Algorithmen

Prof. M. Kaufmann*

Sommersemester 2016

Stand: 20. April 2016, 17:30

*Mitschrift: Volodymyr Piven, Alexander Peltzer, Korrektur: Demen Güler, Sebastian Nagel, Aktualisierungen und Korrektur 2013 von Tobias Fabritz, Simon Kalt, Jan-Peter Hohloch, kleine Erweiterung 2015 von Lennard Boden und David-Elias Künstle. Korrigiert und erweitert 2016 von Finn Ickler

Inhaltsverzeichnis

Vorwort

Beim vorliegenden Text handelt es sich um eine inoffizielle Mitschrift. Als solche kann sie selbstverständlich Fehler enthalten und ist daher für Übungsblätter und Klausuren nicht zitierfähig.

Korrekturen und Verbesserungsvorschläge sind jederzeit willkommen und können als Request im Repository[1][2] eingereicht werden. Alternativ kann gerne unter Beachtung der Lizenz ein Fork erstellt werden.

Der Text wurde mit LaTeX2e in Verbindung mit dem $\mathcal{AMS}$-TeX-Paket für die mathematischen Formeln gesetzt.

Einleitung

Dozent:

» Professor Dr. Michael Kaufmann
Tel.: +49-7071-2977404
Email: mk@informatik.uni-tuebingen.de

Vorlesungszeiten:

» Dienstag 14-16 Uhr

» Donnerstag 10-12 Uhr

Literatur:

» Algorithmen - Kurz gefasst, Schöning, Springer

[1] SS13: https://bitbucket.org/urm31/algoskript/
[2] SS15: https://bitbucket.org/de_kuenstle/algoskript-ss15

1 Komplexität

1.1 $\mathcal{O}$-Notation

$f(n) = \mathcal{O}(g(n))$ (obere Schranke), wenn

$$\exists\, c,\ n_0 > 0\ \forall\, n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$$

$f(n) = \Omega(g(n))$ (untere Schranke), wenn

$$\exists\, c,\ n_0 > 0\ \forall\, n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)$$

$f(n) = \Theta(g(n))$ (scharfe Schranke), wenn

$$\exists\, c_1, c_2,\ n_0 > 0\ \forall\, n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Anmerkung:
Mit $f(n) = \mathcal{O}(g(n))$ ist gemeint $f(n) \in \mathcal{O}(g(n))$
$\Theta(g(n)) = \mathcal{O}(g(n)) \cap \Omega(g(n))$

1.2 Nützliche Rechenregeln

Logarithmen:

$$\begin{aligned}
\log(a \cdot b) &= \log a + \log b \\
\log_a b &= \frac{\log b}{\log a} \\
\log a^b &= b \cdot \log a \\
b^{\log_b a} &= a
\end{aligned}$$

Andere nützliche Regeln:

» Stirling-Approximation:

$$n! = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right)$$

Eine einfachere Abschätzung ist:

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$$

» Gauß'sche Summenformel:

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

» geometrische Reihe:

$$\sum_{k=0}^{n} q^k = \frac{q^{n+1} - 1}{q - 1}$$
$$\sum_{k=0}^{\infty} q^k = \frac{1}{1-q} \quad \text{für } |q| < 1$$

Also zB für $q = \frac{1}{2}$: $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots = 2$

» Ebenfalls gilt (Ableitung):

$$\sum_{k=0}^{\infty} k \cdot q^{k-1} = \frac{1}{(1-q)^2} \quad \text{für } |q| < 1$$

» n-te harmonische Zahl:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + \mathcal{O}(1)$$

2 Rekursionen

» Merge-Sort:

$$T(n) = \underbrace{2 \cdot T(\frac{n}{2})}_{\text{2 Teilprobleme}} + \underbrace{n}_{\text{Aufwand des Aufteilens}} \quad \text{und } T(1) = 1$$

» Schaltkreise: $C(n) \leq C(\frac{2}{3}n) + \frac{n}{3}$

» parallele Algorithmen: $T(n) = T(\frac{9}{10}) + \mathcal{O}(\log n)$

» Selection: $T(n) \leq \frac{2}{n} \cdot \sum_{k=\frac{n}{2}}^{n-1} T(k) + \mathcal{O}(n)$

2.1 3 Methoden zur Berechnung

2.1.1 Schätzen und Beweisen

» $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$
Schätzen: $T(n) = \mathcal{O}(n \log n)$
Beweisen: Behauptung $T(n) \leq c \cdot n \log n$ für c groß genug.
Induktion:
Induktionsanfang:

$$T(1) = 1 \checkmark$$

Induktionsschritt:
Es gilt

$$\begin{aligned} T(n) &\leq 2 \cdot c \cdot \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) + n \\ &= 2 \cdot c \cdot \frac{n}{2} \cdot (\log n - \log 2) + n \\ &= c \cdot n \cdot ((\log n) - 1) + n \leq c \cdot n \log n \end{aligned}$$

für $c \geq 1$

» $C(n) \leq C(\frac{2}{3}n) + \frac{n}{3}$
Schätzen: $C(n) = \mathcal{O}(n)$

Beweis. $C(n) \leq d \cdot n$ für konstante d

$$C(1) = 1 \leq d \text{ für } d \geq 1$$

$$C(n) \leq d \cdot \frac{2}{3}n + \frac{n}{3} \leq d(\frac{2}{3}n + \frac{n}{3}) \text{ für } d \geq 1 = d \cdot n$$

□

» $A(n) = 2 \cdot A(\frac{n}{2}) + \sqrt{n}$
Schätzen:$A(n) = \mathcal{O}(n)$
Versuchter Beweis:
$A(n) \leq c \cdot n$ für c groß.
$A(1) = 1$
$A(n) = 2 \cdot c \cdot \frac{n}{2} + \sqrt{n} \not\leq c \cdot n$ so nicht!

2.1.2 Ausrechnen

$$\begin{aligned}
A(n) &= 2 \cdot A\left(\frac{n}{2}\right) + \sqrt{n} \\
&= 2 \cdot \left(2 \cdot A\left(\frac{n}{4}\right) + \sqrt{\frac{n}{2}}\right) + \sqrt{n} \\
&= 4 \cdot A\left(\frac{n}{4}\right) + 2 \cdot \sqrt{\frac{n}{2}} + \sqrt{n} \\
&= 8 \cdot A\left(\frac{n}{8}\right) + 4\sqrt{\frac{n}{4}} + 2\sqrt{\frac{n}{2}} + \sqrt{n} \\
&\dots \\
&= 2^i \cdot A\left(\frac{n}{2^i}\right) + \sum_{j=0}^{i-1} 2^j \cdot \sqrt{\frac{n}{2^j}} \qquad \text{(i-ter Schritt)} \\
&= 2^i \cdot A\left(\frac{n}{2^i}\right) + \sqrt{n} \cdot \sum_{j=0}^{i-1} 2^{\frac{j}{2}}
\end{aligned}$$

Schritt ist durch Induktion zu zeigen:

$$\begin{aligned}
&= 2^i \cdot \left(2 \cdot A\left(\frac{n}{2^{i+1}}\right) + \sqrt{\frac{1}{2^i}}\right) + \sqrt{n} \cdot \sum_{j=0}^{i-1} 2^{\frac{j}{2}} \\
&= 2^{i+1} \cdot A\left(\frac{n}{2^{i+1}}\right) + \sqrt{n} \cdot \sum_{j=0}^{i} 2^{\frac{j}{2}} \checkmark
\end{aligned}$$

Sei $i = \log n$, da dies der maximale Fall ist, wenn n eine Zweierpotenz:

$$\begin{aligned} & 2^{\log n} \cdot A(1) + \sqrt{n} \sum_{j=0}^{\log n - 1} \sqrt{2}^j \\ &= n + \sqrt{n} \cdot \mathcal{O}(\sqrt{n}) \\ &= \mathcal{O}(n) \end{aligned}$$

2.1.3 Master Theorem

Löse $T(n) = a \cdot T(\frac{n}{b}) + f(n) \ \ (a \geq 1,\ b > 1,\ f(n) \geq 0)$
Beispielsweise für Merge-Sort: $a = b = 2, f(n) = n$

`Satz 2.1.1.` *Sei $a \geq 1, b > 1$ konstant, $f(n), T(n)$ nicht negativ mit*

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

wobei $\frac{n}{b}$ für $\lceil \frac{n}{b} \rceil$ oder $\lfloor \frac{n}{b} \rfloor$ steht. Dann gelten

1. *Für $f(n) = O\left(n^{\log_b a - \epsilon}\right)\ , \epsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$*
2. *Für $f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$*
3. *Für $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)\ , \epsilon > 0$ und $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ für $c < 1$ und n genügend groß, so ist: $T(n) = \Theta(f(n))$*

Beispiel:

- $T(n) = 9 \cdot T(\frac{n}{3}) + n$, $a = 9, b = 3, f(n) = n$
  $\log_3 9 = 2 \Rightarrow$Fall 1 $\Rightarrow T(n) = \Theta(n^2)$
- $T(n) = T(\frac{n}{2}) + 1$, $a = 1, b = 2, f(n) = 1$
  $\log_b a = \log_2 1 = 0 \Rightarrow$Fall 2 $\Rightarrow T(n) = \Theta(\log n)$
- $T(n) = 4 \cdot T(\frac{n}{2}) + n^4$
  $f(n) = n^4 \geq \Theta(n^{2+2})$✓
  $\frac{n^4}{4} = 4 \cdot (\frac{n}{2})^4 \leq c \cdot n^4$✓ Fall 3
  $\Rightarrow T(n) = \Theta(n^4)$
- $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$, $a = 2, b = 2, f(n) = n \log n$
  $log_2 2 = 1$
  $n \log n < n^{1+\epsilon} \Rightarrow$ nicht anwendbar! (da „zwischen“ Fall 2 und 3!)

$T(n) = a \cdot T(\frac{n}{b}) + f(n)$, $a \geq 1,\ b > 1,\ f(n) \geq 0$
Beweis:
Annahme: $n = b^i, i \in \mathbb{N}$
das heißt, Teilprobleme haben Größe $b^i, b^{i-1}, \ldots, b, 1$

Lemma 2.1.2. *Sei* $a, b \geq 1, f(n) \geq 0, n = b^i, i \in \mathbb{N}$ *mit*

$$T(n) = \begin{cases} \Theta(1) & \text{für } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n) & \text{für } n = b^i \end{cases}$$

Dann gilt $T(n) = \Theta(n^{\log_b a}) + \sum_{j=1}^{(\log_b n)-1} a^j f(\frac{n}{b^j})$

Beweis.

$$\begin{aligned} T(n) &= f(n) + a \cdot T\left(\frac{n}{b}\right) \\ &= f(n) + a\left(f\left(\frac{n}{b}\right) + a \cdot T\left(\frac{n}{b^2}\right)\right) \\ &= f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 \cdot T\left(\frac{n}{b^2}\right) \\ &= f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 \cdot T\left(\frac{n}{b^2}\right) + a^3 \cdot T\left(\frac{n}{b^3}\right) \\ &= \dots \\ &= f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 \cdot f\left(\frac{n}{b^2}\right) + \dots + a^i T\left(\frac{n}{b^i}\right) \\ &= f(n) + a \cdot f\left(\frac{n}{b}\right) + \dots + \underbrace{a^{\log_b n} \cdot T(1)}_{=\Theta(n^{\log_b a})} \\ &= \Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{(\log_b n)-1} a^j f\left(\frac{n}{b^j}\right) \end{aligned}$$

□

3 Fälle, je nachdem wie groß $f(n)$ ist.
Anschaulich:
Entweder dominiert die Wurzel (Fall 3) oder das Blattlevel (Fall 1) oder es fallen alle Terme gleich ins Gewicht (Fall 2, $\log_b n$ Terme)

Lemma 2.1.3. *Sei* $a, b \geq 1, f(n)$ *definiert auf Potenzen von b und sei* $g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$
Ist $f(n) = \mathcal{O}\left(n^{\log_b a - \epsilon}\right), \epsilon \geq 0$, *so ist* $g(n) = \mathcal{O}\left(n^{\log_b a}\right)$.

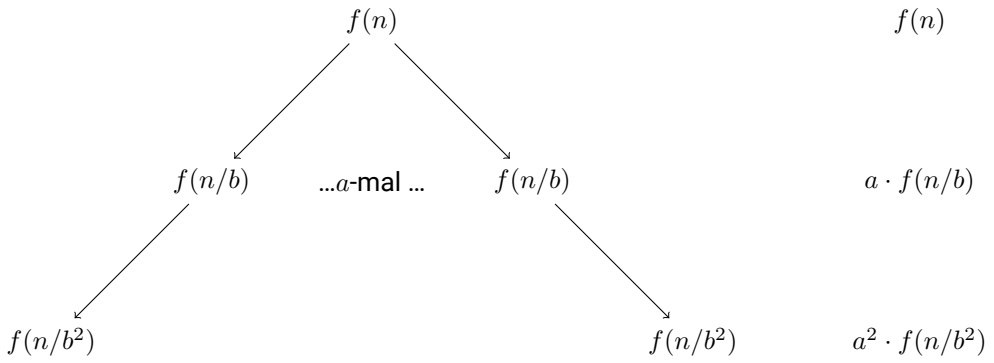


Abbildung 2.1: Rekursionsbaum

Beweis. Es gilt:

$$
\begin{aligned}
& f\left(\frac{n}{b^j}\right) = \mathcal{O}\left(\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) \\
\Rightarrow\ & g(n) = \mathcal{O}\left(\sum_{j=0}^{\log_b n-1} a^j \cdot \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) \\
& = \mathcal{O}\left(n^{\log_b a-\epsilon} \cdot \sum_{j=0}^{\log_b n-1} \left(\frac{a \cdot b^\epsilon}{b^{\log_b a}}\right)^j\right) \\
& = \mathcal{O}\left(n^{\log_b a-\epsilon} \cdot \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j\right) && \text{(geom. Reihe)} \\
& = \mathcal{O}\left(n^{\log_b a-\epsilon} \cdot \frac{(b^\epsilon)^{\log_b n-1}}{b^\epsilon - 1}\right) \\
& = \mathcal{O}\left(n^{\log_b a-\epsilon} \cdot \underbrace{\frac{n^{\epsilon-1}}{b^\epsilon - 1}}_{\in\mathcal{O}(n^\epsilon)}\right) \\
& = \mathcal{O}\left(n^{\log_b a-\epsilon} \cdot n^\epsilon\right) = \mathcal{O}\left(n^{\log_b a}\right)
\end{aligned}
$$

□

3 Suchen in geordneten Mengen

Gegeben ist ein Universum U, mit linearer Ordnung $(U, <)$. Sei nun $S \subseteq U$ die Menge der zu verwaltenden Schlüssel. Wir betrachten nun verschiedene Algorithmen in S ein Element zu suchen.

1. *lineare Suche:* S liege als geordnetes Array vor. Nun sei next, das Element, das als nächstes betrachtet wird.
   Starte mit next $\leftarrow 0$
   Schleife: next $\leftarrow$ next $+ 1$
   Abfrage: Ist $a < S$[next]?
   Laufzeit:
   best case: $\mathcal{O}(1)$
   worst case: $\mathcal{O}(n), n$ Iterat., $|S| = n$
   average case: $\frac{n}{2}$ Iterat. $= \mathcal{O}(n)$
2. *binäre Suche*: S liege wieder als sortiertes Array vor. 2 Grenzen oben , unten als Indizes. Wir suchen wie in Algorithmus 1 beschrieben.
   Laufzeit (best case): $T(n) = 1$ (zu suchendes Element genau in der Mitte)

`Algorithmus 1` Binary Search

```
1:  function SUCHE(a, S)
2:      unten ← 0
3:      oben ← (n − 1)                          // Array enthält n Elemente
4:      while unten < oben do
5:          next ← ⌈(oben − unten)/2⌉ + unten
6:          if S[next] = a then
7:              return next
8:          else if S[next] < a then
9:              unten ← next + 1
10:         else
11:             oben ← next − 1
12:         end if
13:     end while
14:     return −1
15: end function
```

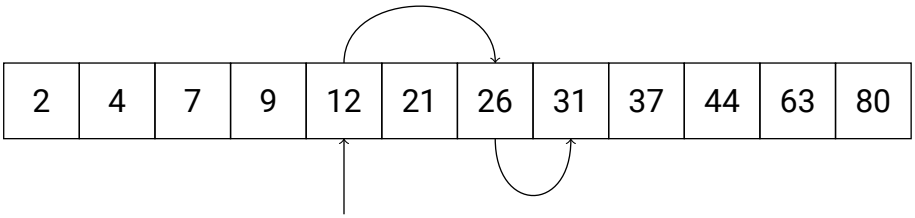


Abbildung 3.1: Interpolationssuche

Laufzeit (worst case):

$$\begin{aligned} T(n) &= 1 + T\left(\frac{n}{2}\right) \\ &= 1 + 1 + T\left(\frac{n}{4}\right) \\ &= 1 + 1 + 1 + T\left(\frac{n}{8}\right) \\ &= i + T\left(\frac{n}{2^i}\right) && \text{(i-ter Schritt)} \\ &= \log n + T(1) = O(\log n) \end{aligned}$$

3.1 Interpolationssuche

next $\leftarrow \left\lfloor \frac{a - S[\text{unten}]}{S[\text{oben}] - S[\text{unten}]} \cdot (\text{oben} - \text{unten}) \right\rfloor + \text{unten}$

Laufzeit: worst case: $\mathcal{O}(n)$
average case: $\mathcal{O}(\log \log n)$

Interpolieren und Springen sukzessive in Sprüngen der Länge $\sqrt{\text{oben} - \text{unten}}$, bis das richtige Suchintervall gefunden ist. Suchintervall hat Größe $\sqrt{\text{oben} - \text{unten}}$.

3.2 Erweiterung der Suche in geordneten Mengen

$|S| = n$

a.) Lineare Suche $\mathcal{O}(n)$

b.) Binäre Suche $\mathcal{O}(\log n)$

c.) Interpolationssuche $\mathcal{O}(\log\log n)$ im Mittel.

3.2.1 Variante von Reingold

S als Array $S[1],\ldots,S[n]$ füge künstliche Elemente ein $S[0],\ldots S[n+1]$
next $\leftarrow$ (unten -1) + $\lceil \frac{a-S(\text{unten}-1)}{S(\text{oben}+1)-S(\text{unten}-1)} \cdot (\text{oben - unten} + 1)\rceil$
Starte mit $n \leftarrow$ oben $-$ unten $+1$

(1) Interpoliere: $a : S[next]$

(2) falls "=" fertig
falls $>$: lineare Suche in $S[next+\sqrt{n}], S[next+2\cdot\sqrt{n}]\cdots$ bis $a < S[next+(i-1)\sqrt{n}]$
Es gilt : Nach i Vergleichen gilt: $S[next+(i-2)\sqrt{n}) \leq a < S[next+(i-1)\sqrt{n}]$
falls $<$: analoges Vorgehen

(3) rekursiv im Suchintervall der Größe $\sqrt{n}$ weitersuchen.

3.2.2 Analyse

Sei C die mittlere Anzahl von Vergleichen um auf Teilproblem der Größe $\sqrt{n}$ zu kommen, dann:

$$
\begin{aligned}
&T(n) \leq C + T(\sqrt{n})\\
&T(1) = 1\\
&T(2) = 1\\
&T(n) = \mathcal{O}(\log\log n) \text{falls } C \text{ konstant.}
\end{aligned}
$$

Berechnung:

$$
\begin{aligned}
T(n) &= C + T(\sqrt{n})\\
&= C + C + T\left(n^{\frac{1}{4}}\right)\\
&= ... =\\
&= i\cdot C + T\left(n^{\frac{1}{2^i}}\right) \stackrel{!}{=} a\cdot C + T(2)
\end{aligned}
$$

Nun soll berechnet werden für welches a gilt:

$$\begin{aligned} n^{\frac{1}{2^a}} &= 2 \\ \log n^{\frac{1}{2^a}} &= log2 \\ \frac{1}{2^a} \cdot \log n &= 1 \\ \log n &= 2^a \\ \log\log n &= a \end{aligned}$$

Wahrscheinlichkeitsannahme

Elemente $a_1 \dots a_n$ und auch a sind zufällige Elemente aus (a_0, a_{n+1}) gezogen nach Gleichverteilung.
Sei p_i Wahrscheinlichkeit, dass $\geq i$ Vergleiche nötig sind.
$\Rightarrow C_i = \sum_i i \cdot \underbrace{(p_i - p_{i+1})}_{\text{Prob(genau i Vergleiche.)}} = 1(p_1 - p_2) + 2(p_2 - p_3) + 3(p_3 - p_4) = \sum_{i \geq 1} p_i$

$p_1 = 1 = p_2$
Was ist jetzt p_i ?
Falls i Vergleiche nötig sind, weicht der tatsächliche Rang von a in der Folge $a_1, \dots, a_n$ um mindestens $(i-2) \cdot \sqrt{n}$ von next ab.
(Rang von a: Anzahl von a_i mit $a_i < a$)
das heißt: $p_i \leq$ prob $((Rang(a) - next) \geq (i-2)\sqrt{n})$
bedeutet prob $((y - \mu(y)) \geq (i-2) \cdot \sqrt{n})$ mit unten = 1, oben = n, next = $p \cdot n$
$p = \frac{a - S[unten-1]}{S[oben+1] - S[unten-1]}$

next ist die erwartete Anzahl der Elemente $\leq a$
also der erwartete Rang von a.
Sei y Zufallsvariable, die Rang von a in $a_1, \dots, a_n$ angibt, sowie $\mu(y)$ Erwartungswert.

Tschebyscheff'sche Ungleichung:

$$\text{prob } ((y - \mu(y)) > t) \leq \frac{Var(y)}{t^2}$$

$Var(x) = E((x - E(x))^2)$

Es gilt: Erwartungswert: $p \cdot n$
Varianz: $(1-p) \cdot p \cdot n$

Erwartungswert μ
prob(genau j Elemente < a) = $\binom{n}{j} p^j (1-p)^{n-j}$

Deshalb:

$$\begin{aligned}\mu &= \sum_{j=1}^{n} j \cdot \binom{n}{j} p^j (1-p)^{n-j} \\ &= n \cdot p \cdot \sum_{i=1}^{n} \binom{n-1}{j-1} p^{j-1} (1-p)^{n-j} \\ &= n \cdot p \cdot \sum_{j=0}^{n-1} \binom{n-1}{j} p^j (1-p)^{n-(j+1)} \\ &= n \cdot p \cdot 1 \\ &= n \cdot p\end{aligned}$$

$$p_i \leq \text{prob}\left(\underbrace{Rang}_{y} - \underbrace{next}_{\mu} \geq \underbrace{(i-2)\sqrt{n}}_{t} \right)$$

$$\leq \frac{Var(y)}{t^2} = \frac{p(1-p)\cdot n}{(i-2)^2 \cdot n} \underset{\text{da } 0\leq p\leq 1}{\overset{p\cdot(1-p)\leq \frac{1}{4}}{\leq}} \frac{1}{4(i-2)^2}$$

Es gilt

$$C \leq \sum_i p_i \leq 2 + \sum_{i\geq 3} \frac{1}{4(i-2)^2} \leq 2 + \frac{1}{4} \sum_{i\geq 1} \left(\frac{1}{i}\right)^2 \leq 2 + \frac{\pi^2}{24} \approx 2.4$$

Im Mittel brauchen wir also $\mathcal{O}(\log\log n)$ Versuche, da $T(n) = 2.4 \log\log(n)$
worst case:

$T_{\text{worst}}(n) = T_{\text{worst}}(\sqrt{n}) + (\sqrt{n}+1) = O\left(\sqrt{n} + \sqrt{\sqrt{n}} + \sqrt{\sqrt{\sqrt{n}}} + \ldots\right) = \mathcal{O}(\sqrt{n})$

Es sind also $\sqrt{n}$ Sprünge nötig um Problem auf Größe $\sqrt{n}$ zu bringen. aber: Annahme der Gleichverteilung kritisch.

4 Sortieren

$A: 11, 3, 4, 5, 4, 12 \quad \rightarrow \quad B: 3, 6, 6, 7, 11, 12$

Verfahren: 1.Minimumsuche + Austausch mit 1. Element usw. $\mathcal{O}(n^2)$
2.Insertionsort $\mathcal{O}(n^2)$
3.Bubble Sort $\mathcal{O}(n^2)$
4.Quicksort, Heapsort, MergeSort, BucketSort

4.1 Quicksort(Divide and Conquer)

Eingabe $S \quad a_1, \ldots, a_n$

1. Wähle Pivotelement a_1
   Teile $S \backslash \{a_1\}$ auf in $A = \{a_i \mid a_i \leq a_1 \text{ für } i \geq 2\}$, $B = \{a_i \mid a_i > a_1 \text{ für } i \geq 2\}$
   Speichere A in $S[1], \ldots, S[i]$ a_1 in $S[j+1]$, B in $S[j+2], \ldots S[n]$
2. Sortiere A und B rekursiv.

Analyse: Schlechtester Fall: wenn a_1 minimal $(A = \varnothing)$ oder a_1 maximal $(B = \varnothing)$.
$\Rightarrow$ # Vergleiche: $(n-1) + (n-2) + (n-3) + \ldots + 1 = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$
Mittlere Analyse: Modelliere Elemente paarweise verschieden, jede der $n!$ möglichen Permutationen der Eingabe ist gleich wahrscheinlich
oBdA: $S = \{1, 2, \ldots, n\}$
$S[1]$ mit Wahrscheinlichkeit $\frac{1}{n}$ für $k = 1, 2, \ldots, n$
Teilprobleme der Größe $k-1$ bzw. $n-k$ erfüllen wieder die W' annahmen.

Sei $\overline{QS}(n)$ die mittlere Anzahl von Vergleichen an Feld der Größe n. Es gilt:

$$\overline{QS}(0) = \overline{QS}(1) = 0$$

sowie für $n \geq 2$:

$$\begin{aligned}\overline{QS}(n) &= n + E(\overline{QS}(A) + \overline{QS}(B)) \\ &= n + \frac{1}{n} \cdot \sum_{k-1}^{n} \overline{QS}(k-1) + \overline{QS}(n-k) \\ &= n + \frac{2}{n} \cdot \sum_{k=1}^{n-1} \overline{QS}(k)\end{aligned}$$

Es lassen sich nun die Formeln

$$n \cdot \overline{QS}(n) = n^2 + 2 \sum_{k-1}^{n-1} \left(\overline{QS}(k)\right) \tag{i}$$

$$(n+1) \cdot \overline{QS}(n+1) = (n+1)^2 + 2 \cdot \sum_{k-1}^{n} \left(\overline{QS}(k)\right) \tag{ii}$$

bilden und damit gilt

$$\begin{aligned}\text{(ii) - (i)} &= (n+1) \cdot \overline{QS}(n+1) - n \cdot \overline{QS}(n) = (n+1)^2 - n^2 + 2 \cdot \overline{QS}(n) \\ (n+1) \cdot \overline{QS}(n+1) &= 2n + 1 \ + \ (n+2) \cdot \overline{QS}(n) \\ \overline{QS}(n+1) &\overset{\frac{2n+1}{n+1} \leq 2}{\leq} 2 + \frac{n+2}{n+1} \cdot \overline{QS}(n) \\ &= 2 + \frac{n+2}{n+1} \cdot \left(2 + \frac{n+1}{n} \cdot \overline{QS}(n-1)\right) \\ &= 2 + (n+2) \cdot \left(\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots + \frac{2}{2} + \frac{2}{1}\right) \\ \overline{QS}(n) &= 2 + 2 \cdot (n+2) \cdot \underbrace{\left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \ldots + 1\right)}_{\text{harmonische Reihe}} \\ &< 2 + 2 \cdot (n+2) \cdot (\ln n + 1) \\ &\Rightarrow \mathcal{O}(n \cdot \log(n))\end{aligned}$$

4.2 Heap Sort

Ähnlich zu „Sortieren durch Min-Suche“.

`Definition 4.2.1.` *Heap ist Binärer Baum:*
Er ist entweder leer, oder hat Wurzelknoten mit linkem und rechtem Teilbaum.

Dabei gilt für Baum T und Knoten v in T:

$$\text{Tiefe}(v) = \begin{cases} 0 & \text{, falls } v \text{ Wurzel von } T \\ \text{Tiefe}(\text{Parent}(\text{v})) + 1 & \text{, sonst} \end{cases}$$

$$\text{Höhe}(v) = \begin{cases} 0 & \text{, falls } v \text{ Blatt} \\ \max(\{\text{Höhe}(u) \mid u \text{ Kind von } v\}) + 1 & \text{, sonst} \end{cases}$$

Allgemein gilt für Baum T:

$$\text{Tiefe}(T) = \max(\{\text{Tiefe}(v) \mid v \in T\})$$
$$\text{Höhe}(T) = \text{Höhe}(\text{Wurzel}(T)) = \max(\{\text{Höhe}(v) \mid v \in T\})$$

Heap wird durch binären Baum dargestellt, wobei jedes Element aus S einem Knoten aus h zugeordnet wird.
Heapeigenschaft: für Knoten mit u, v mit parent$(v) = u$ gilt $S[u] \leq S[v]$

Beobachtung:
Beim Heap stet das Minimum in der Wurzel. Sortieren durch Min-Suche.
1. Suche: Min $\rightarrow$ Wurzel $\mathcal{O}(1)$
2. Lösche Min aus Suchmenge und repariere Heap.

Ablauf:

- » Nimm Blatt und füge es zu Wurzel hinzu
- » Lass es „nach unten sinken“ (vertausche mit kleinstem Kind)
- » # Iterationen $\leq$ Höhe(T)

Höhe sollte also klein sein
$\Rightarrow$ Ausgewogene Bäume:

- » Alle Blätter haben Tiefe k oder $k + 1$
- » Blätter auf Tiefe $k + 1$ stehen ganz links.

„Kanonische Form“: Es gilt auf Tiefe $1, 2, \ldots, k$ liegen $2^1, \ldots, 2^k$ Knoten.

In Abb. 4.1 ist ein Beispiel-Heap als Baum dargestellt. Dieser ist noch nicht ausgewogen. Eine mögliche ausgewogene Variante für diesen Baum ist in Abb. 4.2 dargestellt.

Es gilt für Knoten mit Nummer i: $parent(i)$ hat die Nummer $\lfloor \frac{i}{2} \rfloor$, Kinder haben $2i$ und $2i + 1$. Der Heap wird nur konzeptuell als Baum dargestellt. Implementiert wird dieser als Array mit einer Indizierung ab 1. Der Heap wird wie in Algorithmus 2 beschrieben initialisiert.

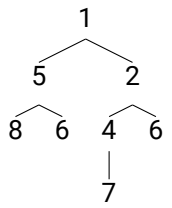


Abbildung 4.1: Unausgewogener Heap Baum

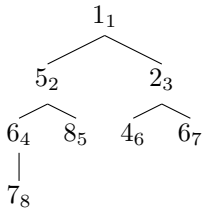


Abbildung 4.2: Ausgewogener Heap Baum mit Nummerierung der Elemente

Algorithmus 2 Initialisierung des Heaps h

```
for all a ∈ S do
    INSERT(a, h)
end for
function INSERT(a, h)
    n ← size(h) +1
    A(n) ← a
    i ← n
    j ← n
    done ← (i = 1)
    while ¬done do
        j ← ⌊i/2⌋
        if A(j) > A(i) then
            swap(A(i), A(j))
            i ← j
        else
            done ← true
        end if
    end while
end function
```

4.2.1 Laufzeit

Initialisierung: $\mathcal{O}(n \cdot \text{Höhe}(H))$
eigentliche Sortierung: $\mathcal{O}(n \cdot \text{Höhe}(H)$
Höhe(H) ist $\mathcal{O}(\log n)$, da Baum ausgewogen ist.

`Satz 4.2.2.` *Lemma: Hat ein ausgewogener Baum die Höhe k, so hat er mindestens 2^k Knoten.*

Beweis. Der kleinste Baum der Höhe k hat nur ein Blatt mit Abstand k zur Wurzel. Auf Stufe i gibt es 2^i Knoten für $0 \leq i \leq k-1$ plus einen Knoten auf Stufe k. Damit ist die Anzahl der Knoten

$$\sum_{i=0}^{k-1} 2^i + 1 = 2^k$$

□

Damit ist $n \geq 2^k$ und $\log n \geq k = \text{Höhe}(H)$

`Satz 4.2.3.` *HeapSort läuft in Zeit $\mathcal{O}(n \log n)$, auch im Worst-Case.*

Beachte $\overline{QS} \leq 2 \cdot n \log n$
Frage: Wie ist HeapSort-Konstante?
(Diese Frage wurde von Prof. Kaufmann offen gelassen ;))

4.3 MergeSort

Prinzip: mische 2 sortierte Folgen zusammen zu einer.
$\left.\begin{matrix}(1,2,4,7)\\(5,6,8)\end{matrix}\right\} \to (1,2,4,5,6,7,8)$
Es werden immer die aktuell minimalen Elemente verglichen $(\mathcal{O}(1))$ und dann das kleinste ausgeschrieben.
Idee:
Starte mit Teilfolgen der Länge 1. (n Teilfolgen)
Mache daraus Teilfolgen der Länge 2 ($\frac{n}{2}$ Teilfolgen)
usw. bis aus 2 Teilfolgen eine entsteht.
Laufzeit:
Teilfolgen haben nach der i-ten Iteration die Länge $\leq 2^i$. Ist $2^i \geq n \Rightarrow i \geq \log n$ Iterationen
Pro Iteration mache $\mathcal{O}(n)$ Vergleiche. $\Rightarrow$ Insgesamt $\mathcal{O}(n \log n)$ Vergleiche.
Allgemeiner:
„m-Wege-Mischen": $\mathcal{O}(\log_m n)$ Iterationen. Dabei fügt man immer m Folgen zusammen.

4.4 Bucket Sort

(Sortieren durch Fachverteilung)
Gegeben: Wörter über Alphabet Σ. Sortiere lexikographisch. Als Ordnung gilt

$$a < B$$
$$a < aa < aba$$
$$\text{mit } |\Sigma| = m$$

1. Fall: Alle Wörter haben Länge 1. Stelle m Fächer bereit (für $a, b, c, \ldots, z$). Diese Fächer müssen sortiert sein. Wirf jedes Wort in entspr. Fach. Konkateniere die Inhalte der Fächer.
   $\Rightarrow$ Laufzeit: $\mathcal{O}(n) + \mathcal{O}(m)$
   Implemetiere einzelne Fächer als lin. Listen.

2. Fall: Alle Wörter haben Länge k. Sei Wort $a^i = a_1^i a_2^i \cdots a_k^i$ das i-te Wort, $1 \leq i \leq n$
   Idee: Sortiere zuerst nach dem letzten Zeichen, dann dem vorletzten, usw. Damit sind Elemente, die zum Schluss ins gleiche Fächer fallen, richtig geordnet.
   Laufzeit: $\mathcal{O}((n+m)k)$
   Problem: Wollen leere Listen überspringen beim Aufsammeln der Listen. Also Ziel: statt $\mathcal{O}((n+m)k)$ besser $\mathcal{O}(n \cdot k + m)$.
   Erzeugen Paare $\left(j, a_j^i\right),\ 1 \leq i \leq n,\ 1 \leq j \leq k$.
   Sortiere nach 2. Komp., dann nach der ersten, dann liegen im j-ten Fach die Buchstaben sortiert, die an der j-ten Stelle vorkommen.
   $\Rightarrow$ leere Fächer werden übersprungen.
   $\Rightarrow$ Laufzeit: $\mathcal{O}(n \cdot k + m)$

3. Fall: (Der allgemeine Fall)
   Wort a^i habe Länge l_i Sei

   $$L = \sum_{i=1}^{n} l_i$$
   $$R_{max} = \max\{l_i\}$$

   Idee: Sortiere Wörter der Länge nach und beziehen zuerst die langen Wörter ein.
   Algorithmus:

   a) Erzeuge Paare $\left(l_i,\ \text{Verweis auf } a^i\right)$

   b) Sortiere Paare durch Bucketsort nach 1.Komp

   c) Erzeuge L Paare $\left(j, a_j^i\right), 1 \leq i \leq n, 1 \leq j \leq l_i$ und sortiere zuerst nach 2. Komp., dann nach 1. Komp. Diese liefert lineare Listen Nichtleer(j), $1 \leq$

$j \leq r_{max}$

d) Sortiere nun a^is durch Bucketsort wie oben unter Berücksicht von $L(k)$

`Satz 4.4.1.` *Bucketsort sortiert n Wörter der Gesamtlänge L über Alphabet $\sum$, $|\sum| = m$ in Zeit $\mathcal{O}(m + L)$*

Beispiel für den 2. Fall:

`Beispiel 1.` *124, 223, 324, 321, 123*

Fächer	1	2	3	4
1.Iter.	321		223 123	124
2.Iter.		321 223 123 124 224		
3.Iter.	123 124	223 224	321	

4.5 Auswahlproblem

Finde den Median (mittleres Element)
formal: Gegeben Menge S von n Zahlen und Zahl $i, 1 \leq i \leq n$. Finde x mit $|\{y|y > x\}| = i$

1. Idee: Sortiere und ziehe dann das i-te Element $\Rightarrow \mathcal{O}(n \log n)$

Bessere Version:

1. Teile n Elemente in $\frac{n}{5}$ Gruppen der Größe 5, sortiere sie und bestimme Median jeder Gruppe.
2. Suche rekursiv den Median x der Medianе.
3. Teile Menge S bezüglich x in 2 Mengen S_1, S_2. Sei $|S_1 \cup x| = k$ und $|S_2| = n - k$
4. ```
   If i = k then return x
   else if i < k then Auswahl  (S_1,  i)
   else Auswahl  (S_2,  i − k)
   ```

Wieso Mediane?
$\frac{1}{2} \cdot \frac{n}{5}$ der Medianе sind größer als x. Die Gruppen dieser Medianе liefern jeweils 3
   ```

Elemente, die größer sind als x; also sind mindestens $\frac{3}{10}n$ Elemente größer als x und genauso $\frac{3}{10}n$ kleiner.
$\Rightarrow |S_1|, |S_2| \leq \frac{6n}{10}$
Um $\frac{6}{10}$ schrumpft die Menge rekursiv mindestens.
Laufzeit:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{falls n konstant ist} \\ T(\frac{n}{5}) + T(\frac{5n}{10}) + c \cdot n & \text{sonst} \end{cases}$$

Behauptung: $T(n) = c \cdot n$ für c konstant, also $T(n) = \mathcal{O}(n)$

5 Dynamische Mengen

Operationen: Einfügen, Streichen, Vereinigen, Spalten, etc.

5.1 Suchbäume

Beispiel: $\{2, 3, 5, 7, 11, 13, 17\}$ (Abbildung 5.1,Abbildung 5.2).

Knoten haben:

- Key-Element
- Verweise zu Kindern (lson, rson)
- oft Verweis zu parent

5.1.1 Knotenorientierte Speicherung

- 1 Element pro Knoten
- Alle Elemente im linken Teilbaum von Knoten v sind kleiner als alle im rechten Teilbaum
- keine Elemente in den Blättern

Dies ist die weiterhin verwendete Speicherform in diesem Skript.

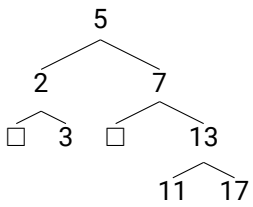
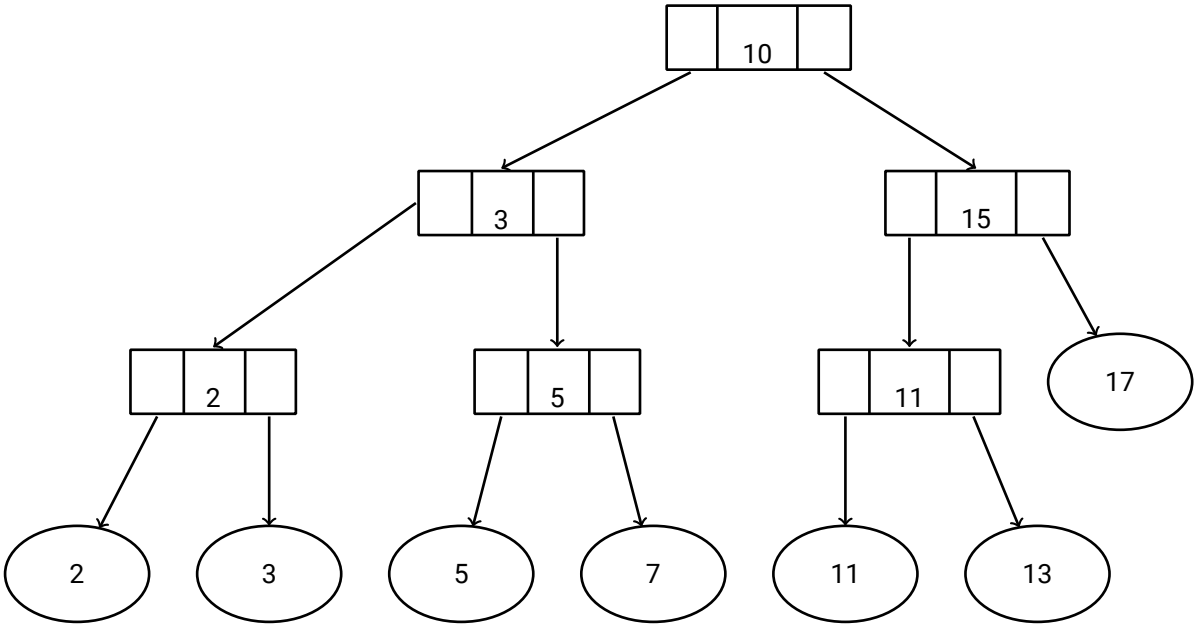


Abbildung 5.1: Knotenorientierter Suchbaum

5.1.2 EXKURS: Blattorientierte Speicherung

1 Element pro Blatt, Elemente aus linkem Teilbaum $\leq$ (Hilfs-) Schlüssel an $v \leq$ Element aus rechtem Teilbaum. Ein Beispiel für diese Speicherung befindet sich in Abbildung 5.2.



Anmerkung:
Beispielsweise 10 ist `nicht` Element dieses Baumes. sie wird lediglich als Hilfsschlüssel verwendet. 11 dagegen ist sowohl Schlüssel, als auch Element.

Abbildung 5.2: Blattorientierte Speicherung

5.1.3 Suchen in Suchbäumen

Beschrieben in Algorithmus 3

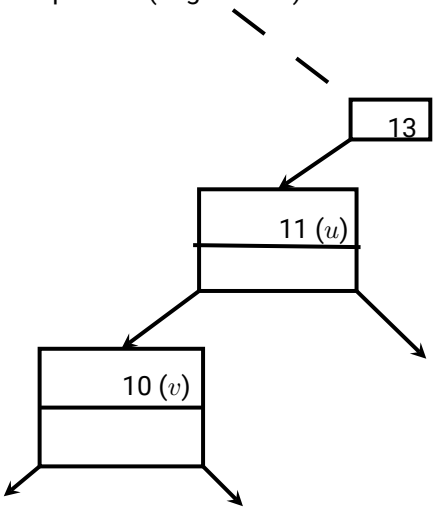
5.1.4 Einfügen in Suchbäumen

Einfügen (x) : zuerstSuchen(x)
Sei u der zuletzt besuchte Knoten u hat ≤ 1 Kind (oder $x \in S$).
Falls $x <$ key(u), erzeuge neues Kind v von u mit key(v) = x als lson(u), andernfalls als

Algorithmus 3 Suchen in Suchbäumen

```
1: function SUCHEN(x)
2:     u ← Wurzel
3:     found ← false
4:     while u exists and !found do
5:         if KEY(u) = x then
6:             found ← true
7:         else
8:             if KEY(u) > x then
9:                 u ← LSON(u)
10:            else
11:                u ← RSON(u)
12:            end if
13:        end if
14:    end while
15:    RETURN(found)
16: end function
```

rson (u).
Beispielhaft (Füge 10 ein):



5.1.5 Löschen in Suchbäumen

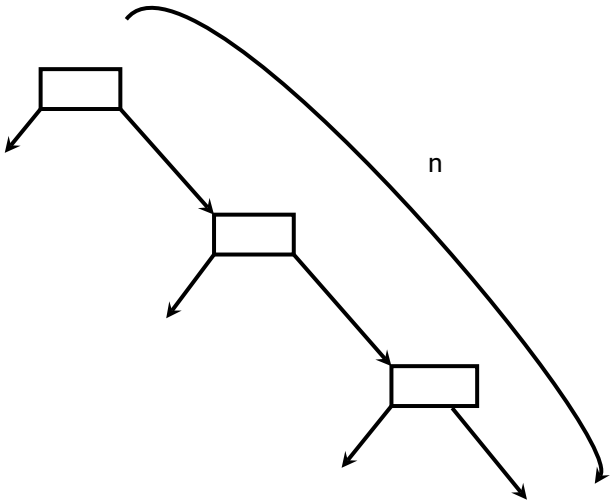
Streichen (x); zuerst Suchen(x),
Suchen endet in Knoten $u,\ x \in S$

```
1. u ist Blatt
streiche u, rson, lson, Verweis von parent(v) auf undef.
2. u hat 1 Kind
Streiche u, setze w als Kind von parent(u) an die Stelle von u.
3. u hat 2 Kinder
Suche v mit größtem Info-Element im linken Teilbaum von u,
(einmal nach links, dann immer nach rechts!  ⇒  v hat keinen rson)
Ersetze u durch v & Streiche v unten (wie Fall 1 oder 2)
```

Laufzeit $\mathcal{O}(h+1)$, wobei h Höhe des Suchbaums ist.

5.1.6 Diskussion

Ein Suchbaum kann sich sehr schlecht verhalten. Werden etwa aufsteigende Zahlen eingefügt wie beschrieben, so ergibt sich eine entartete Baumform, die stark an eine Liste erinnert. Beispiel:



.....

Idee, dies zu lösen:

1 Hoffe, dass es nicht vorkommt. (Unwahrscheinlich, dass ein solcher Input erfolgt)

2 Baue Baum von Zeit zu Zeit neu auf.

5.2 Balancierte Bäume

Sei u Knoten im Suchbaum. Die Balance von u sei definiert als

$$Bal(u) = \text{Höhe(rson)} - \text{Höhe(lson)}$$

Setze Höhe des undefinierten TBs $= -1$.
Beispiele s. Abbildung 5.3

Abbildung 5.3: $Bal(u) = 0 - 1 = -1$, $Bal(x) = 0 - 2 = -2$

`Definition 5.2.1.` *Ein binärer Baum T heißt AVL-Baum, falls gilt*

$$\forall u \in T : |Bal(u)| \leq 1$$

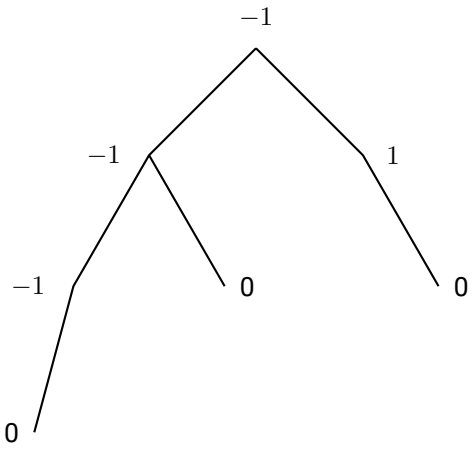


Abbildung 5.4: Ein AVL Baum. An den Knoten ist die Balance des jeweiligen Teilbaums angegeben.

`Definition 5.2.2.` *Fibonacci-Bäume (Abbildung 5.5)*

$$T_0, T_1, T_2, \ldots$$

sind definiert durch $T_0 =$ *leer,* $T_1 = \bullet$, $T_2 = /$,
T_h *siehe Abbildung 5.5*
Fibonacci-Zahlen:

$$F_0 = 0, F_1 = 1, F_h = F_{h-1} + F_{h-2}$$

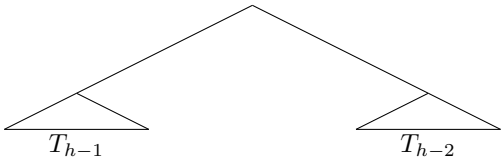


Abbildung 5.5: Fibonacci-Baum

Behauptung: T_h enthält genau F_h Blätter (für alle $h \geq 0$)
Beweis per Induktion. (klar)
Behauptung: AVL-Bäume der Höhe h haben $\geq F_h$ Blätter.

Beweis. » $h = 0$ • gilt für einen Knoten

» $h = 1$ / oder \ oder /\

» $h \geq 2$: Erhalten den blattminimalen AVl-Baum der Höhe h durch Kombination von blattmin. AVL-Bäumen der Höhe $h-1$ und $h-2$.

$\Rightarrow \geq F_{h-1} + F_{h-2} = F_h$ Blätter.
$F_h = h$-te Fibonacci -Zahl.

$$F_h = \frac{\alpha^h - \beta^h}{\sqrt{5}}$$

mit $\alpha = \frac{1+\sqrt{5}}{2}, \beta = \frac{1-\sqrt{5}}{2}$.

□

`Lemma 5.2.3.` *AVL-Bäume mit* n *Knoten haben Höhe* $\mathcal{O}(\log n)$

Beweis. Baum hat $\leq n$ Blätter, also $F_h \leq n$, damit gilt

$$\frac{\alpha^h - \beta^h}{\sqrt{5}} \leq n$$

Da $|\beta| < 1$ gilt:

$$\begin{aligned}
\frac{\alpha^h - \beta^h}{\sqrt{5}} &\geq \frac{\alpha^h}{2\sqrt{5}} \\
\Rightarrow \alpha^h &\leq 2\sqrt{5} \cdot n. \\
h \cdot \log \alpha &< \log(2\sqrt{5}) + \log n \\
h &\leq \frac{\log 2\sqrt{5} + \log n}{\log \alpha} = \mathcal{O}(\log n)
\end{aligned}$$

□

Beim Einfügen/Streichen kann die Balance gestört werden, z.B. auf -2 oder 2 (siehe anschließendes Beispiel). Wir fordern aber für einen AVL Baum, dass ständig gilt

$$\forall u \in T : |Bal(u)| \leq 1$$

und wissen von Fibonacci-Bäumen, dass die Höhe $\mathcal{O}(\log n)$ beträgt. Es müssen also bei Operationen am Baum Korrekturen ausgeführt werden, dass die AVL-Eigenschaften erhalten bleiben.

5.2.1 Einfügen(w) in AVL-Bäumen

Angenommen es wurde die AVL-Bedingung zerstört und es ist eine Balance von ± 2 entstanden. Dann sei u der tiefste dieser „unbal." Knoten. O.B.d.A. sei $Bal(u) = 2$

Sei w der neue Knoten. Der wurde rechts eingefügt. Sei v rechtes Kind von u

Fall 1) Bal(v) = 1

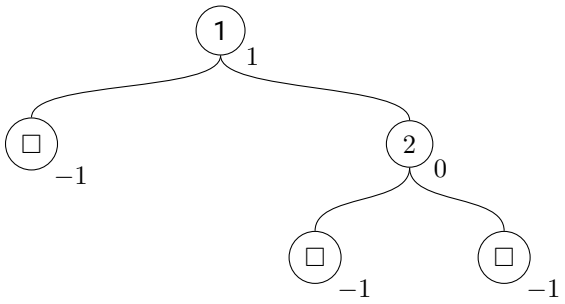
Es gilt: 1) Links-Rechts Ordnung wird aufrechterhalten:
$T_L \leq u \leq T_A \leq v \leq T_B$
2) Nach Rotation haben u und v Balance 0.
Knoten in T_A, T_B, T_L behalten ihre alte Balance.
3) Höhe(v) nach Rotation = Höhe(u) vor Einfügen(w).

Fall 2) Bal(v) = -1
Nach Einfügen von w folge dem Pfad von w zur Wurzel.
Berechne auf diesem Pfad alle Balancen neu.
Tritt Bal. -2/2 auf, führe Rotation/Doppelrotation aus.
Laufzeit: $\mathcal{O}(\log n)$ (Rot./Doppelrot. in $\mathcal{O}(1)$)

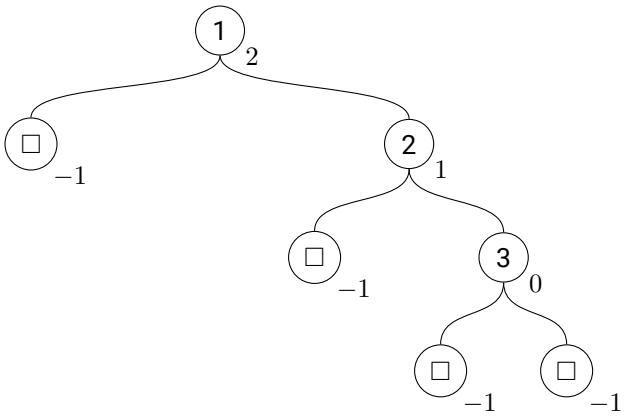
Fall 3) Bal(v) = 0
Kann nicht vorkommen. AVL-Bedingung vorher schon verletzt.

Beispiel für Einfügeoperation

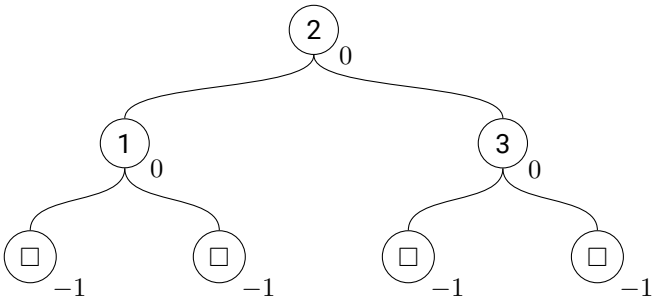
Es soll in den folgenden Baum das Element 3 eingefügt werden:



Dieser hat jedoch nach einer Einfügung die nachfolgende Gestalt



Da nun am obersten Knoten die Balance 2 entstanden ist, muss an der Wurzel rotiert werden. Dies führt zu folgendem Baum:



Es sind dabei an allen Knoten die jeweiligen Balancen angegeben.

5.2.2 Streichen(w) in AVL-Bäumen

Annahme: es gibt Knoten mit Bal ± 2.
Sei u der tiefste solche Knoten.
O.B.d.A. sei $Bal(u) = 2$.
Sei v das rechte Kind von u

Fall 1) Bal(v) = 0
Höhe insgesamt hat sich nicht geändert gegenüber vor Streichen. Können hier abbrechen!

Fall 2) Bal(v) = 1
Beobachte: Der Teilbaum hat jetzt kleinere Höhe als zuvor. Balancen oben drüber ändern sich!
$\Rightarrow$ Iteriere Rebal.-Prozess weiter oben.

`Satz 5.2.4.` *Balancierte Bäume (AVL) erlauben Suchen/Einfügen/Streichen in Zeit* $\mathcal{O}(\log n)$*, wobei* n *Zahl der Knoten.*

5.2.3 Anwendung (Schnitt von achsenparallelen Liniensegmenten)

Ziel: Anzahl der Schnittpunkte von achsenparallelen Liniensegmenten berechnen. Die Lage wird als allgemein angenommen. Ein naiver Algorithmus, der paarweise alle Elemente vergleicht hätte eine Laufzeit von $\mathcal{O}(n^2)$. Wir wählen also eine bessere Alternative:

PlaneSweep

x-Struktur: geordnete Liste von Endpunkten nach x-Koor. statisch, durch Sortieren erzeugt.
y-Struktur: repräsentiert einen Zustand der Sweepline L (dynamisch). Speichern in L horiz. Segmente, die im Moment von L gekreuzt werden geordnet nach y-Koordinate.
$\rightarrow$ AVL-Baum unterstützt Einf./Streichen in Zeit $\mathcal{O}(\log n)$

Vertikale Segmente: (x, y_u, y_o)
Wollen berechnen # horiz. Segmente mit y-Koord. zwischen y_u und y_o
$\rightarrow$ berechne Rang(y_u), Rang(y_o). (# Elemente, die kleiner sind)
Rang(y_o) - Rang(y_u) = # Schnittpunkt auf vert. Segment.
zu tun: Bestimme Rang(x) in AVL-Baum.

Merke in jedem Knoten die Zahl der Knoten im linken Teilbaum (l count).
Suchen(x): Beim Rechtsabbiegen erhöhe Rangzähler um (l count + 1) $\rightarrow$ Rang(x) in

$\mathcal{O}(\log n)$
$\Rightarrow$ Gesamtlaufzeit: $\mathcal{O}(n \cdot \log n)$
Wollen wir die Kreuzungen explizit bestimmen, dann brauchen wir eine Laufzeit von $\mathcal{O}(n \cdot \log n + k)$

5.3 (2,4)-Bäume (blattorientiert)

(2,4)-Bäume gehören zu den (a,b)-Bäumen.

`Definition 5.3.1.` *Seien $a, b \in \mathbb{N}$ mit $a \geq 2$, $b \geq 2a - 1$. T heißt (a,b)-Baum falls gelten:*

a) alle Blätter von T haben gleiche Tiefe

b) alle Knoten haben $\leq b$ Kinder

c) alle Knoten (außer Wurzel) haben $\geq a$ Kinder

d) Wurzel hat ≥ 2 Kinder

Das Ziel ist nun die Menge $S = \{x_1 < x_2 < ... < x_n\}$ abzuspeichern. Dabei sollen die Schlüssel in den Blättern aufsteigend von links nach rechts geordnet abgespeichert werden.

1. Schlüssel in Blättern.
2. Innere Knoten v mit d Kindern hat Elemente $K_1(v), \ldots, K_{d-1}(v)$
   $k_i(v)$ = Inhalt des rechtesten Blattes im i-ten Unterbaum.

`Beispiel 2.`

$$S = \{2, 5, 7, 11, 15, 17, 19\}$$

Klar: Tiefe ist $\mathcal{O}(\log n)$

5.3.1 Suchen in (2,4)-Bäumen

Suchen(k) ✓ Suche nach k liefert Blatt $k' = \min \{x \in S \mid k \leq x\}$

5.3.2 Einfügen in (2,4)-Bäumen

Einfügen(k): Zuerst *Suchen*(k) liefert Blatt v_i mit *Schlüssel* $(v_i) < k <$ *Schlüssel*(v_i)

`Sei` w `der parent von` v `:`

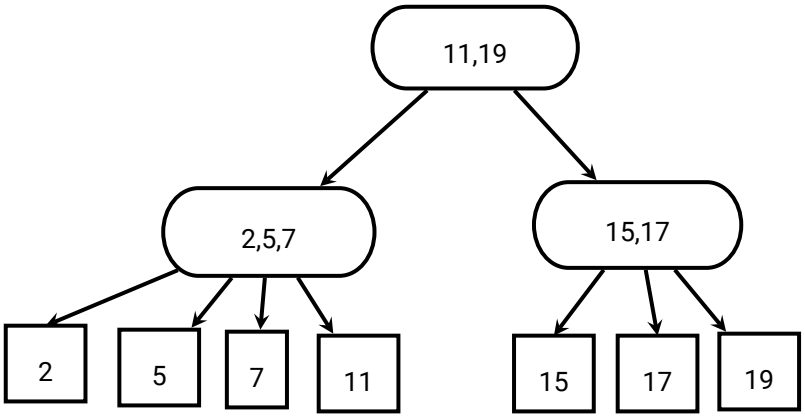


Abbildung 5.6: (2,4)-Baum

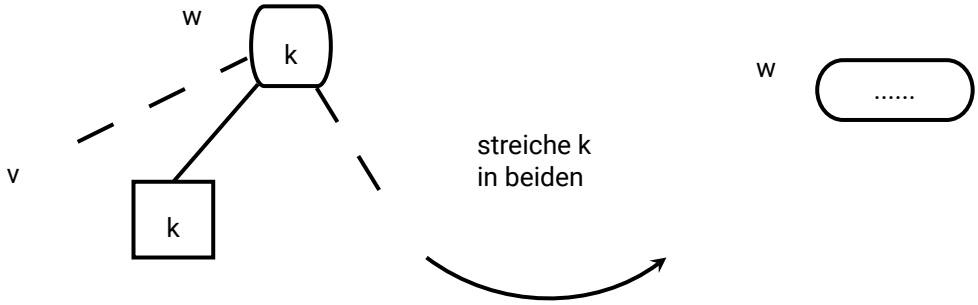
```
while  w  hat 5 kinder do Spalte( w )
 w   ⇐  parent( w )
```

Laufzeit: $\mathcal{O}(1 +$ # Spaltungen)

5.3.3 Streichen in (2,4)-Bäumen

Streiche(k) : Zuerst Suchen$(k) \rightarrow$ Endet in Blatt v mit Schlüssel k.

1. Fall: K steht auch in parent$(v) = w$



2. v rechtestes Kind w sei Vorgänger von v und $v^{'}$ der linke Nachbar mit Schlüssel $k^{'}$. Streiche v und Ersetzte Vorkommen von k durch $k^{'}$. Streiche $k^{'}$ im $parent(w)$, da nun höchster Schlüssel. (vgl. Abbildung 5.7)

```
while w hat ein Kind
```

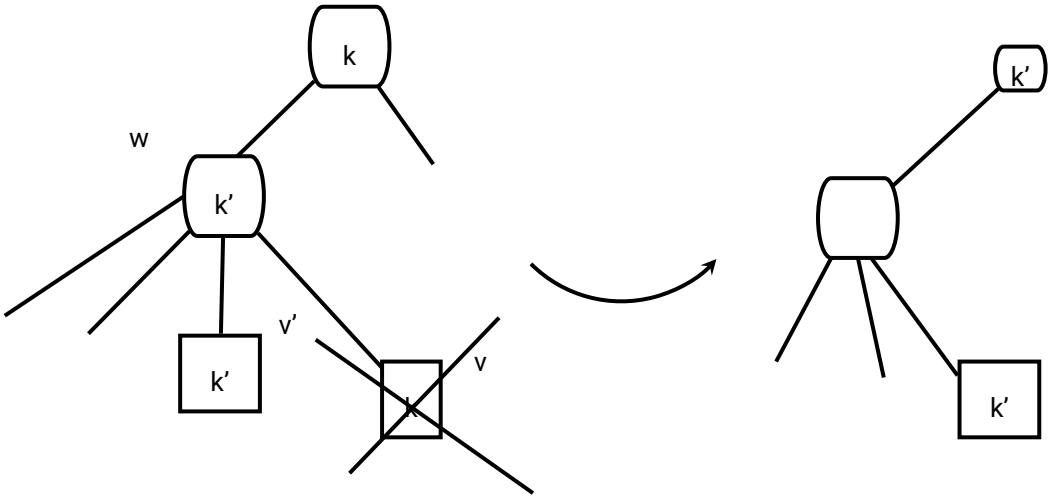
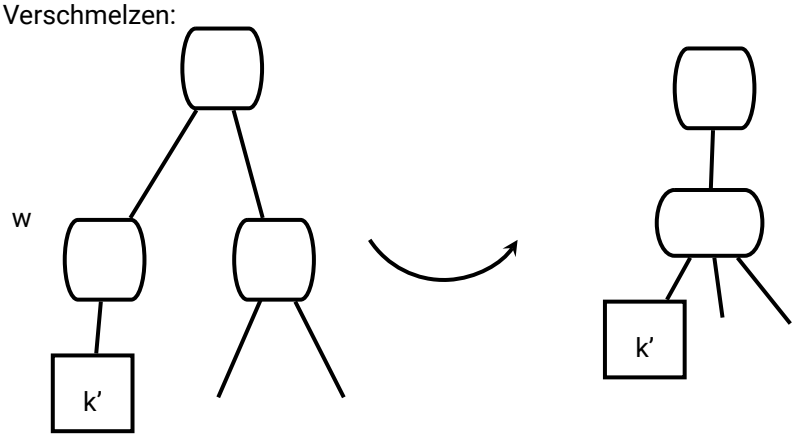


Abbildung 5.7: Bsp. v rechtestes Kind von $parent(v)$ streichen (blattorientiert)

`do Verschmelze oder Stehlen od`

Verschmelzen:

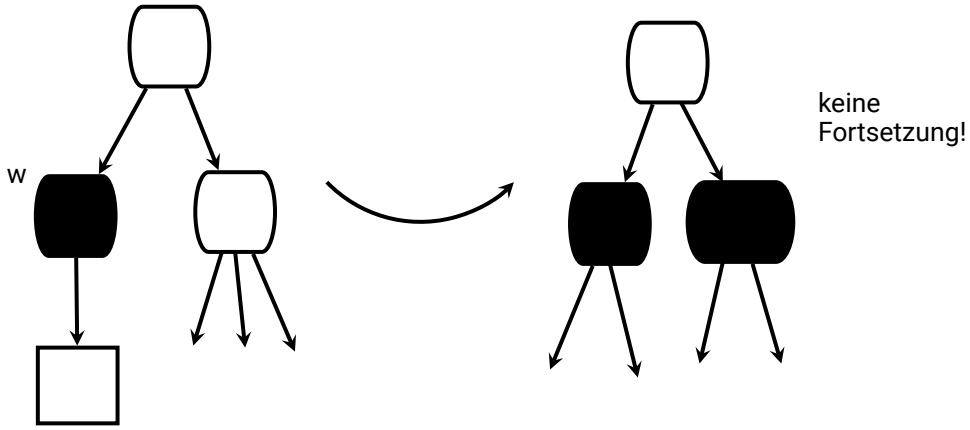


Falls Geschwisterknoten Grad 2 hat.

verschmelze Geschw.-knoten zu einem Knoten Grad 3.

Setze fort!

Stehlen: Geschwisterknoten von w hat 3 oder 4 Kinder: Gibt uns an w ab.



5.3.4 Laufzeitanalyse

Nach Spaltung hat der Baum eine bessere Struktur, deshalb: *Amortisierte Analyse:*
(= Kosten pro Operation gemittelt über n Operationen)

`Lemma 5.3.2.` *In einem (2,4) Baum sind die amortisierten Kosten der Operationen Einfügen/Streichen* $\mathcal{O}(1)$.

Beweis. Beschreibung des Zustands des Baums T:
pot(T)

$$
\begin{aligned}
= \quad & 2 \cdot \text{\# Knoten von Grad } 1 \\
+ \quad & 1 \cdot \text{\# Knoten von Grad } 2 \\
+ \quad & 0 \cdot \text{\# Knoten von Grad } 3 \\
+ \quad & 2 \cdot \text{\# Knoten von Grad } 4 \\
+ \quad & 4 \cdot \text{\# Knoten von Grad } 5
\end{aligned}
$$

Invariante:

- $pot(T) \geq 0$
- bei Spalten/Verschmelzen/Stehlen ist nur ein Knoten nicht auf Grad 2,3,4
- Vor Einfügen /Streichen haben alle Knoten den Grad 2,3,4.

Einzeloperationen haben Kosten von $\mathcal{O}(1)$.
Behauptung: Spalten / Verschmelzen verringern Potential. Stehlen erhöht es nicht.
Beweis:
Stehlen. Knoten w trug 2 Einheiten zum Potential bei und sein Nachbarknoten p. Danach trägt w 1 Einheit bei, Nachbar $\leq p + 1$. □

Spalten: w trägt 4 bei zum Potential, der parent trägt p bei. Danach haben die beiden neuen Knoten 0 und 1, der parent hat $\leq p + 2$ ✓.

Verschmelzen: w trägt 2 bei, Geschwisterknoten von w trägt 1 bei, der parent trägt p bei
$\Rightarrow$ Danach 0 und $\leq p + 1$.

Amortisierte Laufzeit für Einfügen: Tatsächliche Kosten + Potentialerhöhung.
Tatsächliche Kosten 1 + Potentialerhöhung ≤ 2
Folge von f Spaltungen: tatsächliche Kosten f, Potentialerhöhung $\leq f$.
$\Rightarrow$ Amortisierte Kosten von Einfügen: ≤ 3

Streichen analog

5.3.5 Anwendung: Sortieren

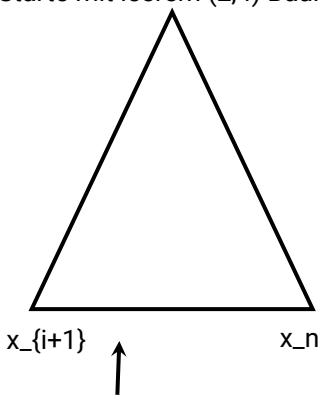
durch Einfügen: $\mathcal{O}(n \log n)$ vorsortierter Folgen: Mache Suchen billiger!
Sei $x_1, \ldots, x_2$ Folge reeller Zahlen.

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

(Zahl der Inversionen). Es gilt $O \leq F \leq \frac{n^2}{2}$

`Satz 5.3.3.` $x_1, \ldots x_n$ *kann in Zeit* $\mathcal{O}(n \log \frac{F}{n})$ *sortiert werden.*

Beweis. Sei $f_i = |\{j \mid i < j, x_i > x_j\}|$ Es gilt $F = \sum_i f_i$
Starte mit leerem (2,4) Baum. Füge Folge in umgekehrter Reihenfolge ein.



Füge $x_n, \ldots, x_1$ ein. Einfügen (x_i): Starte am linkesten Blatt; laufe hoch und drehe an der richtigen Stelle um, laufe runter und füge ein. Ist x_i klein, so laufe nicht sehr hoch.

Es gilt (amortisiert): Kosten für Einfügen $= \mathcal{O}(1 + \log f_i)$
Suchen$(x_i) = \mathcal{O}(\log f_i)$
Einfügen: = $\mathcal{O}(1)$.

Laufe hoch bis Höhe h (Umkehrpunkt):

$$2^{h-1} \leq f_i \Rightarrow h \leq 1 + \log f_i$$

$\Rightarrow$ Gesamtkosten:

$$\begin{aligned}
\sum_i \mathcal{O}(1 + \log f_i) &= \mathcal{O}(n + \sum_{i=1}^{n} \log f_i) \\
&= \mathcal{O}(n + \log \prod_{i=1}^{n} f_i \\
&= \mathcal{O}(n + \log \prod_{i=1}^{n} \frac{F}{n} \\
&\leq \mathcal{O}(n + n \log \frac{F}{n})
\end{aligned}$$

x_i wird nach dem f_i-ten Element in die bisherige Liste eingefügt. □

5.4 B-Bäume (knotenorientiert)

Anmerkung

In allen Skizzen müssten die Pointer jeweils vor und nach einem Element stehen. Dies ist wegen des erhöhten Zeichenaufwandes vernachlässigt worden.

5.4.1 Idee und Definition

Idee: Mehr Daten in einen Knoten

`Definition 5.4.1.` *B-Baum der Ordnung k. $k \geq 2$ ist ein Baum dessen:*

1 Blätter alle gleiche Tiefe haben

2 Wurzel ≥ 2 Kinder und dessen andere inneren Knoten $\geq k$ Kinder Haben.

3 innere Knoten $\leq 2k - 1$ Kinder haben.

Höhe eines B-Baums:

Lemma 5.4.2. *Sei T ein Baum der Ordnung k mit Höhe h und n Blättern. Dann gilt:*

$$2 \cdot k^{h-1} \leq n \leq (2k-1)^h$$

Beweis. Zahl der Blätter ist minimal, wenn jeweils die Minimal-Anzahl von Kindern vorkommen und maximal, wenn jeweils die Maximalzahl vorkommt.

$$2 \cdot \underbrace{k \cdot k \ldots \cdot k}_{h-1} \text{ mal} = n_{\text{min}} \leq n \leq n_{\text{max}} = (2k-1)^h$$

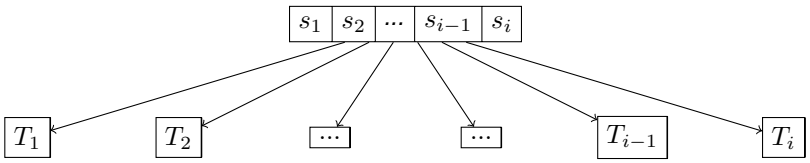
□

Es gilt somit $\log_{(2k-1)} n \leq h \leq 1 + \log_k(\frac{n}{2})$ *(logarithmieren!)*
Operationen: Suchen(x), Einfügen(x), Streichen(x)
Sei u Knoten mit k Kindern und Schlüsseln

$$s_1 < s_2 < \ldots < s_{l-1}$$

mit Unterbäumen

$$T_1, \ldots, T_l$$



Es gilt für alle $v \in T_i$ und Schlüssel s in v:

$$s \leq s_i \text{ falls } i = 1$$
$$s_{i-1} < s \leq s_i \text{ falls } 1 < i < l$$
$$s_{i-1} < s \text{ falls } i = l$$

Algorithmus:

```
Suchen(x): Starte in w= Wurzel

suche kleinstes  s_i  in w mit  x  ≤  s_i

falls  s_i  existiert, dann
if  x =  s_i  then gefunden
```

```
else w  ←  i-tes Kind von w , suche dort weiter
falls  s_i  nicht existiert, dann
w  ←  rechtestes Kind von w, suche dort weiter.
```

Laufzeit: $\mathcal{O}(\log_k n \cdot k)$ (Höhe mal Suche innerhalb eines Knotens)

5.4.2 Einfügen(x)

Suchen(x) endet in Blatt b mit parent v, der l Kinder hat. Nun müssen 2 Fälle unterschieden werden.

1. In Blatt b ist noch genügend Platz für eine Einfügung vorhanden. Dann füge x in b ein.
2. Nach Einfügung von x hätte Blatt b $2k - 1$ Elemente: Teile auf.

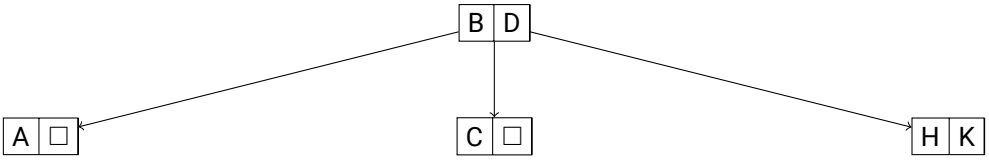
Aufteilen bei Einfügung: Sollte bei einer Einfügung der 2. Fall aufgetreten sein, muss der Blattknoten geteilt werden. Sortiere dazu die Blattelemente aus b und entferne den Median m. Bilde aus den Elementen e mit $e < m$ das Blatt b und aus den verbleibenden Elementen das Blatt b'. Füge dann m als neuen Seperator in den Vaterknoten v ein. Die Blätter b und b' stehen dann jeweils links und rechts von m. Ein Beispiel mit konkreten Werten und Erklärungen findet sich im nächsten Kapitel.

Beachte: Dabei kann der Vaterknoten v ebenfalls mehr als $2k - 2$ Elemente enthalten. Die Einfügung muss also rekursiv nach oben fortgesetzt werden. Wird dabei die Wurzel aufgespalten, so ergibt sich eine neue Wurzel mit 2 Kindern und Tiefe wächst um 1. Dies ist die einzige Möglichkeit, mit der die Tiefe eines B-Baums wachsen kann.

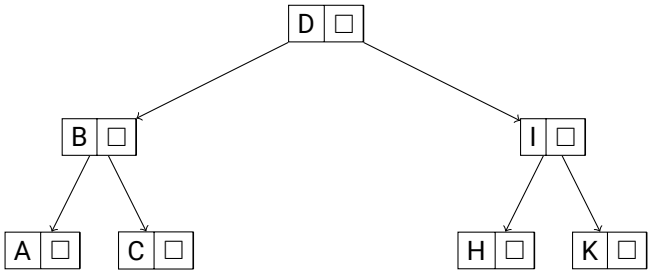
5.4.3 Konkrete Beispiele für die Operationen

Einfügung mit Split

Es wird zu beginn vom folgenden B-Baum ausgegangen:



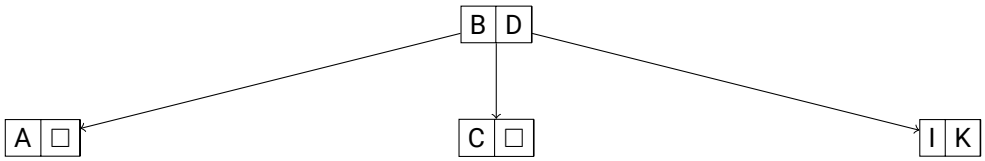
Es soll nun der Wert I eingefügt werden. Dies führt zu mehreren Splits und dem B-Baum



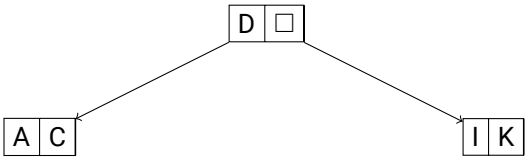
Nachdem I eingefügt wurde, war der Knoten H, I, K überbelegt. Das mittlere Element wird nach oben geschoben. Auch dort ist der Knoten nun überbelegt und wird gesplittet. Dabei wächst die Tiefe des Baums um 1.

Löschen mit Merge

Es wird zu beginn vom folgenden B-Baum ausgegangen:



Es soll nun der Wert B gelöscht werden



Nachdem B gelöscht wird, müssen Kindknoten gemerged werden.

5.5 Randomisierte Suchbäume

» Treaps(Aragorn/Seidel)

» Skiplists (Pugh)

5.5.1 Skiplists

Gegeben: $S = \{x_1, \ldots, x_n\}$, $p = (\frac{1}{2})$
Definiere Datenstruktur bezüglich Leveling aus Listen:
Füge $\pm\infty$ als Spezialelemente in alle Listen ein. Elemente der Liste L_i werden von links nach rechts verbunden. Verbinde zusätzlich alle Elemente x vom höchsten Level bis Level 1. Element x bildet also einen Turm mit $l(x)$ Knoten. Wir haben nun eine Baumstruktur mit Intervallen, die auf Level $i+1$ mehrere Intervalle auf Level i umfassen. Dabei sei $c(I)$ die Zahl der Kinder von Intervall I

$S = L_1 \supseteq L_2 \supseteq \ldots \supseteq L_r = \emptyset$

Zufälliges Leveling:

L_{i+1} ergibt sich aus L_i indem jedes Element $x \in L_i$ mit Wahrscheinlichkeit $p = \frac{1}{2}$ an L_{i+1} weitergereicht wird. $l(x)$ für $x \in S$ sind unabhängige Zufallsvariable nach einer Geometrischen Verteilung mit $p = \frac{1}{2}$.

Geometische Verteilung:
Wirf Münze, bis Zahl fällt. X ist Anzahl der benötigten Würfe.

$E(X) = \frac{1}{p}, Var(X) = \frac{1-p}{p}$

Erwarteter Platzbedarf für zufällige Skiplist mit $|S| = n$ ist also $2n = \mathcal{O}(n)$

`Lemma 5.5.1.` *Zahl der Level r bei zufälligem Leveling hat Erwartungswert* $E(r) = \mathcal{O}(\log n)$. *Es gilt* $r = \mathcal{O}(\log n)$ *mit hoher Wahrscheinlichkeit.*

Beweis. Sei $r = \max\{l(x) | x \in S\}$ und $l(x)$ sind geometrisch verteilte Zufallsvariablen mit $p = \frac{1}{2}$. Sei X_i die Zufallsvariable für $x_i \in S$.
Es gilt:
$P(X_i > t) < (1-p)^t$ und $P(\max_i X_i > t) < n \cdot (1-p)^t = \frac{n}{2^t}$ für $p = \frac{1}{2}$
Mit $t = \alpha \log n$ und $r = 1 + \max_i X_i \Rightarrow P(r > \alpha \log n) \leq \frac{1}{n^{\alpha \log n - 1}}$ □

Suchen(x)

laufe Intervalle levelweise (r$\rightarrow$ 0) ab:
Sei $I_j(x)$ das Intervall auf Level j, das x enthält.
Liegt x auf dem Rand von zwei Intervallen, weise x dem linken Intervall zu.
Folge $I_r(x) \supseteq I_{r-i}(X) \supseteq \cdots \supseteq I_1(x)$ lässt sich als Pfad von Wurzel zum Blatt ansehen.

Laufzeit

$$\mathcal{O}\left(\sum_{j=0}^{r} 1 + c\left(I_j(y)\right)\right)$$

Anzahl Level: $\mathcal{O}(\log n)$, Anzahl Schritte nach unten.
Der Suchpfad von x zurück zu Level v wird beschrieben durch:
Gehe nach oben, wenn möglich; wenn nicht gehe nach links.
Schritt nach links gibt es mit Wahrscheinlichkeit $\frac{1}{2}$ jeweils $\Rightarrow$ Erwartete Anzahl der Schritte nach links ist genauso groß, wie erwartete Anzahl der Schritte nach oben. $\Rightarrow$ Laufzeit $\mathcal{O}(\log n)$

Einfügen(x)

Suche zuerst nach x. Füge es in L_1 ein und hänge zwei Zeiger um $\Rightarrow \mathcal{O}(\log n)$
Laufe hoch und mache Zufallsexperiment, füge nacheinander eventuell Kopien ein.
Ist $l(x) > r$, kreiere neues Level, $l(x) \leftarrow r + 1$, $r \leftarrow r + 1$. Bestimme Suchpfade durch Intervalle von $l(x)$ aus und spalte durch Einfügen von x. Erwartete Zahl von Kopie ist 2 $\Rightarrow \mathcal{O}(1)$ erwartet.
$\Rightarrow$gesamt $\mathcal{O}(\log n)$

Streichen(x)

Während dem Suchlauf nach x lösche von oben nach unten alle Vorkommen von x durch Zeigerumhängen. Laufzeit: $\mathcal{O}(\log n)$.

`Satz 5.5.2.` *In zufälliger Skiplist für S der Größe n können Suchen/ Einfügen/ Streichen in erwarteter Zeit von $\mathcal{O}(\log n)$ ausgeführt werden.*

Weitere Operationen:

join(S_1, x, S_2): ersetzte S_1, S_2 durch $S_1 \cup \{x\} \cup S_2$ und $\forall y \in S_1, z \in S_2 : y < x < z$
Idee: hänge $L_i(S_1), L_i(S_2)$ aneinander $\forall i$ und füge dann x ein

paste(S_1, S_2): analog

split(x, S): Umkehrung von join()
Idee: streiche x, verweise auf $\pm\infty$ statt nächstes Element

Alles in $\mathcal{O}(\log n)$!

Beispiel

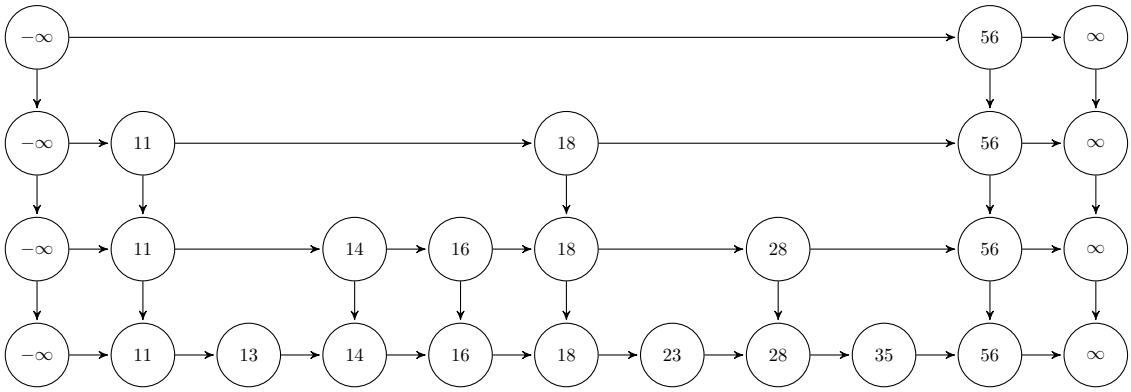
Es sollen beispielhaft die Elemente

$$23, 16, 18, 35, 11, 28, 56, 13, 14$$

mit Hilfe der Zufallsbitfolge

$$10100110010100011101$$

in eine Skiplist eingefügt werden. Dies ergibt die folgende Skiplist.



6 Hashing

6.1 Definition

Sei U ein Universum und $S \subset U$ eine Schlüsselmenge mit $|S| << |U|$. Elemente werden in eine Hashtafel T gespeichert. Wir wollen nun die Operationen

- Zugriff(a,S)
- Einfügen(a,S)
- Streichen(a,S)

unterstützen. Dabei definieren wir

- Hashtafel $T[0, 1, ..., m-1]$
- Universum $U = [0, 1, ..., N-1]$
- Hashfunktion $h : U \to [0, 1, ..., m-1]$ als

$$a \mapsto h(a)$$

  wobei Element a an Stelle $T[h(a)]$ liegt

6.1.1 Beispiel

Seien $N = 50$, $m = 3$, $S = \{2, 21\}$ sowie

$$h(x) = x \mod 3$$

(Divisionsmethode). Dann ergibt sich die Hashtafel

0	1	2
2	21	–

6.2 Hashing mit Verkettung

Es handelt sich um eine der beiden Ideen. Die andere wäre Hashing mit offener Adressierung (s. Kapitel 6.3).

6.2.1 Idee

Jeder Tafeleintrag ist eine Liste. Die i-te Liste enthält alle $x \in S$ mit $h(x) = i$.

6.2.2 Laufzeit

WorstCase: $\mathcal{O}(|S|) = \mathcal{O}(n)$ im Mittel aber *wesentlich* besser.

Annahmen

- $h(x)$ kann ich $\mathcal{O}(1)$ ausgerechnet werden.
- $\left|h^{-1}(i)\right| = \frac{N}{m}, \forall i \in \{0, \ldots, m-1\}$
- Für eine Folge von Operationen gilt: Wahrscheinlichkeit, dass das j-te Element in der Folge ein festes $x \in U$ ist, beträgt $\frac{1}{N} \Rightarrow$ Operationen sind unabhängig und gleichverteilt
  Ist x_k Argument der k-ten Operation, so gilt: $P(h(x_k) = i) = \frac{1}{m}$ (also auch Hashfuktionswerte gleichverteilt)

Beweis für mittlere Kosten

`Definition 6.2.1.`

$$\delta_h(x, y) = \begin{cases} 1 & x \neq y \wedge h(x) = h(y) \\ 0 & \textit{sonst} \end{cases}$$

$\delta_h(x, S) = \sum_{y \in S} \delta_h(x, y)$ *(Zahl der Kollisionen mit x)*
Kosten von Operationen (Zugriff/Streichen/Einfügen) $XYZ(x) = 1 + \delta_h(x, S)$

`Satz 6.2.2.` *Die mittleren Kosten von* $XYZ(x)$ *sind* $1 + \beta = 1 + \frac{n}{m} = 1 + \frac{|S|}{|T|}$

Beweis. Sei $h(x) = i$ und p_{ik} sei die Wahrscheinlichkeit, dass Liste i genau k Elemenete enthält. Dann ist:

$$p_{ik} = \binom{n}{k} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{n-k}$$

Die mittleren Kosten sind:

$$\begin{aligned} E(\text{Listenlänge}) &= \sum_{k=0}^{n} p_{ik}(1+k) \\ &= \sum_{k=0}^{n} p_{ik} + \sum_{k=0}^{n} k\binom{n}{k}\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{n-k} \\ \left[\text{da } k\binom{n}{k} = n\binom{n-1}{k-1}\right] &= 1 + \frac{n}{m}\sum_{k=1}^{n}\binom{n-1}{k-1}\left(\frac{1}{m}\right)^{k-1}\left(1-\frac{1}{m}\right)^{n-k} \\ &= 1 + \frac{n}{m}\left(\frac{1}{m} + \left(1-\frac{1}{m}\right)\right)^{n-1} \\ &= 1 + \frac{n}{m} = 1+\beta \end{aligned}$$

□

6.2.3 Größe von β

Für $\beta \leq 1$ ist $\mathcal{O}(1)$ erwartet.
Für $\beta \geq \frac{1}{4}$ ist Platzausnutzung für Tafel gut.
Wobei folgende Laufzeiten gelten: Für die Tafel gilt $\mathcal{O}(m)$, für die Listen gilt $\mathcal{O}(n)$, also insgesamt $\mathcal{O}(n+m) \sim \mathcal{O}(n)$ für $m = \frac{n}{\beta}, \beta \geq \frac{1}{4}$. Jedoch können Einfügen, Striechen β schnell schlecht werden lassen. Wird also β zu groß oder zu klein, dann rehashen wir.

Rehashen

Benutze Folge von Hashtafeln $T_0, T_1, ...$ der Größe $m, 2m, 4m, ...$
In T_i der Größe $2^i \cdot m$ sei nun $\beta = 1$. Dann wird umgespeichert auf T_{i+1}, was zu $\beta = \frac{1}{2}$ führt. Bei $\beta = \frac{1}{4}$ gehe von T_i zu T_{i-1} was zu $\beta = \frac{1}{2}$ führt. Mit amort. Analyse kann man nun zeigen, dass mittlere Laufzeit immernoch $\mathcal{O}(n)$ ist. Auch mit Rehashing.

6.2.4 Länge der längsten Liste

Sei S zufällig aus U gewählt. Also gilt

$$\text{prob}(h(x_k) = i) = \frac{1}{m}$$

für $k \in S, i = [0, 1, ..., m-1]$. L sei die Länge der längsten Liste und $l(i)$ sei die Länge der Liste i. Es gilt nun

$$\text{prob}(l(i) \geq j) \leq \binom{n}{j} \cdot \left(\frac{1}{m}\right)^j$$

und es gilt weiter

$$\begin{aligned}\text{prob}(\max\{l(i) \geq j\}) &\leq \sum_{i=0}^{m} \text{prob}(l(i) \geq j) \\ &\leq m \cdot \binom{n}{j} \cdot \left(\frac{1}{m}\right)^j \\ &= m \cdot \frac{n!}{j!(n-j)!m^j} \\ &\leq n \cdot \left(\frac{n}{m}\right)^{j-1} \cdot \frac{1}{j!}\end{aligned}$$

Also ist der Erwartungswert

$$\begin{aligned}E(L) &= \sum_{j \geq 1} \text{prob}(\max\{l(i) \geq j\} \\ &\leq \sum_{j \geq 1} \min\left(1, n \cdot \left(\frac{n}{m}\right)^{j-1} \cdot \frac{1}{j!}\right)\end{aligned}$$

Sei

$$j_0 = \min\left\{j, n \cdot \left(\frac{n}{m}\right)^{j-1} \cdot \frac{1}{j!}\right\} \leq \min\{j, n \leq j!\}$$

da $\frac{n}{m} \leq 1$. Und mit $j! \geq \left(\frac{j}{2}\right)^{\frac{j}{2}}$ folgt nun

$$j_0 = \mathcal{O}\left(\frac{\log n}{\log\log n}\right)$$

Und damit folgt nun

$$E(L) = \sum_{j=1}^{j_0} 1 + \sum_{j > j_0} \frac{1}{j_0^{j-j_0}} = \mathcal{O}\left(\frac{\log n}{\log\log n}\right)$$

Im Mittel sind also Längen $\mathcal{O}(1)$ aber es gibt auch Listen der Länge $\mathcal{O}\left(\frac{\log n}{\log\log n}\right)$

6.3 Hashing mit offener Adressierung

Die Idee ist, dass nur ein Element pro Tafeleintrag gespeichert wird. Es wird eine Folge von Hashfunktionen $h_1, h_2, ...$ benutzt. Ist der Eintrag $T[h_1(x)]$ belegt, probiere Funktion h_2.

6.3.1 Beispiel

Es sei

$$h_i(x) = (h(x) + i) \mod m$$

Damit wird versucht Elemente in benachbarte Felder zu schieben, so lange bis man ein freies Feld findet. (Linear probing)

Alternativ sei

$$h_i(x) = (h(x) + c_1 \cdot i + c_2 \cdot i^2) \mod m$$

(quadratic probing). Hier wird weiter gesprungen; man verhindert so, in einem "Block" stecken zu bleiben.

6.4 Perfektes Hashing

6.4.1 Idee

Die Hashfunktion sollte injektiv sein, dann entstehen keine Kollisionen

6.4.2 Theorie

S sei fest. Dann geht man in 2 Stufen vor

1. Stufe: Hashing mit Verkettung. Ergibt Listen wie in vorherigem Kapitel beschrieben
2. Stufe: für jede Liste einzelne eigene injektive Hashfunktion. Jede Liste wird also zu einer Hashtafel.

Wahl einer injektiven Hashfunktion

Sei $U = \{0, ..., N-1\}$, $h_k = \{0, ..., N-1\} \to \{0, ..., m-1\}$ sowie $k \in \{1, ..., N-1\}$ mit

$$h_k(x) = ((k \cdot x) \mod N) \mod m$$

Wähle h_k injektiv, wobei Injektivität wie folgt gemessen wird

$$b_{ik} = |\{x \in S | h_k(x) = i\}|$$

für $1 \leq k \leq N-1, 0 \leq i \leq m-1$. Dann ist

$$b_{ik}(b_{ik} - 1) = |\{ \underbrace{(x, y)}_{\text{Paare in Konflikt}} \in S^2 | x \neq y, h_k(x) = h_k(y) = i\}|$$

und die Zahl der Paare, die für h_k insgesamt in Konflikt stehen ist gegeben durch

$$B_k = \sum_{i=0}^{m-1} b_{ik}(b_{ik} - 1)$$

Dann gilt

$$B_k < 2 \Leftrightarrow h_k|s \text{ injektiv}$$

Als Abschätzung sei $|S| = n)$ und $b_{ik} = \frac{n}{m}$ für alle i (S gleichmäßig verteilt), dann folgt

$$B_k = \sum_{i=0}^{m-1} \frac{n}{m} \cdot \left(\frac{n}{m} - 1\right) = n \cdot \left(\frac{n}{m} - 1\right)$$

Falls dann gilt $m \geq n^2$, dann folgt $B_k < 2$

`Lemma 6.4.1.` *Mit vorherigen Voraussetzungen und N Primzahl gilt*

$$\sum_{k=1}^{N-1} \sum_{i=0}^{m-1} b_{ik}(b_{ik} - 1) \leq 2 \cdot \frac{n \cdot (n-1)}{m} \cdot (N-1)$$

Beweis in Mehlkorn I

`Bemerkung 6.4.2.` *Die Durchschnittliche Anzahl der Konflikte ist $< \frac{2n(n-1)}{m}$. Also folgt, dass für $m > n(n-1)$ die durchschnittliche Anzahl kleiner als 2 ist. Es existiert also ein k, so dass die Funktion injektiv wird.*

Nun folgt

`Korollar 6.4.3.` *Mit obigen Voraussetzungen gilt*

1. *$\exists k \in \{1, ..., N-1\} : B_k \leq 2 \cdot \frac{n(n-1)}{m}$*
2. *Sei $A = \{k|B_k > \frac{4(n(n-1))}{m}\}$ und sei $|A| > \frac{N-1}{2}$, dann gilt*

$$\sum_k B_k \geq \sum_A B_k \geq \frac{N-1}{2} \cdot \frac{4(n(n-1))}{m} = 2 \cdot \frac{n(n-1)}{m} \cdot (N-1)$$

Dies ist ein Widerspruch. Also sind mindestens $\frac{N-1}{2}$ aller h_k haben $B_k \leq \frac{4(n(n-1))}{m}$

Es folgt ebenfalls

`Korollar 6.4.4.` *Mit obigen Voraussetzungen gilt*

1. *Ein $k \in \{1, ..., N-1\}$ mit $B_k \leq 2 \cdot \frac{n(n-1)}{m}$ kann in Zeit $\mathcal{O}(m + N \cdot n)$ berechnet werden. Teste dafür für alle k wie viele Konflikte es gibt. Das geht jeweils in $\mathcal{O}(n)$. Der Aufbau des Feldes kostet $\mathcal{O}(m)$*

2. *Sei $m = n(n-1)+1$. Dann gibt es ein k mit $h_k|S$ injektiv. Wobei k in Zeit $\mathcal{O}(n^2+N\cdot n)$ gefunden werden kann.*
3. *Sei $m = 2n(n-1)+1$. Dann ist die Hälfte der $h_k|S$ injektiv. Ein injektives $h_k|S$ kann randomisiert in Zeit $\mathcal{O}(n^2)$ gefunden werden.*
4. *Sei $m = n$. Ein k mit $B_k \leq 2(n-1)$ kann in Zeit $\mathcal{O}(n \cdot N)$ gefunden werden.*
5. *Sei $m = n$. Randomisiert kann ein k mit $B_k \leq 4(n-1)$ in Zeit $\mathcal{O}(n)$ gefunden werden.*

6.4.3 Realisierung

1. Stufe

Bei $m = n$ gibt es h_k, so dass Länge der Listen $\mathcal{O}(\sqrt{n})$ ist, da $B_k \leq \mathcal{O}(n)$

2. Stufe

Wende Hashing auf jede Liste getrennt an. Sei $|S| = n = m$. Dann

1. Wähle k mit $B_k \leq 4(n-1) < 4n$. Sei
$$h_k(x) = (kx \mod N) \mod m$$
2. Sei $w_i = \{x \in S | h_k(x) = i\}$, $b_i = |w_i|$ und sei $m_i = 2 \cdot b_i \cdot (b_i - 1) + 1$. Wähle k_i mit
$$h_{k_i} = (k_i x \mod N) \mod m_i$$
so dass $h_{k_i}|w_i$ injektiv.
3. Sei $s_i = \sum_{j<i} m_j$. Speichere x in $T[s_i + j]$ wobei
$$i = (kx \mod N) \mod m, j = (k_i x \mod N) \mod m$$

Beispiel

$S = \{5, 6, 10, 12, 17, 21, 23, 42, 59\},\ m = |S|,\ k = 42$

$m = 10$:

0	1	2	3	4	5	6	7	8	9
21		1	5		6		23		10
			12		59				17
			42						

$\Rightarrow$ Kollision in 3,5 und 9

$b_3 = 3 \Rightarrow m_3 = 2 \cdot 3 \cdot 2 + 1 = 13 \Rightarrow h_{k_3} = (k_3 x \mod 127) \mod 13$
$b_5 = 2 \Rightarrow m_5 = 2 \cdot 2 \cdot 1 + 1 = 5 \Rightarrow h_{k_5} = (k_5 x \mod 127) \mod 5$

$b_9 = 2 \Rightarrow m_9 = 5 \Rightarrow h_{k_9} = (k_9 x \mod 127) \mod 5$

Wähle $k_3 = k_5 = k_9 = 1$, da dann alle Abb. injektiv:

0	1	2	3			4	5		6	7	8	9	
21		1	3	5	12		1	4		23		0	2
			42	5	12		6	59				10	17

Platzbedarf

Es gilt

$$m = \sum_{i=0}^{n-1} m_i = \sum_{i=0}^{n-1} 2b_i(b_i - 1) + 1 = n + 2 \underbrace{B_k}_{\leq 4n} \leq 9n = \mathcal{O}(n)$$

Laufzeit

1. $\mathcal{O}(n)$
2. $3 \sum_i \mathcal{O}(w_i) = \mathcal{O}(n)$

Zusammenfassung

`Satz 6.4.5.` *Mit obigen Voraussetzungen kann in linearer Zeit/Platz eine perfekte Hashtafel gebaut werden. Wobei die Hashfunktionen eine Zugriffszeit von $\mathcal{O}(1)$ haben. (randomisiert)*

6.4.4 Dynamischer Fall

Starte mit leerer Hashtafel und füge $xs_1, xs_2, ...$ ein. Bei i-ter Iteration liegen $xs_1, xs_2, ..., xs_i$ vor. Es können folgende Probleme auftreten

- » Auf Stufe 1 zu viele Konflikte. Daraus folgt, dass B_k zu groß wird.
- » Auf Stufe 2 kann es passieren, dass die Hahsfunktion nicht mehr injektiv ist oder die Tafeln zu klein sind.

Der Tafelaufbau ist in Algorithmus 4 beschrieben.

Algorithmus 4 Aufbau von Hashtafel(n) mit n Werten aus xs für gegebene $N, h_k, b_{i,k}$

```
B ← 0
w ← {w_0, ..., w_{n-1}}
k ← {k_0, ..., k_{N-1}}
xs ← {xs_0, ..., xs_{n-1}}
function TAFELAUFBAU
    for i ← 0 to n − 1 do
        T_i ← NEWTABLE()
        k_i ← RANDOM(0, N − 1)
    end for
    m ← n
    for i ← 0 to n − 1 do
        INSERT(xs_i, k_i, n, N)
    end for
end function

function INSERT(x, n)
    i ← (k_x mod N) mod n
    j ← (k_i · x mod N) mod m_i
    w_i ← w_i ∪ {x}
    B ← B + 2b_i
    b_i ← b_i + 1
    if B > 4(n − 1) then
        TAFELAUFBAU()
    else
        if FREE(T[s_i + j]) then
            HASHINSERT(T[s_i + j], x)
        else                                          // h_k nicht injektiv
            if m_i < 2b_i · (b_i − 1) + 1 then
                repeat ← true
                while repeat do
                    ENLARGEAREA(w_i)
                    finde neue Funktion für w_i
                    k_i ← RANDOM(0, N − 1)
                    DELETEAREA(T[s_i, s_i + m_i])
                    for x ∈ w_i do
                        j ← (k_i · x mod N) mod m_i
                        if FREE(T[s_i + j]) then
                            HASHINSERT(T[s_i + j], x)
                            repeat ← false
                        else
                            repeat ← true
                            break
                        end if
                    end for
                end while
            end if
        end if
    end if
end function

function ENLARGEAREA(w_i)
    m_i ← 4b_i(b_i − 1) + 1
    s_i ← m
    m ← m + m_i
    T[s_i, s_i + m_i] ist reserviert für w_i
end function
```

Laufzeit

Tafelaufbau passiert wenn $B < 4(n-1)$. Für zufälliges k gilt nach Lemma

$$B_k = \sum b_{ik}(b_{ik}-1) \leq 4(n-1)$$

mit Wahrscheinlichkeit $k \geq 0.5$. Um ein geeignetes k zu finden brauchen wir also ≤ 2 Versuche im Mittel. Damit ergibt sich insgesamt eine Laufzeit von

$$\mathcal{O}(n) + \mathcal{O}(\sum (b_{ik})^2) = \mathcal{O}(n) + \mathcal{O}(B_k) = \mathcal{O}(n)$$

Platzbedarf

m_i wächst immer. Werte sind $m_{i_1}, m_{i_2}, ...$ Dabei gilt $m_{i_{p+1}} \geq 2 \cdot m_{i_p} + 1$. Nun folgt

$$\sum_{p \geq 1} m_{i_p} \leq \sum_{p \geq 0} \frac{m_{ik}}{2^p} \leq m_{ik} \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + ...\right) \leq 2m_{ik}$$

Das ergibt einen Gesamtplatz von

$$\sum_i 2m_{ik} \leq \sum_i 2(4b_{ik}(b_{ik}-1)+1) \leq 8 \cdot 4(n-1) + 2n \leq 34n$$

`Satz 6.4.6.` *Sei N Primzahl, $S \subseteq U$, $|S| = n$, $|U| = N$. Eine perfekte Hashtafel der Größe $\mathcal{O}(n)$ kann online in $\mathcal{O}(n)$ erwarteter Zeit berechnet werden mit Zugriffszeit $\mathcal{O}(1)$*

7 Graphen

7.1 Definition

Ein Graph sei definiert durch $G = (V, E)$ wobei V (Vertices) eine endliche Menge von Knoten sei und $E \subseteq V \times V$ (Edges) eine endliche Menge von Kanten. Dabei heißt $e = (v, w) \in E$ Kante von v nach w und w heißt Nachbarknoten von v (adjazent). Ein *Pfad* in G ist eine Folge $(v_0, v_1, ..., v_k)$ von Kanten mit $k \geq 0$ und $(v_i, v_{i+1} \in E \, \forall \, 0 \leq i \leq k-1$ (Pfad von v_0 nach v_k). Falls $v_0 = v_k$ und $k \geq 1$ heißt der Pfad Zykel. Falls $v_i \neq v_j$ für $i \neq j$ heißt der Pfad einfach. Wir schreiben

$$v \overset{*}{\rightarrow}_G w$$

für einen Pfad von v nach w. Der Graph heißt *zyklisch* falls er Zykel enthält, sonst *azyklisch*. G heißt *Baum*, falls

a) V enthält genau ein v_0 mit inadj$(v_0) = 0$

b) $\forall v \in V \backslash \{v_0\}$: inadj(v) $= 1$

c) G ist azyklisch

7.2 Darstellung von Graphen

7.2.1 Darstellung im Computer

Wir nehmen an, dass $V = \{1, 2, ..., n\}$. Dann ergeben sich die Darstellungen

1. Darstellung: *Adjazenzmatrix A*. Dabei gilt in der Matrix

$$a_{i,j} \begin{cases} 1 & \text{falls} (i, j) \in E \\ 0 & \text{sonst} \end{cases}$$

   wobei diese Matrix in einem ungerichteten Graphen symmetrisch ist. Dabei verbraucht die Matrix einen Platz von $\mathcal{O}(n^2)$. Dies ist gut, falls $|E| = m$ ungefähr so groß ist wie n^2.

2. Darstellung: *Adjazenzlisten*. Dabei wird für jedes v die Knotenmenge gespeichert: Dabei ist

$$\text{Out: adj}(v) = \{w \in V | (v, w) \in E\}$$

und

$$\text{Inadj}(v) = \{w \in V | (w, v) \in E\}$$

Für ungerichtete Graphen ergibt sich

$$\text{Outadj}(v) = \{w \in V | (v, w) \in E\}$$

Dabei haben wir einen Platzverbrauch von $\mathcal{O}(n + m)$. Als Nachteil ergibt sich beim Zugrif auf Kante (v, w) kostet $\mathcal{O}(addj(v))$ wobei

$$\text{addj}(v) = |\{w \in V | (v, w) \in E\}|$$

Beispiel

Sei $G = (V, E)$ mit $V = \{1, 2, 3, 4, 5\}$ und $E = \{(1, 2), (1, 3), (1, 4), (4, 5), (5, 1), (3, 5)\}$. Für G ergibt sich die folgende Adjazenzmatrix:

$-$	1	2	3	4	5
1	0	1	1	1	0
2	0	0	0	0	0
3	0	0	0	0	1
4	0	0	0	0	1
5	1	0	0	0	0

Der zugehörige Graph ist in Abbildung 7.1 dargestellt.

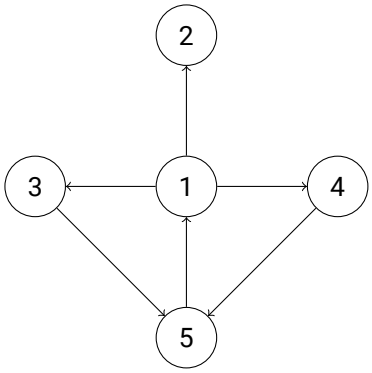


Abbildung 7.1: Gerichteter Graph $G = (V, E)$

7.3 Topologisches Sortieren

Sei $G = (V, E)$ ein gerichteter Graph (Digraph). Abbildung

$$\text{num: } V \to \{1, 2, ..., n\}$$

und $n = |V|$ heißt topologische Sortierung falls für alle $(v, w) \in E$ gilt

$$\text{num}(v) < \text{num}(w)$$

`Satz 7.3.1.` *Die Abbildung* num *existiert genau für azyklische Graphen*

Beweis. » $\Rightarrow$: Annahme G zyklisch. Sei $(V_0, ..., v_k = v_0)$ ein Zykel. Es muss gelten:

$$\text{num}(v_0) < \text{num}(v_1) < ... < \text{num}(v_k) = \text{num}(v_0)$$

Dies ist ein Widerspruch.

» $\Leftarrow$: Sei G azyklisch. Behauptung: G enthält Knoten mit indeg $= 0$ (Anzahl eingehender Kanten)

Beweis. Induktion über Größe von V. Für $|V| = 1$ trivial. Für $|V| > 1$: Entferne beliebigen Knoten $v \in V$. Dann erhalte ich damit $G' = (V, E')$ mit $V' = V \backslash \{v\}$ und $E' = E \cap (V' \times V')$. Nach Induktionsannahme ist G' azyklisch und enthält Knoten v' mit indeg$(v') = 0$. Entferne v' aus G' und erhalte G''. G'' enthält auch wieder ein v'' mit indeg$(v'') = 0$. Lese den Beweis nochmal, wobei als v nun v'' gewählt ist. Entweder ist indeg$(v') = 0$ in G oder indeg$(v'') = 0$ in G, denn es können nicht beide Knoten (v', v'') und (v'', v') in E existieren wegen Zykelfreiheit. □

□

7.3.1 Algorithmus

Beschreibung (Algorithmus 5):

» $|V| = 1$ trivial

» $|V| > 1$: Wähle v mit indeg$(v) = 0$, entferne dann v und erstelle rekursiv G' Sei num': $V' \to \{1, ..., |V'|\}$ Dann ist num für G:

$$\text{num}(w) = \begin{cases} \text{num}'(w) + 1 & \text{falls } w \neq v \\ 1 & \text{sonst} \end{cases}$$

Dabei ergeben sich als Kosten $\mathcal{O}(n + m)$ für die Zeit und den Platzverbrauch, wobei $|V| = n, |E| = m$. Dabei werden Adjazenzlisten benutzt, da sonst der Platz quadratisch wäre. Außerdem brauchen wir ein Array *indeg*, welches die Zahl der eingehenden Kanten zählt. (Abbildung 7.2)

Algorithmus 5 Topologisches Sortieren

count $\leftarrow$ 0
while $\exists v \in V : indeg(v) = 0$ do
    count $\leftarrow$ count$+1$
    num(v) $\leftarrow$ count
    streiche v und ausgehende Kanten
end while
if count$< |V|$ **then** G zyklisch
end if

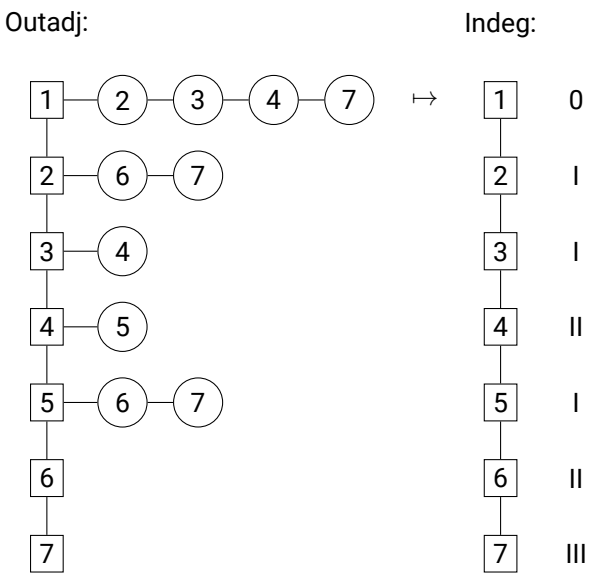


Abbildung 7.2: Beispiel zum topologischen Sortieren

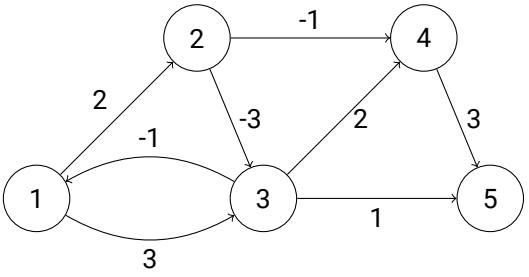


Abbildung 7.3: Graph mit Pfadkosten (negativer Zykel)

7.4 Billigste Wege

Gegeben sei ein Netzwerk (V, E, c) mit $G = (V, E)$ und $c : E \to \mathbb{R}$

Ziel ist es den Pfad mit den geringsten Kosten zwischen zwei gegebenen Knoten zu finden. Dabei sind die Kosten eines Pfades $p = v_0, ..., v_k$ gegeben durch

$$c(p) = \sum_{i=1}^{k-1} c((v_i, v_{i+1}))$$

Dabei unterscheiden wir folgende Probleme

1. Single source - single sink shortest path
2. single source shortest path
3. all pairs shortest path

7.4.1 Single source shortest path

Für $u \in V$ sei $P(s, u)$ die Menge aller Pfade von s nach u. Wir definieren als Kosten

$$\delta(u) = \begin{cases} \infty & \text{falls } P(s,u) = \emptyset \\ \inf\{c(p) | p \in P(s,u)\} & \text{sonst} \end{cases}$$

Ein negativer Zykel p hat dabei $c(p) < 0$.

`Lemma 7.4.1.` *Sei $u \in V$ Dann*

i) $\delta(u) = -\infty$ *gdw. u erreichbar über negativen Zykel, der von s aus erreichbar ist.*

ii) $\delta(v) \in \mathbb{R}$*: Es existiert ein billigster Weg von s nach u mit Kosten $\delta(u)$*

Beweis. i) » $\Rightarrow$: trivial

» $\Leftarrow$: Sei $C = \sum_{i \in E} |c(e)|$ und sei p ein billigster Weg von s nach n mit $c(p) < -C$. Es folgt, dass p einen negativen Zykel enthalten muss.

ii) $\delta(u) < \infty$, d.h. es gibt Pfad von s nach u. Es wird nun Behauptet, dass

$$\delta(u) = \text{min}\{c(p) | p \in P(s,u) \text{ und p einfach}\}$$

Beweis. Sei p' ein billigster, einfacher Weg von s nach u. Ist die Behauptung falsch, so gibt es einen nicht einfachen Weg q mit $c(q) < c(p')$. Da es keine negativen Zykel gibt gilt: Durch Wegnahme des Zykels aus q entsteht ein billigerer Weg q' mit $c(q') \leq c(q) < c(p')$. Dies ist ein Widerspruch. □

□

Beispiel *1. Fall*: G ist azyklisch. Wir nehmen an, dass G topologisch sortiert ist. Dann ergibt sich als Algorithmus

---

`Algorithmus 6` Single Source Shortest Path — Graph azyklisch

---

d(s) $\leftarrow$ 0, Pfad(s) $\leftarrow$ s
d(v) $\leftarrow \infty \forall v \in V \backslash \{s\}$
`for` $i \leftarrow s+1$ to n do
  d(v) $\leftarrow \text{min}_u\{d(n) + c((u,v)) | (u,v) \in E\}$
  Pfad(v) $\leftarrow$ Pfad(u*) + v // Konkatenation
`end for`

7.4.2 Dijkstra's Algorithmus

2. Fall: Kanten in G haben nur positive Kosten, Zykel erlaubt
Gegeben ein Graph $G = (V, E)$ und eine Kostenfunktion $c : E \to \mathbb{R}^+$. Weiterhin zwei Mengen S, S' mit Source $s \in S$. Hierbei ist S die Menge der Knoten $u \in V$ mit bekanntem $\delta(u)$. S' ist die Menge der Knoten aus $V \setminus S$, die Nachbarn in S haben.

`Lemma 7.4.2.` *Sei $w \in S'$, sodass $d'(w)$ minimal, dann ist $d'(w) = \delta(u)$*

Beweis. Sei p billigster Weg von s nach w mit allen Knoten (bis auf w) in S. Annahme: es gibt billigeren Weg q von s nach w. q muss einen ersten Knoten v in $V \setminus S$ haben. Nach Wahl von w ist $d'(v) > d'(w)$. Da alle Kanten nicht negativ, gilt $c(q) \geq d'(v) \geq d'(u) = c(p)$. Folglich ist $c(q) \geq c(p)$, wodurch die Annahme widerlegt ist. □

Es reicht also, sich bei der Wahl von w auf S' zu beschränken.

`Algorithmus 7` Dijkstra's Algorithmus

```
S ← {s}
d(s) ← 0
S' ← OUT(S)                          // Knoten welche von S aus direkt erreichbar sind
d'(u) ← c((s,u)) ∀u ∈ S'
d'(u) ← ∞ ∀u ∈ V \ (S ∪ S')
while S ≠ V do
    wähle w ∈ S' mit geringstem d'(w)
    d(w) ← d'(w)
    S ← S ∪ {w}
    for all u ∈ OUT(w) do
        if u ∉ S then
            S' ← S' ∪ {u}                // falls bereits zuvor u ∈ S' kein Unterschied
            d'(u) ← min{d'(u), d(w) + c(w,u)}
        end if
    end for
end while
```

Laufzeit Sei $n = |V|$, $m = |E|$. Implementieren S, S' als Bitvektor, d, d' als Arrays.

» Schleife über alle $u \in \text{OUT}(w)$ in $o(|\text{OUT}(w)|)$. Insgesamt also:

$$\sum_w |\text{OUT}(w)| = \mathcal{O}(n+m)$$

» n-mal Wahl des Minimums aus S': $\mathcal{O}(n^2)$

» Gesamtlaufzeit: $\mathcal{O}(n^2 + m)$

Alternativ: Speichere S' mit d' Werten als Heap, geordnet nach d'. Hiermit ergibt sich ein Laufzeit von $\mathcal{O}((n+m)\log n)$. Gut für dimere Graphen. Mit einem Fibonacci-Heap ist eine Laufzeit von $\mathcal{O}(n \log n + m)$ möglich.

3.Fall: Negative Kanten, aber keine negativen Zykel

7.4.3 Bellman-Ford Algorithmus

Neues Szenario: Erlaube negative Kanten (keine negativen Zykel).

```
function RELAX(v, w)
    d(w) ← min(d(w), d(v) + c((v,w)))
end function
```

Beobachtung: Relax-Operation erhöht keine d-Werte.

Algorithmus: Sei

$$d(v) = \begin{cases} 0 & \text{für } v = s \\ \infty & \text{sonst} \end{cases}$$

Iteriere Relax-Operation. Idee: d-Werte werden immer kleiner, aber höchstens bis δ-Wert Frage: Reihenfolge der Relax

Lemma 7.4.3. *Sei $w \in V$ und $\delta(w) < \infty$ und sei (v, w) die letzte Kante auf billigstem Weg zu w. Ist $d(v) = \delta(v)$ und wird RELAX(v, w) durchgeführt, so ist danach auch $d(w) = \delta(w)$.*

Algorithmus

```
for i <- 1 to n - 1 do
    forall $(v, w) \in E$ do
        relax(v, w)
    od
od
```

Lemma 7.4.4. *Für $i = 0, ..., n-1$ gilt: Nach Phase i ist:*
$d(w) = \delta(w)$ für alle $w \in V$, für die es einen billigsten Pfad der Länge i von s nach w gibt.

Beweis durch vollständige Induktion. 1. $i = 0$: $d(s) = \delta(s)$

2. $i \to i + 1$: Sei w Knoten mit billigstem Weg der Länge $i + 1$ von s nach w. Dessen letzte Kanten sei (v, w). Also gibt es einen billigsten Weg der Länge i von s nach $v$und nach Ind. Annahme ist nach Phase i $\delta(v) = d(v)$. In Phase $i+1$ wird RELAX(v, w) aufgerufen und $d(w) \leftarrow \delta(v) + c(v, w) = \delta(v)$. $\Rightarrow$ Nach Phase $n - 1$ ist $d(v) = \delta(v)$ für alle v. $\Rightarrow$ Laufzeit: $\mathcal{O}(n \cdot m)$.

□

4. Fall: negative Kosten und negative Zykel erlaubt

1. $n - 1$ Phasen nach Bellman-Ford, alte d-Werte
2. nochmal $n - 1$ Phasen ausführen, neue d-Werte
3. vergleiche alte mit neuen d-Werten

7.4.4 All pairs shortest paths

Annahme: Keine negativen Zykel. Sei $V = \{1, \ldots, n\}$ Für $i, j \in V$ definiere:

$\delta_k(i, j) =$ Kosten des billigsten Weges von i nach j dessen innere Knoten $\leq k$ sind

Für den Graphen in Abbildung 7.4 ergeben sich folgende Werte für δ:

$$\begin{aligned}
&\delta_0(1,4) = \infty \\
&\delta_1(1,4) = \infty \\
&\delta_2(1,4) = 4 + 2 = 6 \\
&\delta_3(1,4) = -3 + 2 + 2 = 1 \\
&\delta_4(1,4) = 1
\end{aligned}$$

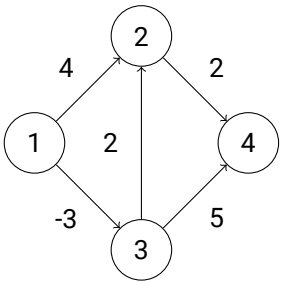


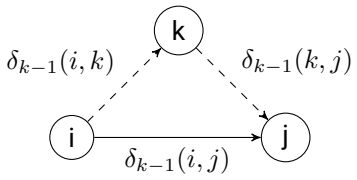
Abbildung 7.4: Graph mit Kantenkosten/gewichteter Graph

$$\delta_0(i,j) = \begin{cases} c(i,j) & \text{falls } (i,j) \in E \\ 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

$\delta_n(i,j) = \delta(i,j) =$ Kosten des kürzesten Wehes von i nach j

Frage: Wie berechnet sich δ_k aus δ_{k-1}? Es gibt zwei Möglichkeiten:

1. Es kann kein neuer Knoten verwendet werden, dann: $\delta_k(i,j) = \delta_{k-1}(i,j)$
2. Es kann ein neuer Knoten verwendet werden, dann: $\delta_k(i,j) = \delta_{k-1}(i,k) + \delta_{k-1}(k,j)$



Also:

$$\delta_k(i,j) = \min\{\delta_{k-1}(i,j), \delta_{k-1}(i,k) + \delta_{k-1}(k,j)\}$$

Algorithmus:

```
for k ← 1 to n do
   for all i, j ∈ V do
      δ_k(i, j) = min{δ_{k−1}(i, j), δ_{k−1}(i, k) + δ_{k−1}(k, j)}
   end for
end for
```

Alternativ n mal Bellman-Ford für jeden Knoten als source s durchführen. Laufzeit: $\mathcal{O}(n \cdot nm)$. Besser: Übertrage Kantengewichte auf nicht negative Werte. Dann n mal Dijkstra anwenden. Dies ergibt eine Laufzeit von $\mathcal{O}(n \log n + m)$. Zur Neugewichtung der Kanten soll folgendes gelten:

Lemma 7.4.5. *$\forall u, v \in V$ sei p der billigste Pfad $u \to v$ mit Kantengewicht $c \in \mathbb{R}$. $\Leftrightarrow \forall u, v \in V$ sei p der billigste Pfad $u \to v$ mit Kantengewicht $c' \geq c$. Wähle $c'(u, v) + h(u) - h(v)$ für Funktion $h : V \to \mathbb{R}$. Dann gilt für Pfad $p = (v_0, \ldots, v_k)$:*

$$\begin{aligned} c'(p) &= \sum_{i=0}^{k-1} c'(v_i, v_{i+1}) \\ &= \sum_{i=0}^{k-1} c(v_i, v_{i+1}) + h(v_i) - h(v_{i+1}) \\ &= \ldots \\ &= c(p) + h(v_0) - h(v_k) \end{aligned}$$

Somit ergibt sich:

$$\begin{aligned} & c(p) = \delta(v_0, v_k) \\ \Leftrightarrow\ & c'(p) = \delta'(v_0, v_k) \qquad \textit{(wie } \delta \textit{ nur mit } c' \textit{ statt } c\textit{)} \end{aligned}$$

Falls p Zykel $(v_0 = v_k) \Rightarrow c(p) = c'(p)$.

Wähle h so, dass alle $c' \geq 0$. Erweitere G um einen Knoten $s \notin V$ und die Kanten $(s, v), \forall v \in V$ mit $c(s, v) = 0$. Wähle $h(v) = \delta(s, v)$. Dann gilt:

$$h(v) \leq h(u) + c(u, v) \text{ für alle } (u, v) \in E$$

und somit:

$$0 \leq c(u, v) + h(u) - h(v) = c'(u, v)$$

h wird durch einmaligen Bellman-Ford von s aus in $\mathcal{O}(n \cdot m)$ berechnet.

7.5 Durchmusterung von Graphen

Es gibt DFS (Depth-First-Search) und BFS (Breadth-First-Search). Bestimmung aller Knoten, die von vorgegebenem $v \in V$ erreichbar sind.

$S \leftarrow \{n\}$
markiere alle Kanten als unbenutzt
`while` $\exists e \leftarrow (u,v) \in E,\ u \in S$ und (u,v) unbenutzt `do`
    markiere e als benutzt
    $S \leftarrow S \cup \{v\}$
`end while`

Probleme:

1. Realisierung benutzt $\leftrightarrow$ unbenutzt
2. Finden geeigneter Kanten
3. Realisierung von S

Lösungen:

1. Verwende Adjazenzlisten, markiere mit Trennzeiger in Liste. Alle Knoten links davon sind benutzt, alle rechts davon unbenutzt.
2. $\tilde{S} \subset S$. In S befinden sich alle Knoten, für die noch nicht alle Kanten gesehen wurden. (Trennzeiger noch nicht ganz rechts)
3. Operationen: INIT, INSERT, $w \in S$. Lege S als boolesches Feld an. Dann alle Operationen in $\mathcal{O}(1)$. Operationen auf S: INIT, INSERT, $w \in \tilde{S}$, wähle $w \in \tilde{S}$, streichen, $\tilde{S} = \emptyset$?. Verwende Stack (dann ergibt sich DFS) oder Queue (dann ergibt sich BFS).

```
function ExploreFrom(Knoten s)
    S ← {s}
    S̃ ← {s}
    for all v ∈ V do
        p(v) ← adjHead(v)
    end for
    while S̃ ≠ ∅ do
        w ← p(v), verschiebe p(v)
        if w ∉ S then
            Insert(w,S), Insert(w,S̃)
        else
            Delete(w,S̃)
        end if
    end while
end function
```

Laufzeit
$\mathcal{O}(n_s m_s)$ mit $n_s = |V_s|,\ m_s = |E_s|, V_s = \{v \in V | s \xrightarrow{*} v\}$
(Indizierter Teilgraph $(V_s, E_s) \subseteq (V, E)$, falls $V_s \subseteq V$

`Satz 7.5.1.` *Zusammenhangskomponenten können bei ungerichtetem Graphen in $\mathcal{O}(n+n)$ berechnet werden.*

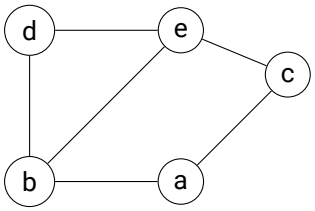


Abbildung 7.5: Beispielgraph für Tiefen- und Breitensuche

```
for all s ∈ V do
   if s ∉ S then
      ExploreFrom(s)
   end if
end for
```

Für jede Zusammehangskomponente wird ExploreFrom 1-mal aufgerufen

7.5.1 Depth-First-Search (Tiefensuche)

bei $\tilde{S}$ = Keller (stack)
DFS startet in a:
S: a,b,d,e,c

$$\tilde{S}: \begin{array}{|c|}\hline c \\ e \\ d \\ b \\ a \\ \hline \end{array}$$

7.5.2 Breadth-First-Search (Breitensuche)

bei $\tilde{S}$ = Schlange (queue)
subsection 7.5.1
BFS startet in a:
S: a,b,c,d,e
$\tilde{S}$: c b a $\rightarrow$ e d c b $\rightarrow$ ...

7.5.3 DFS rekursiv

$\tilde{S}$ als Stack.

Algorithmus 8 DFS

```
function DFS(Knoten v)
   for all (v, w) ∈ E do
      if w ∉ S then
         S ← S ∪ {w}
         dfsnum(w) ← count1, coun1++
         INSERT((v,w), T)                          // (v, w) ist Baumkante
         DFS(w)
         compnum(w) ← count2, count2++
      else
         if v →*_T w then
            INSERT((v,w),F)                        // (v, w) ist Vorwärtskante
         else
            if w →*_T v then
               INSERT((v,w),B)                     // (v, w) ist Rückwärtskante
            else
               INSERT((v,w),C)                     // (v, w) ist Querkante
            end if
         end if
      end if
   end for
end function
```

wobei dfsnum zählt als wievielter Knoten w besucht wird und compnum sagt als wievielter Knoten w abgeschlossen wird. T ist Baumkanten und F ist Vorwärtskanten, B Rückwärtskanten, C Querkante, S die schon gesehenen Knoten. Die Notation $v \to_T^* w$ bedeutet, dass w von v aus über Baumkanten erreichbar ist.

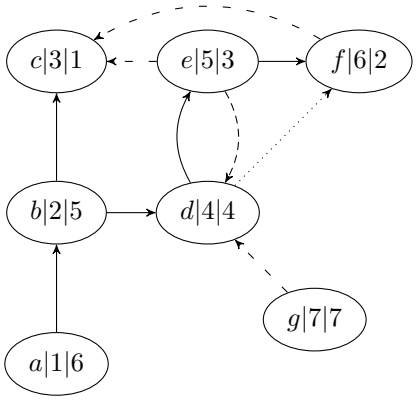


Abbildung 7.6: Rekursive Tiefensuche

`Lemma 7.5.2.` *Sie $G = (V, E)$ ein gerichteter Graph. Es gelten*

1. *DFS auf G hat lineare Laufzeit $\mathcal{O}(n + m)$*
2. *T, B, F, C ist Partition von E*
3. *T entspricht dem Aufrufbaum der Rekursion*
4. *$v \to_T^* w \Leftrightarrow$ dfsnum$(v) \leq$ dfsnum$(w) \wedge$ compnum$(w) <$ compnum(v)*
5. *$\forall (v, w) \in E$ gilt*
   a) *$(v, w) \in T \cup F \Leftrightarrow$ dfsnum$(v) \leq$ dfsnum(w)*
   b) *$(v, w) \in B \Leftrightarrow$ dfsnum$(w) <$ dfsnum$(v) \wedge$ compnum$(v) <$ compnum(w)*
   c) *$(v, w) \in C \Leftrightarrow$ dfsnum$(w) <$ dfsnum$(v) \wedge$ compnum$(w) <$ compnum(v)*

Aus $5.$ folgt, dass Berechnung der Partition in T, F, B, C aus dfsnum/compnum folgt. Zykelfrei bedeutet, dass es keine Rückwärtskanten in G gibt, d.h. $\forall (v, w) \in E$: compnum$(v) >$ compnum(w). Also ist num$(v) = n + 1 -$ compnum(v) topologische Sortierung.

7.5.4 Starke Zusammenhangskomponente (SZKs)

Gegeben sei ein gerichteter Graph $G = (V, E)$. Dieser heißt stark zusammenhängend, wenn es für alle $v, w \in V$ ein $v \to_E^* w$ gibt. Eine SZK von G ist ein maximal stark zusammenhängender Teilgraph von G.

`Definition 7.5.3.` *Sei (V', E') eine SZK. Knoten $v \in V'$ heißt Wurzel der SZK wenn*

$$\textit{dfsnum}(v) = \min\{\textit{dfsnum}(w) \ : w \in V'\}$$

Idee: Sei $G_{\text{aktuell}} = (V_{\text{akt}}, E_{\text{akt}})$ der von den schon gesehenen Knoten aufgespannt wird. Verwalten die SZKs von G_{akt}. Am Anfang ist $V_{\text{akt}} = 1$ und $E_{\text{akt}} = \emptyset$. Betrachte Kante (v, w):

1. $(v, w) \in T$: w kommt zu V_{akt} hinzu. Bildet also eine eigene SZK
2. $(v, w) \notin T$: mische eventuell mehrere SZKs zu einer

7.5.5 Durchmustern

DFS, Berechnung von SZKs, DFSNUM, COMPNUM.

Wurzeln: Folge von Wurzeln von nicht abg. Komp. in aufsteigender Reihenfolge von DFSNUM.

Unfertig: Folge von Knoten v, für die DFS(v) aufgerufen wurde, aber SZK nicht abgeschlossen wurde. Aufsteigend nach DFSNUM.

Fälle für Knoten g

1. Kante (g, d): nichts, da d Abgeschlossen. *Invariante:* Es gibt keine Kanten, die in abgschlossener Komponente starten und in nichtabgeschlossener enden.
2. Kante (g, h): h ist neuer Knoten. (g, h) ist Baumkante. h ist neue SZK. Füge h zu unfertig und Wurzeln hinzu.
3. Kante (g, c): Vereinige 3 SZKs mit Wurzeln g, e, b. Lösche g, e aus Wurzeln.

Algorithmus

```
Unfertig <- {}
Wurzeln <- {}
for all $v \in V$ do
    InUnfertig <- false // Boolesches Feld das Vorhandensein in Unfertig anzeigt
od
in dfs(v):
    push(v, Unfertig)
    InUnfertig(v) <- true
    count1 <- count1 + 1
    dfsnum(v) <- count1
    S <- S \cup {v}
    push(v, Wurzeln)
```

```
for all (v, w) \in E do
    if w \not\in S then
        dfs(w)
    else
        if InUnfertig(w) then
            while dfsnum(w) < dfsnum(top(Wurzeln)) do
                pop(Wurzeln)
            od
        fi
    fi
od
count2 <- count2 + 1
compnum(v) <- count2
if v = top(Wurzeln) then
    repeat
        w <- top(Unfertig)
        InUnfertig(w) <- false
        pop(Unfertig)
    until w = v
    pop(Wurzeln)
fi
```

Laufzeit Normales DFS in $\mathcal{O}(n+m)$ zusätzlich wird jeder Knoten (jeweils?) einmal in INUNFERTIG, UNFERTIG, WURZELN aufgenommen und gelöscht.

$$\Rightarrow \mathcal{O}(n+m)$$

7.6 Minmal aufspannende Bäume (MSTs)

Sei G ein zusammenhängender ungerichteter Graph. Sei $c : E \to \mathbb{R}^+$ eine Kostenfunktion. Wollen minimal aufspannenden Teilgraph $G' = (V, E_T), E_T \subseteq E$. G' zusammenhängend und $c(E_T) = \sum_{e \in E_T} c(e)$ minimal. *Beob.:* G' ist azyklisch. (Beweis durch Widerspruch). Daher: Minimal aufspannender Baum (minimum spanning tree, MST).

7.6.1 Kruskal-Algorithmus (Greedy)

1. Sortiere Kanten, sodass $c(e_1) < c(e_2) < \cdots < c(e_n)$.
2. $E_T = \emptyset$, createsets(u), $V_1 \leftarrow \{1\}, V_2 \leftarrow \{2\}$
3. `for` $i \leftarrow 1$ to m do
   `if` $(V, E_T \cup \{e_i\})$ azyklisch `then`

$E_T \leftarrow E_T \cup \{e_i\}$
end if
end for

`Satz 7.6.1.` *Der Kruskal-Algorithmus ist korrekt.*

Beweis. Nenne Kantenmenge $E' \subseteq E_T$ „gut“, falls sie zu einem MST erweiterbar ist.

1. *Behauptung:* Sei $E' \subseteq E_T$ gut und sei $e \in E \setminus E'$ die billigste Kante, sodass $(V, E' \cup \{e\})$ azyklisch ist. Dann ist auch $E' \cup \{e\}$ gut. $\rightarrow$ Indirektion

   Beweis. Sei $T_1 = (V, E_1)$ ein MST mit $E' \subseteq E_1$. T_1 existiert, da E' gut. Ist $e \in E_1$, dann fertig. Falls nicht, betrachte Graph $H = (V, E_1 \cup \{e\})$. H enthält einen Zykel, auf dem e liegt. Da $(V, E' \cup \{e\})$ azyklisch, enthält der Zykel auch Kante aus $E_1 \setminus (E' \cup \{e\})$. Sei e_1 eine solche Kante. Betrachte $T_2 = (V, (E_1 \setminus \{e_1\}) \cup e)$. T_2 ist aufspannend und $c(T_2) = c(T_1) + c(e) - c(e_1)$. Da e billigste Kante, gilt $c(e) \leq c(e_1)$ und damit $c(T_2) \leq c(T_1)$. Da T_1 MST nach Voraussetzung, ist auch T_2 MST. $\Rightarrow E' \cup \{e\}$ ist gut, denn kann zu T_2 erweitert werden. □

□

Wir ersetzen nun den Test im Algorithmus. Dies ist in Algorithmus 9 beschrieben. Find(u) liefert Namen der Menge in der u ist. Es gibt $2n$ Finds und $n - 1$ Unions.

`Algorithmus 9` Kruskals Algorithmus mit Union-Find

```
1: function KRUSKAL(G)
2:     for i ← 1 to m do
3:         e_i ← (u, v)
4:         a ← FIND(u)
5:         a ← FIND(v)
6:         if a ≠ b then
7:             UNION(a, b)
8:             E_T ← E_T ∪ {e_i}
9:         end if
10:    end for
11: end function
```

Union-Find Datenstruktur

1. Namensfeld R[1,..,n] -> [1,..,n] wobei $R(x)$ der Name der Menge, zu der x gehört. Da ergibt etwa
   Find(x): return R(x)

Union(a,b):

```
for i <- 1 to n do
    if R(i) = a then R(i) <- b
od
```

Damit erhält man eine Laufzeit von Find in $\mathcal{O}(1)$ und Union in $\mathcal{O}(n)$. Gesamt also $\mathcal{O}(m + n^2 m \log m)$

Wir wollen nun den Union von $\mathcal{O}(n)$ so verbessern, dass er nicht alle Elemente besucht. Merke für jede Menge die Elemente, die dort enthalten sind. Behalte bei Union immer den Namen der größeren Menge. Benötigte Funktionen sind nun in Algorithmus 10 dargestellt. Worst case Union: $\mathcal{O}(n), |a| = \frac{n}{2}, |b| = \frac{n}{2}$.

Algorithmus 10 Funktionen für Union-Find

```
 1: function CREATESET
 2:     for i ← 1 to n do
 3:         R(x) ← x                          // Speichert Wurzel des Baumes
 4:         ELEM(x) ← x                       // Speichert Elemente des Baums
 5:         SIZE(x) ← x
 6:     end for
 7: end function

 8: function FIND(x)
 9:     return R(x)
10: end function

11: function UNION(a, b)
12:     if SIZE(a) < SIZE(b) then             // a muss größer/gleich b sein
13:         SWAP(a, b)
14:     end if
15:     for all x ∈ ELEM(b) do
16:         R(x) ← a
17:         e_a ← ELEM(a)
18:         INSERT(x, e_a)
19:     end for
20: end function
```

Satz 7.6.2. *Mit der beschriebenen Struktur können createset, $n-1$ Unions und $2n$ Finds in Zeit $\mathcal{O}(n \log n + m)$ ausgeführt werden.*

Beweis. Createset und $2m$ Finds gehen in $\mathcal{O}(n + m)$. Wir Behaupten nun, dass $n - 1$ Unions $\mathcal{O}(n \log n)$ kosten. Ein Union(a,b) vereinigt zwei Mengen mit n_a, n_b Elementen. Sei $n_b < n_a$, dann folgt eine Laufzeit von $\mathcal{O}(1 + n_b)$. Damit haben wir eine Gesamtzeit

von $\mathcal{O}(\sum_{i=1}^{n-1} n_i + 1)$ mit n_i Größe der kleineren Menge beim i-ten Union. Wechselt ein Element die Menge, so trängt es 1 zu n_i bei. Element j wechselt r_j die Menge. Es gilt also

$$\sum_{i=1}^{n-1} n_i = \sum_{j=1}^{n-1} r_j$$

Behauptung: $r_j \leq \log n$ für alle Knoten j

Beweis. Wenn j in Menge mit l Elementen ist und wechselt, dann kommt j in eine Menge der Größe $\geq 2l$. Das heißt beim k-ten Wechsel ist j in Menge mit mindestens $\geq 2^k$ Elementen. Da es nur n Elemente gibt gilt $2^k \leq n$ und daraus folgt $k \leq \log n$ □

Insgesamt folgt nun

$$\sum_{i=1}^{n-1} n_i = \sum_{j=1}^{n-1} r_j = n \log n$$

Damit ist die Laufzeit für Kruskals Algorithmen $\mathcal{O}(m \log n)$ □

7.6.2 Union-Find

Halte für jede Menge Baum. Damit geht Union in $\mathcal{O}(1)$. Für Find läuft man im Baum hoch zur Wurzel. Das geht in $\mathcal{O}(\text{Tiefe}(\text{Baum}))$. Für weitere Verbesserungen führe folgende Optimierungen ein

1. Gewichtete Verinigungsregel: Hänge kleinen an großen Baum
2. Pfadkomprimierung: Laufe Pfad hoch. Hänge alle Zeiger auf Wurzel

`Satz 7.6.3.` *n Unions + m Finds gehen in Zeit $\mathcal{O}(n + m \cdot \alpha(m + n, n))$ wobei α invers zur Ackermannfunktion ist. Das ist kleiner als 5 für alle realistischen Werte.*

7.6.3 PRIMs Algorthmus

Beschrieben in Algorithmus 11. Dabei liegen in E_T die Kanten des Spannbaums. Korrektheitsbeweis analog zu vorher: Induktiv über gute Erweiterungen.

Laufzeit

Priority Queue mit Schlüsseln $\{c(w) | w \notin T \wedge c(w) = \min\{c(u, w), u \in T\}\}$. Suche w mit kleinstem $c(w)$, entspricht deleteMin bei Dijkstra. Wird w in T eingefügt, so tue für alle $(w, x) \in E$ mit $x \notin T$ und $c(x) \leftarrow \min\{c(x), c(w, x)\}$. Das entspricht DecreasyKey bei Dijkstra.

Algorithmus 11 Prims Algorithmus

```
1: function PRIM(G)
2:     T ← {v}
3:     while T ≠ {V} do
4:         e ← (u, w) ∈ E mit u ∈ T ∧ w ∉ T und c(e) minimal
5:         E_T ← E_T ∪ {e}
6:         T ← T ∪ {w}
7:     end while
8: end function
```

Implementierung

Priority Queue als primären Heap. Dabei gehen Operationen in $\mathcal{O}(\log n)$. Insgesamt also $\mathcal{O}(m \log n)$. Statt bin. Heap verwende (a,b)-Baum mit $a = \max(2, \frac{m}{n})$ und $b = 2a$. Damit ergibt sich eine Laufzeit von $\mathcal{O}(m \frac{\log n}{\log \frac{m}{n}})$ Das ist gut, wenn m etwa n^2 ist.

7.7 Zweifach Zusammenhangs Komponente von ungerichteten Graphen

Graph heißt 2-fach zusammenhängend, falls $G - \{v\}$[1] zusammenhängend für alle $v \in V$
Eine 2ZK ist maximal 2-fach zusammenhängender Teilgraph. $v \in V$ heißt Artikulationspunkt, wenn $G - \{v\}$ nicht zusammenhängend

2ZK's: $\{1, 2\}, \{6, 7\}, \{2, 3, 4\}, \{4, 5, 6\}$
Artikulationspunkte: $2, 4, 6$

7.7.1 DFS für ungerichtete Graphen

Es gibt Baumkanten T und Rückwärtskanten B, jedoch keine Vorwärts- und Querkanten.

Idee:
v ist kein Artikulationspunkt, falls ein Baumnachfolger w von v eine Rückwärtskante "vor" v hat
x "vor" v $\Leftrightarrow dfsnum(x) < dfsnum(v)$
Falls v Wurzel des dfs-Baumes und mehrere Zweige, so ist v Artikulationspunkt.

[1] $G' = (V \backslash \{v\}, (E \backslash \{(v, x) \forall x \in V\}) \backslash \{(x, v) \forall x \in V\})$

`Definition 7.7.1.` $low(u) \leftarrow \min\{dfsnum(u); \min\{dfsnum(v)$ *mit Pfad* $u \xrightarrow[T]{*} Z \rightarrow v\}\}$

`Algorithmus 12` Tiefensuche auf ungerichteten Graphen

```
1:  function DFS(Knoten v)
2:      for all (v,w) ∈ E) do
3:          if w unbesucht then
4:              dfsnum(w) ← count; count++
5:              low(w) ← dfsnum(w)
6:              if low(w) < low(v) then
7:                  low(v) ← low(w)
8:              end if
9:              if low(w) > dfsnum(v) then
10:                 artpunkt(v) ← TRUE
11:             end if
12:         else
13:             if dfsnum(w) < low(v) then
14:                 low(v) ← dfsnum(w)
15:             end if
16:         end if
17:     end for
18: end function
```

In (7) wird low-Wert von w an low(v) übergeben, falls er kleiner ist
In (14) wird Rückwärtskante von v nach w berücksichtigt
In (10) wird erkannt, ob der Zweig, der aus v in Richtung w startet keine Rückwärtskante "vor" v enthält $\Rightarrow$ v ist Artikulationspunkt

Laufzeit: $\mathcal{O}(n+m)$ (Verfeinerung von DFS)

Spezialfall: v Wurzel

```
if dfsnum(v) == 1&&∃w1 ≠ w2 mit (v,w1),(v,w2) ∈ T then
    artpunkt(v) ← TRUE
end if
```

Was fehlt: Berechnung der 2ZK wenn Artikulationspunkte gegeben ($\rightarrow$ Übung)

7.8 EXKURS: Stable Marriage

Vollständiger bipartiter Graph aus n Männern und n Frauen.
Jede Person hat Rangliste $\{0, ..., n-1\}$ für das andere Geschlecht:

» $w(x, Y)$ Gewicht von Mann aus gesehen

» $w(Y, x)$ Gewicht von Frau aus gesehen

(a,A),(b,B),(c,C) nicht stabil:
c findet B attraktiver als C und B findet c attraktiver als b

Suche perfektes Matching M, sodass es kein $(x, Y), (Y, x)$ mit $w(x, Y) > w(x, X)$ und $w(Y, x) > w(Y, y)$

7.8.1 Algorithmus

Alle Männer sind frei.
Wähle einen freien Mann, der wählt seine beste Frau, die ihn noch nicht abgelehnt hat.
Diese akzeptiert oder lehnt ab, je nachdem ob das Angebot ihren Status verbessert.
Evtl. wird ihr bisheriger Mann frei.
Es gilt: Verheiratete Frauen bleiben verheiraten, aber nicht unbedingt mit demselben Mann

Termination

Alle Frauen bekommen irgendwann ein Angebot, bleiben ab dann verheiratet.
Männersicht:
Macht Mann n Angebote haben vorher $n - 1$ abgelehnt. Diese letzte muss frei sein, diese akzeptiert.

Korrektheit

Es gibt *kein* (x,X)(y,Y), wo x Y attraktiver findet als X (x hat Y vor X gefragt) *und gleichzeitig* Y x attraktiver findet als y (denn Y hatte x abgelehnt)

Laufzeit

$$\mathcal{O}\left(\sum_{i=1}^{n} i\right) = \mathcal{O}(n^2)$$[2]

Frage: Erwartete Laufzeit, wenn Referenzen der Männer zufällig sind, die der Frauen beliebig aber fest.
Idee:

[2] → Buch: Kleinberg/Tardos

Jeder Mann wählt seine Präferenz nach und nach (wenn er sie braucht). x macht immer einer zufälligen Frau ein Angebot, der er noch kein Angebot gemacht hat oder einfach: Er wählt *immer eine* zufällige Frau.
Wann haben alle Frauen mind. 1 Angebot?
$\rightarrow$ "Coupon Collector Problem"

7.8.2 Coupon Collector Problem - Laufzeit

Wie viele Bälle müssen auf n Körbe geworfen werden, bis jeder Korb $\geq 1\times$ getroffen ist?

Ergebnis: $\Theta(n \log n)$

$P(\text{Korb i nach k Würfen nicht getroffen}) = \left(1 - \frac{1}{n}\right)^k \approx e^{-\frac{k}{n}}$
Mit $k = c \cdot n$: $P(\text{Korb i nach k Würfen nicht getroffen}) = e^{-c}$
Mit $k = c \cdot n \log n$: $P = e^{-c \cdot \log n} = \left(\frac{1}{n}\right)^c$

$\Rightarrow$ Mit hoher Wahrscheinlichkeit ist nach $(c-1) \cdot n \log n$ Versuchen kein Korb leer. Wähle $c > 1$.

8 Patternmatching

8.1 Patternmatching auf Strings

Sei T ein String über dem Alphabet Σ. Dabei sei $T[1..n], 1 \leq i \leq j \leq n$ ein Teilstring von Stelle i bis Stelle j. Gegeben sei nun ein String $T[1..n]$ und ein Pattern $P[1..m]$ mit $m << n$. Suche nun P in T.

8.1.1 Naive Lösung

Man schiebt das Pattern durch den String und vergleicht jeweils alle Stellen.

```
for s <- 1 to n-m+1 do
    Teste ob T[s..s+m-1] = P
od
```

Dies findet alle Vorkommen und hat eine Laufzeit von $\mathcal{O}(n \cdot m)$

8.1.2 Algorithmus von Knuth/Morris/Pratt

Idee

Verschiebe P nicht nur um 1. Betrachte P. Berechne für alle q

$$\Pi(q) = \max\{k | P[1..k] \text{ ist echtes Suffix von } P[1..q]\}$$

Verschiebe P so, dass $P[\Pi(q-1)+1]$ auf $T[j]$. Wir suchen das kleinste $s' > s$ so dass $P[1..k] - T[s'+1, .., s'+k]$ mit $s'+k = s+q$.

```
P         ababaa
Index     123456
PI(i)     001231
```

Algorithmus für Präfixfunktion

```
Präfixfunktion(P)
```

```
PI(1) <- 0, k <- 0
for q<- 2 to m do
    while k>0 and P[k+1] != P[q] do
        k <- PI(k)
    od
    if P[k+1]=P[q] then
        k <- k+1
    fi
    PI(q) <- k
od
```

Laufzeit

In der while-Schleife befinden wir uns nur wenn $k > 0$ und innerhalb der while-Schleife wird k immer erniedrigt. In jeder for-Iteration wir k einmal erhöht, d.h. es steigt nur n mal. Es kann also auch nur maximal m Erniedrigungen geben. Damit haben wir Laufzeit $\mathcal{O}(m)$

Algorithmus

```
Gesamt
Sei n,m,T,P,PI gegeben

q <- 0
for i <- 1 to n do
    while q > 0 and P[q+1] != T[i] do
        q <- PI(q)
    od

    if P[q+1] = T[i] then
        q <- q+1
    fi

    if q=m then
        // P kommt ab (i-m+1)-ter Stelle vor
        q <- PI(q)
    fi
od
```

Laufzeit

Amortisierungsargument wie bei Präfix, also Laufzeit $\mathcal{O}(n)$

Beispiel

Siehe Wikipedia[1]

8.1.3 Algorithmus von Boyer/Moore

```
Naiv:

s<-0
while s <= n-m do
    j <- m
    while j > 0 and P[j] = T[s+j] do
        j <- j-1
    od

    if j = 0 then
        //P kommt ab Stelle s+1 vor
        s<- s+1    // ersetze durch s <- s+ gamma[0]
    else
        s <- s+ 1  //ersetze durch s <- s + max{gamme[j], j-lambda[T[s+j]]}
                   //hierzu brauchen wir "gutes Suffix" und schlechter Buchstabe
    fi
od
```

Berechne λ als letztes Vorkommen(P, m, Σ) und γ als gutes Suffix(P, m).

Schlechter Buchstabe

sei $P[i] \neq T[s+j]$ ein gefundenes Mismatch. Das ist gut wenn $T[s+j]$ gar nicht vorkommt, sonst verschiebe bis zum Vorkommen. Dann sei

$$k = \max\{i | T[s+j] = P[i]\} \text{ falls T[s+j] in P, sonst } 0$$

`Lemma 8.1.1.` *s kann um $j-k$ verschoben werden.*

Beweis. 1. $k = 0$: $T[s+j]$ kommt nicht vor, ist also schlechter Buchstabe. Wir verschieben um j bis hinter diesen Buchstaben

[1] http://de.wikipedia.org/wiki/Knuth-Morris-Pratt-Algorithmus

2. $k < j$: $T[s+j]$ kommt in P links von Stelle j vor und $j - k > 0$. Verschiebe P um $j - k$.

□

Brauchen für jedes $x \in \Sigma$ die am weitesten rechts stehende Position von a in P also $a = P[\lambda(a)]$.

```
Berechne Lambda:

for all a \in Sigma do
    lambda(a) <- 0
od
for j <- 1 to m do
    \lambda[P[j]] <- j
od
```

Gutes Suffix

Idee: Falls Mismatch $P[j] \neq T[s+j]$ verschiebe P um

$$\gamma[j] \rightarrow m - \text{max}\{k | P[1..k] \text{ ist Suffix von } P[j+1, .., m]\}$$

Ähnlich wie Präfixfunktion

8.1.4 Random Ansatz: Fingerabdruck (Karp/Rabin)

Sei $\Sigma = \{0, ..., 9\}$, x, y sind dann natürliche Zahlen. $F_p : \mathbb{Z} \rightarrow \mathbb{Z}_p$ mit $F_P(z) = z \mod p$ und p Primzahl.

Algorithmus

```
p <- zufällige Primzahl zwischen 2 und mn^2log(mn^2)
match <- false
i <- 1

while not match and 1 <= i <= n-m+1 do
    if F_p(X(i)) = F_p(Y) then
        match <- true
    else
        i <- i+1
        berechne F_p(X(i))
    fi
od
```

Beispiel

Sei X: 2 3 7 5 8 6 1 2 3 5 und m=4

$$F_p(X(i+1)) = (10(F_p(X(i)) - 10^{m-1}x_i) + x_{i+m}) \mod p$$

Es ist nötig 10^{m-1} vorzuberechnen, ansonsten geht es in $\mathcal{O}(1)$

`Satz 8.1.2.` *Algorithmus von Karp/Rabin läuft in $\mathcal{O}(n+m)$ und hat Fehlerwahrschienlichkeit von $\mathcal{O}\left(\frac{1}{n}\right)$*

Beweis. Was ist prob$(F_p(Y) = F_p(X(i))|Y \neq X(i))$. Sei $a = Y, b = X(i). a \mod p = b \mod p \Leftrightarrow p|(a-b)$ Es gilt $|a-b| \leq 10^m \leq 2^{4m}$. Damit $|a-b| < 2^{4m}$ hat $\leq (4m)$ verschiedene Primfaktoren. Haben p aus dem Bereich $[2, mn^2 \log(mn^2)]$. In diesem bereich gibt es

$$\mathcal{O}\left(\frac{mn^2 \log(mn^2)}{\log(mn^2 \log(mn^2)}\right) = \mathcal{O}(mn^2)$$

Primzahlen. Für festes i gilt also

$$\text{prob}(F_p(Y) = F_p(X(i))|Y \neq X(i)) = \mathcal{O}\left(\frac{4m}{mn^2}\right) = \mathcal{O}\left(\frac{1}{n^2}\right)$$

Gesamt also

$$\mathcal{O}((n-m+1)\frac{1}{n^2}) = \mathcal{O}(\frac{1}{n})$$

□

Von Monte Carlo nach Las Vegas

1. Falls $F_p(X(i)) = F_p(Y)$ teste in $\mathcal{O}(m)$ ob $X(i) = y$. Falls nicht starte naiven Algorithmus. Das ergibt eine erwartete Laufzeit von

$$\mathcal{O}((n+m)(1-\frac{1}{n}) + (n \cdot m)\frac{1}{n}) = \mathcal{O}(n+m)$$

2. Falls $F_p(X(i)) = F_p(Y)$ teste in $\mathcal{O}(m)$ ob $X(i) = y$. Falls nicht starte von dort mit neuem zufälligem p. Das ergibt eine neue Fehlerwahrscheinlichkeit von prob(t. Iteration des Algorithmus) = $\mathcal{O}(\frac{1}{n^t})$

8.1.5 Preprocessing von Suffixbäumen

Sei S ein String der Länge n und P ein Pattern der Länge m. Preprocessing des Patterns geht in $\mathcal{O}(m)$. Jetzt machen wir ein Preprocessing des Strings in $\mathcal{O}(n)$. Damit erhalten wir insgesamt eine Laufzeit von $\mathcal{O}(m)$ anstatt $\mathcal{O}(n+m)$.

Beispiel zu Suffixbäumen

String sei a a b b a b b und einem Sonderzeichen am Ende.

Patternsuche geht einfach: Ablesen des Patterns im Baum. Ablesen der Stellenzahlen unter den gefundenen Patterns.

Definition

Baum T für String S hat n Blätter, nummeriert von 1 bis n

- » Jeder innere Knoten hat ≥ 2 Kinder
- » Jede Kante ist mit Teilwort aus S markiert
- » Kanten aus demselben Knoten haben Markierungen mit versch. ersten Zeichen
- » Konkatentationen der Kantenmarkierungen von Wurzel zum Blatt i ergibt den Suffix der bei i beginnt.

Laufzeit für Patternsuche ist $\mathcal{O}(m+k)$ wenn es k Vorkommen des Pattern gibt. Wenn ein Baum bereits aufgebaut ist geht die Suche in $\mathcal{O}(|P|)$ wobei $|P|$ die Länge des Patterns ist.

Aufbau des Baumes

1. Starte mit Einzelkante für $S[1...n]$
2. *Iteriere:* Sei T_i Baum der Suffixe $S[1...n], ..., S[i...n]$. Starte in Wurzel. Suche einen möglichst Langen Pfad für Suffix $S[i+1...n]$ in T_i, der mit Präfix von $S[i+1...n]$ übereinstimmt.
   a) Pfad endet auf Kante (u, v). Dort endet Übereinstimmung. Teile (u, v) in (u, w) und (w, v) und markiere entsprechend. Füge neue Kante an $(w,$ Blatt$(i+1)$ ein mit Markierung restl. Suffix von $S[i+1...n]$
   b) Pfad endet auf Knoten v. Füge neue Kante $(w,$ Blatt$(i+1)$ ein

Laufzeit ist $\mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(n^2)$

Verallgemeinerung

Es gebe nun mehrere Strings $S_1, S_2, ..., S_q$, die mit $\$_1, \$_2, ...$ enden. Wir konkatenieren nun die Strings und bauen Suffixbaum für den Superstring auf in $\mathcal{O}(|S_1| + |S_2| + ...)$. Da bei diesem Vorgehen über Stringgrenzen hinweg Suffixe entstehen, muss man bei einem Endzeichen ein Blatt entstehen lassen.

Gegeben sei nun S_1, S_2 und wir suchen den größten gemeinsamen Teilstring in den beiden Strings.

1. Konstruiere Suffixbaum für S_1 und S_2
2. Innere Knoten werden mit 1/2 markiert falls sie zum Blatt mit Suffixen aus S_1 bzw S_2 führen.
3. Gemeinsame Teilstrings werden repräsentiert durch Pfade zu doppelt beschrifteten Knoten
4. Tiefensuche von Wurzel über solche 1/2 Knoten liefert größten gemeinsamen Teilstring

Laufzeit ist $\mathcal{O}(|S_1| + |S_2|)$

Korollare

1. Gegeben sei ein String S und Pattern $P_1, P_2, ...$ Finde alle Vorkommen von $P_1, P_2, ..., P_k$. Dies geht in Zeit $\mathcal{O}(\sum_{i=1}^{k} |P_i| + z)$ wenn z die Anzahl der Vorkommen ist.
2. Gegeben seien S_1, S_2. Suche alle maximalen Wiederholungen.

   ```
   S_1:  abbacbaa
   S_2:  abbcbbbb
   ```

   Dies geht analog zum größten gemeinsamen Teilstring
3. Suche den am häufigsten Vorkommenden Teilstring. Das ist der Teilstring bis zum Knoten mit den meisten Kindern, da jedes Kind für ein Vorkommen des Knotens steht. Am einfachsten geht das, wenn man sich bei jedem Knoten die Anzahl der Kinder merkt
4. Suche die kürzesten Pattern aus $\Sigma*$, die nicht in S_i vorkommen. Laufe BFS zur ersten Kante, die Mehrfachbeschriftungen hat, oder wo eine Kante aus Σ fehlt. $\mathcal{O}(\Sigma|S|)$
5. Suche das längste Palindrom in String S. Wir konkatenieren dazu S mit $\overline{S}$ und suchen den längsten Teilstring.

8.1.6 Anwendung: Datenkompression

Das Zvi-Lempel Verfahren.

`Definition 8.1.3.` *Gegeben sei String S der Länge n. Prior$_i$ sei das längste Präfix von $S[i..n]$, welches zugleich Teilstring aus $S[1...i-1]$ ist. Sei $l_i \leftarrow$ |Prior$_i$| und für $l_i > 0$ sei s_i die Anfangsposition des ersten Vorkommens von Prior$_i$ in $S[i...i-1]$.*

`Bsp: S = abaxcabaxabz`

```
Prior_7 = bax
l_7 = 3
s_7 = 2
```

Idee: Falls $S[1..i-1]$ schon dargestellt und $l_i > 0$, so stelle die nächsten l_i Stellen nicht explizit dar, sondern durch das Paar (s_i, l_i)

```
Bsp: S = abacabaxabz
-> ab(1,1)c(1,3)x(1,2)z
S = (ab)^16
-> ab(1,2)(1,4)(1,8)

Algorithmus:
i <- 1
repeat berechne l_i, s_i
    if l_i >0 then
        return (s_i,l_i)
        i<- i+l_i
    else
        return S[i]
        i <- i+1
until i>n
```

9 Dynamisches Programmieren

9.1 Idee:

optimale Lösung setzt sich aus optimalen Lösungen für kleinere Teilprobleme zusammen
löse rekursiv und setze zusammen.

Beispiel: Schnittproblem für Stab der Länge n.
$\rightarrow$ zerschneide Stab in kleine Stücke der Länge
$\rightarrow$ Teil der Länge i kostet pi
$\rightarrow$ maximiere Gesamterlös r_n.
1. Lass Stab ganz $p_n + r_n$
2. 1 Stück mit Länge 1 $p_1 + r_{n-1}$
3. 1 Stück mit Länge 2 $p_2 + r_{n-2}$

9.2 Schnittproblem

```
Algorithmus:
 1: function SCHNITT(n,p)                                  // Input: n, Preisliste p
 2:     q := −∞;
 3:     if n = 0 then
 4:         return 0;
 5:     end if
 6:     for i ∈ [1, n] do
 7:         q := max(q, p[i] + Schnitt(n-1);
 8:     end forreturn q;
 9: end function
```

Analyse: $T(n) =$ Aufruf von $Schnitt()$ $T(0) = 1$ $T(n) = 1 + \Sigma_j = 0n - 1(T(j))$
dynamische Programmierung: berechne Teillösungen!

```
function SCHNITT(n,p)
    r[0] = 0;
    for j ∈ [1, n] do
```

```
        q := ∞;

        for i ∈ [1, j] do
            q := max(q, p[i] + r[j-i])
        end forr[j] := q;
        end forreturn r[n];
    end function
```

Laufzeit:
$T(n) = \sum_{j=1}^{n} \frac{n(n+1)}{2} = \mathcal{O}(n^2)$
Beachte: Größe j wird erst betrachtet, wenn die kleineren Größen alle feststehen.
Rekonstruktion der optimalen Lösung:

Merke für r[j] den 1. Schritt; also ersetze q = max(q, ...) durch

```
if q < p[i] + r[j-1] then
    q := p[i] + r[j-1];
s[j] := i;
end if
```

Das Stück s[j] := i heißt, das Stück am Stab der Länge 1.

9.3 Matrizenmultiplikation

Matrizenmultiplikation $A \cdot B$
Voraussetzung: Anzahl Spalten von A = Anzahl Zeilen von B.

```
for i ∈ [1, A.Zeilen()] do

    for j ∈ [1, B.Spalten()] do
        c_{i,j} := 0

        for k ∈ [1, A.Spalten()] do
            c_{i,j} := c_{i,j} + a_{i,k} · b_{k,j}
        end for
        end for
    end forreturn C;
```

Berechne $A \cdot B \cdot C \cdot D \cdot E \cdot F$
Bsp.:
A := (4 x 2)

A · B [:= (2 x 3)] $\Rightarrow$ ergibt [4 x 3] $4 \cdot 3 \cdot 2 = 24$ Operationen
B · C [:= (3 x 1)] $\Rightarrow$ ergibt [4 x 1] $4 \cdot 1 \cdot 3 = 12$ Operationen
C · D [:= (1 x 2)] $\Rightarrow$ ergibt [4 x 2] $4 \cdot 2 \cdot 1 = 8$ Operationen
D · E [:= (2 x 2)] $\Rightarrow$ ergibt [4 x 2] $4 \cdot 2 \cdot 2 = 16$ Operationen
E · F [:= (2 x 3)] $\Rightarrow$ ergibt [4 x 3] $4 \cdot 3 \cdot 2 = 24$ Operationen
Reihenfolge spielt große Rolle:
$(((((A \cdot B) \cdot C) \cdot D) \cdot E) \cdot F) \rightarrow 84$ Operationen
$(A \cdot (B \cdot (C \cdot (D \cdot (E \cdot F))))) \rightarrow 69$ Operationen

$\rightarrow$ brauche `Optimale Klammerung`
Teilproblem: $A_i, ..., A_j$ wird geklammert $(A_i ... A_k)(A_{k+1} ... A_j)$
d. h. $\forall i \leq k \leq j$
berechne $(A_i ... A_k)$ und $(A_{k+1} ... A_j)$; multipliziere $A_{i,k}$ mit $A_{k+1,j}$. Wähle dazu bestes k.
`Beachte:` Innerhalb der Teilprobleme muss Klammerung optimal sein.
Also berechne nun Kosten für alle Teilprobleme $A_i \cdot \cdots \cdot A_j$ für alle $1 \leq i \leq j \leq n$. $\Rightarrow$ MIN[1, n] ist Minimum aller Multiplikationen für $A_1 \cdot A_2 \cdot ... \cdot A_n$.
Sei A_i Matrix mit Dimension $p_{i_1} \times p_i$
rekursiv $\Rightarrow$ MIN[i, j] = min$\{MIN[i,k] + MIN[k+1,j] + p_{i-1} \cdot p_k \cdot p_j\}$
MIN[i, j] = 0 für $i = j$
Speichern noch $s[i,j]$ ab mit k als den Wert, der MIN[i, j] bestimmt.

```
n := Anzahl Matrizen;
for i ∈ [1, n] do
    MIN[i, i] := 0;
end for
for l ∈ [2, n] do

    for i ∈ [1, n − l + 1] do
        j := i + l -1;
MIN[i, j] := ∞;

        for k ∈ [i, j − 1] do
            q := MIN[i, k] + MIN[k+1, j] + p_{i−1} · p_k · p_j
            if q < Min[i, j] then
                MIN[i, j] := q;
s[i, j] := k;
            end if
            end for
        end for
    end forreturn (MIN[1, n], s);
```

`Laufzeit:` 3 geschachtelte Vorschleifen mit Länge $\leq n$
innerer Schleifenrumpf: $\mathcal{O}(1)$
$\Rightarrow \mathcal{O}(n \cdot n \cdot n) = \mathcal{O}(n^3)$

genauer: l = 2, ..., n-1
$\sum_{l=2}^{n-1}(n-l)\cdot(l-1) = \sum_{l=2}^{n-1}(n-l-1)\cdot l = \sum_{l=2}^{n-1} nl - l^2 - l$

9.4 Greedy-Algorithmen

Optimierungsproblem: lokale Entscheidungen
$\rightarrow$ globales Optimum

9.4.1 Beispiel: Jobauswahl

n Jobs $a_1, ..., a_n$ mit Startzeit s_i, Endzeit e_i. a_i und a_j kompatibel, falls Intervalle nicht überlappen
Aufgabe: Maximale Anzahl kompatibler Jobs.
Bsp:

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
e_i	4	5	6	7	9	9	10	11	12	14	16

besser: $\{a_1, a_4, a_9, a_1 1\}$
Geht es besser?

9.4.2 dynamisches Programmieren

optimale Lösung besteht aus optimalen Teillösungen
$\rightarrow$ Wähle die beste Kombination aus optimalen Teillösungen.
Greedy: Betrachte nur *eine* Kombination, nicht alle.
Seien $S_{i,j}$ die Jobs, die starten, nachdem a_i endet, und die enden, bevor a_j startet.
Suche maximale Anzahl kompatibler Jobs in $S_{i,j}$.
Sei $A_{i,j}$ so eine Menge und sei $a_k \in A_{i,j}$. Ist $a_k \in$ OPT, so ergeben sich Teilprobleme $S_{i,k}$ und $S_{k,j}$ mit
$A_{i,k} = A_{i,j} \cap S_{i,k}$ und $A_{k,j} = A_{i,j} \cap S_{k,j}$ und $A_{i,j} = A_{i,k} \cup \{a_k\} \cup A_{k,j}$
Optimale Menge $A_{i,j}$ setzt sich zusammen aus a_k und sich ergebenen optimalen Mengen $S_{i,k}$ und $S_{k,j}$.

→ *Dynamisches Programmieren:* $c[i,j]$ optimale Anzahl von Jobs für $S_{i,j}$.
$c[i,j] = 0$ falls $S_{i,j} = \emptyset$
$c[i,j] = max\{c[i,k] + c[k,j] + 1\}$ ($a_k \in S_{i,j}$
→ Laufzeit: $\mathcal{O}(n^3)$

`Name:` Max. unabhängige Menge in Intervallgraphen
Greedy:
Idee: Wähle den Job als ersten, der als erstes endet.
→ a_1: lasse alle weg, die mit a_1 nicht kompatibel sind.
→ Iteriere.
Einfach, wenn Jobs sortiert nach Endzeitpunkten.
Zeige durch Widerspruch, dass allgemein immer eine optimale Lösung existiert, wo der Job mit dem frühesten Ende drin ist.
In optimaler Lösung: Kann den 1. Job, falls er nicht am frühesten endet, durch einen anderen Job ersetzen, der früher endet (Der Rest der Jobs davon nicht betroffen).

`Algorithmus:`
Felder s, e halten Index k (k letzter gefundener Job)
Jobs nach Endzeit sortiert.
Aufruf Jobauswahl(0) liefert a_o fiktiv mit $c_0 = 0$

```
function JOBAUSWAHL(k) i := k + 1;

    while i ≤ n ∧ s[i] < e[k] do
        i++;
    end while
    if i ≤ n then
        return {a_i} und Jobauswahl(i)
    else
       return ∅
    end if
end function
```

Laufzeit: $\mathcal{O}(n)$ jeder Job nur einmal betrachtet
`iterativ:`

```
A := {A_1};
k := 1;
for i ∈ [2, n] do

    if s[1 ≥ e[k] then
        A := A ∪{a_i}; k := 1;
    end if
    end forreturn A;
```

Allgemein:

- bestimme Teilstruktur des Problems
- rekursive Lösung
- Greedy-Variante: nur 1 Teilproblem
- zeige, dass Greedy zum Optimum führt
- rekursiver Algorithmus
- iterativer Algorithmus

Geht das immer?
Beispiel: Gewichtetes Rucksackproblem
n Gegenstände $g_1, ..., g_n$; g_i hat Wert v_i und Gewicht w_i. Träger kann $\leq W$ Kilogramm tragen.
Maximiere Gesamtwert der Ladung.

`dynamische Programmierung:`
Wertvollste Ladung mit $\leq W$ Kg Gewicht besteht aus einem g_j mit Wert v_j und dem Rest mit Gewicht $\leq W - w_j$.
Greedy:
Strategie:

- g_j mit größtem Wert
- g_j mit kleinstem Gewicht
- g_j mit größtem Wert pro Kilogramm

Beispiel:
kleinstes Gewicht; w = 50
$w_1 = 10 \quad v_1 = 280 \quad \rightarrow w_1, w_2 \rightarrow 380$
$w_2 = 20 \quad v_2 = 100$
$w_3 = 30 \quad v_3 = 120$ besser ist w_1, w_3

größter Wert pro Kilo;
$w_1 = 10 \quad v_1 = 60 \quad \rightarrow (greedy) w_1, w_2 \rightarrow 380$
$w_2 = 20 \quad v_2 = 100$
$w_3 = 30 \quad v_3 = 120$ besser ist w_2, w_3

`dynamische Programmierung:` $\mathcal{O}(n \cdot W)$
−−−−−−−−−−−−−−−−−−−−−−
1 | 2 | 3 |..... |17| ..... |Q|.....|W|
−−−−−−−−−−−−−−−−−−−−−−

Wertvollste Ladung mit Gewicht = 23 (//sollte überarbeitet werden)

Variante: Gegenstände sind teilbar.
Greedy: (funktioniert hier)

```
Ordne Gegenstände nach Wert pro Kilogramm;
Lege g_1 in den Rucksack;
i := 2
while W nicht erreicht do
    Stecke g_i in den Rucksack; i++;
end while                          // Teile g_i auf und fülle Rucksack mit Teil von g_i;
```

Laufzeit: $\mathcal{O}(n)$

Abbildungsverzeichnis

Algorithmenverzeichnis