   fi

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

Satz: Die worst-case reellen Kosten sind $\mathcal{O}(1)$ bei *Insert*, *DecreaseKey* und $\mathcal{O}(C)$ (bei *Initialize*) und *ExtractMin* Operationen.

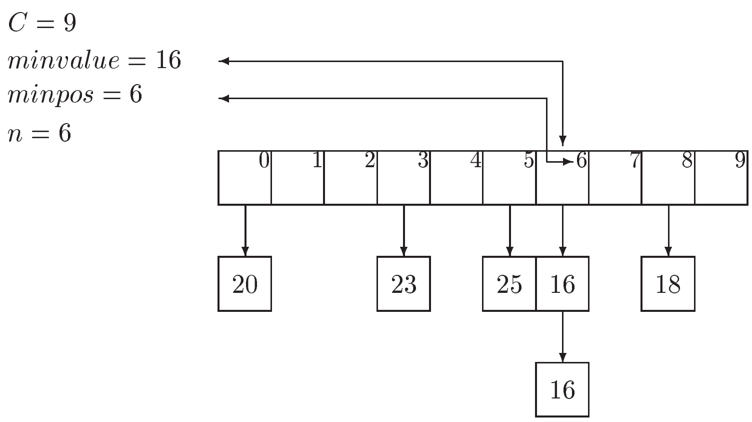
Beweis: Wird x am Anfang der Liste eingefügt, so müssen bei *Insert* nur einige Zeiger umgehängt sowie n, $minpos$ und $minvalue$ angepaßt werden, was wieder nur ein paar Zeigeroperationen sind. Die Aktualisierung von n, $minpos$ und $minvalue$ benötigt auch nur konstante Zeit. Für das Finden des nächstkleinsten 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 auch nur wenige Zeigeroperationen und je eine Zuweisung an n und Key(x) nötig. ◇

1.7.1.2 2-Level-Buckets

Bei einem großen Wertebereich der zu speichernden Schlüssel, das heißt beim großen C, und einer geringen Anzahl tatsächlich abgelegter Datenelemente sind 1-Level-Buckets in zweifacher Hinsicht ungünstig:

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

2-Level-Buckets versuchen dem mit folgender Idee abzuhelfen: Es gibt ein Top-Bucket (*btop*) bestehend aus $|\sqrt{C+1}|$ Buckets und zu jedem Bucket in *btop* gibt es noch ein weiteres Bottom-Bucket (*bbots*), das auch aus B=$|\sqrt{C+1}|$ Buckets besteht. Das i-te *bbot* nimmt Elemente auf, deren Schlüssel im Interval $[i\mathrm{B}, (i+1)\mathrm{B}]$ liegen. Um ein Element in ein Bucket einzufügen, wird zuerest das passende Bucket in *btop* gesucht. Dann wird in dem dazugehörigen *bbot* das Element (wie bei 1-Level-Buckets) eingefügt.



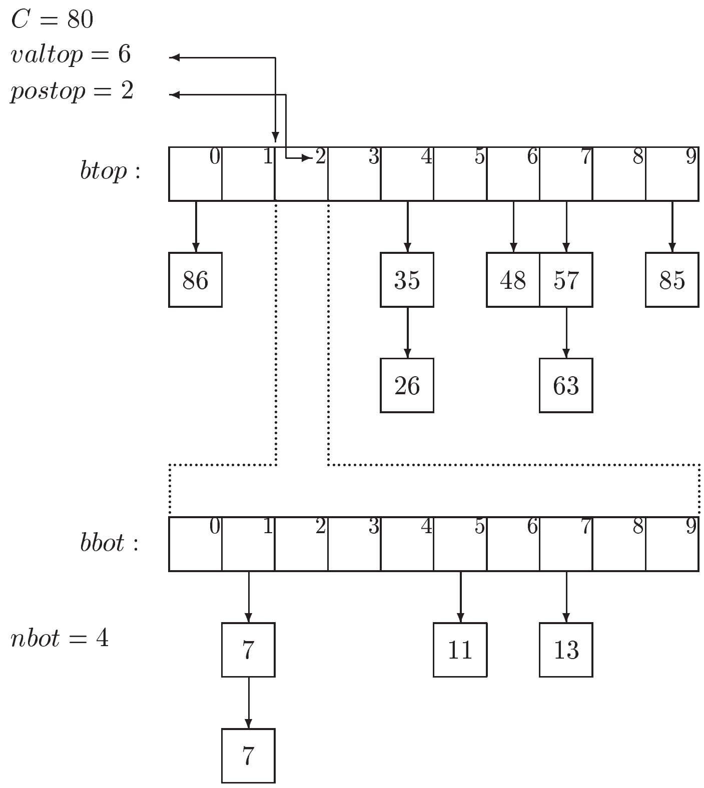
Um sowohl Platz als auch Zeit zu sparen, kann man durch leichte Modifizierung nur ein einziges Bottom-Bucket einsetzen:
Verwende zwei Arrays (Top-Bucket und Bottom-Bucket) der Länge:

$$B = |\sqrt{C+1}| + 1$$

Dabei enthält das Top-Bucket die meisten Elemente in grob vorsortierter Form, nur die Elemente mit den kleinsten Schlüsseln werden im Bottom-Bucket dann endgültig sortiert vorgehalten. 2-Level-Buckets können mit folgenden Elementen aufgebaut werden:

- Das Array *btop* ($[0..B-1]$) nimmt in jedem Bucket Elemente mit Schlüsseln aus einem Intervall der Länge B auf; die Schlüssel stammen aus allen Intervallen außer dem niedrigsten.
- Das Array *bbot* ($[0..B-1]$) nimmt in jedem Bucket Elemente mit genau einem Schlüssel auf; die Schlüssel stammen nur aus dem niedrigsten.
- *valtop* ist die linke Intervallgrenze des niedrigsten Intervalls.
- *postop* enthält den Index des Buckets in *btop* für das niedrigste Intervall (das aber in *bbot* gespeichert wird).

- *nbot* ist die Anzahl der Elemente in *bbot*.
- n ist die Anzahl der Elemente in *bbot* und *btop*.



Dabei gilt:

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

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

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

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

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 $\geq k$ und $\leq k + C$ eingefügt. Dies stellt sicher, daß ein Element aus *bbot* nie nach *btop* wandert.

Lemma: 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

$valtop + B - 1 + C$:

$$\begin{aligned}
&\text{größtmöglicher Schlüssel in } btop \\
&= valtop + BB - 1 \\
&= valtop + B + (B-1)B - 1 \\
&= valtop + B + (\lceil\sqrt{C+1}\rceil + 1)\lceil\sqrt{C+1}\rceil - 1 \\
&\geq valtop + B + C + 1 + \lceil\sqrt{C+1}\rceil - 1 \\
&\geq valtop + B - 1 + C \\
&= \text{größtmöglicher erlaubter Schlüssel} \qquad \diamond
\end{aligned}$$

Annahme: Vor dem ersten *ExtractMin* werden nur Elemente mit $0 \leq$ Schlüssel $\leq$ C eingefügt.

i) *Initialize*:

```
valtop = postop = nbot = n = 0;
Initialisiere bbot, btop;
```

ii) *Insert(x)*:

```
überprüfe Invarianten;
i := (⌊(Key(x) − valtop)/B⌋ + postop) mod B;
if i == postop then
    Füge x in bbot[Key(x) − valpop] ein;
    nbot := nbot + 1; n := n + 1;
else
    Füge x in btop[i] ein;
fi
```

iii) *ExtractMin*

```
co suche kleinstes Element in bbot oc
j := 0 ; while bbot[j] == ∅ 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] == ∅ do
        postop := (postop + 1) mod B;
        valtop := valtop + B;
    od
    while btop[postop] ≠ ∅ do
        entferne beliebiges Element x aus btop[postop];
        Füge x im bbot[Key(x) − valtop] ein;
        nbot := nbot + 1;
    od
fi
```

iv) *DecreaseKey(x, k)*

```
enferne x aus seinem momentanen Bucket;
falls nötig, aktualisiere nbot
Key(x) := k;
Insert(x);
```

Lemma: Worst-case (reelle) Laufzeit bei 2-Level-Buckets sind für *Insert* und *DecreaseKey* $\mathcal{O}(1)$, für *ExtractMin* $\mathcal{O}(n+\sqrt{C})$ und $\mathcal{O}(\sqrt{C})$ (bzw. $\mathcal{O}(1)$) für *Initialize*.

Beweis:
- *Insert*: $\mathcal{O}(1) + 1 = \mathcal{O}(1)$
- *DecreaseKey*: $\mathcal{O}(1) + (\leq 0) = \mathcal{O}(1)$
- *ExtractMin* $\mathcal{O}(\sqrt{C} + \sqrt{C} + \#\text{ Elemente } \begin{array}{c} btop \\ \downarrow \\ bbot \end{array})$. ◇

Potential: Anzahl von Elementen im Buckets im *btop*.

Satz: Amortisierte 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*: $\mathcal{O}(\sqrt{C} + \#\text{ Elemente } \begin{array}{c} btop \\ \downarrow \\ bbot \end{array}) - \#\text{ Elemente } \begin{array}{c} btop \\ \downarrow \\ bbot \end{array} = \mathcal{O}(\sqrt{C})$ ◇

1.7.1.3 k-Level-Buckets

Die Verallgemeinerung von 2-Level-Buckets führt zu k-Level-Buckets. Diese bestehen dann aus k-Arrays der Länge $|\sqrt[k]{C}|$. Dadurch lassen sich die Speicher- und Zeitkomplexität weiter verbessern, der Implementierungsaufwand steigt jedoch stark an.
Zeitkomplexität: $\boxed{\mathcal{O}(C^{\frac{1}{4}})}$

Literartur:

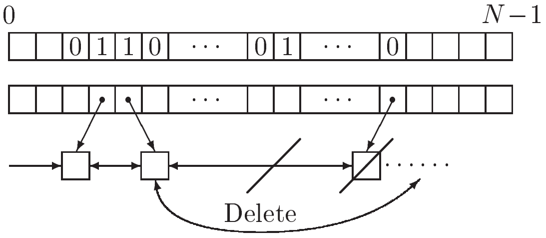
- Aluja, R.K., K. Mehlhorn, J.B. Orbin, R.E. Tarjan
  Faster Algorithmus for the shortest Path Problem
  J.ACM 37 (1990) p.213-223

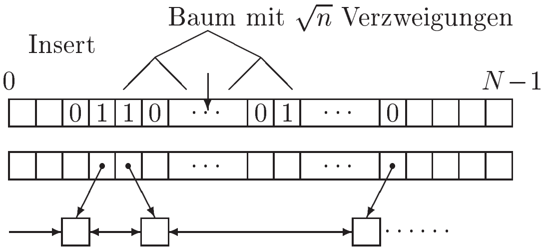
- Cherkassky, B.V., A.V. Goldberg, T.Ratzig
  Shortest Path Algorithmus: Theory and Experimental Evaluation
  Math. Prog. 73(1996) p.129-174

- Cherkassky, B.V., A.V. Goldberg, C. Silverstein:
  Buckets, Heaps, Lists and monoton Priority Queues.
  Proc. 8th Soda, ACM p83-92, 1997 Hot-Queue O($\sqrt{\log C}$)

1.7.2 Von-Ende-Boas-PQ

Von Ende Boas, P., R. Kaas, E.Zijlstra:
Universum U, $|U| = N$, $U \subseteq \mathbb{N}$
Hier: $U = \{0, 1, \ldots, N-1\}$
Bessere Priority-Queue für den Fall $n \geq \log N$



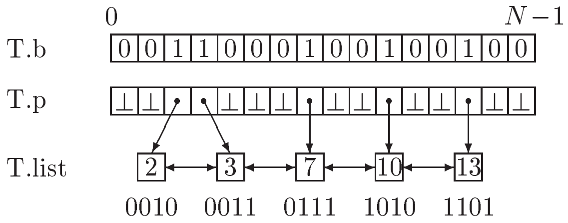


Sei $k \in \mathbb{N}$, $k \geq 2$; $k' = \left\lceil \frac{k}{2} \right\rceil$, $k'' = \left\lfloor \frac{k}{2} \right\rfloor$
$x \in [0, \ldots, 2^k - 1] \leq k$ Bits
$x' = \left\lfloor \frac{x}{2^{k''}} \right\rfloor, x'' := x \mod 2^{k''}$ (x' vordere Hälfte, x'' hintere Hälfte der Bits)
Sei $S = \{x_1, \ldots, x_m\} \subseteq [0, \ldots, 2^k - 1]$

Definition: Eine k-Struktur T für S besteht aus:

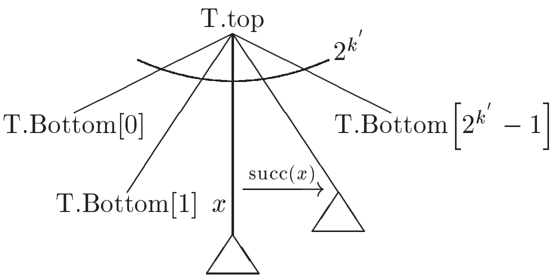
1. der Zahl T.size $= |\{x_1, \ldots, x_m\}| = |S| = m$
2. einer doppelt verketteten Liste T.list, die die Elemente von S in aufsteigender Reihenfolge enthält
3. einem Bitvektor T.b$[0 \ldots 2^k - 1]$ mit T.b$[i]$ = $\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}$ falls $\begin{smallmatrix} i \notin S \\ i \in S \end{smallmatrix}$ 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 $x \in [0 \ldots 2^{k'} - 1]$ ist T.bottom$[x]$ eine k''-Struktur für die Menge $\{x_i''; \ 1 \leq i \leq m; \ x = x_i'\}$

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



T.top ist 2-Struktur $\{0, 1, 2, 3\}$
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 $\min\{y \in \text{S};\ y > x\}$ in der k-Struktur T.

```
if k = 1 or T.Size ≤ 2 then
    naive Suche
else
    if T.size = 0 or x ≥ max in T.list then
        return(Succ(x) gibt's nicht)
    else
        x' := ⌊x / 2^{k''}⌋;
        x'' := x mod 2^{k''};
        if T.top.b[x'] = 1 and x'' < max{T.bottom[x']} then
            return(x' · 2^{k''} + Succ(x'', T.bottom[x']))
        else
            z' := Succ(x', T.top), return(z' · 2^{k''} + min{T.bottom[z']})
        fi
    fi
fi
```



Kosten: $\text{T}(k) \le c + \text{T}(\lceil \frac{k}{2} \rceil) = \mathcal{O}(\log k)$

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

Beweis: $\surd$ $\diamond$

Insert Operation:

- falls x bereits in Priority-Queue, dann fertig

- bestimme *Succc*(x, *T*), füge x davor ein
- bestimme x' und x''
- behandle die entsprechende 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 $\mathcal{O}(\log k)$

Delete Operation:

Komplexität von *Delete* in k-Struktur: $\mathcal{O}(\log k)$
Kosten der Initialisierung: $\sim$ Größe der Datenstruktur
Platzbedarf für k-Struktur: Sei $\mathrm{S}(k)$ der Platzbedarf für eine k-Struktur.
Dann gilt:
$\mathrm{S}(1) = c$
$\mathrm{S}(k) = c2^k + \mathrm{S}(k') + 2^{k'}\mathrm{S}(k')$ für $k \geq 2$
Wir ersetzen zur Vereinfachung: $\mathrm{S}(k) = c2^k + \mathrm{S}(\frac{k}{2}) + 2^{\frac{k}{2}}\mathrm{S}(\frac{k}{2})$

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

Beweis:
Setze: $\mathrm{S}(k) := c' \cdot 2^k \log k$ (besser: Zeige $\mathrm{S}(k) := c'2^k \log k$ funktioniert)
Für $k = 1$ klar
Rekursionsschritt:

$$\begin{aligned}
\text{Platz für } k\text{-Struktur} &\leq c2^k + c'2^{\frac{k}{2}}(\log k - 1) + 2^{\frac{k}{2}}c'2^{\frac{k}{2}}(\log k - 1) \\
&= c2^k + c'2^{\frac{k}{2}}(1 + 2^{\frac{k}{2}})(\log k - 1) \\
&= c2^k + c'2^{\frac{k}{2}}(2^{\frac{k}{2}} \log k + \log k - 2^{\frac{k}{2}} - 1) \\
&\leq c2^k + c'2^{\frac{k}{2}}(2^{\frac{k}{2}} \log k + \log k - 2^{\frac{k}{2}}) \\
&\leq c'2^k \log k \text{ falls} \\
&\qquad c2^k + c'2^k \log k + c'2^{\frac{k}{2}} \log k - c'2^k \leq c'2^k \log k \\
&\qquad \Leftrightarrow c'2^{\frac{k}{2}} \log k \leq (c' - c)2^k \\
&\qquad \Leftrightarrow \frac{c' - c}{c'} \geq \frac{\log k}{2^k} \text{ (Gilt für } c' \text{ groß genug.)}
\end{aligned}$$

$\diamond$

Satz: 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 $\mathcal{O}(N \log \log N)$.

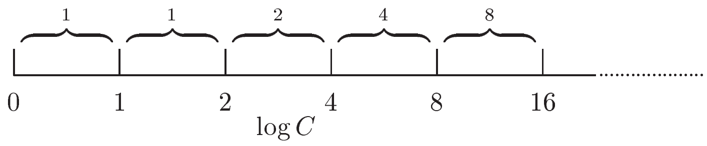
Beweis: s.o. $\diamond$

Literatur zu van Emde Boas-Priority Queue:

1. Mehlhorn Bd. 1, S.290-296
2. van Emde Boas, P., R. Kaas, E. Zijlstra: Design and Implementation of an efficient priority queue, Math. Systems Theory 10 (1977), S. 99-127

1.7.3 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, ähnlich von $\mathcal{O}(\sqrt{C})$ auf $\mathcal{O}(\log C)$. C bezeichne wie bei den 2-Level-Bauckets die maximale Differenz zwischen zwei Schlüsseln im Heap. Die Grundidee besteht darin, anstelle einer Hierarchie von Buckets konstanter Größe, Buckets 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.



Randbedingung:

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

Implementierung:

- B:= $\lceil \log(C+1) \rceil + 1$
- Buckets $b[0..\mathrm{B}]$
- Schranken für Backets $u[0..\mathrm{B}+1]$;
  $u[\mathrm{B}+1] = \infty$
- Bucket-Nr $b\_no[x]$; Index der aktuellen Buckets für x

Invarianten:

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

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

Operationen:

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

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

2. *Insert(x)*: Um ein neues Element einzufügen, wird linear nach dem richtigen Buckets gesucht, wobei die Suche beim größten Bucket beginnt.

i :=B;
<u>while</u> $u[i] >$ Key(x) <u>do</u> $i := i - 1$ <u>od</u>
füge x in $b[i]$ ein;

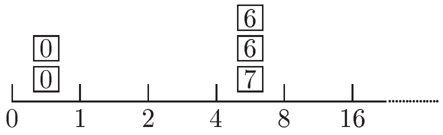
3. *DecreaseKey*(x, K): Hierbei wird analog zur Prozedur *Insert* linear nach dem Bucket gesucht, in den das Element mit dem veränderten Schlüssel eingeordnet werden muß. Der einzige Unterschied besteht darin, daß die Suche in diesem Fall beim aktuellen Bucket des Elements beginnen kann, da der Schlüssel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

   $i := b\_no[x]$;
   entferne x aus $b[i]$;
   Key(x) := k; <u>co</u> Überprüfung, das alles stimmt <u>oc</u>
   <u>while</u> $(u[i] > k)$ <u>do</u> $i := i - 1$ <u>od</u>
   füge x in $b[i]$ ein;

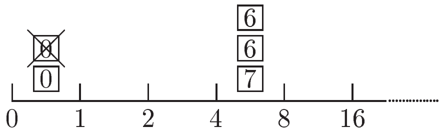
4. *ExtractMin*: Hierbei wird zuerst nach dem kleinsten nicht leeren Bucket gesucht und der kleinste dort enthaltene Schlüssel k festgestellt. Wenn der kleinste Bucket bereits ein Element enthält, so kann dieses ausgegeben werden und es ist sonst nichts zu tun.
   Anderfalls wird $u[0]$ auf k gesetzt und die Bucketgrenzen gemäß der Invarianten neu gesetzt. Danach wird ein minimales Element aus dem kleinsten Bucket, der nun nicht mehr leer sein kann, ausgelesen und entfernt. Von dieser "Aufräumarbeit" sind nur der kleinste nicht-leere Bucket und die davor liegenden leeren Buckets betroffen.

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

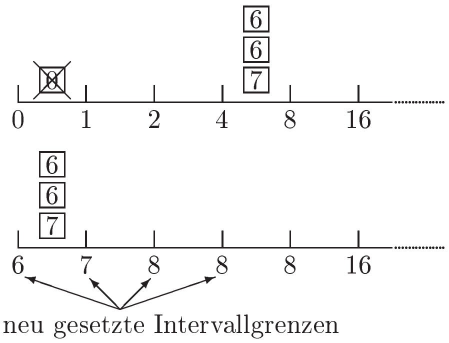
   Beispiel:



1. ExtractMin



2. ExtractMin



$b[i] \neq \emptyset \Rightarrow$ Intervallgröße $\leq 2^{i-1}$
Innerhalb $b[i]$ stehen $i-1$-Buckets zur Verfügung mit Intervallen $[k, k+1[, [k+2, k+4[, \ldots [k+2^{i-2}, k+2^{i-1}[$ bzw. höchstens $u[i+1]$. Da alle Schlüsseln 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:

$$\text{Potential:} \quad \sum_{x \in \text{Radix-Heap}} b\_no[x]$$

Amortisierte Komplexität:

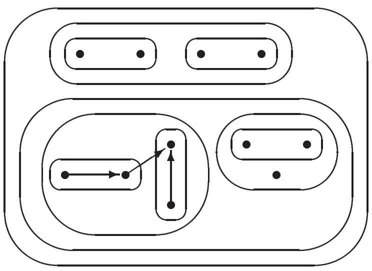
i) *Initialize*: $\mathcal{O}(1)$ bzw. $\mathcal{O}(B)$
ii) *Insert*: $\mathcal{O}(B)$
iii) *DecreaseKey*:$\mathcal{O}(\Delta b\_no[x] - \Delta b\_no[x] + 1) = \mathcal{O}(1)$
iv) *ExtractMin*: $\mathcal{O}(\mathrm{B} + \sum_{x \in b[i]} \Delta b\_no[x] - \sum_{x \in b[i]} \Delta b\_no[x]) = \mathcal{O}(B) = \mathcal{O}(1)$

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

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

1.8 Union-Find-Datenstrukturen

1.8.0 Motivation

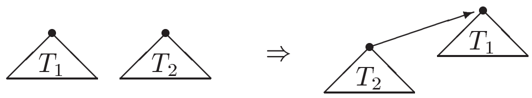


- *Union(T_1, T_2)*: Vereinige T_1 und T_2
  $T_1 \cap T_2 = \emptyset$
- *Find(x)*: Finde den Repräsentanten der größten Menge, in der sich x gerade befindet.

1.8.1 Union-Find-Datenstruktur

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

1.8.1.2 Gewichtete Union (erste Verbesserung)

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

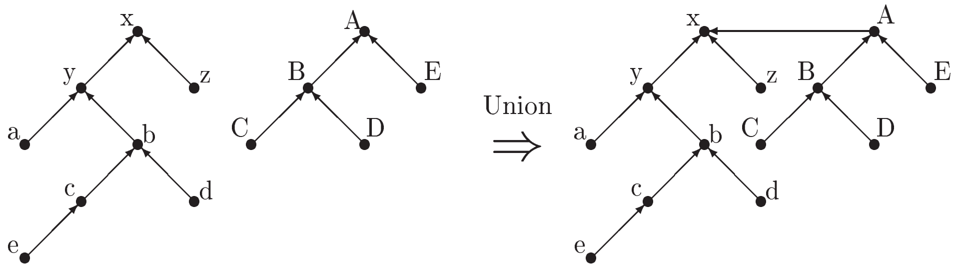
- Zeit für *Find* $= \mathcal{O}(\log n)$
- Zeit für *Union* $= \mathcal{O}(1)$

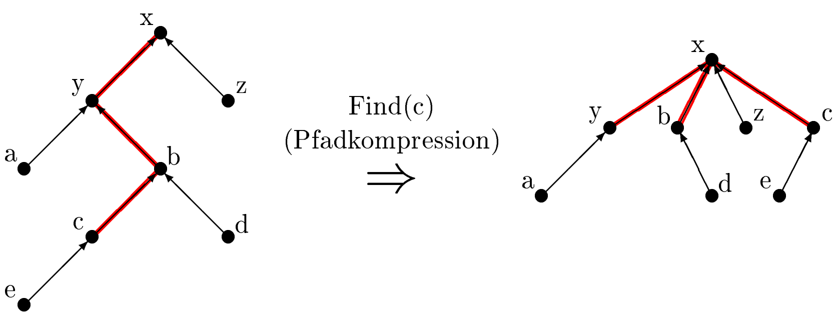
Es gilt auch: Tiefe des Baumes in worst-case : $\boxed{\Omega(\log n)}$

1.8.1.3 Pfad-Kompression mit gewichteten Union (zweite Verbesserung)

Problem: Folge von k *Find*- und *Union*-Operationen auf einen Operation mit n Elementen, darunter $n-1$ *Union*.

Implementierung: Gewichtete *Union*, *Find*'s für Pfad-Kompression:





Bemerkung: Nach der Definition ist $\log^* = \min\{i \geq 1; \underbrace{\log\log\log\ldots\log n}_{i-\log\text{'s}} \leq 1\}$.

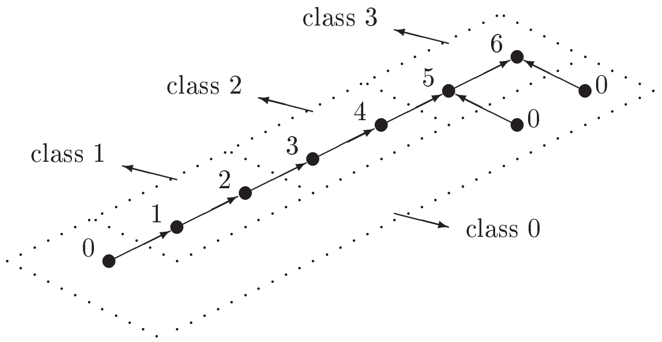
Beispiele:
$\log^* 0 = \log^* 1 = 1$
$\log^* 2 = 1$
$\log^* 3 = 2$
$\log^* 16 = 2^{2^2} = 3$
$\log^* 2^{65536} = 2^{2^{2^{2^2}}} = 5$

> **Satz:** In der obigen Situation 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: $a_0 = 0, a_i = 2^{2^{\cdot^{\cdot^{2}}}}\Big\}{\scriptstyle i-2'\text{en}}$ für $i \geq 1$



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

Definiere eine geignete Potentialfunktion:

- Potential := $c\sum_x \text{dist}(x)$ $\quad$ c ist eine geignete Konstante > 0

Beobachtungen:

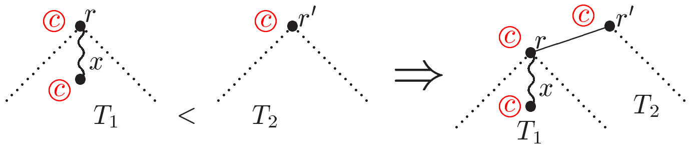
i) Sei T ein Baum in der aktuellen Union-Find-Stuktur (mit Pfad-Kompression), seien x, y Knoten im T, y Vater von $x \Rightarrow$ class(x) $\leq$ class(y).

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

Amotisierte Kosten Find(x):
Sei $x_0 \to x_1 \to x_2 \dots x_k = r$ die Pfad von x_0 zur Wurzel. Kanten (x_{i-1}, x_i) mit $\text{class}(x_{i-1}) = \text{class}(x_i)$. Es gibt höchstens $\log^* n$-Kanten (x_{i-1}, x_i) mit $\text{class}(x_{i-1}) < \text{class}(x_i)$. Ist $\text{class}(x_{i-1}) = \text{class}(x_i)$ und $i < k$ (also $x_i \neq r$), dann ist $\text{dist}(x_{i-1})$ vor der *Find(x)*-Operation ≥ 2, nachher gleich 1.
Damit können die Kosten für alle Kanten (x_{i-1}, x_i) mit $\text{class}(x_{i-1})$=$\text{class}(x_i)$ aus dem Potentialverringerung bezahlt werden und es ergeben sich amortisierte Kosten:

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

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



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

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

◇

1.8.1.4 Erweiterung

1) Bessere obere Schranke $\alpha(k,n), k \geq n$. Ackermannfunktion: $A(m,n)$:
   $A(0,n) = 2n \; ; \; n \geq 0$
   $A(m,0) = 2 \; ; \; m \geq 1$
   $A(m+1, n+1) := A(m, A(m+1, n))$

$n \longrightarrow$; $m \downarrow$

0	2	4	6	8
2	4	8		
2				
2				
2				
⋮				

Die Ackermannfunktion steigt asymptotisch schneller als jede primitiv-rekursive Funktion.

Definition: Die Klasse der primitiv rekursiven Funktionen[2] (auf den natürlichen Zahlen) ist induktiv wie folgt definiert:

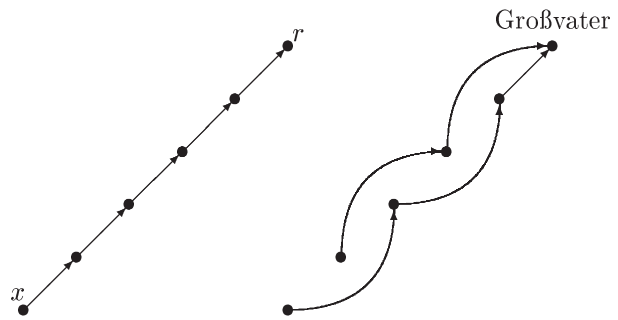
1. Alle konstanten Funktionen sind primitiv rekursiv.
2. Alle identischen Abbildungen (Projektionen) sind primitiv rekursiv.
3. Die Nachfolgerfunktion auf den natürlichen Zahlen ist primitiv rekursiv.
4. Jede Funktion, die durch Einsetzung (Komposition) von primitiv rekursiven Funktionen entsteht, ist selber primitiv rekursiv.
5. Jede Funktion, die durch sog. primitive Rekursion aus primitiv rekursiven Funktionen entsteht, ist primitiv rekursiv. Primitive Rekursion bedeutet, daß die definition von $f(n+1,\cdots)$ zurückgeführt wird auf $f(n,\cdots)$. Formaler, f muß ein Gleichungssystem der folgenden Form erfüllen:

$$\begin{aligned} f(0,\cdots) &= g(\cdots) \\ f(n+1,\cdots) &= h(f(n,\cdots),\cdots) \end{aligned}$$

wobei g, h bereits primitiv rekursive Funktionen sind.

$\alpha(n,n) = \min\{i; A(i,i) \geq n\}$
Ebenso entsprechend die untere Schranke.

2) Varianten



[2] U.Schöning, Theoretische Informatik kurz gefaßt, BI-Verlag, 1993 ', p. 105.

Kapitel 2

Selektieren und Sortieren

2.1 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
Bekannt: Fälle $i = 1,\ n$ (Minimum, Maximum bestimmen)
Standardalgorithmus: $n - 1$ Vergleiche
Zeigen: Diese Anzahl ist optimal:

Beweis: Interpretiere jeden Algorithmus als Turnier. Ein Spiel wird vom kleineren Element gewonnen. Beobachtung: Jedes Element außer dem Gesamtsieger muß mindestens ein Spiel verloren haben $\Rightarrow n - 1$ Vergleiche notwendig.

$\diamond$

Nun: Bestimmung des Vize-Meisters (bzw. zweitkleinstes Element)

Satz: Das zweitkleinste Element kann mit $n + \lceil \log_2 n \rceil - 2$ Vergleichen bestimmt werden.

Beweis: einem K.O. Turnier: $(n - 1)$ Vergleiche. Beobachtung: Das zweitkleinste Element ist unter den „Verlierern“ gegen das Minimum zu suchen. Deren Anzahl ist $\lceil \log_2 n \rceil$. Bestimme unter diesen Elementen wiederum das Minimum $\rightsquigarrow$ zweitkleinstes Element $(\lceil \log_2 n \rceil - 1)$ $\diamond$

Literatur:

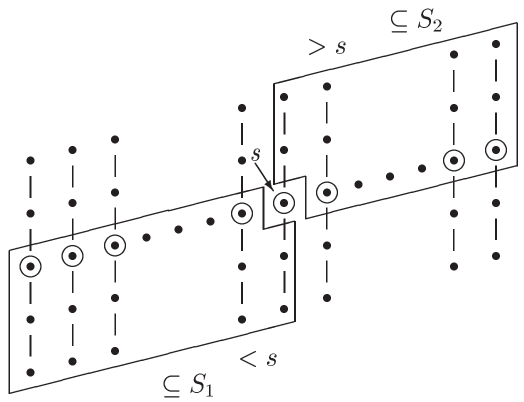
- Carroll, Lewis: Lawn Tennis Tournaments. St. Jones Garzette (Aug. 1, 1883), p. 5-6. Reprinted in the complete work of lewis Carroll. New York. Modern Librarry 1947.
- Pratt, V.R. and F.F. Yao: On lower bounds for computing the i-th largest element. Proc. 14. Anm IEEE. Swat (1973), p. 70-81.
- Knuth, D. E. Vol. 3

Definiere: $i = \lceil \frac{n}{2} \rceil$: Median (das mittlere Element) $= \lceil \frac{n}{2} \rceil$ - kleinstes Element.

2.2 Der Blum-Floyd-Pratt-Rivest-Tarjan Selektions-Algorithmus

Blum, M. and RW. Floyd and V.R. Pratt and R.L. Rivest and R.E. Tarjan: Time bounds for selection. JCSS (1973), p. 448-461.
Sei m eine kleine ungerade Zahl (etwa $5 \leq m \leq 21$). Sei $S := \{a_1, \ldots, a_n\}$. Suche i-kleinstes Element in S. Wir betrachten folgenden Algorithmus BFPRT:

1. Teile S in $\lceil \frac{n}{m} \rceil$ Blöcke auf, $\lfloor \frac{n}{m} \rfloor$ davon mit m Elementen
2. Sortiere jede diese Blöcke
3. Sei S' die Menge der $\lceil \frac{n}{m} \rceil$ Mediane der Blöcke Bestimme rekursiv den Median dieser Mediane (also das $\left\lceil \frac{|S'|}{2} \right\rceil$-kleinste Element). Nenne diesen Median s.
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 $n \geq 3m - 1$



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.
   Sonst (falls $|S_2| > n - i$) bestimme rekursiv das $(i - |S_1| - 1)$-kleinste Element in S_2

Komplexitätsbetrachtung (# Vergleiche im worst-case):

Sei $\mathrm{T}(n)$ 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:

$$\mathrm{T}(n) \leq \underbrace{\mathrm{T}\left(\left\lceil \frac{n}{m} \right\rceil\right)}_{3.} + \underbrace{\mathrm{T}\left(\left\lfloor \frac{3}{4} n \right\rfloor\right)}_{5.} + \underbrace{\left\lceil \frac{n}{m} \right\rceil C_m}_{2.} + \underbrace{\left\lfloor \frac{n}{2} \right\rfloor}_{4.}$$

Satz: Der Selektions-Algorithmus BFPRT bestimmt das i-kleinste Element von n Elementen mit $\mathcal{O}(n)$ Vergleichen (und Zeit).

Beweis: Behauptung:

$$\mathrm{T}(n) \leq c \cdot n \qquad ; \text{wobei } c = c(m) \text{ konstant ist.}$$

Behauptung ist ok, falls

$$
\begin{aligned}
\mathrm{T}(n) \leq & \mathrm{T}\left(\left\lceil \frac{n}{m} \right\rceil\right) + \mathrm{T}\left(\left\lfloor \frac{3}{4}n \right\rfloor\right) + \left\lceil \frac{n}{m} \right\rceil C_m + \left\lfloor \frac{n}{2} \right\rfloor \leq cn, \text{ dies gilt, falls} \\
& \left\lceil \frac{n}{m} \right\rceil c + \left\lfloor \frac{3}{4}n \right\rfloor c + \left\lceil \frac{n}{m} \right\rceil C_m + \left\lfloor \frac{n}{2} \right\rfloor \leq cn \qquad | \cdot \frac{1}{n} \quad \text{(Ind.A.)} \\
& \Leftrightarrow \text{(bis auf } \lceil\ \rceil, \lfloor\ \rfloor) \ \frac{c}{m} + \frac{3}{4}c + \frac{C_m}{m} + \frac{1}{2} \leq c \\
& \Leftrightarrow -\frac{c}{m} - \frac{3}{4}c + c \geq \frac{C_m}{m} + \frac{1}{2} \\
& \Leftrightarrow c \geq \frac{\frac{C_m}{m} + \frac{1}{2}}{1 - \frac{3}{4} - \frac{1}{m}}
\end{aligned}
$$

Schlußbemerkung: $m = 11 \rightsquigarrow c = c(m) \approx 20$ $\diamond$

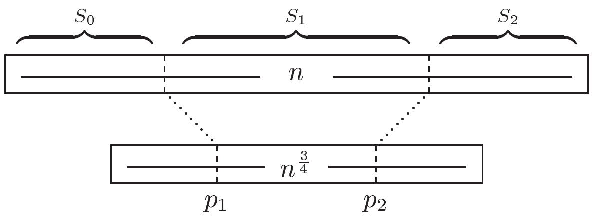
2.3 Randomisierter-Median-Algorithmus

Problem: Bestimme den Median von n Elementen:

1. Wähle $n^{\frac{3}{4}}$ Elemente zufällig aus dem n Elementen aus.
2. Sortiere diese $n^{\frac{3}{4}}$ Elemente mit einem $n \log n$-Algorithmus.
3. Setze

   $p_1 := \max\{\frac{n^{\frac{3}{4}}}{2} - \sqrt{n}, 1\}$-kleinsten Element der $n^{\frac{3}{4}}$ Elemente.

   $p_2 := \min\{\frac{n^{\frac{3}{4}}}{2} + \sqrt{n}, n^{\frac{3}{4}}\}$-kleinsten Element der $n^{\frac{3}{4}}$ Elemente.



4. Partitioniere n Elemente in

   $S_0 := \{\text{Elemente} < p_1\}$
   $S_1 := \{p_1 \leq \text{Elemente} \leq p_2\}$
   $S_2 := \{p_2 < \text{Elemente}\}$

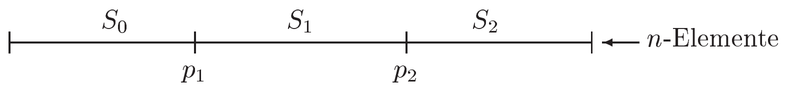
5. Falls $|S_0| \geq \lceil \frac{n}{2} \rceil$ oder $|S_2| \geq \lceil \frac{n}{2} \rceil$ oder $|S_1| \geq 4 \cdot n^{\frac{3}{4}}$ dann wiederhole den Algorithmus
   ansonsten sortiere S_1 und liefer das $\lceil \frac{n}{2} \rceil - |S_0|$-kleinste Element davon ab.

Satz: Obiger Algorithmus bestimmt den Median von n-Elementen mit einer erwarteten Anzahl von $\frac{3}{2}n + o(n)$ Vergleichen.

Beweis:

i) Korrektheit: klar.

ii) Anzahl der Vergleichen einer Iteration:
$\mathcal{O}(n^{\frac{3}{4}} \log n^{\frac{3}{4}})$ + Kosten der Partitionierung
Kosten der Partitionierung: zu naiv, bessere Ansatz:



Wähle zuerst mit der Wahrscheinlichkeit je $\frac{1}{2}$ aus, ob Element x mit p_1 oder p_2 verglichen wird, mache zwei Vergleiche nur, falls nötig. Erwartete Anzahl Vergleiche dafür ist gleich:

$$= n \cdot \frac{1}{2}\left(\frac{|S_0|}{n} \cdot 1 + \frac{|S_1| + |S_2|}{n} \cdot 2\right) + n \cdot \frac{1}{2}\left(\frac{|S_2|}{n} \cdot 1 + \frac{|S_0| + |S_1|}{n} \cdot 2\right) =$$

$$= \frac{n}{2}\left(\frac{|S_0| + |S_2|}{n} + 2\frac{\overbrace{|S_0| + |S_1| + |S_2|}^{n} + |S_1|}{n}\right) =$$

$$= \frac{n}{2}(1 - n^{\frac{1}{4}} + 2 + 2 \cdot n^{-\frac{1}{4}}) = \frac{3}{2} + o(n)$$

Wir zeigen, daß der Algorithmus mit der Wahrscheinlichkeit $\geq 1 - \mathcal{O}(n^{-\frac{1}{4}})$ nur eine Iteration benötigt (daraus folgt, daß insgesamt Anzahl der Vergleiche $\leq \frac{3}{2}n + o(n)$ ist). Dafür verwenden wir Hilfsmittel aus der Stochastik:

- ***Zufallsvariable*** (ZV): X, Werte $\in \{0, 1\}$ (binäre Zufallsvariable) mit

  $$X = \begin{cases} 1 \text{ mit WS } p \\ 0 \text{ mit WS } q = 1 - p \end{cases}$$

- ***Erwartungswert*** einer ZV: $\mathrm{E}(X) = \sum_{x \in \text{ Wertebereich}} \mathrm{WS}(X = x) \cdot x$
  (X ist diskret, d.h. der Wertebereich von X ist endlich)

- ***Markov-Ungleichung***: $\mathrm{WS}(X \geq t) \leq \frac{\mathrm{E}(X)}{t}$ für X nicht negativ

- ***Tschebyscheff-Ungleichung***: $\mathrm{WS}(|X - \mathrm{E}(x)| \geq t) \leq \frac{\mathrm{Var}(X)}{t^2}$

- ***Binomialverteilung***: Seien $X_1, \cdots, X_n$ unabhängige, identisch gleichverteilte Zufallsvariablen mit Wertebereich $\{0, 1\}$, $\mathrm{WS}(X_i = 1) = p$;

  $$X := \sum_{i=1}^{n} X_i$$

  X ist gemäß der Binomialverteilung verteilt, Wertebereich $= \{0, 1, \cdots, n\}$:

  $$\mathrm{WS}(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$
  $$\mathrm{E}(X) = n \cdot p$$
  $$\mathrm{Var}(X) = n \cdot p \cdot (1 - p) = n \cdot p \cdot q.$$

  X ist verteilt gemäß der Binomialverteilung mit den Parametern n und p, in Zeichen $X \sim \mathrm{B}(n, p)$.

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 der aller Elemente $\leq$ Median auswählen.
Was ist also die Wahrscheinlichkeit dafür, daß keine neue Auswahl der $n^{\frac{3}{4}}$ Elemente stattfinden muß?
Setze Binäre Zufallsvariable für jedes $X_i, \cdots, X_n$ mit:

$$X_i = \begin{cases} 1 & \text{,falls Element } i \text{ ausgewählt und in der unteren Hälfte der Wert liegt} \\ 0 & \text{,sonst} \end{cases}$$

$X = \sum X_i$ binomialverteilt mit Parametern n und $\frac{1}{2}n^{-\frac{1}{4}}$
$\mathrm{E}(X) = \frac{1}{2}n^{\frac{3}{4}}$, $\mathrm{Var}(X) = n \cdot \frac{1}{2}^{-\frac{1}{4}}(1 - \frac{1}{2}n^{-\frac{1}{4}}) = \frac{1}{2}n^{\frac{3}{4}}(1 - o(1))$

Die Wahrscheinlichkeit hierbei ist:

$$\mathrm{WS}(|S_0| \geq \frac{n}{2}) = \mathrm{WS}(X \leq \frac{1}{2}n^{\frac{3}{4}} - \sqrt{n}) \leq \mathrm{WS}(|X - \mathrm{E}(X)| \geq \sqrt{n}) \leq$$
$$\leq \frac{\frac{1}{2}n^{\frac{3}{4}}(1 - o(1))}{n} \leq \frac{1}{2}n^{-\frac{1}{4}}(1 - o(1))$$

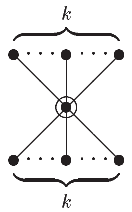
Die Wahrscheinlichkeit für den Fall, daß man von vorne anfangen muß:

$$\mathrm{WS} \Rightarrow 1 - \left(\frac{1}{2}n^{-\frac{1}{4}}(1 - o(1))\right)$$

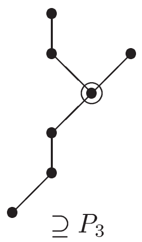
Die anderen beiden Wahrscheinlichkeitsbedingungen ($\mathrm{WS}(|S_2| \geq \lceil \frac{n}{2} \rceil)$ und $\mathrm{WS}(|S_1| \geq 4 \cdot n^{\frac{3}{4}})$) ergeben analoge Abschätzungen $\Rightarrow$ Widerholung mit WS $\leq \mathcal{O}(n^{-\frac{1}{4}})$ $\diamond$

2.4 Der Schönhage-Paterson-Pippenger-Median-Algorithmus

Definition: Sei $k \in \mathbb{N}\backslash\{0\}$. P_k ist die folgende partielle Ordnung:



$2k + 1$ Elemente



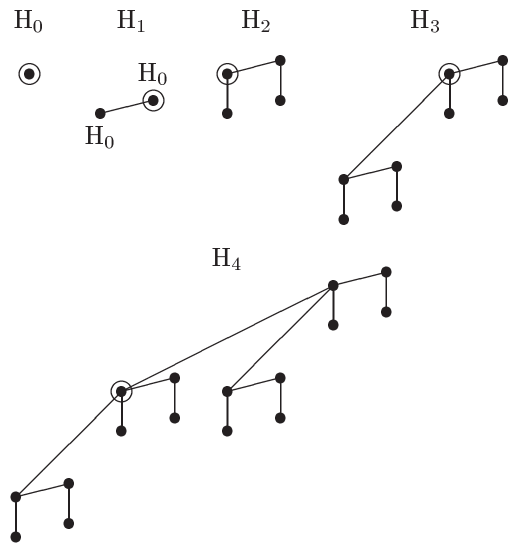
$\supseteq P_3$

Spezielle Binomialbäume mit „Zentren".

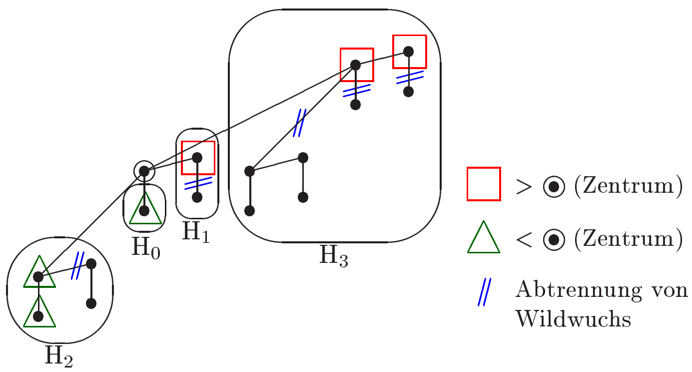
Definition:

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.



Das 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

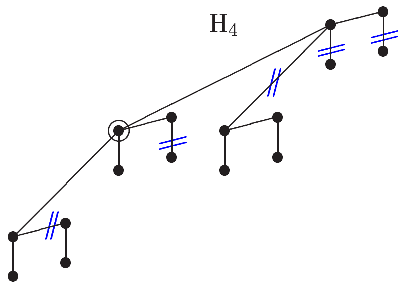
- sein Zentrum

- 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}.

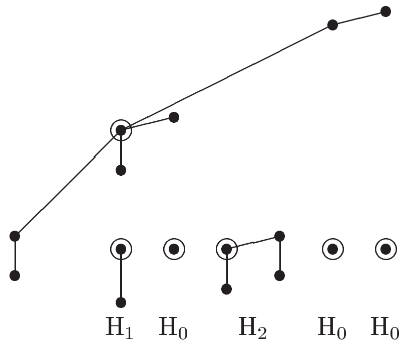
c) H_{2k+1} kann so zerlegt werden, daß die Zusammenhangskomponente des Zentrums genau 2^k Knoten $\geq$ dem Zentrum enthält, indem höchstens $2^{k+1} - 1$ Kanten entfernt werden.
H_{2k} kann so zerlegt werden, daß die Zusammenhangskomponente des Zentrums genau 2^k Knoten enthält, die alle $\leq$ dem Zentrum sind, indem höchstens $2^k - 1$ 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.

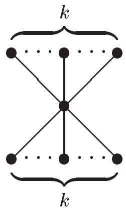
Bemerkung: Bei jedem Konstruktionsschritt wird ein Vergleich durchgeführt, um zu bestimmen, welcher der beiden Teilbäume das kleinere Zentrum hat. Im Algorithmus von Schönhage, Paterson und Pippenger werden aus Teilstücken H_r größere Bäume H_{r+1} zusammengebaut, wodurch schrittweise eine partielle 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 $k \leq 2^h - 1$. Im folgenden 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 der Teile a) bis d) des Zerlegungslemmas:

a) **Lemma 1:** 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 ($r \geq 1$): $C_r = 1{+}2C_{r-1}$ und $C_0 = 0$. Damit folgt sofort $C_r = 2^r{-}1$. ◇

b) **Lemma 2:** H_r kann in folgende disjunkte Bereiche unterteilt werden:

   – sein Zentrum,
   – eine Reihe H_1, H_3, ..., H_{r-1} (r gerade) bzw. ..., H_{r-2} (r ungerade) von Unterbäumen, deren Zentren über dem von H_r liegen,
   – eine Reihe H_0, H_2, ..., H_{r-2} (r gerade) bzw. ..., H_{r-1} (r ungerade) von Unterbäumen, deren Zentren unter dem von H_r liegen.

   Beweis: durch Induktion über r. Induktionsanfang: für H_0 gilt die Behauptung. Induktionsannahme: die Behauptung gelte für H_{r-1}.

   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 $H_1, H_3, ..., H_{2h-3}$ (Zentren über z) und $H_0, H_2, ..., H_{2h-2}$ (Zentren unter z) partitionieren. Zusammen mit dem H_{2h-1}, dessen Zentrum über z liegt, ergibt sich die Induktionsbehauptung für H_{2h}.
   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, ..., H_{2h-1} (Zentren über z) und H_0, H_2, ..., H_{2h-2} (Zentren unter z) partitionieren. Zusammen mit dem H_{2h}, dessen Zentrum unter z liegt, ergibt sich die Induktionsbehauptung fr H_{2h+1}.

   ◇

Wir bezeichnen im folgenden mit H_{2h}^- den Baum, der entsteht, wenn wir H_{2h} so zerlegen, da alle Elemente oberhalb des Zentrums wegfallen. Mit H_{2h+1}^+ bezeichnen wir den Baum, der entsteht, wenn wir H_{2h+1} so zerlegen, da alle Elemente unterhalb des Zentrums wegfallen.

c) **Lemma 3:** 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 2. 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}^-. 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:

$$E^-(2h) = 1 + \sum_{i=0}^{h-1} E^-(2i) \overset{I.A.}{=} 1 + \sum_{i=0}^{h-1} 2^i = 1 + \underbrace{\sum_{i=1}^{h} 2^{i-1}}_{2^h-1} = 2^h.$$

2. Sei $r = 2h+1, h > 0$. Wir betrachten die Partitionierung von H_{2h+1} mit Zentrum z wie in Lemma 2. 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}^+:

$$\begin{aligned} K^+(2h+1) =& h + 1 + \sum_{i=1}^{h} K^+(2(i-1)+1) \\ \overset{I.A.}{=}& h + 1 + \sum_{i=1}^{h}(2^i - 1) = 1 + \sum_{i=1}^{h} 2^i \\ =& 1 + \underbrace{\sum_{i=1}^{h+1} 2^{i-1}}_{2^{h+1}-1} - 1 = 2^{h+1} - 1. \end{aligned}$$

   Für die Zahl $E^+(2h+1)$ der Elemente in H_{2h+1}^+ gilt:

$$E^+(2h+1) = 1 + \sum_{i=1}^{h} E^+(2(i-1)+1) \overset{I.A.}{=} 1 + \underbrace{\sum_{i=1}^{h} 2^{i-1}}_{2^h-1} = 2^h.$$

◇

d) **Lemma 4:** 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 $k = k_0 2^0 + k_1 2^1 + ... + k_{h-1} 2^{h-1}$ und die Partitionierung von H_{2h} mit Zentrum z wie in Lemma 2.

– 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 uns aber nach Lemma 2 $K^+(2i+1) = 2^{i+1} - 1$ 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 höchstens h Kanten aufgebrochen. Genau k Elemente oberhalb von 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:

$$\sum_{i=0}^{h-1} k_i E^+(2i+1) = k.$$

– 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 3 $K^-(2i) = 2^i - 1$ Kanten, also:

$$\sum_{i=0}^{h-1} k_i (2^i - 1) \leq k - 1.$$

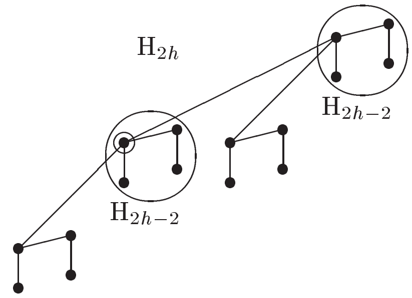
Für jedes i mit $k_i = 0$, schneide H_{2i} ganz weg. Dabei werden höchstens h Kanten aufgebrochen. Genau k Elemente unterhalb von z bleiben zurück, da jedes $k_i = 1$ für 2^i in k steht, und ein H^-_{2i} genau $E^-(2i) = 2^i$ Elemente enthält, also:

$$\sum_{i=0}^{h-1} k_i E^-(2i) = k.$$

Damit ergibt sich fr die Gesamtanzahl aufzubrechender Kanten eine obere Schranke von $3k + 2h$. Lemma 3 liefert uns darüberhinaus die gewünschte Aussage ber die Form der abgetrennten Teile. ◇

Betrachte H_{2h}

- „größer“: $H_{2h-1}, H_{2h-3}, \ldots, H_1$
- „kleiner“: $H_{2h-2}, H_{2h-4}, \ldots, H_0$

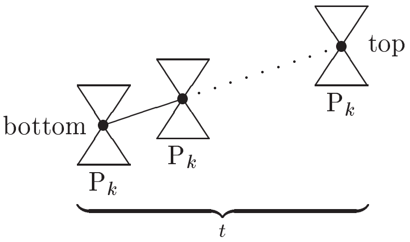


$\mathrm{U}(h)$:= Anzahl der Elemente in $H_{2h} \geq$ Zentrum: $\mathrm{U}(h) = 2\mathrm{U}(h-1) = 2^h\mathrm{U}(0) = 2^h$
$\mathrm{D}(h)$:= Anzahl der Elemente in $H_{2h} \leq$ Zentrum: $\mathrm{D}(h) = 2\mathrm{D}(h-1) = 2^h\mathrm{D}(0) = 2^h$

Anzahl der Kanten, die entfernt werden müssen:

$$\left.\begin{aligned} \mathrm{C}_u(h) &\leq 2 + 2\mathrm{C}_u(h-1) \\ &= 2 + 4 + 2^3 + \ldots + 2^h \\ &= 2^{h+1} - 2 \\ \mathrm{C}_d(h) &\leq 1 + 2\mathrm{C}_d(h-1) \\ &= 2^h - 1 \end{aligned}\right\} \quad \mathrm{C}_H \leq 2^{h+1} - 2 + 2^h - 1 \approx 3 \cdot 2^h$$

Kette von P_k's:



Gesamtzahl der Elemente:

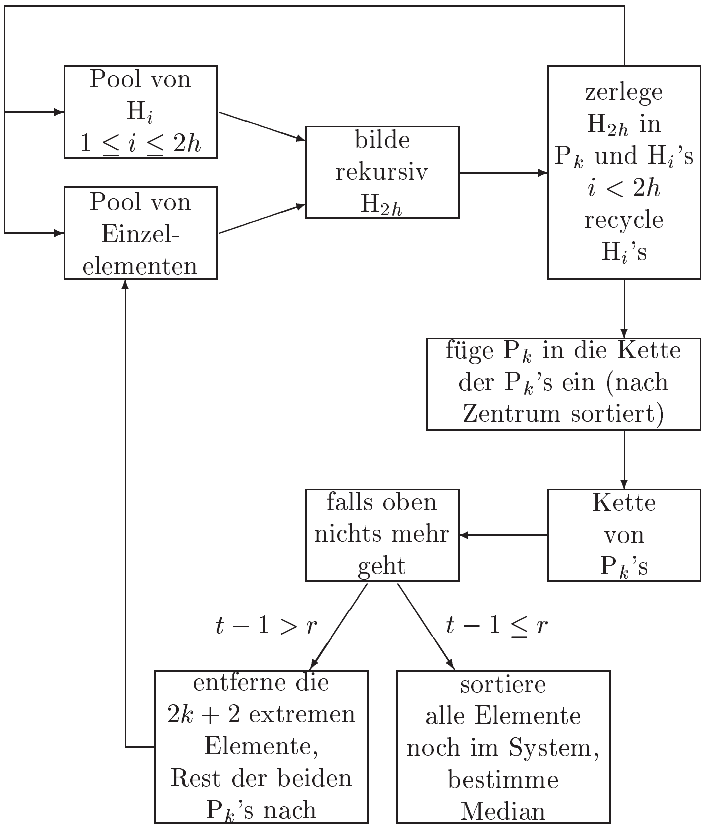
n
$t(2k+1)$ in den P_k's
$r = n - t(2k+1)$ Rest

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

$k := \left\lfloor n^{\frac{1}{4}} \right\rfloor$
h sdg. $2^{h-1} \leq k < 2^h$



Definiere: $r :=$ Anzahl der noch im H_{2h} Produktionsprozeß steckenden Elemente.
(für jedes $i < 2h$ höchstens ein $H_i \Rightarrow r \leq \sum\limits_{i=0}^{2h-1} = 2^{2h} - 1$)
$R :=$ 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.

$$m = t + 2\frac{n-R}{2(k+1)} = t + \frac{n-R}{k+1}$$

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

$$\leq \left(\frac{n-R}{k+1} + t\right)\left[\underbrace{2k}_{1} + \underbrace{3k+2h}_{2} + \underbrace{\log\frac{n}{2k+1}}_{4}\right] + \underbrace{R\log R}_{5} + \underbrace{r}_{3}$$

$k := \left\lfloor n^{\frac{1}{4}} \right\rfloor$, h so daß $2^{h-1} \leq k < 2^h$

$$\begin{aligned}
\text{Damit } r &= \mathcal{O}(k^2) \\
t &= \mathcal{O}(k^2) \text{ zum Schluß} \\
R &= \mathcal{O}(k^3)\text{, und damit die} \\
\text{Anzahl der Vergleiche } &= \mathrm{T}(n) \leq 5n + o(n)
\end{aligned}$$

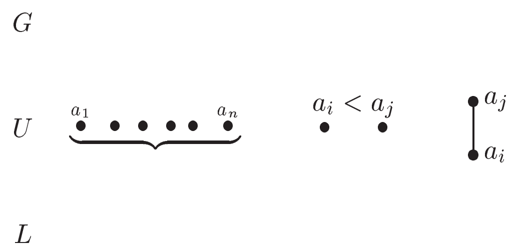
Verbesserte Version: $T(n) = 3n + o(n)$
Dor/Zwick: $2,95n + o(n)$

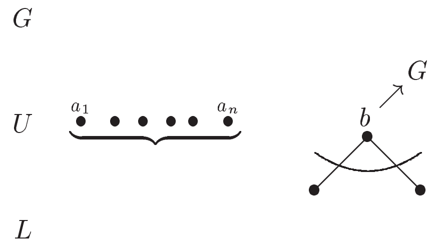
Bester bekannter Algorithmus: K.Zwick ($2.95n + o(n)$)
Literatur: SPP: Finding the Median. JCSS 13(1976) p.84-199

2.5 Eine untere Schranke für die Medianbestimmung

Satz: Jeder (vergleichsbasierte) Medianalgorithmus benötigt in worst-case mindestens $\lceil \frac{3n}{2} \rceil - 2$ 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. 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 zu stellen bevor die Antwort feststellen kann(d.h. U soll möglichst ungeordnet bleiben). Durch die (konsisten!) Antworten des Gegenspielers auf die "$\stackrel{?}{<}$"-Queries des Algorithmus entsteht ein DAG. Gegenspieler hält DAG "einfach".
(Angenommen $y > z$ und $y > x \Rightarrow y$ soll sehr groß sein $\Rightarrow y \to G$)





$G \quad \leq \dfrac{n+1}{2} - 1$

$U \quad \underbrace{\overset{a_1}{\bullet} \ \bullet \ \bullet \ \bullet \ \bullet \ \overset{a_n}{\bullet}}$  Solange ein Element unverglichen ist, ist nicht klar, welches Median ist.

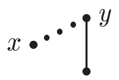
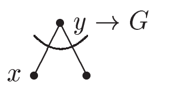
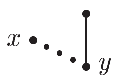
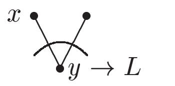
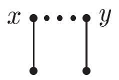
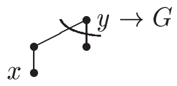
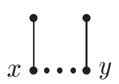
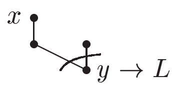
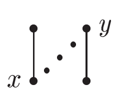
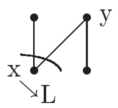
$L \quad \leq \dfrac{n+1}{2} - 1$

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: 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) $x, y \in U$

	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:
$|L|$ wird $\frac{n-1}{2}$ oder $|G|$ 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).

Analyse:
$C :=$ Anzahl der Vergleiche
$P :=$ Anzahl der Paare in U

i) In der Phase 1 gilt folgende Invariante: $C - P + 2|U| \geq 2n$
Beweis: vollständige Induktion:
Induktionsanfang: $0 - 0 + 2n \geq 2n$
Induktionsschritt: Gemäß der Tabelle. $\diamond$

ii) Sei C # der Vergleiche am Ende der Phase 1, U die am Ende der Phase 2: In der Phase 2 werden noch $\geq |U| - 1 - |P|$ Vergleiche nötig. Die Anzahl der Vergleiche für alle Phasen:

$$\text{Anzahl der Vergleiche} \geq C + |U| - 1 - |P|.$$

Am Ende der Phase 1 gilt ja $C \geq 2n + |P| - 2|U|$ (wg. Invariante).
$\Rightarrow$ Anzahl der Vergleiche:

$$\begin{aligned} \geq 2n + |P| - 2|U| + |U| - 1 - |P| = 2n - |U| - 1 \\ \geq \frac{3}{2}n - \frac{3}{2} = \\ = \left\lceil \frac{3}{2}n - 2 \right\rceil \text{ da } |U| \leq \frac{n+1}{2} \end{aligned}$$

$\diamond$

2.6 Eine bessere untere Schranke

Literatur: Samuel W. Bent, John W. John: “Finding the median requires $2n$ comparisons.” Proceedings of the 17th Annual ACM Symposium on Theory of Computing 1985, p.213–216.

Satz: Sei T ein Entscheidungsbaum für die Bestimmung des i-kleinsten von n verschiedenen Elements, mit $i^2 \geq \log\left[\binom{n}{i}\frac{1}{n-i+1}\right] + 3$. Dann gilt, mit

$$p := 2\sqrt{\log\left[\binom{n}{i}\frac{1}{n-i+1}\right] + 3} - 2,$$

$$\text{Höhe}(T) \geq \log\left[\binom{n}{i}\frac{2^{n-p}}{n-i+1}\right]$$

Bemerkung: $i = \lceil \frac{n}{2} \rceil \to \binom{n}{\lceil \frac{n}{2} \rceil} \approx \mathcal{O}(\frac{2^n}{\sqrt{n}}) \Rightarrow \text{Höhe}(T) \geq 2n - o(n)$ (untere Schranke für Median).

Beweis: Gegenspielerargument + Abzählargument:
„T hat viele Blätter“, A Teilmenge der Schlüssel, $|A| = i$;
T_A Teilbaum von T, alle Blätter von T_A sind auch Blätter von T.
Zu zeigen: T_A hat viele Blätter. ($\binom{n}{i}$ Möglichkeiten, T_A’s zu konstruieren, Anzahl Blätter in T_A: 2^{n-p}, jedes Blatt von T kommt in höchstens $n - i + 1$ T_A’s vor)

Beobachtungen: Jedes Blatt w von T liefert folgende Informationen:

answer(w) $\xrightarrow{\text{liefert}}$ i-kleinstes Element x
little(w) $\xrightarrow{\text{liefert}}$ $i-1$ Elemente $< x$
big(w) $\xrightarrow{\text{liefert}}$ $n-i$ Elemente $> x$

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 (keine zus. Blätter durch Abschneiden)
- für jedes Blatt w von T_A gilt: little(w) $\subseteq A$

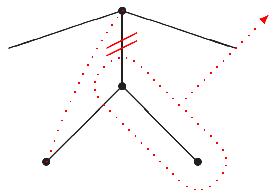
Setze $\bar{A}$ das Komplement von A,

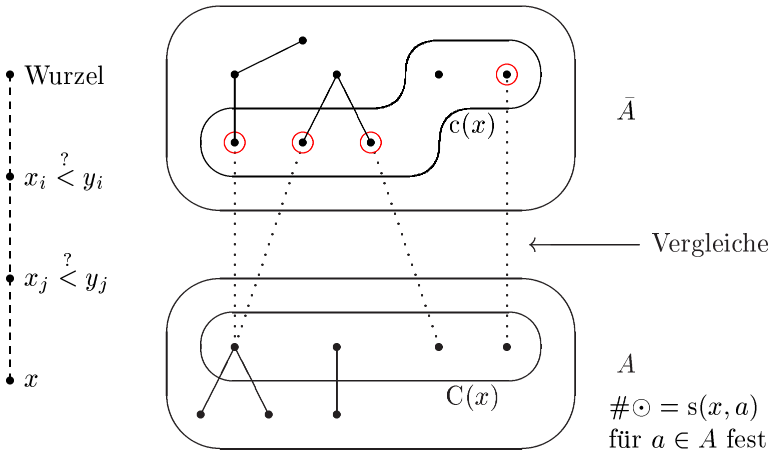
$$r := \sqrt{\log\left[\binom{n}{i}\frac{1}{n-i+1}\right] + 3}$$

$$s := r - 1$$

Es gilt: $p = r + s - 1$

Die Konstruktion (das "Zurechtstutzen") von T erfolgt in zwei Phasen:





Erste Phase von der Wurzel nach unten (Breitensuche)
Betrachte Knoten x. Definiere:

- C(x) sind die Elemente $a \in A$ für die es kein $b \in A$ gibt und einen Knoten auf dem Pfad von der Wurzel von T zu x, an dem kein Vergleich mit einem anderen Element $b \in A$ erfolgt ist mit dem Ergebnis $a < b$.

- $\mathrm{c}(x)$ sind entsprechend die in x bekannten Minima in $\bar{A}$.
- $\mathrm{s}(x,a)$ $(a \in A)$ ist die Anzahl der Elemente $\in \bar{A}$ (genauer: $\in \mathrm{c}(x)$) mit denen a auf dem Pfad von Wurzel von x verglichen wurde.

Regeln für Phase 1:
Es werden in Knoten x die Elemente a und b verglichen:

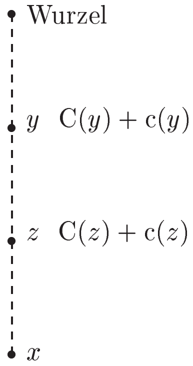
Regel 1.1: Falls $a \in A$ und $b \in A$ behalte x in T_A bei.

Regel 1.2: Falls $a \in \bar{A}$ und $b \in \bar{A}$ behalte x in T_A bei.

Regel 1.3: Sei o.B.d.A. $b \in \bar{A}$, $a \in A$. Ersetze den Unterbaum in T mit Wurzel x mit dem Unterbaum, dessen Wurzel das Kind von x ist, das dem Ergebnis entspricht (d.h. lasse den Vergleich $a < b$ aus, nach Voraussetzung $A < \bar{A}$).

Phase 1 läuft solange $|\mathrm{C}(x)| \leq r$. Ein Knoten auf einem Pfad in T von der Wurzel, bei dem $|\mathrm{C}(x)|$ erstmals $= r$ wird, heißt *kritisch*.Jeder Pfad in T von der Wurzel zu einem Blatt enthält genau einen kritischen Knoten.

Betrachte in T_A einen Pfad von der Wurzel zu einem kritischen Knoten x. Sei y ein Knoten auf diesem Pfad, z sein Kind.



Dann gilt:

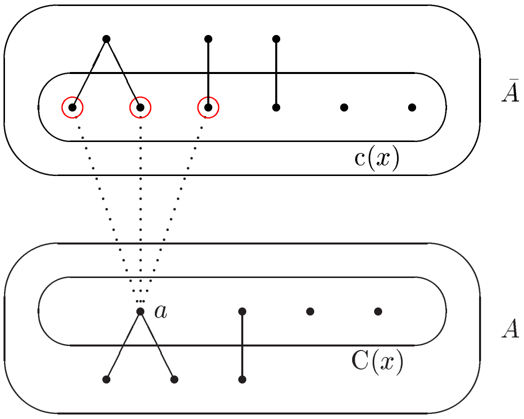
$$|\mathrm{C}(z)| + |\mathrm{c}(z)| \geq |\mathrm{C}(y)| + |\mathrm{c}(y)| - 1$$

Da $|\mathrm{C}(\text{Wurzel})| = |A| = i$ und $|\mathrm{c}(\text{Wurzel})| = |\bar{A}| = n-i$, müssen überhalb eines jeden kritischen Knoten x mindestens

$$i - |\mathrm{C}(x)| + n - i - |\mathrm{c}(x)| = n - r - |\mathrm{c}(x)|$$

Vergleiche erfolgen. Von jedem kritischen Knoten abwärts arbeitet der Gegenspieler nach einer Strategie für Phase zwei. Sei x ein solcher kritischer Knoten. Dann ist $|\mathrm{C}(x)| = r$.

Phase 2: Sei $a \in \mathrm{C}(x)$ ein Element mit minimalem $\mathrm{s}(x,a)$.



$\#\odot = \mathrm{s}(x,a)$
für $a \in A$ fest

Fall 1: $\mathrm{s}(x,a) \geq s$.

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 $\mathrm{C}(x) \setminus \{\mathrm{answer}(w)\}$ und Elementen in $\bar{A}$ erfolgt sind).

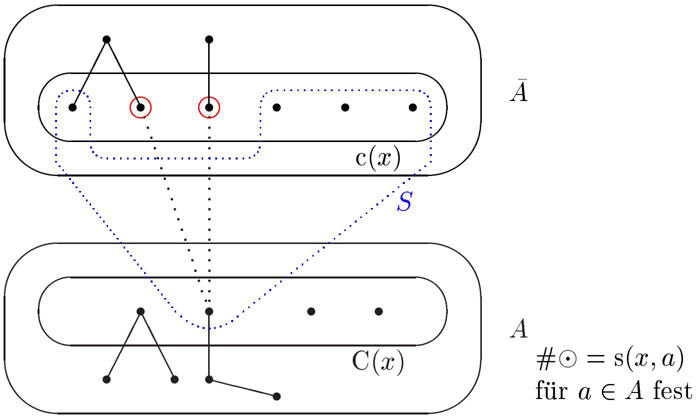
Also:

$$\begin{aligned} \mathrm{Tiefe}(T) &\geq n - 1 + s(r-1) \\ &= n - r - s - 1 + (s+1)r \\ &= n - r - s + 2 + \log\left[\frac{\binom{n}{i}}{n-i+1}\right] \\ &> \log\left[\binom{n}{i}\frac{2^{n-p}}{n-i+1}\right] \end{aligned}$$

In diesem Fall folgt also die Behauptung direkt!

Fall 2: $s(x,a) < s$

Fall 2.1:



$|\mathrm{c}(x)| \geq p$. Sei $S := \mathrm{c}(x) \cup \{a\} \setminus$Elemente, die in $\mathrm{s}(x,a)$ 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 a und b die beiden Elemente, die in y verglichen werden):

Regel 2.1: falls $a, b \in S$, dann bleibt y erhalten.

Regel 2.2: andernfalls, sei oBdA. $a \in A \setminus S$ oder $b \in \bar{A} \setminus S$; ersetze y mit seinem Unterbaum durch das Kind von y und dessen Unterbaum, das der Antwort von $a < b$ entspricht.

Also, 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.
Zusammen mit Phase 1 ergibt sich eine Tiefe von T_A:

$$\begin{aligned} &\geq n - r - |\mathrm{c}(x)| + |S| - 1 \\ &\geq n - r - |\mathrm{c}(x)| + |\mathrm{c}(x)| + 1 - (s-1) - 1 \\ &= n - r - s + 1 \\ &= n - p \end{aligned}$$

Fall 2.2: $|\mathrm{c}(x)| < p$. $S := \mathrm{C}(x)$
Regeln so, daß der Algorithmus als Antwort das Maximum von S bestimmt. Tiefe von T_A:

$$\begin{aligned}
&\geq n - r - |\mathrm{c}(x)| + |S| - 1 \\
&\geq n - r - (p-1) + r - 1 \\
&= n - p
\end{aligned}$$

In Wirklichkeit: Jeder Pfad in T_A von x zu 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 muß, falls T_A w auch enthält, little$(w) \subseteq A$ sein. Für das Element in $A-$ little(w) gibt es $\leq n-i+1$ Möglichkeiten $\Rightarrow$ Anzahl Blätter von $T \geq \frac{1}{n-i+1}\binom{n}{i} 2n^{n-p} \Rightarrow$ Tiefe$(T) \geq \log \left[\binom{n}{i} \frac{2^{n-p}}{n-i+1}\right]$.

$\diamond$

Einige Referenzen:

1. L. Hyafil: Bounds for selection. SIAM J.Compute 5(1976), p.109-114.

2. S. Beut and John W. John: Finding median requires $2n$ compression. Proc 17. STOC (1985), p.213-216.

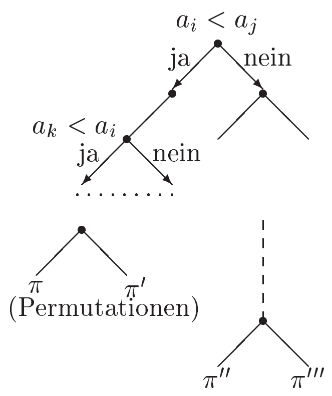
3. John Welliaveetil John: A new lower bound for the set-partitioning problem. SIAM J. Compute 17 (1988), p. 640-647.

4. Doris Dor and Uri Zwick: Median selection requires $(2 + \varepsilon)n$ compression. Proc 37 FOCS (1996), p. 125-134

5. Doris Dor and Uri Zwick: Selecting the median Proc 6 SODA (1995), p. 28-37

2.7 Untere Schranke für (vergleichbasiertes) Sortieren

Gegeben n-Schlüsseln, Queries $a_i \overset{?}{<} a_j$. Dies entspricht einen Entscheidungsbaum.



Beobachtung: Jede Permutation $\pi \in S_n$ kommt in mindestens einem Blatt des Entscheidungsbaums vor ($|S_n| = n!$).$\Rightarrow$ Tiefe des Entscheidungsbaums $\geq \log_2 n!$.

Stirling: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

$$\Rightarrow \text{ Also Tiefe } \geq \log_2\left[\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right] = n \cdot \log_2 n - n \cdot \log_2 e + \frac{1}{2}\log 2\pi n$$
$$= n \log_2 n - \mathcal{O}(n)$$

Satz: Jeder Vergleichsbasierter Sortieralgorithmus benötigt mindestens

$$n \log_2 n - \mathcal{O}(n)$$

Vergleiche.

2.8 Bucketsort im Schnitt

Gegeben n-Schlüsseln $\in [0,1]$. Schlüsseln gleichverteilt. Um diese zu sortieren:

1. Initialisiere Buckets $b_0, b_1, \cdots, b_n$
2. Lege a_i in Bucket $\lceil n \cdot a_i \rceil$
3. Sortiere Schlüssel innerhalb jedes Buckets
4. Kankataniere Buckets

Satz: Wird zum Sortieren Sortieralgorithmus mit der 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 die Zeit $\mathcal{O}(n)$. Die Zeit für Schritt 3 beträgt:

$$\mathcal{O}(\sum_{i=0}^{n} n_i^2)$$

Die Erwartungswert für $\sum n_i^2$ läßt sich folgendermaßen abschätzen:

$$\mathrm{E}\left(\sum_{i=0}^{n} n_i^2\right) \leq c \sum_{0 \leq i < j \leq n} \mathrm{WS}(a_i \text{ und } a_j \text{ enden in gleichen Bucket})$$
$$= c\left(\sum_{0 \leq i < j \leq n} \frac{1}{n+1}\right) = \mathcal{O}(n).$$

◇

2.9 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 zudem ein Divide-and-Conquer-Verfahren.

Devide: Wähle ein Pivot-Element p (z.B. das letzte) und partitioniere $a[l..r]$ gemäß p in zwei Teile $a[l..i-1]$ und $a[i+1]$ (durch geeignetes Vertauschen der Elemente), wobei anschließend $a[i] = p$.

Conquer: Sortiere $a[l..i-1]$ und $a[i+1..r]$ rekursiv.

Algorithmus:

```
proc qs(a,l,r)
{
    if l ≥ 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] ≤ p do i++ od
        while i < j and p ≤ a[j] do j-- od
        if i < j then vertausche a[i] und a[j];
    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 in der Regel $n+1$ Schlüsselvergleiche. Lediglich wenn die beiden Indizes zusammenfallen (was nur bei einem Pivotduplikat passieren kann), benötigt er nur n Vergleiche. Durch geschicktes Einstreuen von if-Abfragen kann man auch 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 gleichgroße Teilintervalle entstehen, das Pivot ist dann stets der Median.

  Anzahl der Schlüsselvergleiche: $\sum_{i=1}^{\log n}(n+1) = \log n(n+1) \approx n \log n$

- Worst-case-Analyse: Z.B. bei eine aufsteigend sortierte Eingabe.

  Anzahl der Schlüsselvergleiche: $\Omega(n^2)$

- Average-case: Da die Laufzeit von Quicksort sehr stark von den Eingabedaten abhängt und sich demnach keine scharfe allgemeingültige Schranke angeben läßt, 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: Die durchschnittliche Anzahl von Schlüsselvergleichen beträgt unter der Annahme , daß alle Permutationen für die Eingabe gleichwahrscheinlich sind, höchstens:

$$C_n = 2(n+1)(\mathrm{H}_{n+1} - 1) \approx 2n \ln n - 0.846..n + o(n) \approx 1.386 n \log n$$

wobei $H_n := \sum_{i=1}^{n} i^{-1}$ die n-te Harmonische Zahl ist.

Beweis: (Variante mit $n-1$ Vergleiche pro Durchlauf)
Sei C_n die Anzahl der Vergleichen bei dem Feld der Länge n. $C_0 = C_1 = 0$.

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

Da

i) für festes $j \in \{1, \cdots, n\}$ das Produkt (in beiden Varianten) das j-kleinstes Element ist und

ii) auch für die rekursiven Aufrufe wieder alle Eingabepermutationen gleichwahrscheinlich sind:

$$\begin{aligned}
\Rightarrow \qquad\qquad C_n &= n - 1 + \frac{2}{n}\sum_{j=0}^{n-1} C_j; \\
nC_n &= n^2 - n + 2\sum_{j=0}^{n-1} C_j; \\
(n-1)C_{n-1} &= (n-1)^2 - (n-1) + 2\sum_{j=0}^{n-2} C_j; \\
nC_n - (n-1)C_{n-1} &= 2n - 1 - 1 + 2C_{n-1}; \\
nC_n &= 2n - 2 + (n+1)C_{n-1}; \qquad / \frac{1}{n(n+1)} \\
\frac{C_n}{n+1} &= 2\frac{n-1}{n(n+1)} + \frac{C_{n-1}}{n} = \\
&= 2\frac{n-1}{n(n+1)} + 2\frac{n-2}{(n-1)n} + \frac{C_{n-2}}{n-1} = \\
&= 2\frac{n-1}{n(n+1)} + \cdots + \frac{C_2}{3} = \\
&= 2\left[\sum_{j=2}^{n} \frac{j-1}{j(j+1)}\right] = \\
&= 2\left[\sum_{j=2}^{n} \left(\frac{j}{j(j+1)} - \frac{1}{j(j+1)}\right)\right] = \\
&= 2\left[\mathrm{H}_{n+1} - \frac{3}{2} - \sum_{j=2}^{n}\left(\frac{1}{j} - \frac{1}{(j+1)}\right)\right] = \\
&= 2\left[\mathrm{H}_{n+1} - \frac{3}{2} - \left(\frac{1}{2} - \frac{1}{(n+1)}\right)\right] = \\
&= 2\left[\mathrm{H}_{n+1} - 2 + \frac{1}{(n+1)}\right]
\end{aligned}$$

$\Rightarrow C_n = 2(n+1)\left(\mathrm{H}_{n+1} - 2 + \frac{1}{n+1}\right)$; $\mathrm{H}_n \approx \ln n + 0.57721\cdots + o(1)$
$\Rightarrow C_n \approx 2n\ln n - 4n + o(n) \approx 1.386 n \log n$ $\diamond$

Kapitel 3

Minimale Spannbäume

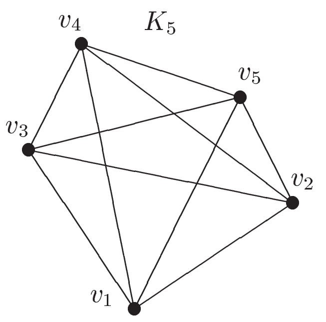
3.0 Grundlagen

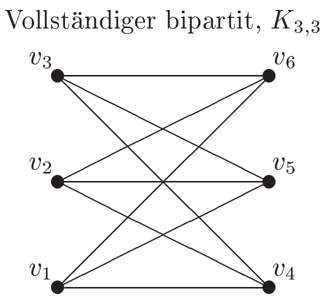
Ein Graph $G = (V, E)$ besteht aus einer Menge von Knoten V und einer Menge Kanten E. Die Anzahl der Knoten bezeichnen wir mit n ($|V| = n$), die Anzahl der Kanten mit m ($|E| = m$). Eine Kante $(v, v) \in E$ heißt ***Schlinge***. Falls E eine Multimenge ist, spricht man von ***Mehrfachkanten***. Kanten (u, v) und (v, u) heißen ***antiparallel***. Graphen ohne Schlingen und Mehrfachkanten bezeichnen wir mit ***einfachen Graphen*** (eng. simple). Formal definiert:

$$E \subseteq \binom{V}{2} =: \{X \subseteq V, |X| = 2\}$$

In diesem Fall bezeichnet man $G = (V, E)$ als einen ***ungerichteten Graphen.***

Graphische Darstellung:





Ist $E \subseteq V \times V$, dann heißt G ***gerichteter Graph***, für $(u, v) \in E$:

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*** $N(v) := \{w; (v, w) \in E\}$ eines Knotens v versteht man die Menge der direkten Nachbarn von v. Der ***Grad*** eines Knotens ist definiert als:

$$deg(v) = \begin{cases} |N(v)| & ; \text{falls } G \text{ ungerichtet und} \\ indeg(v) + outdeg(v) & ; \text{falls G gerichtet.} \end{cases}$$

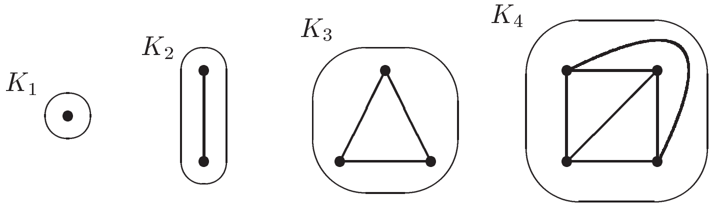
Dabei sind $indeg(v)$ die Anzahl aller Kanten, die v als Endknoten haben, und $outdeg(v)$ die Anzahl aller Kanten, die v als Startknoten haben.

Beobachtung: $\sum_{v \in V} deg(v) = 2|E|$

Korollar: In jedem Graphen ist die Anzahl der Knoten mit ungeradem Grad eine gerade Zahl.

Das ***Komplement*** $\overline{G} = (V, \binom{V}{2} - E)$ 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, Untergraph, ...) von $G = (V, E)$, falls $V' \subseteq V$ und $E' \subseteq E$. H heißt ***(knoten-) induzierter Teilgraph,*** falls H Teilgraph von G ist, und es gilt $E' = E \cap \binom{V'}{2}$.

Ein ***Kantenzug*** ist eine Folge $e_1 := \{v_0, v_1\}, \cdots, e_l := \{v_{l-1}, v_l\}$, wobei alle e_i verschieden sind. v_0 und v_l bezeichnen wir als Endknoten. Die Länge des Kantenzuges bezeichnen wir mit l. Sind bei einem Kantenzug auch alle v_i verschieden, so sprechen wir von einem ***Pfad.*** Ein Kantenzug mit $v_l = v_0$ heißt ***Zykel***. Ein Zykel, in dem alle v_i verschieden sind, heißt ***Kreis***.
Ein Graph G heißt ***zusammenhängend***, wenn es für alle $u, v \in V$ einen Pfad gibt, der u und v verbindet. Ein (kanten-) maximaler zusammenhängender Teilgraph heißt ***(Zusammenhangs-)Komponente***.



Ein Graph G ist ***azyklisch***, wenn er keine Kreise enthält. Wir bezeichnen diesen 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***.

Satz: Ist $T = (V, E)$ ein Baum, dann ist $|E| = |V| - 1$.

Beweis: Induktion über die Anzahl der Knoten:
$n = 0, 1$: klar.
$n \to n + 1$: Sei $v \in V$, $|V| = n + 1$, $V' = V - \{v\}$, $deg(v) = 1$.
Solche Knoten müssen existieren, denn ansonsten würde ein Zykel entstehen, und das wäre ein Widerspruch zur Definition eines Baumes. $\diamond$

Korollar: Ein Graph G ist zusammenhängend, g.d.w. G ein Spannbaum hat.

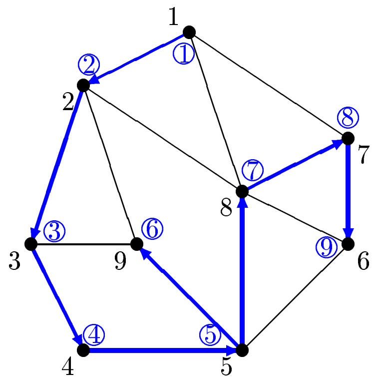
3.1 Traversierung von Graphen

Sei $G = (V, E)$ ein ungerichteter Graph. Anhand eines Beipiels betrachten wir die zwei Algorithmen DFS (Tiefensuche) und BSF (Breitensuche).

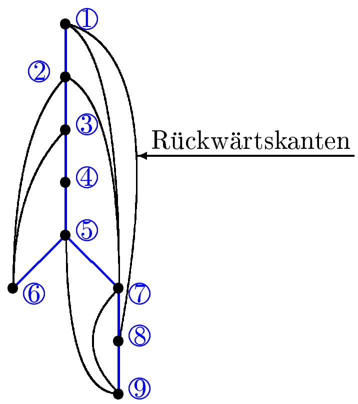
3.1.1 DFS-Algorithmus

```
while ∃ unvisitet v do
    r := pick(random) unvisited node;
    push r onto stack;
    while stack ≠ ∅ do
        v := pop the top element;
        if not visited v then
            mark v visited;
            push all neigbour's of v onto stack;
            DFS_Ops(v);
        fi
    od
od
```

Beispiel für DFS:

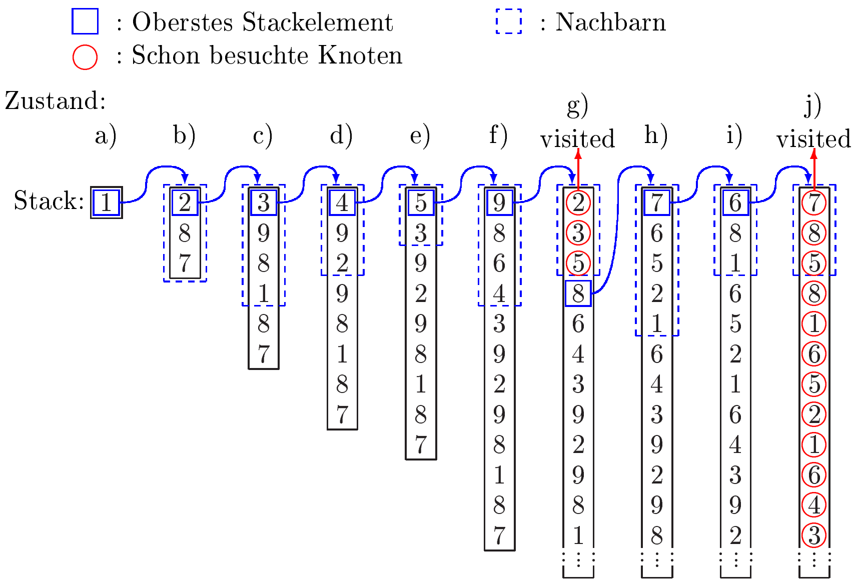


Die markierten Kanten bilden einen Spannbaum:



Wir betrachten nun 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

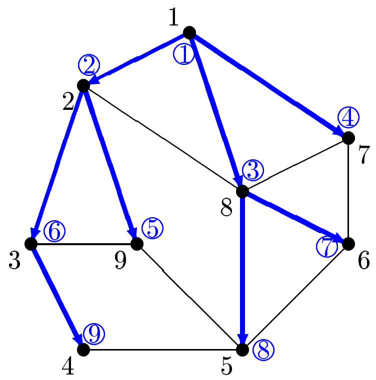
Stack entfernt und so wird das Element 8 zum obersten Stackelement. Im Zustand j) sind alle Elemente markiert, so daß eins nach dem anderen aus dem Stack entfernt werden.



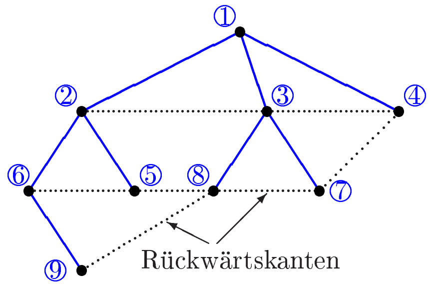
3.1.2 DFS-Algorithmus

```
while ∃ unvisitet v do
    r := pick(random) unvisited node;
    push r into queue;
    while queue ≠ ∅ do
        v := remove front element of queue;
        if not visited v then
            mark v visited;
            append all neigbour's of v to end of queue;
            BFS_Ops(v);
        fi
    od
od
```

Beispiel für BFS:



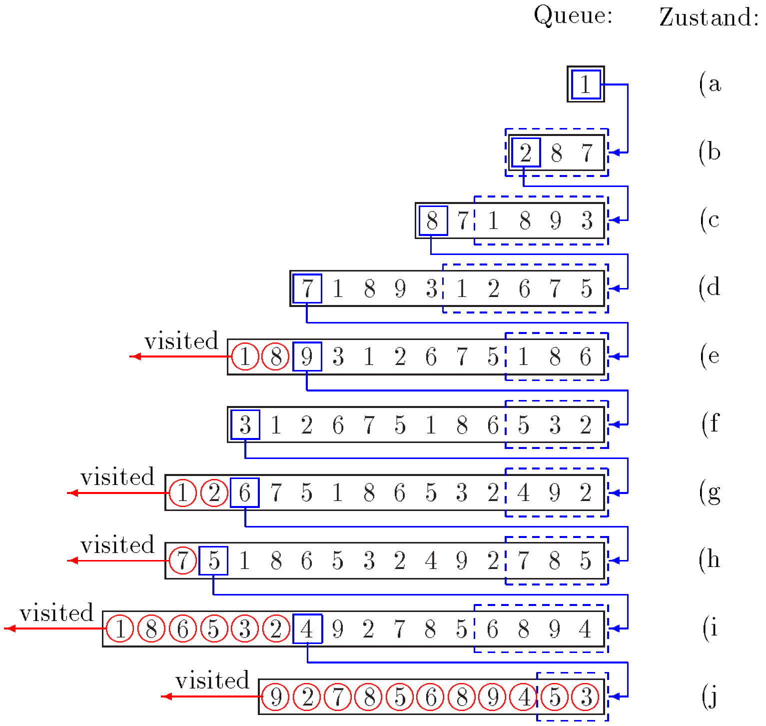
Die markierten Kanten bilden einen Spannbaum:



Wir betrachten nun den Queuezustand: 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.

□ : Vorderstes Queue-Element ⬚ : Nachbarn

○ : Schon besuchte Knoten



3.2 Minimale Spannbäume

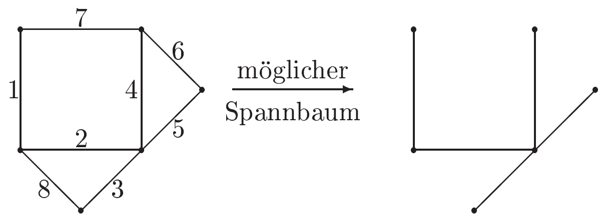
Sei $G = (V, E)$ ungerichteter einfacher Graph gegeben mit folgenden Eigenschaften:

- $w : E \to \mathbb{R}$ eine Gewichtsfunktion,
- $E' \subseteq E$: $w(E') = \sum_{e \in E'} w(e)$,
- $T = (V', E')$ ein Teilgraph von G mit $w(T) = w(E')$,
- sei o.B.d.A. G zusammenhängend, dann definiere:

Definition: T heißt minimaler Spannbaum (MSB) von G, falls T Spannbaum von G ist und gilt:

$$w(T) \leq w(T')$$

Beispiel:



Anwendungen:

- Telekom: Telefonvermittlungen
- Leiterplatinen

3.2.1 Konstruktion von minimalen Spannbäumen

Es gibt zwei Prinzipien für die Konstruktion von minimalen Spannbäumen(Tarjan):

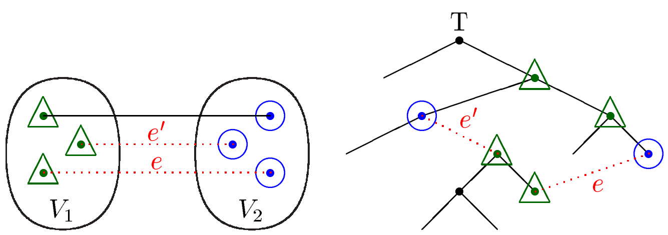
- „blaue“ Regel
- „rote“ Regel

Satz: 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 $E_C = E \cap (V_1 \times V_2)$ 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.

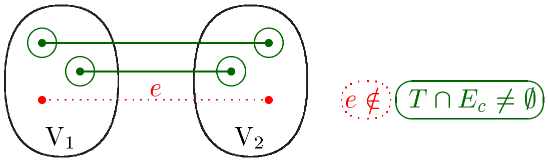
Beweis: durch Widerspruch

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.



Sei $e' \in E_c \cap T$. Dann gilt nach Vorraussetzung 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.

2. Sei $e \in E_c$ minimal. Annahme: e kommt in <u>keinem</u> 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 T' minimaler Spannbaum und $e \in T'$.

$\diamond$

Satz: Sei $G = (V, E)$, $w : E \to \mathbb{R}$, ein gewichteter, zusammenhängender Graph. Dann gilt: (**„rote“ Regel**)

1. Gibt es zu $e \in E$ einen Kreis C in G, der e enthält und $w(e) > w(e') \quad \forall e' \in C \setminus \{e\}$ 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 $w(\hat{T}) < w(T)$. Ein Widerspruch.

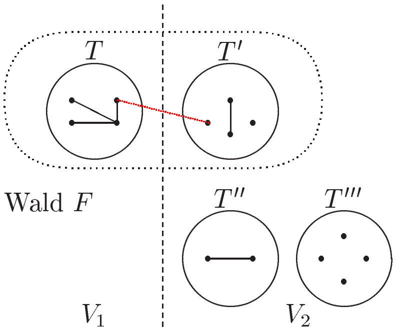
2. Wir nehmen an, e_i liege in jedem minimalen Spannbaum (MSB) von G, und zeigen die Behauptung durch Widerspruch.
Sei T ein beliebiger MSB von G. Entfernen wir e_i aus T, so zerfällt T in zwei nicht zusammenhängende Teilbäume T_1 und T_2. Da e_i auf einem Kreis $c_1 = e_1, \ldots, e_k$ in G liegt, können wir wie unter 1. 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.

$\diamond$

Literatur: R.E. Tarjan: “Data Structures and Network Algorithms”, SIAM CBMS-NSF Reihe Bd. 44.

3.2.2 Generischer minimaler Spannbaum-Algorithmus

Initialisiere Wald F von Bäumen, jeder Baum ist ein singulärer Knoten.
(jedes $v \in V$ bildet einen Baum)
<u>while</u> Wald F mehr als einen Baum enthält <u>do</u>
    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;
<u>od</u>



3.3 Kruskals Algorithmus

```
algorithm Kruskal (G,w):=
⌜
    sortiere die Kanten nach aufsteigendem Gewicht in eine Liste L;
    initialisiere F = {T_i, i = 1,...,n} Wald (n = |V|) mit T_i = {v_i};
        co G = (V,E), V = {v_1,...,v_n} oc
    co implementiere T_i's als in-Trees mit den jeweiligen Wurzeln als
        Repräsentanten; allgemein: Unio-Find-Problem oc
    MSB:=∅;
    for i := 1 to length(L) do
        {v,w} := L_i;
        x := Baum ∈ F, der v enthält; co x :=Find(v) oc
        y := Baum ∈ F, der w enthält; co y :=Find(w) oc
        if x ≠ y then
        co v und w in verschiedenen Bäumen oc
            MSB:=MSB ∪ {{v,w}};
            if |x| ≤ |y| then
                co Baum x enthält ≤ Knoten als Baum y oc
                co y :=Union(x,y) oc
            else
                vereinige x und y, mit y neues Kind der Wurzel von x;
            fi
        fi
    od
⌞
```

Korrektheit: Falls die Gewichte eindeutig sind ($w()$ injektiv), folgt die Korrektheit direkt aus der "blauen" und "roten" Regel. Ansonsten Induktion über die Anzahl der Knoten ($|V|$):

Ind. Anfang: $|V|$ klein:$\surd$

Sei $r \in \mathbb{R}$, $E_r := \{e \in E; w(e) < r\}$.
Es genügt zu zeigen:
Sei $r \in \mathbb{R}$, sei $T_1, \cdots, T_k$ ein minimaler Spannwald (d.h. Spannwald minimalen Gewichtes) für $G_r := \{V, E_r\}$ (d.h.betrachte nur Kanten mit Gewicht $> r$). Sei T ein MSB von G, dann:

Hilfsbehauptung: Die Knotenmenge eines jeden T_i im T induziert einen zusammenhängenden Teilbaum.

Beweis der Hilfsbehauptung: Sei $T_i =: (V_i, E_i)$ und sei $e \in V_i \times V_i$. Falls $e \in T_i$, dann ist nichts zu zeigen. Für den Fall $e \notin T_i$ betrachte den Fundamentalkreis C, den e in T_i erzeugt. Gewicht aller Kanten auf $C - \{e\} < r$. Falls die beiden Endpunkte von e in T durch einen Pfad verbunden sind, der eine Kante $e' \in E - E_r$ enthält, dann liefert die "rote" Regel, daß T nicht minimal ist.
$\Rightarrow$ Die beiden Endpunkte und e sind in T durch einem Pfad aus lauter Kanten mit Gewicht $< r$ verbunden. $\diamond$

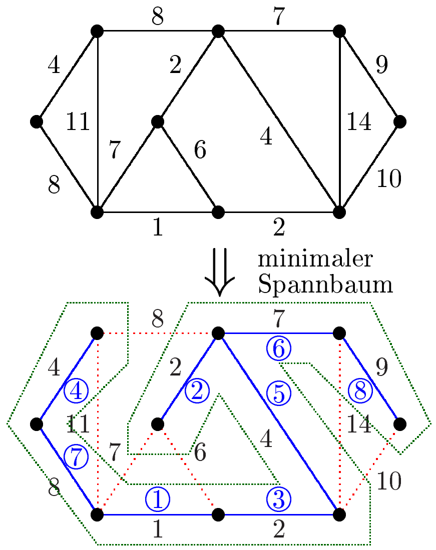
Zeitkomplexität: (mit $n = |V|, m = |E|$)

Sortieren	$m \log m = \mathcal{O}(m \log n)$
$\mathcal{O}(m)$ *Find*-Operationen	$\mathcal{O}(m \log n)$
$n - 1$ *Unions*	$\mathcal{O}(n)$

Satz: Kruskal's MSB-Algorithmus hat die Zeitkomplexität von $\mathcal{O}((m + n) \log n)$

Beweis: s.o. $\diamond$

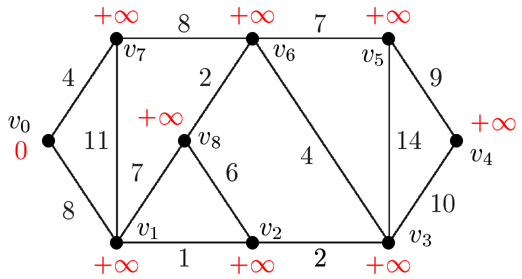
Beispiel für Kruskals Algorithmus:



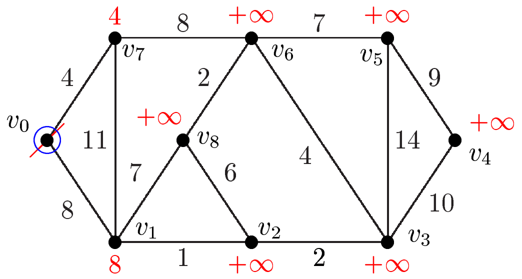
3.4 Prim's Algorithmus, erste Variante

```
algorithm PRIM-MSB (G,w) :=
⌜
    initialisiere Priority Queue PQ mit Knotenmenge V und Schlüssel +∞, ∀v ∈ V;
    wähle Knoten r als Wurzel (beliebig);
    Schlüssel[r] := 0;
    Vorgänger[r] := nil;
    while PQ ≠ ∅ do
        u := FindMin(PQ);
        DeleteMin(PQ);
        for alle Knoten v, die in G zu u benachbart sind do
            if v ∈ PQ and w({u,v}) < Schlüssel[v] then
                Vorgänger[v] := u;
                Schlüssel[v] := w({u,v});
            fi
        od
    od
⌞
```

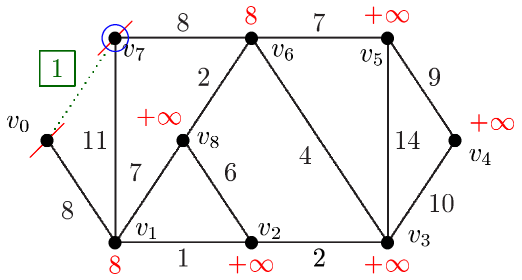
Beispiel für Prim's Algorithmus (erste Variante):



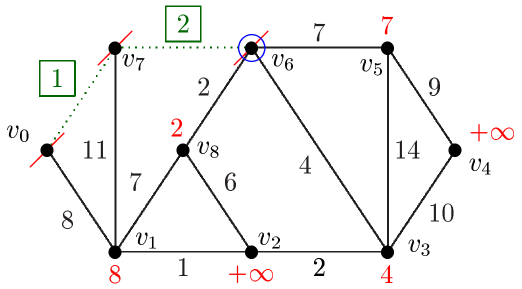
Ausgangszustand:
alle Schlüssel $= +\infty$
aktueller Knoten ($= u$): ⦿
Startknoten: r ($= v_0$)



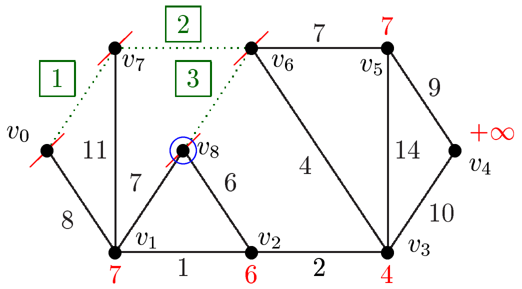
suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w($\{u,v\}$) < Schlüssel[v]: ($v_1 = 8, v_7 = 4$)



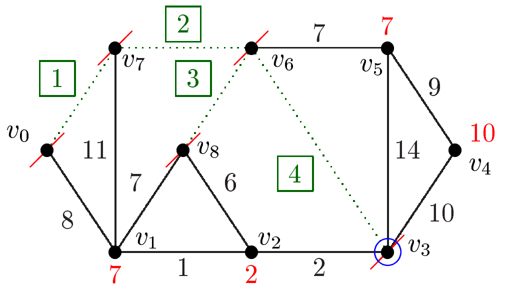
suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w($\{u,v\}$) < Schlüssel[v]: ($v_6 = 8$)



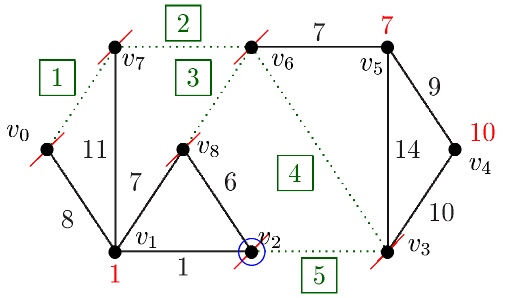
suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w($\{u,v\}$) < Schlüssel[v]: $(v_3 = 4, v_5 = 7, v_8 = 2)$



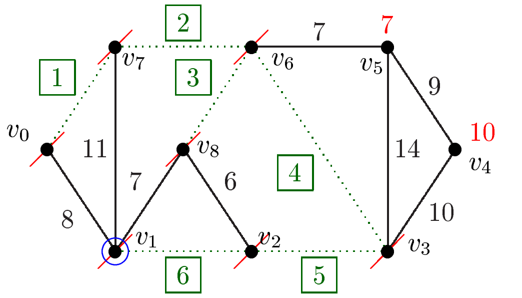
suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w($\{u,v\}$) < Schlüssel[v]: $(v_1 = 7, v_2 = 6)$



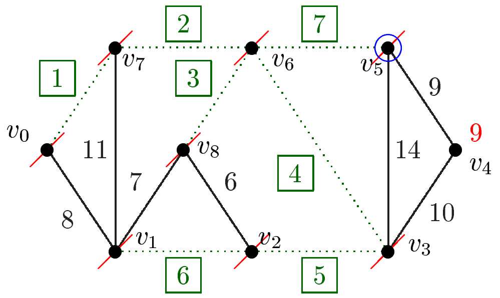
suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w($\{u,v\}$) < Schlüssel[v]: $(v_2 = 2, v_4 = 10)$



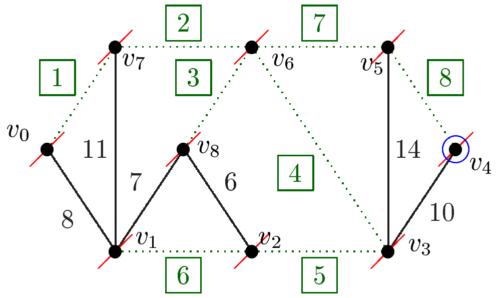
suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w($\{u,v\}$) < Schlüssel[v]: $(v_1 = 1)$



suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w($\{u,v\}$) < Schlüssel[v]: solche Nachbarn existieren nicht



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:

- $n - 1$ *DeleteMin*
- $\mathcal{O}(m)$ sonstige Operationen inclusive *DecreaseKey*

Implementierung der Priority Queue mittels Fibonacci-Heaps:

Initialisierung	$\mathcal{O}(n)$
Find + DeleteMins	$\mathcal{O}(m)$ $(\leq m$ Stück$)$
DecreaseKeys	$\mathcal{O}(m)$ $(\leq m$ Stück$)$
Sonstige Overhead	$\mathcal{O}(1)$

$\Rightarrow$ Laufzeit:

$$\boxed{\rightsquigarrow \mathcal{O}(n) + \mathcal{O}(n \log(n)) + m \cdot \mathcal{O}(1) + m \cdot \mathcal{O}(1) = \mathcal{O}(m + n \log n)}$$

Satz: 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 $\mathcal{O}(m + n \log n)$ (hier: $n = |V|$, $m = |E|$). Dies ist für $m = \Omega(n \log n)$ asymptotisch optimal.

3.5 Prim's Algorithmus, zweite Variante

Idee: Lasse die Priority Queues nicht zu groß werden.
Sei $G = (V, E)$, $|V| = n$, $|E| = m$, w Gewichtsfunktion, k Parameter, der unten festgelegt wird.
Der Algorithmus wird in den folgenden Phasen abgearbeitet:

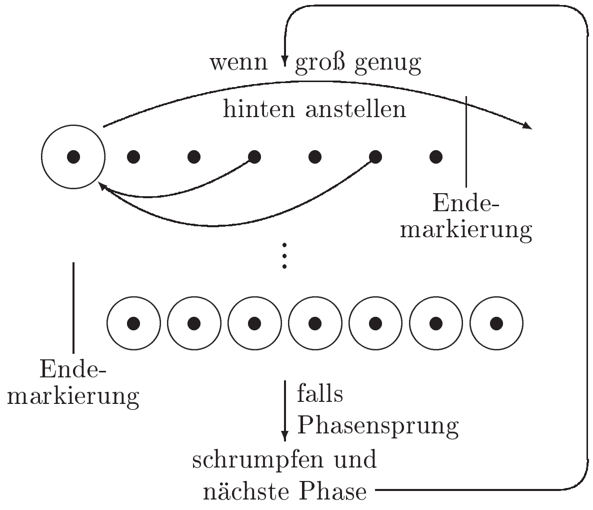
1. Initialisiere eine Schlange von Bäumen, jeder Baum anfangs (Super-) Knoten. Zu jedem Baum: Priority Queue (Fibonacci-Heap) mit den Nachbarn der Knoten im Baum, die selbst nicht im Baum sind. Schlüssel: Gewicht einer leichtesten Kante zu einem Knoten im Baum.

2. Markiere jeden Baum in der Schlange mit der Nummer der laufenden Phase

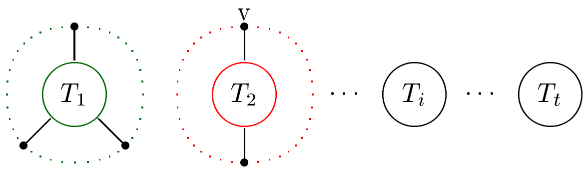
3. <u>while</u> vorderster Baum in der Schlange hat laufende Phasennummer <u>do</u>
   lasse ihn wachsen, solange seine Priority Queue weniger als
   k Elemente enthält (und noch was zu tun ist)
   <u>if</u> Priority Queue zu groß <u>then</u>
   füge Baum mit Phasennummer + 1 am Ende der Schlange ein
   <u>fi</u>
   <u>od</u>

4. Falls „Phasensprung“: Schrumpfe alle Bäume zu Superknoten, reduziere Kantenmenge (Behalte zwischen zwei Knoten, jeweils nur die leichteste Kante).

5. nächste Phase!



Betrachte Schlange von Bäumen:



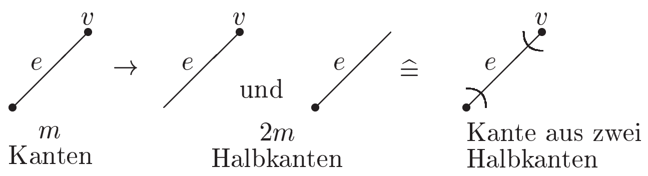
(Arbeit pro Phase ∼ # Knoten, außer Vereinigung zweier Superknoten T_1 und T_2.)
Für jeden (Super-)Knoten v in T_2's Priority Queue:

1. $v \in T_1$: $\surd$ (nichts zu tun)

2. v in Priority Queue von T_1, einfügen bzw. conditional *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 den Bäumen vor. Mit m Kanten ergeben sich $2m$ Halbkanten, d.h. Anzahl proportional zu m.



Zeitaufwand pro Phase: (+ Superknoten zu Beginn)

- Initialisierung: $\mathcal{O}(m)$
- *Delete* und *DeleteMin*: $\leq t$
- sonstige Priority Queue-Operationen, Overhead: $\mathcal{O}(m)$

Da die Priority Queues höchstens k Elemente enthalten, wenn darauf eine Priority Queue-Operation durchgeführt wird, sind die Kosten pro Phase

$$\mathcal{O}(t \log k + m)$$

Setze $k = 2^{\frac{2m}{t}}$ (damit $t \log k = m$). Damit sind die Kosten pro Phase $\mathcal{O}(m)$.

Wieviele Phasen? Sei t die Anzahl der Superknoten am Anfang einer Phase, t' diese Zahl zu Beginn der nächsten Phase. Sei a die durchschnittliche Anzahl ursprünglicher Knoten in den 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 $a = \frac{2m}{n}$, a ist für jede Phase $\leq n - 1$. 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: Für gewichtete, zusammenhängende 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.

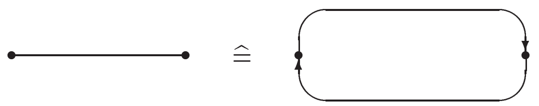
Euklidische minimale Spannbäume stellen ein Problem dar, für das es speziellere Algorithmen gibt. Literatur hierzu:
A.C.-C. Yao: Constructing Minimum Spanning Trees in k-dimensional spaces and related problems. SIAM Journal on Computing 11(4) p. 721–736 (1982).

Kapitel 4

Kürzeste Pfade

4.1 Grundlegende Begriffe

Betrachte Digraph $G = (V, A)$ oder Graph $G = (V, E)$.



Kante von ungerichtetem Graph — Entgegengesetzte (antiparallele) Kanten

Distanzfunktion:

$$\mathrm{d} : A \longrightarrow \mathbb{R}^+ \text{ (bzw. } \longrightarrow \mathbb{R})$$

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.

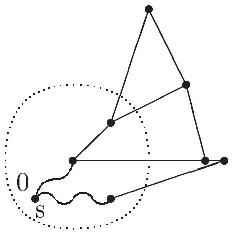
Bemerkung: Es gibt keinen Algorithmus, der das single-pair-shortest-path berechnet, ohne nicht gleichzeitig das single-source-shortest-path-Problem zu lösen.

4.2 Das single-source-shortest-path-Problem

Zunächst nehmen wir an, daß $d \geq 0$ ist. Alle kürzesten Pfade von a nach b sind o.B.d.A. einfache Pfade.

4.2.1 Dijkstra's Algorithmus

Gegeben: $G = (V, A), (A = V \times V)$, Distanzfunktion $d : A \to \mathbb{R}^+ \cup \{+\infty\}$. Startknoten s. G durch Adjazenzlisten dargestellt.




```
algorithm sssp:=
⌜
S := {s}; dis[s] := 0; initialisiere eine Priorityqueue PQ, die
alle Knoten v ∈ V \ {s} enthält mit Schlüssel dis[v] := d(s,v);
for alle v ∈ V − {s} do
    from[v] := s
od
while S ≠ V do
    v := ExtractMin(PQ);
    S := S ∪ {v};
    for alle w ∈ V − S, d(v,w) < ∞ do
        if dis[v] + d(v,w) < dis[w] then
            DecreaseKey(w, dis[v] + d(v,w));
                co DecreaseKey verändert dis[w] oc
            from[w] := v;
        fi
    od
od
⌞
```

Seien $n = |V|$ und $m :=$ Anzahl wirklicher Kanten:

Laufzeit (mit Fibonacci-Heaps):

Initialisierung:	$\mathcal{O}(n)$
ExtractMin:	$n \cdot \mathcal{O}(\log n)$
Sonstiger Aufwand:	$m - \mathcal{O}(1)$ (z.B. *DecreaseKey*)

$\Rightarrow$ Zeitbedarf also: $\boxed{\mathcal{O}(m + n \log n)}$

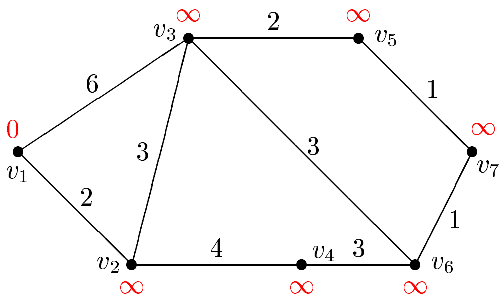
Korrektheit: Wir behaupten, daß in dem Moment, in dem ein $v \in V - \{s\}$ Ergebnis der *FindMin* Operation ist, der Wert dis$[v]$ des Schlüssels von v gleich der Länge eines kürzesten Pfades von s nach v ist.

Beweis: durch Widerspruch: Sei $v \in V - \{s\}$ der erste Knoten, für den diese Behauptung nicht stimmt und sei

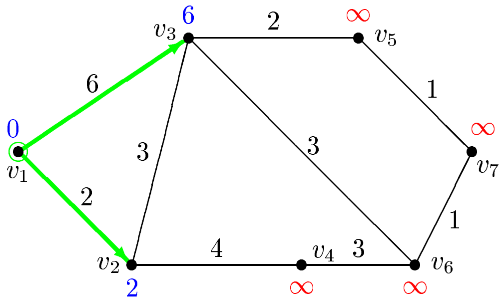
$$s \rightarrow s_1 \rightarrow s_2 \cdots \rightarrow s_r \rightarrow v_1 \rightarrow v_2 \cdots \rightarrow v_q \rightarrow v$$

ein kürzester Pfad von s nach v, mit einer Länge $<$ dis$[v]$. Dabei sind $s_1, \ldots, s_r \in S$, $v_1 \notin S$ [$r = 0$ und/oder $q = 0$ ist möglich]. Betrachte Pfad $s \rightarrow s_1 \cdots \rightarrow s_r \rightarrow v_1$; seine Länge ist $<$ dis$[v]$, für $q \geq 1$ (ebenso für $q = 0$) ist also dis$[v_1] <$ dis$[v]$, Widerspruch zur Wahl von v. $\diamond$

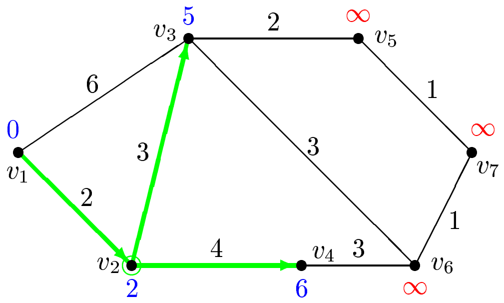
Beispiel für Dijkstra-Algorithmus:



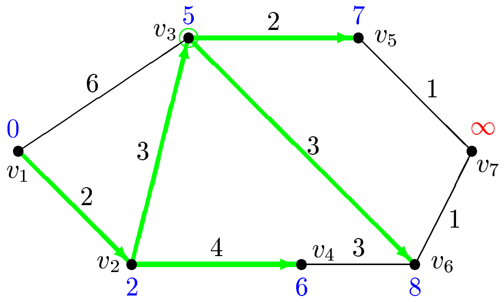
gegeben Graph G;
v_1 ist der Startknoten;
setze v_1 als Bezugsknoten;
setze dis[v_1]=0;
setze dis[Rest]=$+\infty$;



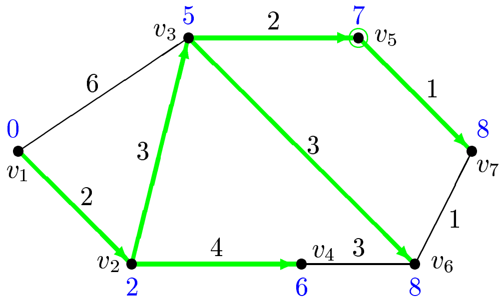
markiere (v_1, v_2);
setze dis[v_2]=2;
markiere (v_1, v_3);
setze dis[v_3]=6;
setze v_2 als Bezugsknoten, da
$d(v_1, v_2) < d(v_1, v_3)$;



markiere (v_2, v_3);
unmarkiere (v_1, v_3), da
$d(v_1, v_2) + d(v_2, v_3) < d(v_1, v_3)$;
setze dis[v_3]=2+3=5;
markiere (v_2, v_4);
setze dis[v_4]=2+4=6;
setze v_3 als Bezugsknoten, da
$d(v_2, v_3) < d(v_2, v_4)$;



markiere (v_3, v_5);
setze dis[v_5]=2+3+5=7;
markiere (v_3, v_6);
setze dis[v_6]=2+3+3=8;
setze v_5 als Bezugsknoten, da
$d(v_3, v_5) < d(v_3, v_6)$;



markiere (v_5, v_7);
setze dis[v_7]=2+3+2+1=8;
alle Knoten wurden erreicht:
$\Rightarrow$ Algorithmus zu Ende

Beobachtung:

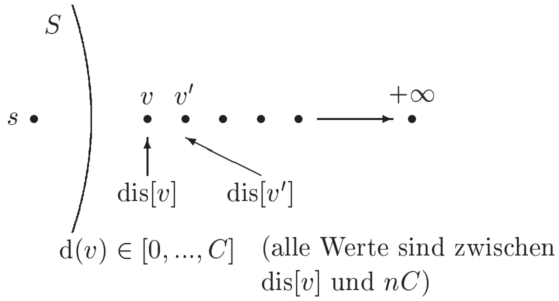
- *ExtractMin* liefert eine monoton steigende Folge von Schlüsseln dis[ ];
- Die Schlüssel in PQ sind stets $\leq$ dis$[v]+C$, wobei v das Ergebnis der vorangehenden *ExtractMin*-Operation (bzw. s zu Beginn) und $C := \max_{u,v \in A}\{d(u,v)\}$ ist.

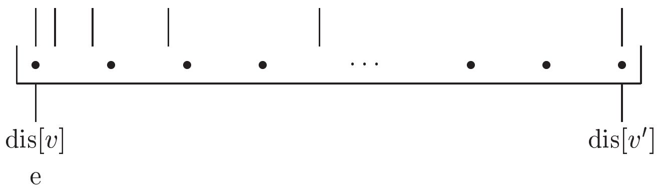
Satz: Dijkstra's Algorithmus (mit Fibonacci-Heaps) löst das Single-Source-Shortest-Path-Problem in Zeit $\mathcal{O}(m + n\log n)$.

Bemerkung:

1. Verwendet man Dijkstra's Algorithmus mit d-Heaps, so erhält man Laufzeit $\boxed{\mathcal{O}(m \log_{2+\frac{m}{n}} n)}$
2. Mikkel Thorup (STOC '98): $\boxed{\mathcal{O}(m)}$

4.2.2 Dikstra's Algorithmus mit Radix-Heaps





Einstufige Radix-Heaps:

Menge von $B = \lceil \log_2(C+1) \rceil + 2$ Buckets (jeder Bucket als doppelt verkettete Liste implementiert). Jeder Bucket hat Größe size, mit

$$\text{size}(i) = \begin{cases} 1 & \text{für } i = 1 \\ 2^{i-2} & \text{für } i = 2, \ldots, B-1 \\ nC+1 & \text{für } i = B \end{cases}$$

Beobachtung:

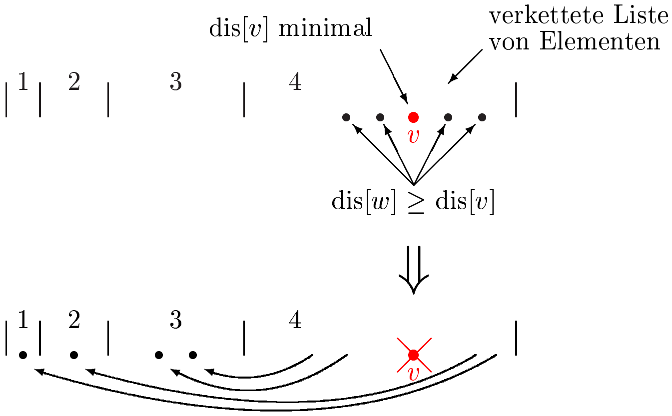
$$\sum_{i=1}^{j-1} \text{size}(i) \geq \min\{\text{size}(j), C+1\} \text{ für } j = 2, \ldots, B$$

Wir ordnen den Buckets Intervalle zu, indem wir für jeden Bucket i eine obere Intervallgrenze u(i) angeben. Das Intervall für Bucket i ist dann $[\text{u}(i-1)+1, \text{u}(i)]$.

Setze zu Beginn $\mathrm{u}(0) := -1$. Die $\mathrm{u}(i)$ ändern sich im Algorithmus, die $\mathrm{size}(i)$ bleiben konstant. Zu Beginn:

$$\left.\begin{array}{ll} \mathrm{u}(i) & := 2^{i-1} - 1 \text{ für } i = 1, \dots, B-1 \\ \mathrm{u}(B) & := nC + 1 \end{array}\right\rangle \begin{array}{l} \text{Größe des dem Bucket i} \\ \text{zugeordneten Intervalls:} \\ u(i) - u(i-1) \leq \mathrm{size}(i) \end{array}$$

- Initialisierung: Einfügen aller Schlüssel in die Buckets: Durchlaufe, für Schlüssel k, die Buckets B, $B-1$, ... bis wir Bucket j finden mit $\mathrm{u}(j) < k$. Füge k in Bucket $j+1$ ein. Kosten: $n \cdot \log C$.
- *DecreaseKey*: Wie bei Insert, beginne Suche mit momentanem Bucket des Schlüssels. Kosten für alle *DecreaseKey* Operationen: $\mathcal{O}(1)$ pro *DecreaseKey* $+\mathcal{O}(n \log C)$.
- *DeleteMin*: Suche von links den ersten nicht leeren Bucket j. Falls $j = 1$ ist alles ok (wegen size(1)=1). Falls $j \geq 2$: Suche minimales Element v im Bucket j (v wird als *FindMin* zurückgegeben und aus der Priority Queue gelöscht), verteile alle anderen Elemente im Bucket j auf die Buckets $1, \dots, j-1$, nachdem die $\mathrm{u}(i)$'s, wie folgt, aktualisiert wurden:
  $\mathrm{u}(0) := \mathrm{dis}[v] - 1$
  $\mathrm{u}(1) := \mathrm{dis}[v]$
  $\mathrm{u}(i) := \min\{\mathrm{u}(i-1) + \mathrm{size}(i), \mathrm{u}(j)\}$ für $i = 2, \dots, j-1$



Kosten für alle *DeleteMins*: $\mathcal{O}(\log C)$ pro *DeleteMin* $+ \ \mathcal{O}(n \log C)$.

Gesamtkosten für Dijkstra's Algorithmus mit einstufigen Radix-Heaps damit:

$$\mathcal{O}(m + m + n \log C + n \log C + n \log C) = \mathcal{O}(m + n \log C)$$

Satz: Dijkstra's Algorithmus mit einfstufigen Radix-Heaps hat Zeitkomplexität $\mathcal{O}(m + n \log C)$.

Verbesserungen:

1. zweistufige Heaps: $\boxed{\mathcal{O}(m + n \cdot \frac{\log C}{\log \log C})}$
2. mehrstufige Heaps (+Fibonacci-Heaps): $\boxed{\mathcal{O}(m + n\sqrt{\log C})}$

Literatur: Akuja, Mehlhorn, Orlin, Tarjan; Faster Algorithms for the Shortest Path Problem. J ACM 37(1990), S. 213-223).

4.2.3 Bellman-Ford-Algorithmus

Dieser Algorithmus ist ein Beispiel für dynamische Programmierung.

Sei $B_k[i]$:= Länge eines kürzesten Pfades von s zu Knoten i, der höchstens k Kanten enthält.
Gesucht ist $B_{n-1}[i]$; für $i = 1, \cdots, n$, Menge der Knoten $V = \{1, \cdots, n\}$

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:

$\underline{\texttt{for}}$ $k = 2$ $\underline{\texttt{to}}$ $n-1$ $\underline{\texttt{do}}$

  $\underline{\texttt{for}}$ $i = 1$ $\underline{\texttt{to}}$ n $\underline{\texttt{do}}$

$$B_k[i] := \begin{cases} 0 & \texttt{, falls } i = s; \\ \min_{j \in N(i)} \{B_{k-1}[i], B_{k-1}[j] + \mathtt{d}(j,i)\} & \texttt{, sonst} \end{cases}$$

  $\underline{\texttt{od}}$

$\underline{\texttt{od}}$

Korrekheit: klar (Beweis durch vollständige Induktion)

Zeitbedarf: Wenn die innere Laufschleife durch eine Schleife über alle Kanten ersetzt wird:

$\boxed{\text{Zeitbedarf: } \mathcal{O}(n \cdot m)}$

4.3 Floyd's Algorithmus für das All-pairs-shortest-path-Problem

Auch genannt „Kleene's Algorithmus“. Dieser Algorithmus ist ein weiteres Beispiel für dynamische Programmierung.

Sei $G = (V, E)$ mit Distanzfunktion d:$A \to \mathbb{R}_0^+$ gegeben. Sei o.B.d.A. $V = \{v_1, \ldots, v_n\}$.
Sei c_{ij}^k := Länge eines kürzesten Pfades von v_i nach v_j, der als innerer Knoten (alle bis auf ersten und letzten Knoten) nur Knoten aus $\{v_i, \cdots, v_k\}$ enthält.

$\underline{\texttt{algorithm}}$ `floyd:=`

⌜

$\underline{\texttt{co}}$ $\forall i : d_{ii} = 0$ $\underline{\texttt{od}}$

$\underline{\texttt{for}}$ `alle` (i,j) $\underline{\texttt{do}}$ $c_{ij}^{(0)} := \mathtt{d}_{ij}$ $\underline{\texttt{od}}$; $1 \leq i,j \leq n$

$\underline{\texttt{for}}$ $k := 1$ $\underline{\texttt{to}}$ n $\underline{\texttt{do}}$

  $\underline{\texttt{for}}$ `alle` (i,j), $1 \leq i,j \leq n$ $\underline{\texttt{do}}$

$$c_{ij}^{(k)} := \min \left\{ c_{ij}^{(k-1)},\ c_{ik}^{(k-1)} + c_{kj}^{(k-1)} \right\}$$

  $\underline{\texttt{od}}$

$\underline{\texttt{od}}$

⌞

$\boxed{\textbf{Laufzeit: } \mathcal{O}(n^3)}$

Korrektheit:
Zu zeigen: $c_{ij}^{(k)}$ des Algorithmus $= c_{ij}^{k}$ (damit sind für $k = n$ die Kosten der kürzesten Pfade durch $c_{ij}^{(k)}$ gegeben).

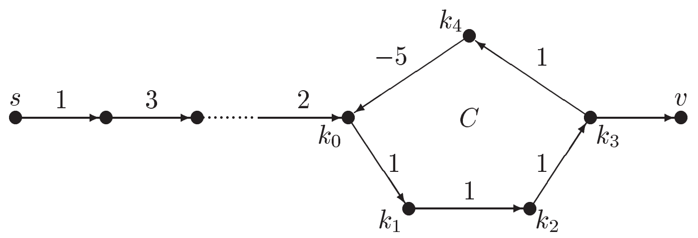
Beweis: Richtig für $k = 0$. Induktionsschluß: Ein kürzester Pfad von v_i nach v_j mit inneren Knoten $\in \{v_1, \dots, v_{k+1}\}$ enthält entweder v_{k+1} gar nicht als inneren Knoten, oder er enthält v_{k+1} genau einmal als inneren Knoten. Im ersten Fall wurde dieser Pfad also bereits für $c_{ij}^{(k)}$ betrachtet, hat also Länge $\geq c_{ij}^{(k)}$. Im zweiten Fall setzt er sich aus einem kürzesten Pfad P_1 von v_i nach v_{k+1} und einem kürzesten Pfad von P_2 von v_{k+1} nach v_j zusammen, wobei alle inneren Knoten von P_1 und $P_2 \in \{v_1, \dots, v_k\}$ sind. Also ist die Länge des Pfades $= c_{i,k+1}^{(k)} + c_{k+1,j}^{(k)}$. ◇

Satz: Floyd's Algorithmus für das All-pair-shortest-path-Problem hat Zeitkomplexität $\mathcal{O}(n^3)$.

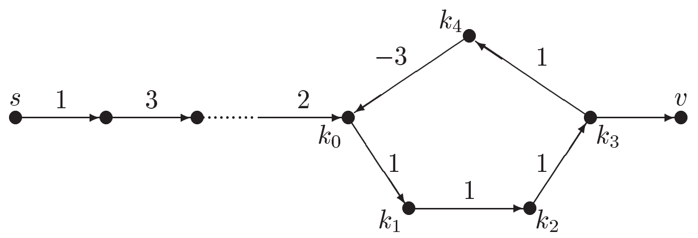
4.4 Digraphen mit negativen Kantengewichten

4.4.1 Grundsätzliches

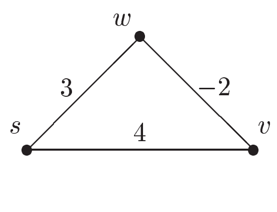
Betrachte Startknoten s und einen Kreis C mit Gesamtlänge < 0 und C ist von s aus erreichbar.



Sollte ein Pfad von s nach C und von C nach v existieren, so ist der kürzeste Pfad nicht definiert. Falls aber die Gesamtlänge des Kreises $C \geq 0$ ist,



dann ist der kürzeste Pfad wohl definiert. Probleme gibt es also nur dann, wenn es einen Zyklus negativer Länge gibt. Nichtsdestotrotz funktioniert Dijkstra-Algorithmus 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 fälchlicherweise als 3 (statt 2).

4.4.2 Modifikation von Bellman-Ford-Algorithmus

$B_k[i]$ gibt nun die Länge eines kürzesten gerichteten s-i-Pfades an, der aus höchstens k Kanten besteht. Jeder beliebige Kantenzug, 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 dagegen einen (von s aus erreichbaren) negativen Kreis, so gibt es einen Knoten $i \in V$, bei dem ein Kantenzug aus n Kanten mit der Länge $B_n[i]$ diesen Kreis häufiger durchläuft als jeder Kantenzug aus maximal $n-1$ Kanten der Länge $B_{n-1}[i]$. Demnach gilt:

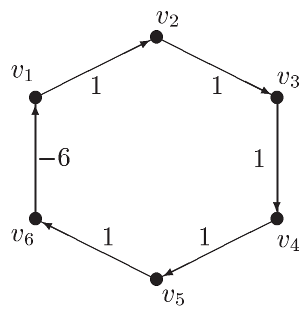
$$B_n[i] < B_{n-1}[i]$$

Man kann also in den Algorithmus aus Bellman-Ford einen Test auf negative Kreise einbauen, indem man in der Rekursion auch $\forall i \in V : B_n[i]$ berechnet und direkt vor der Ausgabe den folgenden Befehl einfügt:

<u>for</u> i 1 to n <u>do</u>
    <u>if</u> $B_n[i] < B_{n-1}[i]$ <u>then</u> stop.

4.4.3 Modifikation von Floyd's-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, g.d.w. ein $c_{ii}^n < 0$ existiert.

Man kann also in den Algorithmus von Floyd einen Test auf negative Kreise einbauen, indem man direkt vor der Ausgabe den folgenden Befehl einfügt:

<u>for</u> i 1 to n <u>do</u>
    <u>if</u> $c_{ii}^n < 0$ <u>then</u> stop.

4.4.4 Der Algorithmus von Johnson

Definition: Sei $d : A \rightarrow \mathbb{R}$ eine Distanzfunktion. Eine Abbildung $r : V \rightarrow \mathbb{R}$ heißt ***Rekalibrierung***, falls gilt:

$$\forall (u, v) \in A : r(u) + d(u, v) \geq r(v)$$

Beobachtung: Sei r eine Rekalibrierung (für d). Setze $d'(u,v) := d(u,v) + r(u) - r(v)$. Dann gilt:

$$d'(u,v) \geq 0$$

Sei $u = v_0 \to \cdots \to v_1 = v$ ein Pfad. Dann ist:

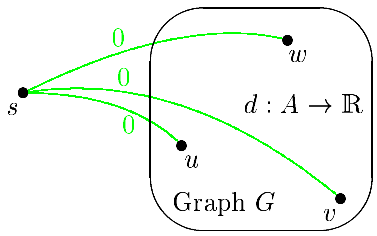
$$d\text{-Länge} = \sum_{i=1}^{r} d(v_i, v_{i-1})$$

Demnach ist:

$$\begin{aligned} d'\text{-Länge } &= \sum_{i=1}^{r} d'(v_i, v_{i-1}) \\ &= \sum_{i=1}^{r} (d(v_i, v_{i-1}) + r(v_i) - r(v_{i-1})) = \\ &= \sum_{i=1}^{r} d(v_i, v_{i-1}) + r(v_r) + r(v_0) \end{aligned}$$

Also ist ein d-kürzester Pfad von $u(v_0)$ nach $v(v_r)$ auch ein d'-kürzester Pfad und umgekehrt. Man kann jetzt mit Hilfe von d' beliebige Algorithmen (Dijkstra, Floyd, etc.) anwenden.

Berechnung einer Rekalibrierung: a-Algorithmus.



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 Floyd). Sei $r(v)$ die dadurch berechnete Entfernung von s zu $v \in V$. Dann r Rekalibrierung, denn es gilt:

$$r(u) + d(u,v) \geq r(v)$$

4.5 Zusammenfassung

	$d \geq 0$	d allgemein
ASSP	Dijkstra (Fibonacci): $\mathcal{O}(m + n \cdot \log n)$ Dijkstra (Radix) $\mathcal{O}(m + n\sqrt{\log C})$	Bellman-Ford: $\mathcal{O}(n \cdot m)$
apsp	Dijkstra: $\mathcal{O}(n \cdot m + n^2 \min\{\log n, \sqrt{\log C}\})$ Floyd: $\mathcal{O}(n^3)^{(*)}$	Johnson: $\mathcal{O}(n \cdot m + n^2 \log n)$ Floyd: $\mathcal{O}(n^3)$

Bemerkung$^{(*)}$: In der Praxis ist der Floyd-Algorithmus bei kleinen n besser als Dijkstra-Algorithmus.

4.6 Transitive Hülle

4.6.1 Min-Plus-Matrix-Produkt und Min-Plus-Transitive Hülle

Ring $\mathbb{R}(+,\bullet)$ — $\nearrow$ Gruppe, $\nwarrow$ Halbgruppe

Semiring $\mathbb{N}(+,\bullet)$ — $\nearrow$ Halbgruppe, $\nwarrow$ Halbgruppe

Semiring über $\mathbb{R} \cup \{\infty\}$ mit Operationen min und $+$, jeweils Halbgruppe.
Distributivgesetz $a + \min\{b,c\} = \min\{a+b, a+c\}$

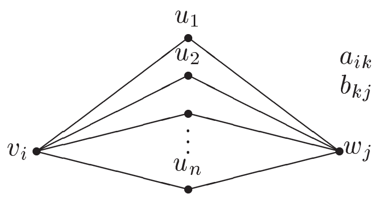
Normale Matrixmultiplikation:

$$A = (a_{ij})_{1\le i,j\le n}, \quad B = (b_{ij})_{1\le i,j\le n}, \quad I = (\delta_{ij})_{1\le i,j\le n}$$

$$C = A \cdot B = (c_{ij})_{1\le i,j\le n}, \quad c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

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: $A = B$
kürzeste Wege von v_i nach w_j in einem Graph ($A = B$)

$$I_{\min,+} = \begin{pmatrix} 0 & & \infty \\ & \ddots & \\ \infty & & 0 \end{pmatrix}$$

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 (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 <u>zwei</u> Kanten enthält. Induktion: $a_{ij}^{(k)}$ ist die Länge des kürzesten Pfades von v_i nach v_j mit höchstens $\underline{k}$ Kanten.
Falls die $\mathrm{d}(v_i, v_j)$ alle ≥ 0, gibt es immer kürzeste Pfade, die $\le n-1$ Kanten enthalten.

Alternative Lösung des All-Pairs-Shortest-Path-Problems: Berechne A^{n-1} (Min-Plus). Es genügt auch $A^{2^{\lceil \log(n-1) \rceil}}$ (nicht $A^2, A^3, A^4, \ldots$ berechnen, sondern wiederholt quadrieren).

$A^* := \min_{i\ge 0}\{A^i\}$ heißt Min-Plus-Transitive Hülle.

Bemerkung: Min wird Komponentenweise gebildet. Wenn $d \ge 0$, dann $A^* = A^{n-1}$.

4.6.2 Boolesche Matrixmuliplikation und Transitive Hülle

Ersetze in 4.6.1. Distanzmatrix durch (Boolesche) Adjazenzmatrix und ersetze Min-Plus durch $\vee, \wedge$, d.h.:

$$C = A \cdot B; \quad c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$$

Setze $a_{ii} = 1, 1 \leq i \leq n$
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

$$A^* := \bigvee_{i \geq 0} A^i$$

ist damit die Adjazenzmatrix der transitiven Hülle des zugrundeliegenden Graphen $(= A^{n-1})$.

Satz: Sei $\mathrm{M}(n)$ die Zeitkomplexität für das Boolesche Produkt zweier $n \times n$ - Matrizen, $\mathrm{T}(n)$ die Zeitkomplexität für die transitive Hülle einer $n \times n$ Booleschen Matrix.
Falls $\underbrace{\mathrm{T}(3n) \leq c\mathrm{T}(n)}_{\substack{\text{sicher erfüllt, falls} \\ \text{T polynomiell}}}$ und $\underbrace{\mathrm{M}(2n) \geq 4\mathrm{M}(n)}_{\substack{\text{sicher erfüllt, falls} \\ \mathrm{M}(n) \geq n^2}}$, dann gilt:

$$\mathrm{T}(n) = \Theta(\mathrm{M}(n))$$

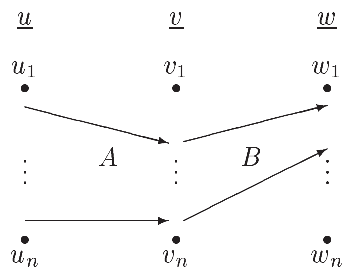
Beweis:

- Matrixmultiplikation $\prec$[1] transitive Hülle:
  Gegeben: A, B; Gesucht: $C = A \cdot B$ (Boolesches Produkt); Setze:

$$L = \underbrace{\begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}}_{3n} \left.\vphantom{\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}}\right\} 3n$$

  L ist Adjazenzmatrix eines tripartiten, gerichteten Graphen, denn:

$$\begin{array}{c|ccc} & \underline{u} & \underline{v} & \underline{w} \\ \hline \underline{u} & 0 & A & 0 \\ \underline{v} & 0 & 0 & B \\ \underline{w} & 0 & 0 & 0 \end{array}$$



[1] $x \prec y$ bedeutet „reduziere x auf y"

Daher kann L^* leicht bestimmt werden:

$$L^* = \begin{pmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

Also gilt: $\mathrm{M}(n) \leq \mathrm{T}(3n) = \mathcal{O}(\mathrm{T}(n))$.

- Transitive Hülle $\prec$ Matrixmultiplikation
  Gegeben: L $n \times n$ Boolesche Matrix; gesucht: L^*; Annahme: n ist Zweierpotenz.
  Teile auf:

$$L = \underbrace{\begin{pmatrix} A \\ C \end{pmatrix}}_{\frac{n}{2}}\!\!\underbrace{\begin{pmatrix} B \\ D \end{pmatrix}}_{\frac{n}{2}} \begin{matrix} \}\frac{n}{2} \\ \}\frac{n}{2} \end{matrix} \; ; \quad L^* = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

Dann gilt:

$E =$	$(A \vee BD^*C)^*$	betrachte alle Pfade von der ersten Hälfte der Knoten zur ersten Hälfte der Knoten
$F =$	EBD^*	analog
$G =$	D^*CE	analog
$H =$	$D^* \vee GF$	analog

Um L^* zu berechnen, benötigen wir zwei Transitive-Hülle-Berechnungen und sechs Matrixprodukte für Matrizen der Dimension $\frac{n}{2} \times \frac{n}{2}$, plus Aufwand für $\vee$, der $\leq c'n^2$ ist. Also:

$$\begin{array}{lll|ll}
\mathrm{T}(n) & \leq 2\mathrm{T}(\frac{n}{2}) + 6\mathrm{M}(\frac{n}{2}) + c'n^2 & & \text{Behauptung:} & \mathrm{T}(n) \leq c\mathrm{M}(n) \\
& & & \text{Beweis:} & \text{durch Induktion:} \\
& & & & n = 1\surd \\
& \leq 2c\mathrm{M}(\frac{n}{2}) + 6\mathrm{M}(\frac{n}{2}) + c'n^2 & & \text{Vorauss.:} & \mathrm{M}(2n) \geq 4\mathrm{M}(n) \\
& & & & \mathrm{M}(n) \geq n^2 \\
& \leq \frac{1}{4}(2c + 6 + 4c')\mathrm{M}(n) & & & \\
& \leq c\mathrm{M}(n) & & &
\end{array}$$

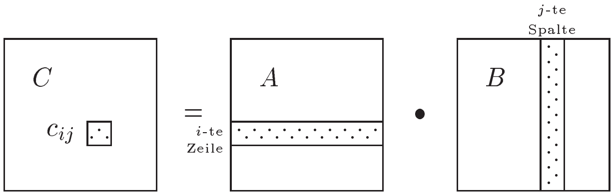
falls $c \geq \frac{1}{4}(2c + 6 + 4c')$, also falls $c \geq 3 + 2c'$.
Also $\mathrm{T}(n) = \mathcal{O}(\mathrm{M}(n))$.

$\diamond$

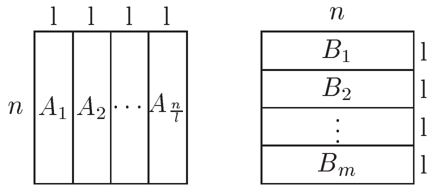
4.6.3 Der 4-Russen-Algorithmus für Boolesche Matrixmultiplikation

Autoren:
V.L. Avlazarov, E.A. Dinic, M.A. Kronod, I.A. Faradzev : “On economical construction of the transitive closure of a directed graph”, Soviet Math. Doklady 11 (1970), p. 1209–1210.

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 ist Teiler von n).



Teile A auf (setze $m := \frac{n}{l}$):



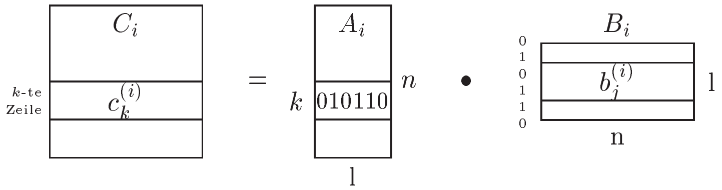
Sei $A = A'_1 \vee A'_2 \vee \cdots \vee A'_m, C_i := A_i \cdot B_i$ für $i = 1, \ldots, m$. Dann gilt

$$C = \bigvee_{i=1}^{m} C_i, \text{ da}$$

$$C = AB = \left(\bigvee_{i=1}^{m} A'_i\right)\left(\bigvee_{i=1}^{m} B'_i\right) = \bigvee_{\substack{1 \le i \le m \\ 1 \le j \le m}} A'_i B'_j = \bigvee_{i=1}^{m} A_i B_i,$$

da $A_i B_j = 0$ für $i \neq j$ (fasse A_i und B_j als $n \times n$ Matrizen auf 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 jedes mögliche $a_k^{(i)}$. 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,...,0];
for j := 1 to 2^⌊log n⌋ − 1 do
    p := ⌊log j⌋;   co p Index der vordersten 1 von j oc
    comb[j] := comb[j − 2^p] ∨ b_{(i−1)⌊log n⌋+p+1}
od
⌞
```

Zeitbedarf:

(a) sequentiell: $\boxed{\mathcal{O}(n^2)}$

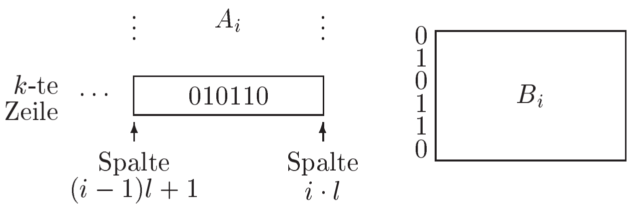
(b) Vektoroperationen der Breite n: $\boxed{\mathcal{O}(n)}$

```
algorithm four-russians(array a, b, c) =
⌜
co die Matrizen a, b, c sind als Vektor von n Zeilenvektoren organisiert oc
const l = ⌊log n⌋ co wir nehmen l|n an oc;
array comb[0..2^(l-1)] of boolean-vector;

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 der j-ten Zeile von A_i in Binärzahl wandeln oc
        nc := 0;
        for k := i·l downto (i-1)·l+1 do
            if a[j,k] then nc := nc+nc+1 else nc := nc+nc fi
        od
        c[j] := c[j]∨comb[nc];
    od
od
⌞
```

Beispiel:



Zeitbedarf:

(a) sequentiell:

$\mathcal{O}\left(\frac{n}{l} \cdot (n^2 + n(l+n))\right) = \mathcal{O}\left(\frac{n^3}{l}\right) = \boxed{\mathcal{O}\left(\frac{n^3}{\log n}\right)}$

(b) Vektorrechner der Breite n (Interpretation eines Bitintervalls als Zahl in $\mathcal{O}(1)$ Zeit):

$\mathcal{O}\left(\frac{n}{l} \cdot (n + n(1+1))\right) = \boxed{\mathcal{O}\left(\frac{n^2}{\log n}\right) \text{ (Vektoroperationen)}}$

Satz: Der 4-Russen Algorithmus berechnet das Produkt zweier Boolescher Matrizen sequentiell in Zeit $\mathcal{O}\left(\frac{n^3}{\log n}\right)$ bzw. mit $\mathcal{O}\left(\frac{n^2}{\log n}\right)$ Vektoroperationen.

Beweis: $\surd$ (s.o.) $\diamond$

4.6.4 Transitive Hülle und DFS

Tiefensuche(DFS):

```
proc DFS-visit(node v) =
⌜
markiere v;
num[v] := ++count;
for alle Nachbarn u von v do
    if u unmarkiert then DFS-visit(u) fi
od
⌞

algorithm DFS =
⌜
for alle Knoten v do unmarkiere(v); num(v) := 0 od;
count := 0;
for alle Knoten v do
    if v unmarkiert then DFS-visit(v) fi
od
⌞
```

Bemerkung: Funktioniert für Graphen und Digraphen.

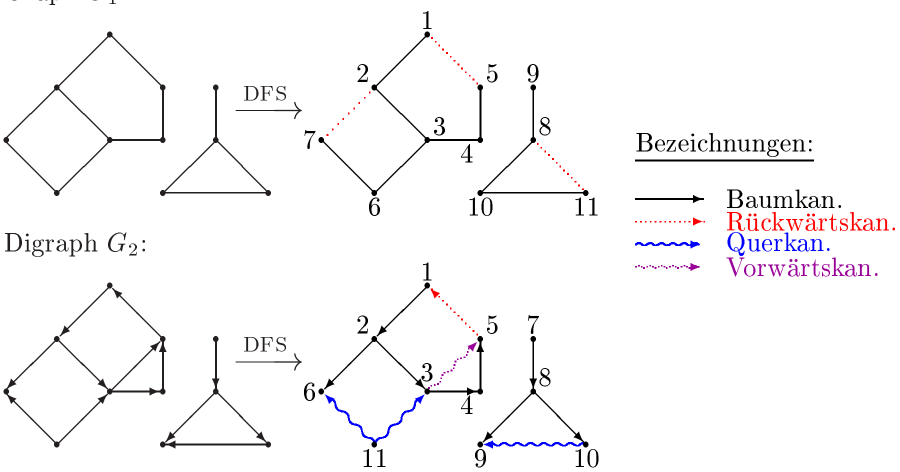
Nachbarn in Graphen und Digraphen:

u Nachbar von v
v Nachbar von u

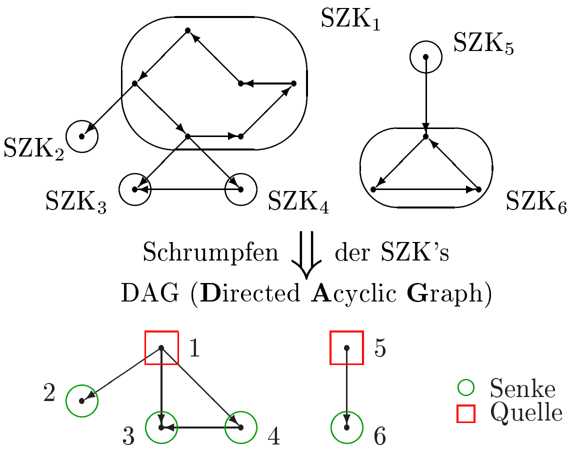
u Nachbar von v

Beispiele für DFS:

Graph G_1:



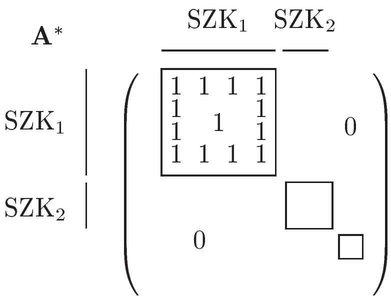
Bemerkung: Alle Baumkanten zusammen ergeben den DFS-Wald (Spannwald).



Bemerkung: SZK: starke Zusammenhangskomponente (in Digraphen)

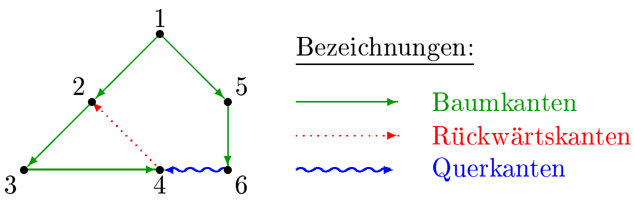
DFS in ungerichteten zyklischen Graphen, n Knoten, m Kanten, $n-1$ Baumkanten, $m-n+1$ Rückwärtskanten.

Zeitaufwand: $\boxed{\mathcal{O}(n+m)}$



A^* aus DFS mit Aufwand $\Theta(n^2)$ in ungerichteten Graphen (Anm.: SZK = ZK).

DFS in gerichteten Graphen (Digraphen), n Knoten, m Kanten, Baum-, Vorwärts-, Rückwärts- und Querkanten:



Zeitaufwand: $\boxed{\mathcal{O}(n+m)}$

Es gilt: u ist von v aus auf einem gerichteten Pfad erreichbar g.d.w. u Knoten in dem DFS-Baum (DFS-visit(v)) mit Wurzel v ist.
Also: Transitive Hülle in Zeit $\mathcal{O}(n \cdot (m+n))$ bzw. $\mathcal{O}(n \cdot m)$. Sehr gut für dünne Graphen (also solche mit relativ wenigen Kanten).

4.7 Ein besserer Algorithmus für das All-pairs-shortest-distance (bzw. paths)-Problem in ungewichteten Digraphen

$$\left.\begin{array}{l}(d_{ij})_{0\leq i,j<n} \text{ Distanzmatrix}\\ D^* \text{ Distanzmatrix für Transitive Hülle}\end{array}\right\}\text{alle Kantenlängen} = 1$$

$$D^{(l)} = (d_{ij}^{(l)})_{o\leq i,j<n} \text{ mit } d_{ij}^{(l)} = \begin{cases} d_{ij}^* \text{ falls } d_{ij}^* \leq l \\ \infty \text{ sonst}\end{cases}$$

<u>algorithm</u> 1 =

<u>co</u> Sei $A = (a_{ij})_{0\leq i,j<n}$ mit $a_{ij} = \begin{cases} 1 \text{ falls } d_{ij} \leq 1 \\ 0 \text{ sonst}\end{cases}$ <u>oc</u>
<u>co</u> Sei B die Boolsche Einheitsmatrix <u>oc</u>
<u>for</u> $l := 2$ <u>to</u> $r+1$ <u>do</u>
    $B := A \cdot B$;
    <u>for</u> $0 \leq i,j < n$ <u>do</u>
        <u>if</u> $b_{ij} = 1$ <u>then</u> $d_{ij}^{(l)} := l$ <u>else</u> $d_{ij}^{(l)} := \infty$ <u>fi</u>;
        <u>if</u> $d_{ij}^{(l-1)} \leq l$ <u>then</u> $d_{ij}^{(l)} := d_{ij}^{(l-1)}$ <u>fi</u>
    <u>od</u>
<u>od</u>

Dieser Algorithmus berechnet $D^{(1)} = D, D^{(2)}, D^{(3)}, \ldots, D^{(r)}$.
Sei ω eine Zahl ≥ 2, so daß die Matrixmultiplikation in der Zeit $\mathcal{O}(n^\omega)$ durchgeführt werden kann (Winograd/Coppersmith: $\omega \leq 2{,}376$).

Zeitaufwand für Algorithmus 1: $\boxed{\mathcal{O}(rn^\omega)}$

<u>algorithm</u> apsd =

Berechne, mit Algorithmus 1, $D^{(l)}$ für $l = 1, \ldots, r$;
$l := r$;
<u>for</u> $s := 1$ <u>to</u> $\left\lceil \log_{\frac{3}{2}} \frac{n}{r} \right\rceil$ <u>do</u>
    <u>for</u> $i := 0$ <u>to</u> $n-1$ <u>do</u>
        finde in der i-ten Zeile von $D^{(l)}$ den Wert d, $\lceil \frac{l}{2} \rceil \leq d \leq l$, so daß d in der i-ten Zeile am wenigsten oft vorkommt;
        $S_i :=$ Menge der Spaltenindizes, in denen dieses d in der i-ten Zeile von $D^{(l)}$ vorkommt;
    <u>od</u>
    $l_1 := \lceil \frac{3}{2} l \rceil$;
    <u>for</u> $0 \leq i,j < n$ <u>do</u>
        <u>if</u> $S_i \neq \emptyset$ <u>then</u> $m_{ij} := \min_{k\in S_i}\{d_{ik}^{(l)} + d_{kj}^{(l)}\}$ <u>else</u> $m_{ij} := \infty$ <u>fi</u>;
        <u>if</u> $d_{ij}^{(l)} \leq l$ <u>then</u> $d_{ij}^{(l_1)} := d_{ij}^{(l)}$
        <u>elseif</u> $m_{ij} \leq l_1$ <u>then</u> $d_{ij}^{(l_1)} = m_{ij}$
        <u>else</u> $d_{ij}^{(l_1)} := \infty$
        <u>fi</u>
    <u>od</u>
    $l := l_1$;
<u>od</u>

apsd berechnet:
Zunächst $D^{(1)}, \dots, D^{(r)}$, dann $D^{(l)}$ für

$$l = r, \left\lceil \frac{3}{2} r \right\rceil, \left\lceil \frac{3}{2} \left\lceil \frac{3}{2} r \right\rceil \right\rceil, \dots, \underbrace{n'}_{\geq n}$$

$$i \bullet \overbrace{\rule{6em}{0.4pt}}^{d} \underset{\bullet}{\overset{k \in S_i}{\downarrow}} \overbrace{\rule{6em}{0.4pt}}^{\lceil \frac{3}{2} l \rceil - d \,\leq\, l} \bullet j$$

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 $r = n^{\frac{3-\omega}{2}}$, dann ergibt sich damit für apsd die Laufzeit $\mathcal{O}\left(n^{\frac{3+\omega}{2}}\right)$.

Satz: 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: $\surd$ $\diamond$

Bemerkung: apsp: Problem: Ausgabegröße kann $\Omega(n^3)$ sein:

$$\boxed{\tfrac{n}{3} \text{ Knoten}} \underbrace{\bullet\!\longrightarrow\!\bullet\!\longrightarrow\!\bullet\!\longrightarrow\!\bullet \cdots \bullet\!\longrightarrow\!\bullet\!\longrightarrow\!\bullet}_{\frac{n}{3} \text{ Knoten}} \boxed{\tfrac{n}{3} \text{ Knoten}}$$

Kürzeste Pfade-Nachfolger-Matrix $N \in [0, \dots, n-1]^{n \times n}$
n_{ij} = Nummer des zweiten Knoten auf „dem" kürzesten Pfad von Knoten i zu Knoten j. So eine Matrixdarstellung für die kürzesten Pfade zwischen allen Knotenpaaren kann in Zeit $\mathcal{O}(n^{\frac{3+\omega}{2}} \cdot \log^c n)$ für ein $c > 0$ bestimmt werden.

Satz: Für Digraphen mit Kantenlängen $\in \{1, 2, \dots, M\}$, $M \in \mathbb{N}$, kann APSD in Zeit $\mathcal{O}((Mn)^{\frac{3+\omega}{2}})$ gelöst werden.

Idee: Ersetze $u \xrightarrow{M' \leq M} v$ durch:

$$u = u_0 \longrightarrow u_1 \longrightarrow u_2 \cdots \longrightarrow u_{M'} = v$$

4.8 Ein Algorithmus für die transitive Hülle in Digraphen mit linearer erwarteter Laufzeit

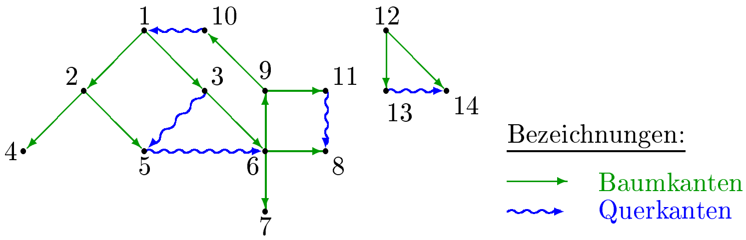
Die Wahrscheinlichkeit eines Digraphen mit n Knoten und m Kanten ist (nur) eine Funktion von n und m. Daraus folgt (wir lassen der Einfachheit halber Schleifen

(=Kanten $v \to v$) zu), daß jede Kante (u, v) mit Wahrscheinlichkeit $\frac{m}{n^2}$ vorhanden ist, falls wir Graphen mit n Knoten und m Kanten betrachten.
Einschub: Breitensuche (BFS): Schlange, queue, FIFO:

$$\longrightarrow \boxed{\text{FIFO}} \longrightarrow$$

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:



algorithm exp-lin-transitive-closure

(C.P. Schnorr, An Algorithm for Transitive Closure with Linear Expected Time, SIAM Journal on Computing 7 (1978), pages 127-133.)

Phasen:

0. Konstruiere die linear geordneten Adjazenzlisten $L_i^r, i = 1, \ldots, n$ des Graphen G^r (entsteht aus G durch Umkehrung aller Knoten).
   Beispiel:

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

   Ersetze ebenfalls alle L_i durch L_i^{rr} ($\to$ 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.
   <u>`for`</u> $i := 1$ <u>`to`</u> n <u>`do`</u>
   $L_i^* := S_i \cup \{j; i \in P_j\} \cup \{j; |S_i| = |P_j| = \lfloor \frac{n}{2} \rfloor + 1\}$;
   <u>`od`</u>
   Bilde $(L_i^*)^{rr}$ für $i = 1, \ldots, n$

Korrektheit: $j \in L_i^*$ g.d.w. (i, j) in transitiver Hülle.

Satz: Sei die Wahrscheinlichkeit eines Graphen (lediglich) eine Funktion der Anzahl n seiner Knoten und der der Anzahl m seiner Kanten. Dann ist der Erwartungswert für die Laufzeit des Schnorr'schen Algorithmus $\mathcal{O}(n + m^*)$, wobei m^* der Erwartungswert für die Anzahl der Kanten in der transitiven Hülle ist.

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,

$$L_{\sigma(\nu)} = (a_t; \mathrm{h}(\nu) < t \leq \mathrm{h}(\nu+1))$$

Sei $\Delta L_{\sigma(\nu)}$ die Menge der a_t in $L_{\sigma(\nu)}$, und sei

$$S_i(t) := \{a_1, a_2, \ldots, a_t\}$$

Wie bereits gezeigt, ist

$$\mathrm{WS}(j \in L_i) = \frac{m}{n^2}, \quad \forall i, j$$

Alle n-Tupel von geordneten Listen $L_1, L_2, ..., L_n$ 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:

$$\mathrm{WS}(\Delta L_{\sigma(\nu)} = A) = (\frac{m}{n^2})^{|A|}(1 - \frac{m}{n^2})^{n-|A|}$$

Lemma: 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:

$$\min\{t; \bar{a}_t \notin A\} \quad \text{hat Erwartungswert} \quad 1 + \frac{k}{n-k}.$$

Beweis: $\mathrm{WS}(\bar{a}_1, \ldots, \bar{a}_r \in A$ <u>und</u> $\bar{a}_{r+1} \notin A) = \left(\frac{k}{n}\right)^r \cdot (1 - \frac{k}{n})$.
Also ist der Erwartungswert für $\min\{t; \bar{a}_t \notin A\}$:

$$\begin{aligned}
&1 + \sum_{r=0}^{\infty} r \left(\frac{k}{n}\right)^r \left(1 - \frac{k}{n}\right) \\
&= 1 + \frac{k}{n}\left(1 - \frac{k}{n}\right) \sum_{r\geq 1} r \left(\frac{k}{n}\right)^{r-1} \\
&= 1 + \frac{k}{n}\left(1 - \frac{k}{n}\right) \frac{1}{\left(1 - \frac{k}{n}\right)^2} \\
&= 1 + \frac{\frac{k}{n}}{1 - \frac{k}{n}} \\
&= 1 + \frac{k}{n-k}
\end{aligned}$$

$\diamond$

Anmerkung:

$$\sum_{r=0}^{\infty} r z^{r-1} = \frac{d}{dz}\frac{1}{1-z} = \frac{1}{(1-z)^2}$$

Lemma: Sei $(\bar{a}_t)_{t\geq 1}$ wie oben, $k \leq \frac{n}{2}$. Dann hat $\min\{t; |\bar{a}_1, \bar{a}_2, \dots, \bar{a}_t| > k\}$ Erwartungswert $\leq \frac{3}{2}(k+1)$.

Beweis: Wegen der Additivität des Erwartungswertes gilt: Der gesuchte Erwartungswert ist

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

Wenn wir jedes $L_{\sigma(\nu)}$ in sich zufällig permutieren, erhalten wir eine Folge $(\bar{a}_t)'_{t\geq 1}$ ◇ von Zufallsvariablen, so daß

$$|\{a_1, \dots, a_t\}| = |\{a'_1, \dots, a'_t\}| \text{ für } t = \mathrm{h}(\nu), \forall \nu$$

Da die a'_t innerhalb eines jeden Intervalls $\mathrm{h}(\nu) < t \leq \mathrm{h}(\nu+1)$ paarweise verschieden sind, gilt für sie die vorhergehende Abschätzung erst recht. Daher 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 $t = \mathrm{h}(y)$; 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.

Sei s der Erwartungswert und sei $|S_i|$ die tatsächliche Größe von S_i nach der Phase 1. Dann

$$s = \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor + 1} k \cdot \mathrm{WS}(|S_i| = k)$$

Die erwartete Anzahl der Schritte (und damit die Kosten) in der BFS sei q. Dann gilt:

$$q \leq \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor + 1} \mathrm{WS}(|S_i| = k) \frac{3}{2} k = \frac{3}{2} s.$$

Also ist der Aufwand des Algorithmus für die Phasen 1 und 2 im Erwartungswert $\leq 3sn$. Da $ns = m^*$, und da die Kosten der anderen Phasen offensichtlich durch $\mathcal{O}(n + m + m^*)$ beschränkt sind, folgt die Behauptung. ◇

4.9 Matrizenmultiplikation à la Strassen

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}; \quad c_{ij} = a_{i1} b_{1j} + a_{i2} b_{2j}$$

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 $\mathrm{M}(n)$ die Anzahl arithmetischer Operationen (Mult, Add, Sub), die Strassen's Algorithmus bei $n \times n$ Matrizen benötigt:

$$\begin{aligned}
\mathrm{M}(1) &= 1 \\
\mathrm{M}(n) &= 7\mathrm{M}\left(\tfrac{n}{2}\right) + 18\left(\tfrac{n}{2}\right)^2
\end{aligned}$$

Expansion dieser Rekursionsgleichung $\rightarrow$

$$\mathrm{M}(n) = 7^{k+1} - 6n^2 = 7 \cdot 2^{k \log_2 7} - 6n^2 = 7n^{\log_2 7} - 6n^2 < 7n^{2,81} - 6n^2$$

```
procedure Matmult(A,B,n) =
kill
⌜
co A und B sind n × n Matrizen oc
if n < 16 then berechne A · B mit klassischer Methode
elseif n gerade then
       spalte A und B in je 4 n/2 × n/2 Matrizen auf und wende Strassen's Formeln
       an. Führe die Multiplikationen rekursiv mit Matmult aus.
   else
       spalte A und B in je eine (n − 1) × (n − 1) Matrix (A11, B11) und
       drei verbleibende Blöcke auf. Wende Matmult rekursiv auf A11, B11 an
       und berechne die anderen Produkte mit klassischer Methode.
   fi
fi
⌞
```

Satz: 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 $4.8n^{\log_2 7}$ arithmetische Operationen.

Beweis:

1. $\surd$

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 klassischer 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 klassischer 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.
   Durch Induktion über die Anzahl der rekursiven Anwendungen von Strassen's Formeln folgt die Behauptung.

3. Sei $\mathrm{M}'(n)$ die Anzahl arithmetischer Operationen. Dann ist:
$$\mathrm{M}'(n) = \begin{cases} 2n^3 - n^2 & \text{falls } n < 16 \\ 7\mathrm{M}'\left(\frac{n}{2}\right) + \frac{18}{4}n^2 & \text{falls } n \geq 16 \text{ gerade} \\ 7\mathrm{M}'\left(\frac{n-1}{2}\right) + \frac{42}{4}n^2 - 17n + \frac{15}{2} & \text{falls } n \geq 16 \text{ ungerade} \end{cases}$$
   Wir definieren für $x \in \mathbb{R}^+$:
$$\overline{\mathrm{M}}(x) = \begin{cases} 2x^3 - x^2 & \text{falls } x < 32 \\ 7\overline{\mathrm{M}}\left(\frac{x}{2}\right) + \frac{42}{4}x^2 & \text{falls } x \geq 32 \end{cases}$$
   Dann gilt:
$$\overline{\mathrm{M}}(n) \geq \mathrm{M}'(n) \text{ für alle } n \in \mathbb{N}$$
   Weiterhin ist
$$\overline{\mathrm{M}}'(x) = \sum_{i=0}^{k-1}\left[7^i \cdot \frac{42}{4}\left(\frac{x}{2^i}\right)^2\right] + 7^k\left(2 \cdot \left(\frac{x}{2^k}\right)^3 - \left(\frac{x}{2^k}\right)^2\right) \text{ für } x \geq 32, \text{ wobei}$$
$$k := \min\left\{l \,\middle|\, \frac{x}{2^l} < 32\right\}$$
   Mit $k = \lfloor \log_2 x \rfloor - 4 = \log_2 x - t$ für $t \in [4, 5]$ erhalten wir:
$$\overline{\mathrm{M}}(x) \leq 7^{\log_2 x}\left(13\left(\frac{4}{7}\right)^t + 2\left(\frac{8}{7}\right)^t\right) \leq 4{,}8 \cdot 7^{\log_2 x} \text{ für } x \geq 32.$$
   Für $x < 32$ direkt nachrechnen.

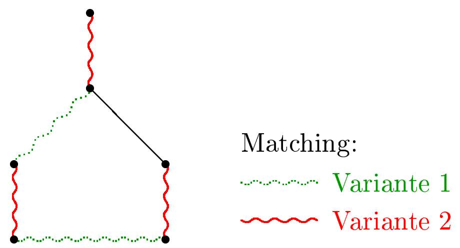
◇

Kapitel 5

Matchings in Graphen

5.1 Grundlagen

Definition: 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:

1. Ein Matching M in $G = (V, E)$ heißt perfekt (oder vollkommen), falls

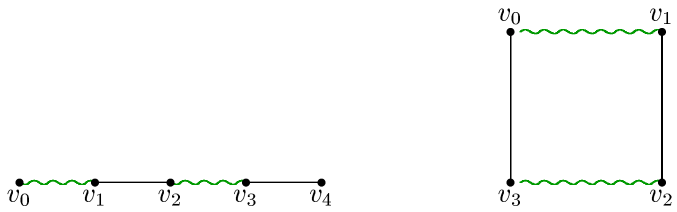
$$|M| = \frac{|V|}{2}$$

2. Ein Matching M heißt „Matching maximaler Kardinalität" (engl. maximum matching) in G, falls es in G kein Matching M' mit $|M'| > |M|$ gibt. ($\rightarrow$ durchgehend geringelt (rot) im Beispiel)

3. Ein Matching M heißt maximal in G, falls es bezüglich „$\subseteq$" maximal ist. (engl. maximal matching, $\rightarrow$ gepunktet geringelt (grün) im Beispiel)

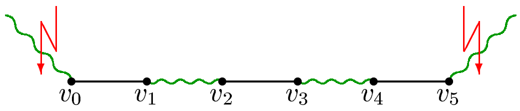
Definition:

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

2. Ein alternierender Pfad heißt augmentierend, falls er bzgl. „$\subseteq$“ maximal ist und an beiden Enden ungematchte Kanten hat.



ungerade Länge

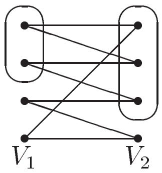
Bemerkung: Es kann keine augmentierenden Kreise geben.

Definition: 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, so heißt G bipartit ($G = (V_1, V_2, E)$).

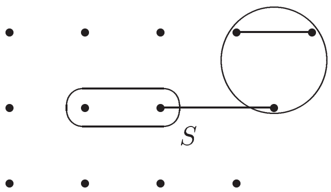
Heiratssatz: (Frobenius, Hall, Rado, König)
Sei $G = (V_1, V_2, E)$ ein bipartiter Graph. G enthält ein Matching der Kardinalität $|V_1|$ genau dann, wenn gilt:

$$\forall V \subseteq V_1 : |N(V)| \geq |V|,$$

wobei N(V) die Nachbarschaft von V (in V_2) bezeichnet.



Satz: 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.

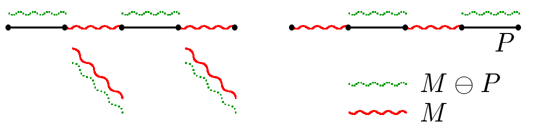


Satz:(Berge, 1957)
Ein Matching hat maximale Kardinalität genau dann, wenn es keinen augmentierenden Pfad dafür gibt.

Definition: 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: 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:
„$\Leftarrow$“



„$\Rightarrow$“ Satz von Berge ◇

Lemma: 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 (gerade Länge)
3. alternierende Pfade

Seien $C_1, \ldots, C_r$ die Zusammenhangskomponenten in $(V,\ N \ominus M)$ dann gilt:

$$M \ominus \underbrace{C_1 \ominus C_2 \ominus \ldots \ominus C_r}_{N \ominus M} = N$$

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 (ev. knotendisjunkt da ZHK's). ◇

Korollar: Satz von Berge

Lemma: Sei M ein Matching der Kardinalität r, und sei s die maximale Kardinalität eines Matchings in $G = (V, E)$. Dann gibt es einen augmentierenden Pfad bzgl. M der Länge $\leq 2\lfloor \frac{r}{s-r} \rfloor + 1$, wobei $s > r$ vorausgesetzt sei.

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. Einer dieser augmentierenden Pfade enthält daher $\leq \lfloor \frac{r}{s-r} \rfloor$ Kanten aus M. ◇

Lemma: 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 $P_1,\ P_2$. Es gilt:

$N \ominus M = P \ominus P'$, also

$|P \ominus P'| \stackrel{\text{Def.}}{=} |(P - P') \cup (P' - P)| = |P| + |P'| - 2|P \cap P'| \geq |P_1| + |P_2| \geq 2|P|$,

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 $M_{i+1} = M_i \ominus P_i$.

Es gilt: $|P_{i+1}| \geq |P_i|$

Lemma: 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 ein augmentierender Pfad bzgl. M', dem Matching nach dem augmentieren mit $P_0,\ P_1, \ldots,\ P_i$. Mit vorigem Lemma: $|P_j| \geq |P_i| + 2|P_i \cap P_j|$ also $P_i,\ P_j$ kantendisjunkt. Da in M' jeder Knoten in P_i gematcht ist (und auch in $M' \ominus P_{i+1} \ominus P_{i+2} \ominus \ldots \ominus P_{j-1}$) können P_i und P_j auch keinen Knoten gemeinsam haben. ◇

Satz: 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 $r := \lfloor s - \sqrt{s} \rfloor$. Per Konstruktion ist $|M_i| = i$, also $|M_r| = r$. Mit dem Lemma über die maximale Länge eines kürzesten augmentierenden Pfades folgt

$$|P_r| \quad \leq \quad 2\left\lfloor \frac{\lfloor s - \sqrt{s} \rfloor}{s - \lfloor s - \sqrt{s} \rfloor} \right\rfloor + 1 \quad \leq \quad 2\left\lfloor \frac{s}{\sqrt{s}} \right\rfloor + 1 \quad = \quad 2\lfloor \sqrt{s} \rfloor + 1.$$

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 := ∅;
while ∃ augmentierender Pfad bzgl. M do
    l := Länge des kürzesten augmentierenden Pfades bzgl. M;
    bestimme eine bzgl. „⊆\ maximale Menge {Q_1, ..., Q_m}
    augmentierender Pfade bzgl. M, die alle Länge l haben und knotendisjunkt sind;
    M := M ⊖ Q_1 ⊖ ... ⊖ Q_m;
od
```

Korollar: In obiger *while*-Schleife finden höchstens $\mathcal{O}\left(|V|^{\frac{1}{2}}\right)$ Durchläufe statt.

5.2 Matchings maximaler Kardinalität in bipartiten Graphen

Gegeben: $G = (V_1, V_2, E)$ bipartiter Graph.

Simultane BFS zur Bestimmung der Länge eines kürzesten augmentierten Pfades.

```
for alle v ∈ V1 ∪ V2 do
    label[v] := 0;
od
R := ∅;  l := 1;
for alle freien Knoten v ∈ V1 do
    for alle Kanten {v, w} ∈ E do
        label[w] := 1;  R := R ∪ {w};
    od
od
while R ≠ ∅ and R enthält keinen freien Knoten do
    L := ∅;  l++;
    for alle w ∈ R, {v, w} ∈ M do
        L := L ∪ {v};
        label[v] := l;
    od
    R := ∅;  l++;
    for alle v ∈ L, {v, w} ∈ E \ M do
        if label[w] = 0 then
            R := R ∪ {w};
            label[w] := l;
        fi
    od
od
R := Menge der freien Knoten in R;
```