Informatik 3 – Algorithmen und Datenstrukturen

Vorlesung: Prof. Harald Hempel
Mitschrift: Max Tandetzky

WS 2007/08

Inhaltsverzeichnis

Organisatorisches

- Zugangsvoraussetzung zur Klausur 50% der Übungsserienpunkte
- keine Online-Testate
- Skripte im Netz, z.B. von Henker
- Übungsserie wird montags ausgegeben und die darauffolgende Woche montags vor der Vorlesung eingesammelt

0 Literatur

- CLR (Cormen, Leisesson, Rivest: Design of Analysis of Algorithmus)

1 Grundlagen

- Herkunft des Wortes Algorithmus: Mathematiker, dessen Namen ähnlich klingt
- Algorithmen terminieren immer und sind deterministisch
- Effizienz von Algorithmen ist sehr wichtig (*Zeit (Anzahl der Operationen und Vergleiche)*, Speicherplatzbedarf)
- Wie wird gemessen (z.B. die Anzahl der Operationen)? Problem ist, dass ein Algorithmus bei einer Eingabe schneller sein kann und ein anderer bei einer anderen. Siehe Abbildung 1 auf Seite 2.

1.1 Zeitbedarf von Algorithmen

- Algorithmus α
- Rechenzeit: t_α, T_α
- Eingabe: ω
- $T_\alpha(w)$ = Anzahl der Operationen und Vergleiche, die α bei Eingabe ω tätigt
- Größe der Eingabe ω: $n = |\omega|$ (Zahl der Buchstaben des Strings)
- $t_\alpha(n) = \max\{T_\alpha(w) : |w| = n\}$ (worst case time)
- in der Praxis sind desweiteren average case time und best case time relevant, da die Eingaben, die sehr viel Zeit brauchen nicht immer häufig vorkommen. Siehe Abbildung 2 auf Seite 2.

Beispiel: Algorithmus zum Sortieren von Zahlen (insertion sort) Eingabe $A[1 \dots n]$ (Array) Darstellung des Verfahrens siehe Abbildung 3 auf Seite 3.

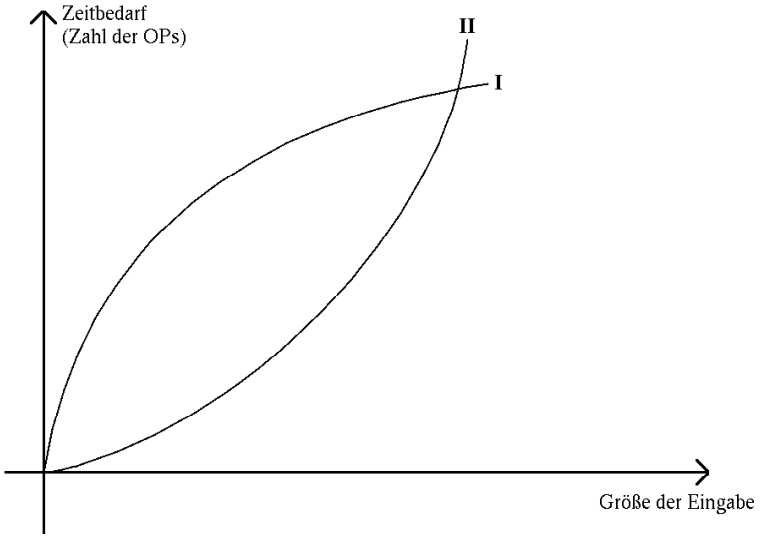


Abbildung 1: Zeitbedarf zweier Algorithmen

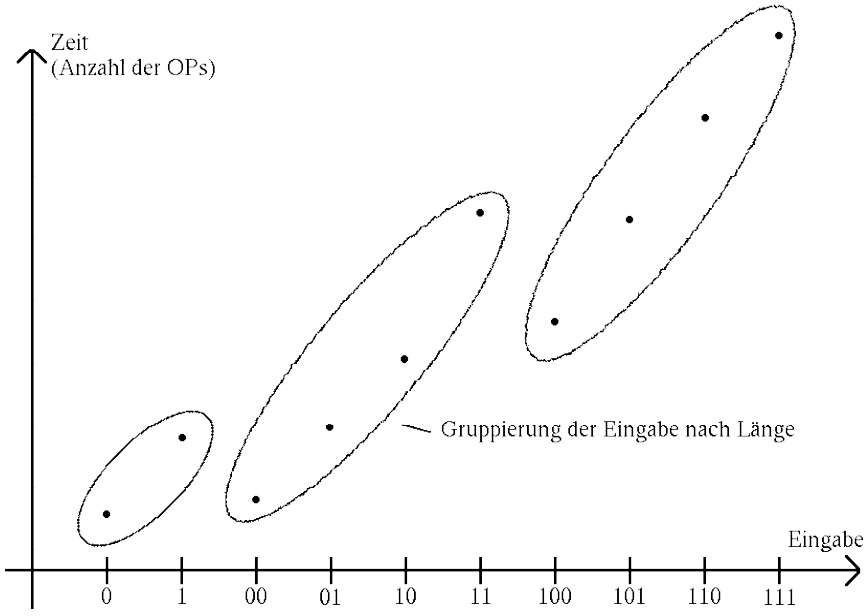


Abbildung 2: Eingaben nach Länge

```
(1) FOR i=2 TO length(A) DO
(2)   hans := A[i]
      //füge hans in bereits sortierten Teil A[1], A[2], ..., A[i-1] ein
(3)   j:=i-1
(4)   WHILE (j>0 und A[j]>hans) DO
(5)      A[j+1]:=A[j]
(6)      j:=j-1
      END
(7)   A[j+1]=hans
    END
```

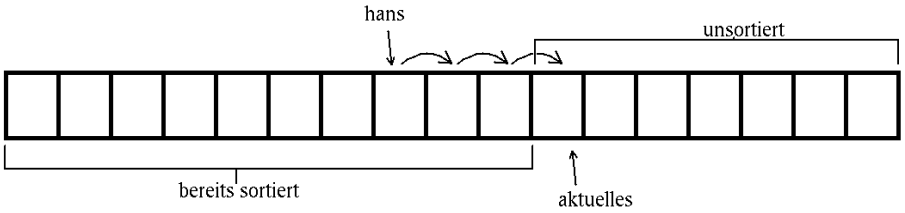


Abbildung 3: Insertion Sort

Zeitbedarf bei Eingabe der Größe n, A[1,... n]

Zeile	Anzahl der Ausführungen	Kosten der Ausführung
1	$n(-1)$ mal	c_1
2	$n-1$ mal	c_2
3	$n-1$ mal	c_3
4	$\sum_{i=2}^{n} \underbrace{t_i}_{\text{Anzahl, wie oft die WHILE-Bedingung für i getestet wird}}$	c_4
5	$\sum_{i=2}^{n}(t_i-1)$	c_5
6	$\sum_{i=2}^{n}(t_i-1)$	c_6
7	$n-1$	c_7

$$\Rightarrow t_\alpha(n) = c_1*(n-1)+c_2(n-1)+c_3(n-1)+c_4*\sum_{i=2}^{n} t_i+(c_5+c_6)*\sum_{i=2}^{n}(t_i-1)+(n-1)*c_7$$

$$a*n-b+c\sum_{i=2}^{n} t_i \text{ für geeignete Konstanten a, b, c}$$

best case: $t_i = 1$ für alle i (d.h. Eingabe ist bereits sortiert)

$$\Rightarrow T_\alpha(n) = d*n+e \text{ für geeignete Konstanten } d, e$$

worst case: $t_i = i$ für alle i (d.h. Eingabe ist falschrum sortiert)

$$\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$$

$$\Rightarrow t_\alpha(n) = f * n^2 + g * n + h \text{ für geeignete Konstanten } f, g, h$$

average case: W – eine Menge von sich ausschließenden Ereignissen (hier die Menge der möglichen Zeitbedarfe unseres Algorithmus)
Jedem $w \in W$ liegt eine Wahrscheinlichkeit zu Grunde, mit der w eintritt – p_w

$$\sum_{w \in W} p_w = 1$$

Die Zufallsvariable X (hier der Zeitbedarf) möge stets einen Wert aus W annehmen. Der Erwartungswert von X, $E[X]$ ist gerade: $E[X] = \sum_{w \in W} w * p_w$

p_w gibt Auskunft über die Wahrscheinlichkeit über das Auftreten eines bestimmten Zeitbedarfs.

uns interessiert:

$$\begin{aligned} E\left[\sum_{i=2}^{n}(t_i - 1)\right] &= \sum_{i=2}^{n} E[t_i] - 1 \\ &= \sum_{i=2}^{n} \frac{i-1}{2} \\ &= \frac{1}{4} n * (n-1) \end{aligned}$$

1.2 Asymptotische Notation

$$n^2 + 1 > n^2$$
$$2n^2 > n^2$$
$$2^n > n^2$$

1.2.1 Definitionen

Sei $g : \mathbb{N} \to \mathbb{R}$ (z.B. Zeitfunktion)

$$\mathcal{O}(g) =_{\text{def}} \{f : \mathbb{N} \to \mathbb{R} \mid (\exists c \in \mathbb{R} : c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)[0 \leq f(n) \leq c * g(n)]\}$$

$$\Omega(g) =_{\text{def}} \{f : \mathbb{N} \to \mathbb{R} \mid (\exists c \in \mathbb{R} : c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\,[0 \leq c * g(n) \leq f(n)]\}$$

$$\Theta(g) =_{\text{def}} \Omega(g) \cap \mathcal{O}(g)$$

$$\mathfrak{o}(g) =_{\text{def}} \{f : \mathbb{N} \to \mathbb{R} \mid (\forall c \in \mathbb{R} : c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\,[0 \leq f(n) \leq c * g(n)]\}$$

$$\omega(g) =_{\text{def}} \{f : \mathbb{N} \to \mathbb{R} \mid (\forall c \in \mathbb{R} : c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\,[0 \leq c * g(n) \leq f(n)]\}$$

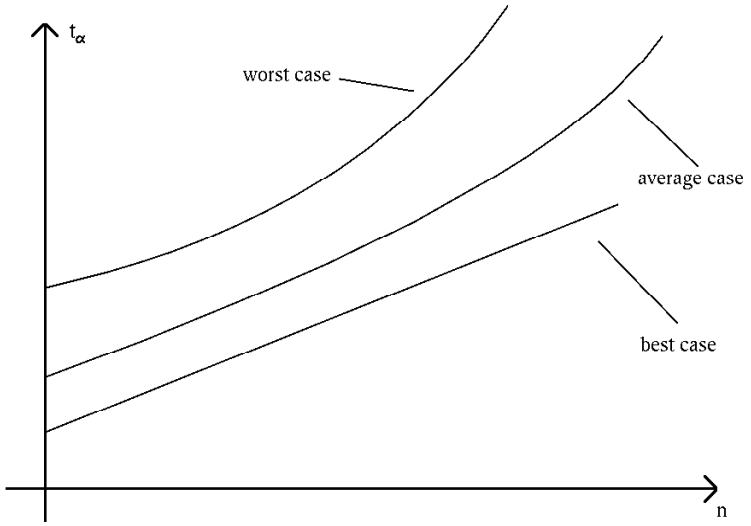


Abbildung 4: Unterschiedliche Fälle

Beispiel

- $g = \mathrm{Id},\ g(n) = n$

$$\mathcal{O}(g) = \left\{ g, 2g, \frac{1}{2}g, \dots \right\}$$

$$\mathcal{O}(n) = \{ n, 2n, \log n, \dots \}$$

Siehe Abbildung 5 auf Seite 6

Erläuterungen

- $f \in \mathcal{O}(g)$ (in einigen Büchern auch $f = \mathcal{O}(g)$)
  - f wächst nicht schneller als g
  - f lässt sich durch Konstante * g „abfangen“ / majorisieren
  - $n \in \mathcal{O}(n^2)$ weil $(\exists c : c > 0)(\exists n_0)(\forall n \geq n_0)[0 \leq \underbrace{f(n) \leq c * g(n)}_{n \leq c * n^2}]$
  - $n \in \mathcal{O}(\frac{1}{100}n^2)$ gilt für $c = 100, n_0 = 0$ und auch für $c = 1, n_0 = 100$
- $f \in \Omega(g) \Leftrightarrow g \in \mathcal{O}(f)$, g wächst nicht schneller als f
- $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$ bedeutet, dass f gleichschnell wächst wie g
  - $n^2 \in \Theta(\frac{1}{100}n^2)$
  - $n \notin \Theta(n^2)$, da $n \in \mathcal{O}(n^2)$ aber $n \notin \Omega(n^2)$

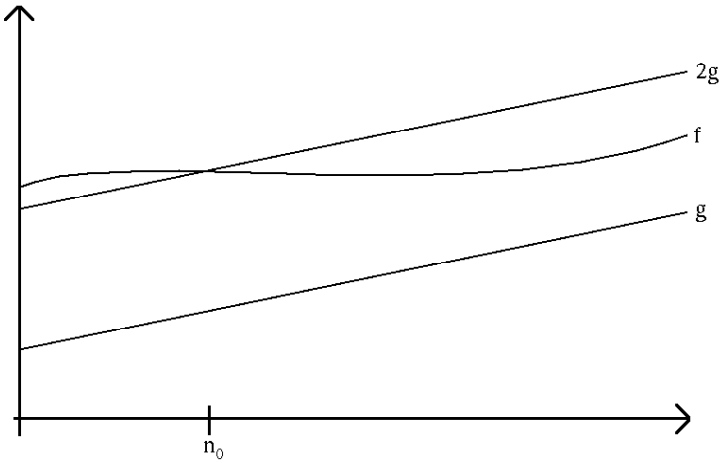


Abbildung 5: $\mathcal{O}$-Notation

- Unterschied zwischen $\mathfrak{o}$ und $\mathcal{O}$ ist, dass Forderung bei $\mathfrak{o}$ *für alle* Konstanten gelten muss
  - $f \in \mathfrak{o}(g)$ – f wächst langsamer als g, bzw. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
  - $n \in \mathfrak{o}(n^2)$ Sei $c > 0$ beliebig. Es gibt stets ein $n_0 \in \mathbb{N}$, so dass für alle $n \geq n_0 \;\; n \leq c * n^2$ gilt, nämlich für $n_0 = \frac{1}{c}$
- $f \in \omega(g) \Leftrightarrow g \in \mathfrak{o}(f)$

Für die Funktionen $f(n) = n^2$, $g(n) = n \log n$ gilt

- $g \in \mathcal{O}(f)$ und sogar $g \in \mathfrak{o}(f)$ und $f \in \Omega(g)$, $g \notin \Omega(f)$

Bemerkung

$$R_{\mathcal{O}} =_{\text{def}} \{(f, g) : f \in \mathcal{O}(g)\}$$

Die Relation hat folgende Eigenschaften, sie ist:

- reflexiv
- transitiv
- nicht symmetrisch
- nicht antisymmetrisch

Damit ist diese Relation eine Quasihalbordnung, damit gibt es eine Äquivalenzrelation

$$f \lesseqgtr g \Leftrightarrow_{\text{def}} f \in \mathcal{O}(g) \wedge \underbrace{g \in \mathcal{O}(f)}_{f \in \Omega(g)} \Leftrightarrow f \in \Theta(g)$$

Bemerkung Es ergibt sich folgende Hierarchie der Funktionen:

1. $\alpha(n)$ (die inverse Ackermann-Funktion)
2. $\log^*(n)$
3. $log^\nu(n)$ $\nu > 2$
4. $(\log\log n)^\epsilon$ ϵ beliebig
5. $(\log n)^\delta$ $\delta < \gamma$
6. $(\log n)^\gamma$ γ beliebig
7. n^β $\beta < 1$
8. n
9. n^α $\alpha > 1$
10. $n^\alpha * (\log n)^\xi$
11. n^θ $\theta \geq \alpha + 1$
12. A^n $A > 1$
13. $A^n n^\tau$
14. B^n $B > A$

Erläuterungen
Es sei $\log = \log_2$, $\ln = \log_e$

$$\log^k(n) = \underbrace{\log(\log(\ldots \log n \ldots))}_{k\text{-mal}}$$

$$\log^*(n) = \min\{i : \log^i(n) \leq 1\}$$

$\log^*(0) =_{\text{def}} 0$, $\log^*(1) = 0$, $\log^*(2) = 1, \log^*(4) = 2$, $\log^*(16) = 3 \ldots$, $\log^*(65536) = 4$

- $\log^*$ wächst extrem langsam, aber $\lim_{n\to\infty} \log^*(n) = \infty$
- α wächst langsamer als $\log^*(n)$; $\alpha \;\widehat{=}$ Inverses der Ackermann-Funktion

Vergleich von Funktionen (Laufzeiten von Algorithmen)

- insertion sort Laufzeit ist $\mathcal{O}(n^2)$

$$\hat{a} * n^2 + \hat{b} * n + c$$

Laufzeit	Problemgröße, die in einer Zeit bewältigt werden kann	Problemgröße, die ein 10x schnellerer Rechner in Zeit T bewältigen kann
n	s	$10 * s$
$n \log n$	s	$\approx 10 * s$
n^2	s	$\approx 3,16 * s$
n^3	s	$\approx 2,15 * s$
2^n	s	$s + 3,3$

Problemgrößen, die in der Zeiteinheit bei einem Algorithmus mit gegebener Laufzeit von einem Rechner, der 1 Millionen Operationen pro Sekunde ausführen kann, erledigt werden können:

	1sec	1min	1h	1d	1 Jahr
n	10^6	$6 * 10^7$	$3,6 * 10^9$	$8,64 * 10^{10}$	$3,13 * 10^{13}$
$n \log n$	62764				
n^2	1000	7745	60.000	293.939	$5,6 * 10^6$
n^3					
2^n	20	25	31	36	44

2 Sortieren

- Sortieren erleichtert das Wiederfinden
- Sortierkriterien (Schlüssel) sind wichtig
- 25% der kommerziellen Rechenleistung fließt in Sortierprozesse

Zunächst: Sortieren durch Vergleichen und Tauschen

- gegeben:
  - $S_1 =$ | key(1) | information(1) |
    $\vdots$
  - $S_n =$ | key(n) | information(n) |
- Suche Permutation $i_1, i_2, \ldots, i_n$ der Indizes $1, \ldots, n$ derart, dass key(i_1)$\leq$key(i_2)$\leq \ldots \leq$key(i_n).
- idealisiert:

  - gegeben: Liste von natürlichen Zahlen $< a_1, \ldots, a_n >$
  - gesucht: Permutation (Umordnung) $< a_1^{'}, \ldots, a_n^{'} >$ von $< a_1, \ldots, a_n >$ sodass $a_1^{'} \leq a_2^{'} \leq \ldots \leq a_n^{'}$.

- divide and conquer
- Idee:
  - Probleme in Teilprobleme Zerlegen (Teile)
  - Teilprobleme lösen
  - Lösung der Teilprobleme zu einer Lösung des Ausgangsproblems zusammensetzen (Herrsche/Kombiniere)

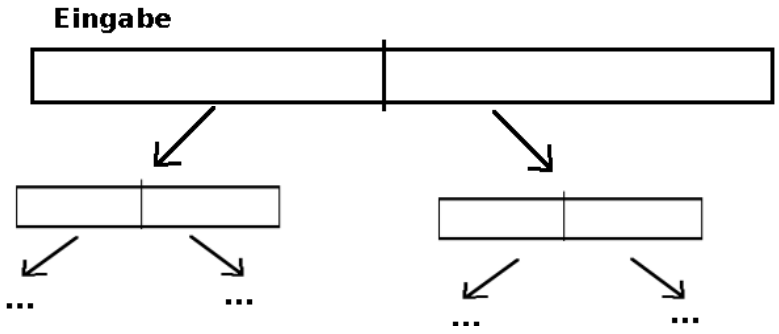


Abbildung 6: Mergesort (Schema)

Mergesort

- Eingabe $< a_1, \ldots, a_n >$
- Durchführung
  - divide: zwei Teillisten $< a_1, \ldots, a_{\frac{n}{2}} >, < a_{\frac{n}{2}+1}, \ldots, a_n >$ erzeugen
  - conquer: diese Teillisten rekursiv sortieren
  - combine: sortierte Teillisten $< a_1^{'}, \ldots, a_i^{'} >$ zusammenfügen

Zum Sortieren:

- haben: zwei sortierte Stapel, kleinste Karte oben
- vergleichen die beiden oben liegenden Karten der zwei Stapel und legen kleinere von beiden in „Ausgabestapel“
- wiederhole bis beide Stapel abgeräumt

siehe Abbildung 7 auf Seite 10

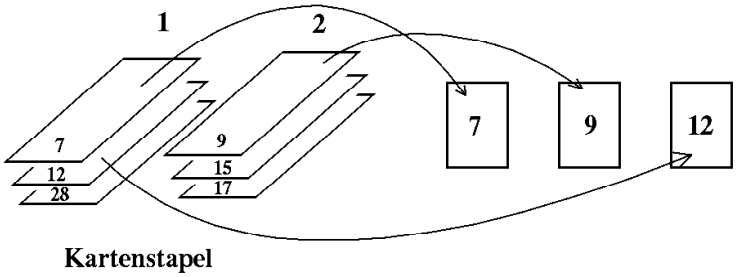


Abbildung 7: Mergesort Schema

Bemerkung in Implementierung evt. Karte ∞ zuerst in jeden Stapel

MERGE(A, p, q, r)

$$\underbrace{A[p] \dots A[q]}_{\substack{\text{linke} \\ \text{Teilliste sortiert}}} \quad \underbrace{A[q+1] \dots A[r]}_{\substack{\text{rechte} \\ \text{Teilliste sortiert}}}$$

```
1   n1:=q-p+1
2   n2:=r-q
3   create array L[1,...,n1+1] and R[1,...,n2+1]
4   for i=1 to n1
5       do L[i]:=A[p+i-1]
6   for j=1 to n2
7       do R[j]:=A[q+j]

8   L[n1+1]: = ∞
9   R[n_2+1]: = ∞

10  i:=1
11  j:=1
12  for k=p to r
13      do if L[i] <= R[j]
14          then A[k]:=L[i]
15          i:=i+1
16      else
17          A[k]:=R[j]
18          j:=j+1
```

MERGESORT(A, p, r) wollen $A[p], \dots, A[r]$ sortieren

```
1   if p < r then do

2       q:=⌊(p+r)/2⌋
```

```
3       MERGESORT(A,p,q)
4       MERGESORT(A,q+1,r)
5       MERGE(A,p,q,r)
```

Vergleich InsertionSort und MERGESORT

InsertionSort

- viele Vergleiche
- wenig Umsortieren

MERGESORT

- wenig Vergleiche
- viel Umsortieren, Umräumen

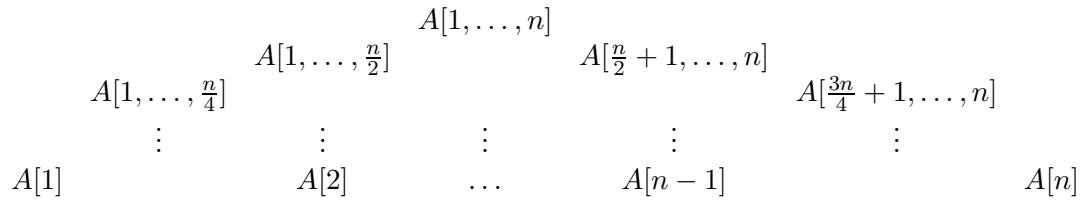
Laufzeitanalyse für MERGESORT

Sei $T(n)$ die Anzahl der Vergleiche und Operationen, die MERGESORT im schlimmsten Fall bei Eingabe von n Zahlen (Feld der Größe n) durchführt.

Wie groß ist $T(n)$? Wie bestimmt man es?

Verschiedene Methoden zum Bestimmen von $T(n)$

a) Rekursionsbaum



Aufwand der Rekursion(-sebenen)

$$1 * (\underbrace{c_1 * n}_{\text{teile}} + \underbrace{c_2 * n}_{\text{kombiniere}}) = c_1 n + c_2 n$$

$$2 * (c_1 * \frac{n}{2} + c_2 * \frac{n}{2}) = c_1 n + c_2 n$$

$$4(c * \frac{n}{4} + c_2 * \frac{n}{4}) = c_1 n + c_2 n$$

$$\vdots$$

$$n * 1$$

Es gibt $\log n$ Zeilen, damit ergibt sich ein Gesamtaufwand von:

$$\approx \text{Konstante} * n * \log n \Rightarrow T(n) \in \mathcal{O}(n \log n)$$

Offenbar gilt ebenso

$$T(n) = \underbrace{T\left(\frac{n}{2}\right)}_{\text{rek. Aufruf}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{rek. Aufruf}} + \underbrace{\hat{c} * n}_{\text{kombiniere}}$$

In Wirklichkeit sieht die Rekurrenz für $T(n)$ aber so aus:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \hat{c} * n$$

Da uns bei Laufzeiten aber nur das asymptotische Verhalten interessiert und die Analyse mit den exakten Werten lediglich die Konstante vor der asymptotischen Laufzeit beeinflusst, werden wir in Zukunft stets mit $\frac{n}{2}$ (oder ähnlich idealisierten Werten) bei der Laufzeitanalyse arbeiten.

2.1 Rekurrenzen

- treten bei Laufzeitanalyse häufig auf, insbesondere bei divide-and-conquer Algorithmen und Algorithmen, die sich selbst rekursiv aufrufen

Die Rekurrenz von MERGESORT war $T(n) = 2 * T\left(\frac{n}{2}\right) + \hat{c} * n,\ T(1) = 0$

a) Rekursionsbaum

b) Iterationsmethode

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + \hat{c}n \\
T\left(\frac{n}{2}\right) &= 2T\left(\frac{n}{4}\right) + \hat{c}\frac{n}{2} \\
&= 2\left(2T\left(\frac{n}{4}\right) + \hat{c}\left(\frac{n}{2}\right)\right) + \hat{c}n \\
&= 4T\left(\frac{n}{4}\right) + 2\hat{c}n \\
&= 4\left(2T\left(\frac{n}{8}\right) + \hat{c}\left(\frac{n}{4}\right)\right) + 2\hat{c}n \\
&= 8T\left(\frac{n}{8}\right) + 3\hat{c}n \\
&\vdots \\
&\text{für } n = 2^k \text{ ergibt sich} \\
&= 2^k * T\left(\frac{n}{2^k}\right) + k\hat{c}n = \underbrace{k}_{=\log n}\hat{c}n \\
&= \hat{c}n \log n
\end{aligned}$$

c) Substitutionsmethode

$$T(n) = 2 * T\left(\frac{n}{2}\right) + \hat{c} * n,\ T(1) = 0$$

Vermutung: $T(n) \leq \hat{c} * n \log n$ (Vermutung zu gewinnen ist oft schwer)

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

Beispiele für schlechte Vermutungen Vermutung: $T(n) \leq k * n$

$$\begin{aligned}
T(n) = 2T\left(\frac{n}{2}\right) + \hat{c} * n &\leq 2 * k * \frac{n}{2} + \hat{c} * n \\
&= k * n + \hat{c} * n \not\leq k * n \\
T(n) &\leq k * n
\end{aligned}$$

$\Rightarrow$ Vermutung war schlecht, denn $k * n$ ist keine obere Schranke

2. Vermutung: $T(n) \leq k * n^2$

$$\Rightarrow T(n) = 2 * T\left(\frac{n}{2}\right) + \hat{c} * n \leq 2 * k * (\frac{n}{2})^2 + \hat{c} * n$$

$$k * \frac{n^2}{2} + \hat{c} * n \leq k * n^2 \text{ für große } n$$

d) Variablensubstitution Sei $T(n) = 2 * T(\sqrt{n}) + \log n$

Setzen $m := \log n$, d.h. $n = 2^m$

$$\Rightarrow T(2^m) = 2 * T\left(2^{\frac{m}{2}}\right) + m$$

$$\text{Setzen } S(m) := T(2^m)$$

$$\begin{aligned}
\Rightarrow S(m) &= 2 * S\left(\frac{m}{2}\right) + m \\
\Rightarrow S(m) &\in \mathcal{O}(m \log m)
\end{aligned}$$

Wegen $S(m) = T(2^m) = T(n)$ und $m = \log n$ folgt $T(n) \in \mathcal{O}(\log n * \log \log n)$

Ganz allgemein kann man Rekurrenzen der Form $T(n) = a * T\left(\frac{n}{b}\right) + f(n),\ a \geq 1, b > 1, f(n) \geq 0$ für genügend große n mit so genannten Mastertheorem einfach lösen. Dabei ist $f(n)$ der Aufwand für Teile und Zusammenfügen siehe Abbildung 8 auf Seite 14

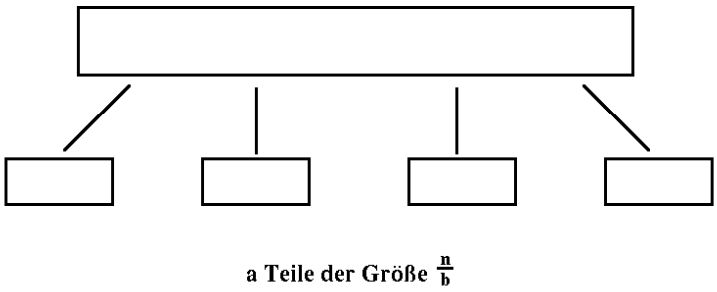


Abbildung 8: Teile- und Herrsche-Schema

2.2 Mastertheorem

Seien $a \geq 1$ und $b > 1$ Konstanten, $f : \mathbb{N} \to R$ eine Funktion und $T : \mathbb{N} \to \mathbb{R}$ eine durch die folgende Rekurrenz definierte Funktion:

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

Dabei interpretieren wir $\frac{n}{b}$ als $\lfloor \frac{n}{b} \rfloor$ oder $\lceil \frac{n}{b} \rceil$.

$T(n)$ besitzt dann folgende asymptotische Schranken:

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{falls } f \in \mathcal{O}(n^{\log_b a - \epsilon}) \text{ für ein } \epsilon > 0 \\ \Theta(n^{\log_b a} * \log n) & \text{falls } f \in \Theta(n^{log_b a}) \\ \Theta(f(n)) & \text{falls } f \in \Omega(n^{log_b a + \epsilon}) \text{ für ein } \epsilon > 0 \text{ und falls } a * f(\frac{n}{b}) \leq c * f(n) \\ & \text{für ein } c < 1 \text{ und genügend große } n \end{cases}$$

Bemerkung

- in allen drei Fällen des Mastertheorems wird das Wachstum von f mit dem Wachstum von $n^{\log_b a}$ verglichen.
- „größere“ der beiden Funktionen bestimmt das Wachstum von T

zu Fall 1

- $f(n)$ nicht nur asymptotisch kleiner als $c * n^{\log_b a}$, sondern

$$f(n) \leq c * n^{\log_b a - \epsilon} = c * \frac{n^{log_b a}}{n^{\epsilon}}$$

zu Fall 3

- $f(n)$ nicht nur asymptotisch größer als $c * n^{\log_b a}$, sondern

$$f(n) \geq \hat{c} * n^{\log_b a + \epsilon} = \left(\hat{c} * n^{log_b a}\right) * n^{\epsilon}$$

- außerdem muss f der Regularitätsbedingung

$$a * f\left(\frac{n}{b}\right) \leq c * f(n)$$

für ein $c < 1$, n genügend groß genügen

Die Fallunterscheidung hat Lücken (es gibt Funktionen, die in keinen der drei Fälle Fallen).

Beispiele

1. f ist kleiner als $n^{\log_b a}$ aber nicht polynomial kleiner, also $f(n) \leq n$, $f(n) \not\leq n^{0,999}$
   Sei $a = 2$, $b = 2$

$$T(n) = 2 * T\left(\frac{n}{2}\right) + \frac{n}{\log n}$$

   Denn $f \in \mathfrak{o}(n)$ und $f \notin \mathcal{O}(n^{1-\epsilon})$ für irgendein $\epsilon > 0$

2. f ist größer als $n^{log_b a}$ aber nicht polynomial größer
   $f(n) \geq n$, $f(n) \not\geq n^{1,001}$
   Sei $a = 2$, $b = 2$

$$T(n) = 2 * T\left(\frac{n}{2}\right) + n \log n$$

   Denn $n \in \mathfrak{o}(n \log n)$ aber für alle $\epsilon > 0$ $f \notin \Omega\left(n^{1+\epsilon}\right)$

3. f erfüllt Regularitätsbedingung nicht
   Seien $a = 2$, $b = 2$

$$T(n) = 2 * T\left(\frac{n}{2}\right) + f(n)$$

   $f(n) = n$ gilt? $2 * \frac{n}{2} \leq c * n$ für ein $c < 1$ und genügend große n gilt nicht
   $f'(n) = n^2$ gilt? $2 * \left(\frac{n}{2}\right)^2 \leq c * n^2$ für ein $c < 1$ und genügend großen n

$$\frac{n^2}{2} \leq c * n^2 \text{ gilt für } c = \frac{3}{4} \text{ und alle } n$$

aber in vielen Fällen lässt sich das Mastertheorem anwenden

Beispiele

- $a = 9,\ b = 3,\ f(n) = n$

$$T(n) = 9 * T\left(\frac{n}{3}\right) + n$$

$\log_b a = \log_3 9 = 2$

$$\begin{array}{ccc} f(n) & = & n^{\log_b a} \\ n & & n^2 \quad \epsilon = 0,5 \end{array}$$

$$n \leq n^{1,5} = n * \sqrt{n} \Rightarrow \text{ 1. Fall des Mastertheorems } T(n) = \Theta(n^2)$$

- $a = 1,\ b = \frac{3}{2},\ f(n) = 1$

$$T(n) = 1 * T\left(\frac{2n}{3}\right) + 1$$

$\log_b a = \log_{\frac{3}{2}} 1 = 0$

$$\begin{array}{ccc} f(n) & = & n^{\log_b a} \\ 1 & & n^0 = 1 \end{array}$$

$$\Rightarrow \text{2. Fall des Mastertheorems } \Rightarrow T(n) \in \Theta(\log n)$$

- $a = 3,\ b = 4,\ f(n) = n * logn$

$$T(n) = 3 * T\left(\frac{n}{4}\right) + n \log n$$

$$f(n) = n \log n$$

$\log_b a = \log_4 3 :\ \frac{1}{2} < \log_4 3 < 1$

$$\begin{array}{ccc} f(n) & & n^{\log_b a} \\ n * \log n & > n > & n^{\log_4 3} \end{array}$$

$$\Rightarrow f(n) \in \Omega\left(n^{\log_4 3 + \epsilon}\right)\ \epsilon = 0,1$$

Regularitätsbedingung?

$$3 * f\left(\frac{n}{4}\right) \leq c * f(n)$$

$$\Rightarrow 3 * \frac{n}{4} * \log \frac{n}{4}$$

$$a * f\left(\frac{n}{b}\right) \leq c * f(n)$$

$$\text{d.h. } 3 * \frac{n}{4} * \log \frac{n}{4} \stackrel{?}{\leq} c * n \log n$$

$$3 * \frac{n}{4} * (\log n - 2) \leq c * n \log n$$

Aber: *linke Seite* $\leq \frac{3}{4} n \log n$.

$$\frac{3}{4} n \log n \leq c * n \log n \text{ für } c = \frac{3}{4} \text{ und alle } n$$

$\Rightarrow$ 3. Fall Mastertheorem $\Rightarrow T(n) \in \Theta(n \log n)$

2.3 Heapsort

Für Sortieralgorithmen ist unter anderem auch folgendes Kriterium wichtig:

> Werden während des Sortierprozesses die zu sortierenden Zahlen in eingegebenen Feld umsortiert oder müssen viele von ihnen extern zwischengespeichert und behandelt werden?

Begriffsbildung: imfelde-Sortieralgorithmus Ist ein Sortieralgorithmus, bei dem nur eine konstante Anzahl von Zahlen zu jedem beliebigen Zeitpunkt außerhalb des Eingabefeldes zwischengespeichert werden müssen.

Beispiele

Algorithmus	imfelde	Zeitbedarf
InsertionSort	ja	$\mathcal{O}(n^2)$
MergeSort	nein	$\mathcal{O}(n \log n)$
HeapSort	ja	$\mathcal{O}(n \log n)$

neue Datenstruktur: *heap* (effiziente Prioritätsschlange)
heap = array A, das man als eine Art (fast) vollständigen Binärbaum auffasst

- vollständiger Binärbaum: alle Blätter haben gleiche Entfernung zur Wurzel
- fast vollständiger Binärbaum: Abstand von zwei Blättern zur Wurzel unterscheidet sich maximal um 1, Baum wird „von links“ aufgefüllt

heap ist ein Feld mit zwei Attributen:

- length(A) – Zahl der Elemente in A
- heap-size(A) – Zahl der Elemente von A, die innerhalb der heap-Struktur von A gespeichert sind

$$\text{heapsize}(A) \leq \text{length}(A)$$

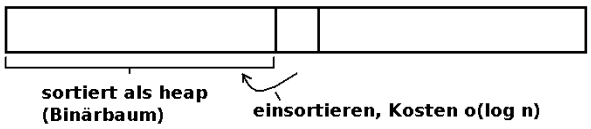


Abbildung 9: HeapSort Schema

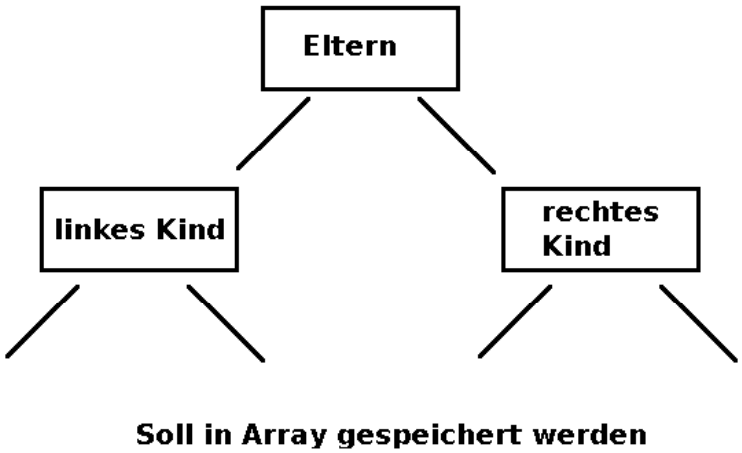
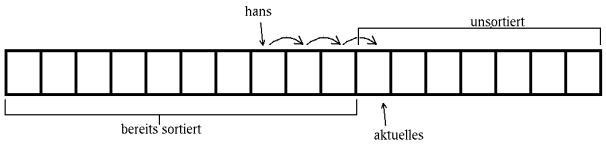


Abbildung 10: Binärbaum

Grundidee von HeapSort

- zum Vergleich: InsertionSort:



- der Vorteil der Binärbaumstruktur: beim Einsortieren müssen nicht mehr alle Elemente durchlaufen werden

Binärbaum im Array speichern:

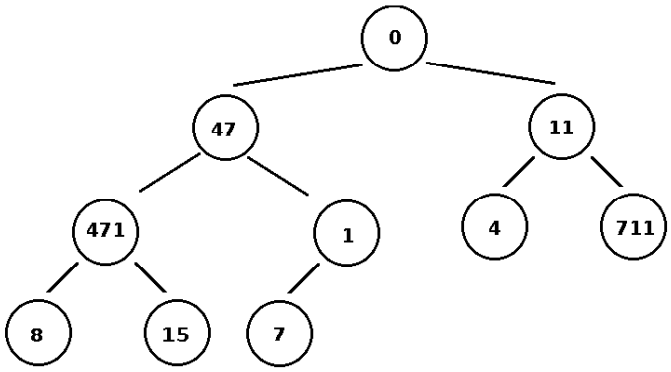
- Wurzel $\rightarrow_{\text{def}} A[1]$
- Ist $A[i]$ ein Knoten in Baum, so befinden sich der Elternknoten sowie linkes und rechtes Kind, left(i) bzw right(i) in den Feldpositionen

$$\text{parent}\,(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

$$\text{left}(i) = 2i$$

$$\text{right}(i) = 2i + 1$$

Bemerkung Berechnung dieser 3 Funktionswerte (Position der Knoten) ist besonders effizient möglich (bitshift)



Schreibe Daten von links nach rechts in Feld

Feldposition	0	47	11	471	1	4	711	8	15	7
	1	2	3	4	5	6	7	8	9	10

Abbildung 11: Heap-Sort Beispiel

Beispiel

sortierte Folge im Binärbaum speichern Man unterscheidet zwischen max-heap und min-heap

max-heap-Eigenschaft: Für jeden Knoten $i > 1$ (alle Knoten außer der Wurzel) gilt:

$$A[\text{parent}(i)] \geq A[i]$$

im Elternknoten steht also eine größere Zahl. Das bedeutet

1. größter Wert (Maximum) steht in der Wurzel
2. in keinem der beiden Teilbäume, die an Knoten i hängen gibt es Werte größer als $A[i]$

min-heap-Eigenschaft wie bei max-heap-Eigenschaft jedoch: $A[\text{parent}(i)] \leq A[i]$

Probleme

1. Wie genau funktioniert das Einsortieren bei Erhaltung der max-heap-Eigenschaft
2. Wie wird aus dem gesamten max-heap die Ausgabe, d.h. die sortierte Folge ausgelesen?

Beispiel Abbildung 12 auf Seite 20 ist ein Beispiel für einen Heap. Dieser Baum erfüllt die max-heap-Eigenschaft

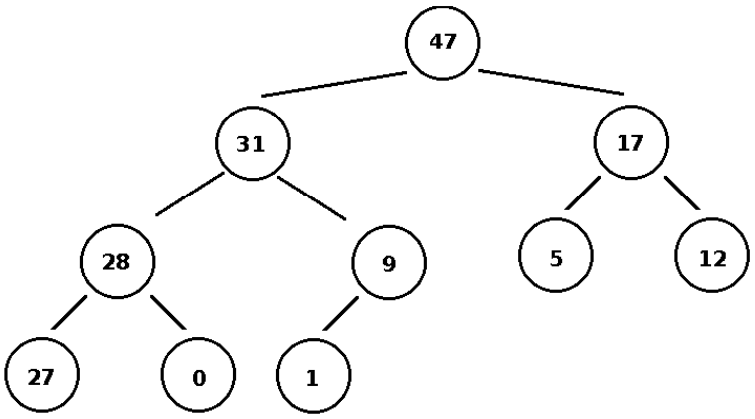


Abbildung 12: Beispiel-Heap

Bemerkungen

- wir werden max-heaps verwenden
- heaps sind fast vollständige Binärbäume und somit kann man zeigen, dass ein heap mit n Knoten eine Höhe von $\Theta(log\, n)$ hat.
- wir werden gleich feststellen, dass alle vernünftigen Operationen auf heaps (einfügen, entfernen,...) in Zeit $\Theta(n)$ möglich sind

a) Erhalten der heap-Eigenschaft (Einfügen):

Situation: Array A, Index i. Siehe Abbildung 13 auf Seite 20

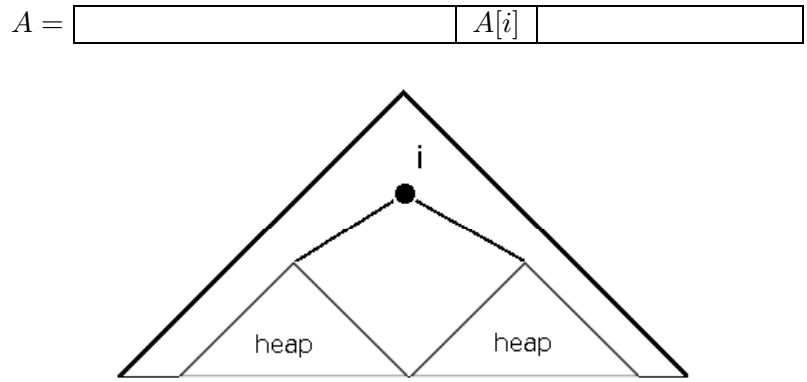


Abbildung 13: 2 Heaps mit kleinem Element als Wurzel

Die in left[i] und right[i] verwurzelten Teilbäume sind max-heaps, aber $A[i]$ ist kleiner als A[left[i]] bzw. A[right[i]], d.h. an Knoten i ist die max-heap Eigenschaft verletzt. Siehe Abbildung 14 auf Seite 21.

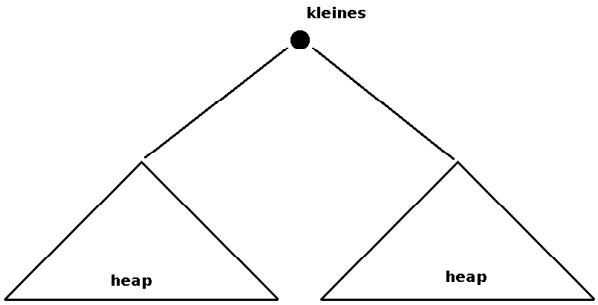


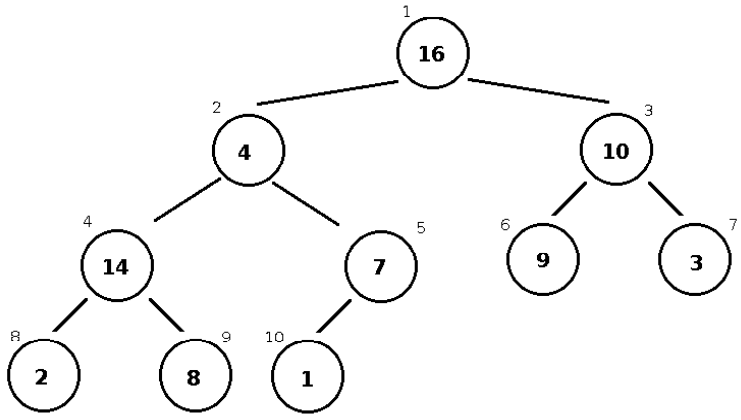
Abbildung 14: Kleines Element als Wurzel von 2 Heaps

folgender Algorithmus stellt die heap-Eigenschaft wieder her:

MAX-HEAPIFY (A, i)

```
1    l :=left(i)
2    r := right(i)
3    if l <= heap-size(A) and A[l] > A[i] then größter := l
4        else größter := i
5    if r <= heap-size(A) and A[r] > A[größter] then größter := r
6    if größter != i then do
7        vertausche A[i] und A[größter]
8        MAX-HEAPIFY (A,größter)
9    od
```

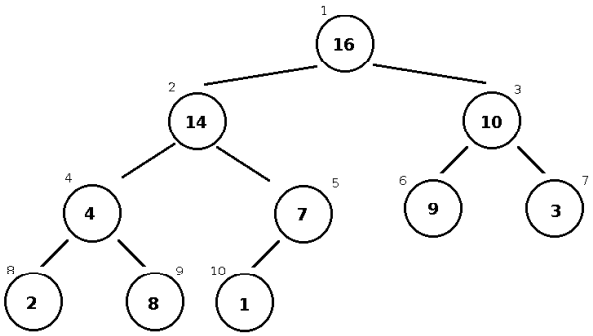
Beispiel



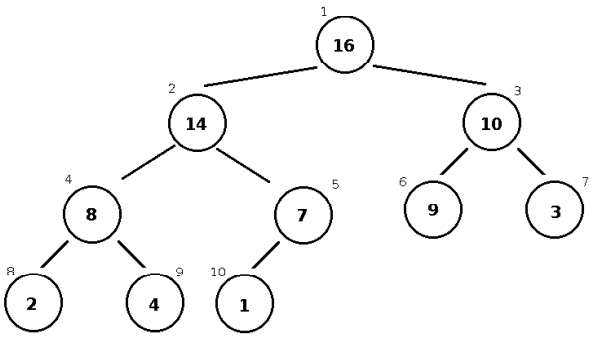
$A =$

16	4	10	14	7	9	3	2	8	1

An Knoten mit Index 2 ist die max-heap-Eigenschaft verletzt $\Rightarrow$ rufe `MAX-HEAPIFY(A,2)` auf.



$\Rightarrow$ `MAX-HEAPIFY(A,4)`



Zeitanalyse Zeile 1-7 $\Theta(1)$, Zeile 8 (MAX-HEAPIFY$\begin{pmatrix} A, & \text{left}(i) \\ & \text{right}(i) \end{pmatrix}$

Sind in an i verwurzelten Teilbaum k Knoten, wieviele Knoten sind dann höchstens im linken bzw. rechten Teilbaum von i

Anzahl der Konten der Teilbäume insgesamt n Knoten.

Gesamtbaum:

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

$$= 2^{k+1} - 1 + 2^k$$

$$= 3 * 2^{k+1} - 1$$

linker Teilbaum:

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

$$= 2 * 2^k - 1$$

Damit ergibt sich für MAXHEAPIFY für folgende Rekurrenz:

$$T(n) \leq T(\frac{2}{3}n) + \Theta(1)$$

Rekurrenz lösen: Mastertheorem
$a = 1$, $b = \frac{3}{2}$, $f(n) = 1$
$n^{\log_b a} = n^0 = 1$

$$\Rightarrow T(n) \in \Theta\left(n^{\log_b a} * \log\, n\right)$$

$$\Rightarrow T(n) \in \Theta(\log\, n)$$

$\Rightarrow$ Ist die Höhe eines heaps gerade h, so ist die Laufzeit von MAX-HEAPIFY(A,i) gerade $\mathcal{O}(h)$

b) Aufbau eines Heaps

MAX-HEAPIFY von unten nach oben in Baum anwenden um ein Feld $A[1, \ldots, n]$, wobei $n = \text{length}(A)$, in einen max-heap zu überführen.

BUILD-MAX-HEAP(A)

```
1    heapsize(A) := length(A)
2    for i := ⌊length(A)/2⌋ downto 1
3         do MAX-HEAPIFY(A,i)
```

Beispiel siehe Abbildung 15 auf Seite 24

$A =$

4	1	3	2	16	9	10	14	8	7

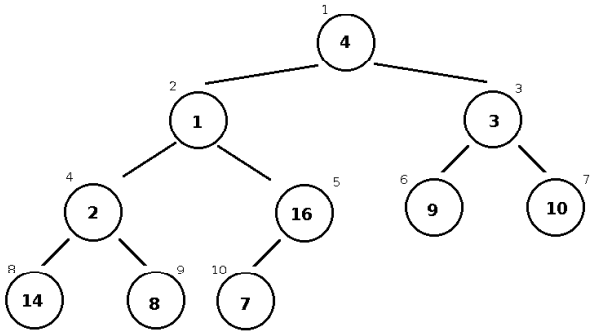


Abbildung 15: Beispiel für ein Ausgangsfeld für BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
```

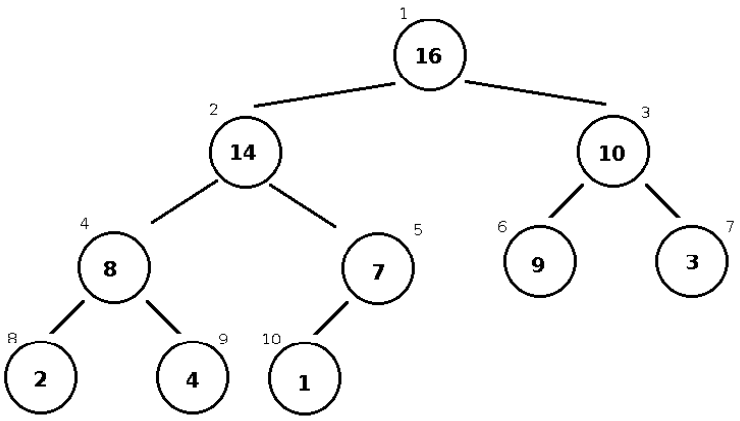
```
heapsize=10

MAX-HEAPIFY (A,5)
MAX-HEAPIFY (A,4)
MAX-HEAPIFY (A,3)
MAX-HEAPIFY (A,2)
MAX-HEAPIFY (A,1)
```

Vorgehen:

MAX-HEAPIFY (A,4): Tausche Element mit Index 4 und 8
MAX-HEAPIFY (A,3): Tausche Element mit Index 3 und 7
MAX-HEAPIFY (A,2): Tausche Element mit Index 2 und 5
MAX-HEAPIFY (A,1): Tausche Element mit Index 1 und 2

Resultierender heap



Korrektheitsbeweis für den Algorithmus

Wieso produziert der Algorithmus einen heap? Wir zeigen dies, indem wir folgende Schleifeninvariante betrachten.

* Zu Beginn einer jeden Ausführung der in der for-Schleife enthaltenen Befehle sind die Knoten $i+1, i+2, \ldots$ Wurzeln von max-heaps.

Wir zeigen jetzt, dass * vor dem ersten Durchlauf der for-Schleife gilt, dass jeder Durchlauf der for-Schleife die Eigenschaft * erhält und dass * in der Tat die Korrektheit des Algorithmus impliziert, wenn die for-Schleife beendet ist.

Vor dem ersten Durchlauf: $i = \frac{n}{2} = \left\lfloor \frac{\text{length}(A)}{2} \right\rfloor$ Die Knoten mit Index $\frac{n}{2}+1, \frac{n}{2}+2, \ldots, n$ Sind Blätter im Baum und somit Wurzeln von max-heaps.

Erhaltung während eines Durchlaufs sei $k := i$
wissen left$(k) > h$ und right $(k) > k$
$\Rightarrow$ da * vor dem Durchlauf mit k gilt, sind left(k) und right(k) Wurzeln von max-heaps. Das ist aber gerade die Voraussetzung, dafür, dass beim Aufruf von MAX-HEAPIFY(A,k) in der for-Schleife der Knoten mit Index k zur Wurzel eines max-heaps befördert wird.

$\Rightarrow$ auch k ist nun Wurzel von max-heaps und zudem stört MAX-HEAPIFY(A,k) die heap-Eigenschaft in den Knoten $k+1, k+2, \ldots$ nicht

$\vdots$

BUILD MAX-HEAP
for $i = \left\lfloor \text{length}\frac{A}{2} \right\rfloor$ downto 1
HEAPIFY(A,i)

$\vdots$

Beendigung der Schleife $i = 0$ (nach dem $i = 1$ Durchlauf).

Gemäß * sind alle Knoten mit Index $i + 1, i + 2, \ldots,$ d.h. alle Knoten des Heap sind Wurzeln von Max-Heaps. Also auch die Wurzel der gesamten Datenstruktur. Folglich erzeugt der Algorithmus BUILD-MAX-HEAP in der Tat einen Max-Heap

Zeitanalyse

Pro HEAPIFY $\mathcal{O}(\log n)$, insgesamt $\frac{n}{2}$ Aufrufe von MAXHEAPIFY $\Rightarrow \mathcal{O}(n \log n)$

Das ist zu grob!

Der Zeitbedarf von MAXHEAPIFY(A, i) hängt von Höhe des Knoten i ab:

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

Die meisten Knoten im Heap haben eine geringe Höhe.

Beobachtung 1 Heap mit n Knoten hat Höhe von $\lfloor \log n \rfloor$

Beobachtung 2 $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ Knoten haben die Höhe h

Genauere Zeitabschätzung für BUILD-MAX-HEAP:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^{h+1}} \underbrace{\mathcal{O}(h)}_{c*h} = \frac{c * n}{2} \underbrace{\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}}_{\to 2} \in \mathcal{O}(n)$$

c) Sortieren mit Hilfe eines Heaps

Idee `Rufe BUILD-MAX-HEAP auf`
`wiederhole ( maximum auslesen → heapsize verringern → HEAPIFY )`

HEAP-SORT

```
1   BUILD-MAX-HEAP(A)
2   for i=length(A) downto 2
3       do vertausche A[1] und A[i]
4           heapsize(A):=heapsize(A)-1
5           MAX-HEAPIFY(A,1)
```

Beispiel

4	1	3	2	16	9	10	14	8	7

$i = 10,$ heapsize $= 10$

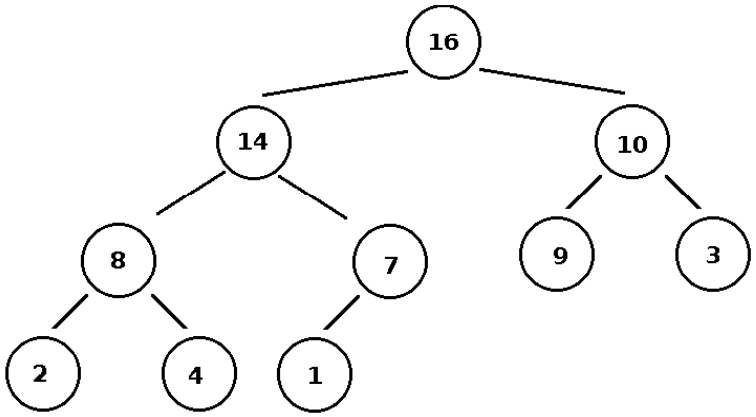


Abbildung 16: Resultierender Max-Heap

Algorithmus schrittweise durchgeführt

16	14	10	8	7	9	3	2	4	1	i=10, heapsize=10
⇓										
1	14	10	8	7	9	3	2	4	16	i=9, heapsize=9
MAX-HEAPIFY										
⇓										
14	10	8	7	9	3	2	4	1	16	i=9, heapsize=9
⇓										
1	10	8	7	9	3	2	4	14	16	i=8, heapsize=8

⋮

Laufzeitabschätzung von HEAP-SORT

Zeile	Aufwand
1	$\mathcal{O}(n)$
2	$n-1$ viel
3	
4	
5	

$\Rightarrow$ Aufwand insgesamt: $\mathcal{O}(n \log n)$

Exkurs: Prioritätsschlangen (priority queue)

Datenstruktur zum Speichern von Daten mit Schlüsseln

Schlüssel	Info

Eine MAX-Prioritätenschlange unterstützt folgende Operationen (d.h. folgende Operationen sind besonders effizient möglich):

(MAX-)INSERT(S,x)	einfügen eines neuen Objektes x in die Datenstruktur (die Menge der Objekte S)
EXTRACT-MAX(S)	liefert das Element von S mit *größtem* Schlüssel und entfernt es aus S
MAXIMUM(S)	liefert das Element von S mit *größtem* Schlüssel
INCREASE-KEY(S,x,k)	erhöht den Wert des Schlüssels vom Dateneintrag x auf k

Beispiel Anwendungen:

- Druckerwarteschlange
- Job-Scheduling auf PC

Es gibt auch Min-Prioritätsschlangen

Prioritätenschlangen lassen sich gut mit Heaps realisieren (Wurzel ist Maximum)

Vergleich Zeitaufwand der Operationen bei verschiedenen Datenstrukturen

Operation	**Heap**	**Doppelt verkettete Liste**
HEAP-MAXIMUM(A)		
return A[1]	$\mathcal{O}(1)$	$\mathcal{O}(n)$
EXTRACT-MAX(A)		
if(heapsize(A)<1)		
then error "heapunderflow"		
max := A[1]		
A[1]:=A[length(A)]		
heapsize(A)=heapsize(A)-1	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
MAX-HEAPIFY(A,1)		
return max		
INCREASE-KEY(S,i,key)		
if key < A[i]		
then error "new key is		
smaller then current key"		
A[i]:=key		
while i> 1 and A[parent(i)] < A[i]	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
do vertausche A[i]↔ A[parent(i)]		
i:=parent(i)		
od		

(MAX-)INSERT(A,key)		
heapsize(A):=heapsize(A)+1	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
A[heapsize(A)]=$-\infty$		
INCREASE-KEY(A,heapsize(A),key)		

2.4 Quicksort

- geht auf Hoare zurück (60er Jahre)
- worst case $\Theta(n^2)$
- average case $\Theta(n \log n)$ mit sehr kleinen Konstanten

Idee Teile und Herrsche:

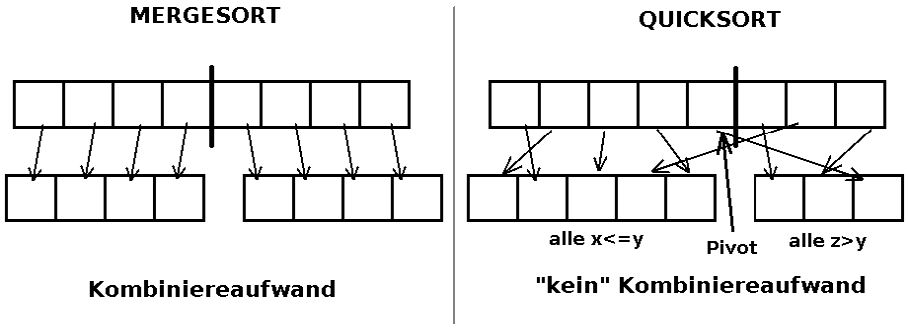


Abbildung 17: Vergleich: Mergesort, Quicksort

Vorteil von Quicksort

- quasi kein Aufwand die Teillisten wieder zu kombinieren

Nachteil von Quicksort

- Index von x muss berechnet werden

Ideen für x

$$A[1], \quad \frac{A[1] + A[n]}{2}$$

Beides nicht günstig

$$\frac{\sum_{i=1}^{n} A[i]}{n}$$

auch „verbesserungswürdig“, da z.B. in Feld mit einer sehr kleinen Zahl und vielen sehr großen Zahlen ein zu kleines Pivot-Element gewählt wird

- teile: siehe Abbildung 17 auf Seite 29
- herrsche: Teilfelder rekursiv sortieren
- kombiniere: Teilfelder hintereinander setzen

Quicksort

```
QUICKSORT(A,p,r)
1   if (r-p) < 3 then sortiere mit Hand/brute force
2       else x <-- MEDIAN(A,p,r)
3            q <-- PARTITION(A,p,r,x)
4            QUICKSORT(A,p,q-1)
5            QUICKSORT(A,q+1,r)
```

siehe Abbildung 18 auf Seite 30

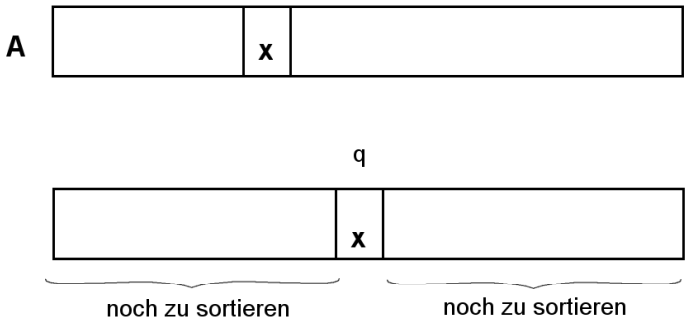


Abbildung 18: Element mit etwa mittlerer Wertigkeit

Wie bestimmt man ein gutes Pivotelement?

Ein paar Hilfsbegriffe: Sei $S = \{s_1, s_2, \ldots, s_n\}$ eine Menge von (paarweise verschiedenen) natürlichen Zahlen und sei $x \in S$. Der Rang von x bezüglich S ist definiert als

$$\text{rang}(x, S) = \text{card } \{y \in S : y < x\}$$

Beispiele / Bemerkungen

- $S = \{7, 12, 5, 8, 17\}$

$$\text{rang}(8, S) = 2$$

$$\text{rang}(17, S) = 4$$

- sind die Elemente in S geordnet, d.h. $s_1 < s_2 < s_3 < \cdots < s_n$, dann ist $\text{rang}(s_i, S) = i - 1$
- Ziel für QUICKSORT: Pivotelement soll ein $x \in A$ sein, so dass $\text{rang}(x, A) \approx \frac{n}{2}$.

Formalisierung dieses Teilproblems

- gegeben:
  - $\underbrace{\text{Feld } A \text{ / Menge } S}_{\substack{\text{wird gleichbe-}\\\text{deutend verwendet}}} = \{\underbrace{s_1, \ldots, s_n}_{\substack{\text{paarw.}\\\text{verschieden}}}\}$
  - $k \in \mathbb{N}$, $0 \leq k \leq n$
- gesucht: das Element aus S mit Rang k

Bemerkung

untere Schranke	obere Schranke	Erklärung
	$\mathcal{O}(n^2)$	für alle x $\text{rang}(x, S)$ ausrechnen
$\Omega(n)$		Jedes Element muss mindestens einmal betrachtet werden
	$\mathcal{O}(n \log n)$	S sortieren, k leicht zu bestimmen

SELECT(S, k) Sei b eine Konstante, z.B. $b = 3$ oder $b = 5$, suche Element mit Rang k.

```
1   if card S < b then sortiere S (z.B. mit INSERTIONSORT) und gib das
    k+1te Element der sortierten Folge aus

2   else
        zerlege S in ⌈card (S)/b⌉ Teilmengen TS_1, TS_2, ..., TS_⌈card (S)/b⌉ von
        denen ⌈card (S)/b⌉ genau b Elemente enthalten und eine die rest-
        lichen Elemente enthalten (hintereinander weg)
3       sortiere Teilfolge, und zwar jede TS_j (mit INSERTIONSORT) und
        bestimme das mittlere Element m_j einer jeden Teilfolge
        (card TS_j ≡ 0 und 2, so ist das mittlere Element das größere
        der beiden in der Mitte stehenden Elemente) (mittleres Element mit
        Index ⌈cardTS_j/2⌉ in sortierter Folge)
4       Rufe SELECT rekursiv auf, um das mittlere Element m der m_j zu be-
        stimmen, d.h. SELECT({m_1, m_2, ..., m_⌈card (S)/b⌉}, ⌈(⌈card (S)/b⌉+1)/2⌉)

5       Zerlege S in drei Teilfolgen von Elementen unter Ausnutzung von m
        in
```

```
            S1 - Elemente < m
            S2 - Elemente = m
            S3 - Elemente > m
            (das macht man mit PARTITION)
6       if card(S1) >= k then SELECT(S1,k)[1]
        else if card(S1)+card(S2) >= k then return m
        else SELECT(S3,k-card(S1)-card(S2))
```

siehe Abbildung 19 auf Seite 32.

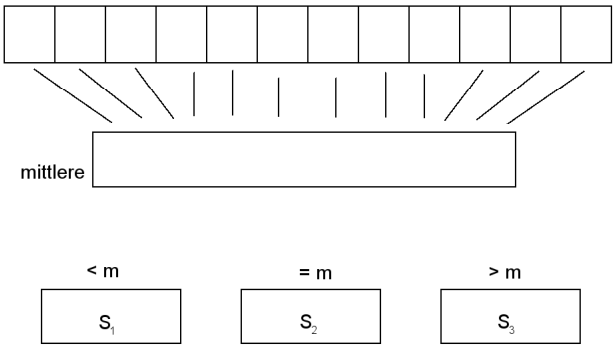


Abbildung 19: Auswahl des mittleren Elements

Beispiel

$$S = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\} \; k = 5$$

Sei $b = 3$

Zerlege Eingangsfolge in Teilfolgen

$$TS_1 = \underbrace{(4, 1, 3)}_{(1,3,4} \; TS_2 = \underbrace{(2, 16, 9)}_{(2.9,16)} \; TS_3 = \underbrace{(10, 14, 8)}_{(8,10,14)} \; TS_4 = \underbrace{(7)}_{(7)}$$

$$\Rightarrow m_1 = 3, \quad m_2 = 9, \quad m_3 = 10, \quad m_4 = 7$$

```
SELECT({3,9,10,7}, 2)
```

$$TS'_1 = \underbrace{(3, 9, 10)}_{(3,9,10)} \; TS'_2 = \underbrace{(7)}_{(7)}$$

$$m'_1 = 9 \; m'_2 = 7$$

```
SELECT({9,7},1)
```

$$m = 9$$

[1] denn wenn in S_1 mehr als k-viele Elemente liegen, muss k auch in S_1 liegen

$$S_1 = (4,1,3,2,8,7)\ S_2 = (9)\ S_3 = (16,10,14)$$

gehen mit $m = 9$ in Programmzeilen 5 und 6 zurück, errechne S_1 bis S_3

$$S_1 = (4,1,3,2,8,7)$$

$$S_2 = (9)$$

$$S_3 = (16,10,14)$$

Wegen card $(S_1) > k \rightarrow$ `SELEKT`(S_1, k)

$$TS_1'' = \underbrace{(4,1,3)}_{(1,3,4)}\ TS_2'' = \underbrace{(2,8,7)}_{(2,7,8)}$$

$$m_1'' = 3\ m_2'' = 7$$

`SELECT({3,7},1)`

$$\Rightarrow m'' = 7$$

Zeile 5 und 6

$$S_1'' = (4,1,3,2)$$

$$S_2'' = (7)$$

$$S_3'' = (8)$$

Ausgabe 8 ???

Zeitabschätzung für SELEKT

Zeile	Zeitaufwand
1	$\Theta(1)$
2	$\Theta(n)$
3	$\underbrace{\left\lceil \frac{\text{card }(S)}{b} \right\rceil}_{\approx \frac{n}{b}}$ viele Teilfolgen, jede benötigt $\Theta(1)$ also insgesamt: $\Theta(n)$
4	$\left\lceil \frac{\text{card } S}{b} \right\rceil$ viele Mediane $\Rightarrow T\left(\frac{n}{b}\right)$
5	$\Theta(n)$
6	Rekursiver Aufruf von SELEKT bezüglich S_1 oder S_3. Wie groß können S_1 oder S_3 im worst case sein? Für jedes m_i gilt: $\frac{b}{2}$ Elemente in B_i sind $\geq m_i$. Für m gilt: $\frac{1}{2}\frac{\text{card } S}{b}$ der m_i sind $\geq m \Rightarrow \underbrace{\frac{1}{2}\frac{\text{card } S}{b} * \frac{b}{2}}_{\frac{\text{card } S}{4}}$ viele Elemente von S sind $\geq m \Rightarrow$ card $S_1 \leq \frac{3}{4}n$. Analog überlegt man sich card $S_3 \leq \frac{3}{4}n$

Rekurrenz:

$$T(n) = T\left(\frac{n}{b}\right) + T\left(\frac{3}{4}n\right) + \Theta(n)$$

Ziel

$$\frac{n}{b} + \frac{3}{4}n < n$$

für $b = 5$ ergibt sich $\frac{n}{5+\frac{3}{4}n}\frac{19}{20}n < n$

$$\Rightarrow T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + \Theta(n)$$

Substitutionsmethode Vermutung: $T(n) \leq c * n$

$$\Rightarrow T(n) \leq c * \frac{n}{5} + c\frac{3}{4}n + \underbrace{\Theta(n)}_{k*n}$$

$$= \frac{19}{20} * c * n + k * n \overset{\text{soll}}{\underset{\text{sein}}{<}} c * n$$

$$\Rightarrow kn < \frac{1}{20}cn$$

$$\Rightarrow 20k < c$$

$$\Rightarrow T(n) \in \Theta(n)$$

zurück zu QUICKSORT

$$\text{MEDIAN}(A, p, r) \;\widehat{=}\; \text{SELEKT}\;(A, \left\lceil \frac{r-p-1}{2} \right\rceil)$$

$\Rightarrow$ Laufzeit MEDIAN: $\Theta(n)$

Schema von Partition

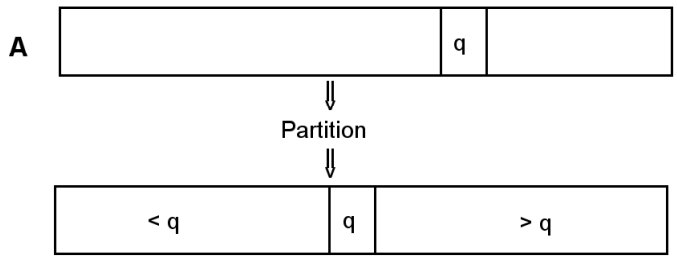


Abbildung 20: Schema von Partition

```
PARTITION(A,p,r,x) // x ist Median von A, hat Index q im Feld A
1   i := p
2   j := r
3   repeat
4       while j <= r and A[i] < x do i := i+1 od
5       while j >= p and A[j] > x do j := j-1 od
6       if i < j then vertausche A[i] <==> A[j]
7       else return j+1
8   (until false)
```

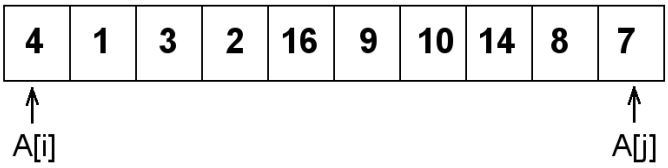


Abbildung 21: Ausgangssituation von Partition

Für $x = 10$: $i = 1, j = 10$. Wandern so lange, bis $i = 5 \rightarrow$ erste While-Schleife durchlaufen. Die zweite While-Schleife wird sofort wieder verlassen, $A[5]$ und $A[10]$ werden getauscht.

i wird erhöht, bis $A[i] \not< x$, also $i = 7$, $j = 10$. Verringere j um eins. Vertausche $A[7]$, $A[9]$

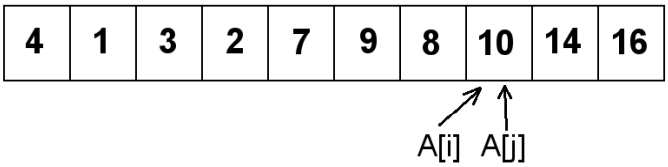


Abbildung 22: Endzustand von Partition

Ergebnis von PARTITION Am Ende gilt:

$$\forall_k p \leq k \leq i \text{ gilt } A[k] \leq x$$

$$\forall_k j < k \leq r \text{ gilt } A[k] \geq x$$

Zeitanalyse Jede while-Schliefe wird maximal $r - p + z$ mal durchlaufen

$$\Rightarrow \mathcal{O}(r - p)$$

$$\Rightarrow \text{ Zeitbedarf } \mathcal{O}(n)$$

Zeitbedarf für QUICKSORT

- Median in $\mathcal{O}(n)$
- Partition in $\mathcal{O}(n)$

$+$ rekursiver Aufruf

$\Rightarrow T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + \Theta(n)$ im schlimmsten Fall, da Median das Element mit Rang $\lceil \frac{n}{2} \rceil$

$$= \Theta(n \log n)$$

Randbemerkung: klassische Variante von QUICKSORT $x \leftarrow A[1]$, wir $x \leftarrow$ MEDIAN

worst case:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

$$\Rightarrow T(n) \in \Theta(n^2)$$

Aber: Die Wahrscheinlichkeit, im ersten Feld jeder (rekursiv zerlegten) Liste immer ein extremes Element zu haben ist sehr unwahrscheinlich, siehe Abbildung 23 auf Seite 36.

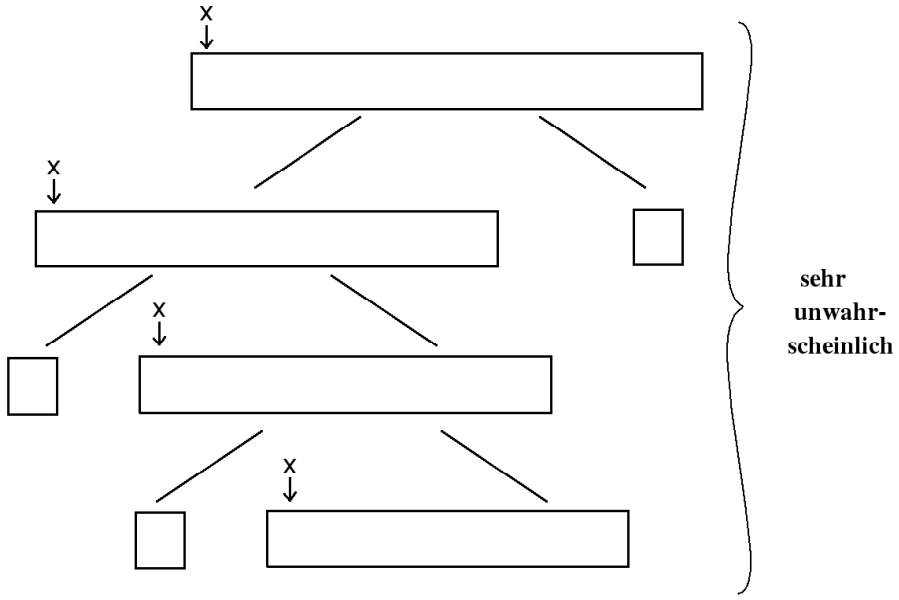


Abbildung 23: Worst-case Verteilung von Quicksort (klassische Variante)

Bemerkung

- es gibt viele weitere Sortierverfahren

Beispiel

STOOGESORT

```
STOOGESORT(A,p,r)
1   if A[p] > A[r} then vertausche A[p] <==> A[r]
2   if p+1 >= r then return A[p,r]
3   k:= ⌊(r−p+1)/3⌋
4   STOOGESORT(A,p,r-k)
5   STOOGESORT(A,p+k,r)
6   STOOGESORT(A,p,r-k)
```

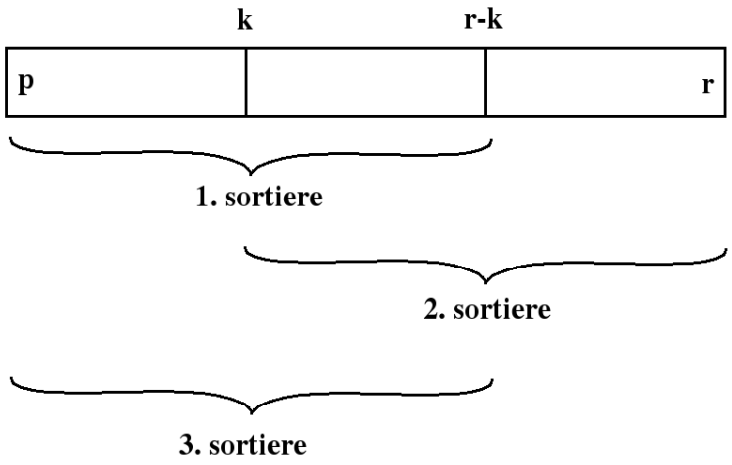


Abbildung 24: Schema von Stoogesort

2.5 Sortieren in Linearzeit

Bisher: Sortieren in $\mathcal{O}(n \log n)$

Frage: Geht es noch schneller? $\mathcal{O}(n \log \log n), \mathcal{O}(n)$?

- zum Sortieren von Zahlen (ohne zusätzliches Wissen über die Zahlen) werden Vergleiche benötigt: $x > y,\ x < y,\ x \geq y,\ x \leq y$.
- geht man davon aus, dass alle Zahlen paarweise verschieden sind, so genügt es offensichtlich nur Vergleiche der Form $x \leq y$ anzustellen
- wir wollen zunächst die Anzahl der zum Sortieren von n Zahlen benötigten Vergleiche betrachten
- hierfür benutzen wir den Begriff des *Entscheidungsbaumes*

Entscheidungsbaum Sei SoAL ein Sortieralgorithmus. Der zu SoAL gehörende Entscheidungsbaum T ist wie folgt definiert:

- Knoten von T: $(i, j) \;\widehat{=}$ Vergleich $A[i] \overset{?}{\leq} A[j]$
- Wurzel von T: derjenige Knoten ($\hat{i}, \hat{j}$, sodass $A[\hat{i}] \overset{?}{\leq} A[\hat{j}]$, der erste Vergleich ist, den SoAL bei Eingabe von $A[1, \ldots, n]$ durchführt
- T ist ein Binärbaum
- Kinder eines Knoten (i, j): Knoten (i_1, j_1) und (i_2, j_2), derart, dass SoAL im Falle von $A[i] \leq A[j]$ als nächsten Vergleich den Vergleich $A[i_1] \overset{?}{\leq} A[j_1]$ durchführt und im Falle von $A[i] > A[j]$ der nächste Vergleich gerade $A[i_2] \overset{?}{\leq} A[j_2]$ ist.
- Blattebene: jedes Blatt enthält eine Permutation π der Indizes $1, 2, \ldots, n$ derart, dass $A[\pi(1)] \leq A[\pi(2)] \leq \ldots A[\pi(n)]$

Beispiel

- Eingabe:

5	3	1	7
A[1]	A[2]	A[3]	A[4]

(1,2,3,4)

1	3	5	7
A[3]	A[2]	A[1]	A[4]

(3,2,1,4)

- INSERTIONSORT

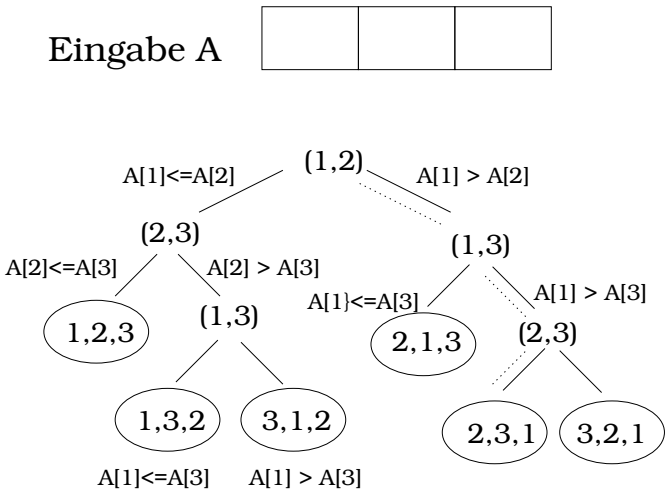


Abbildung 25: Entscheidungsbaum von INSERTIONSORT mit einem dreielementigen Array

- MERGESORT

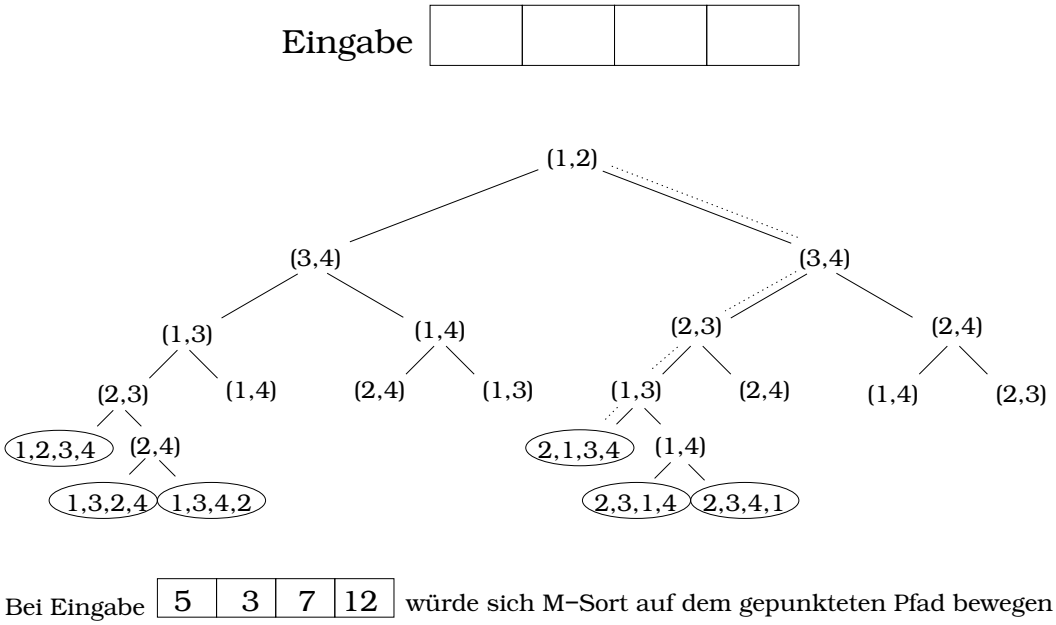


Abbildung 26: Entscheidungsbaum von MERGESORT mit einem vierelementigen Array

Beobachtungen:

1. Ein auf Vergleichen basierender Sortieralgorithmus kann nur dann korrekt sein,

wenn jede der $n!$ Permutationen der Zahlen $1, \ldots, n$ in einem Blatt des Entscheidungsbaumes steht.

2. Die Länge des längsten Pfades entspricht der worst-case-Anzahl von Vergleichen, die der Algorithmus durchführen muss. $\Rightarrow$ Höhe des Baumes = worst-case-Anzahl Vergleiche

3. Eine untere Schranke für die Höhe von Binärbäumen (Entscheidungsbäumen), in denen alle Permutationen als Blätter vorkommen (also von möglichen Sortierverfahren), liefert also auch eine untere Schranke für die Zahl der Vergleiche, die im worst-case gemacht werden müssen.

2.5.1 Satz – Zeitbedarf von Sortieralgorithmen mit Vergleichen

Jedes auf Vergleichen basierende Sortierverfahren benötigt $\Omega(n \log n)$ Vergleiche im worst case

Beweis: Wie hoch muss ein Binärbaum mindestens sein, um $n!$ viele Blätter zu haben? Gemäß den eben gemachten Beobachtungen genügt es die Höhe von Entscheidungsbäumen zu bestimmen, in denen alle Permutationen als Blätter vorkommen.

Sei T ein Entscheidungsbaum der Höhe h mit l Blättern. Da das Sortierverfahren korrekt sein soll, muss $n! \leq l$ gelten. Ein Binärbaum der Höhe h hat höchstens 2^h viele Blätter. Für T bedeutet das

$$n! \leq l \leq 2^h$$

$$\Rightarrow n! \leq 2^h$$

$$\log n! \leq h$$

Für $n!$ gibt es eine schöne Abschätzung: STIRLINGFORMEL:

$$n! \in \sqrt{2\pi n} * \left(\frac{n}{e}\right)^n (1 + \Theta(\frac{1}{n}))$$

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

$$\Rightarrow \log n! \approx n \log \left(\frac{n}{e}\right) + \log \sqrt{2\pi n} \in \Omega(n \log n)$$

$\Rightarrow$ es gibt $c > 0$ so dass asymptotisch

$$c * n \log n \leq \log n!$$

$$\Rightarrow c * n \log n \leq h$$

□

2.5.2 Folgerung

MERGESORT und HEAPSORT (aber auch QUICKSORT in der SELEKT-Variante) sind asymptotisch optimale Sortierverfahren.

Bemerkung

Sortieren in Linearzeit: nur möglich, wenn keine Vergleiche
$\Rightarrow$ benötigen zusätzliche Informationen über die Zahlen.

2.5.3 COUNTINGSORT

geht davon aus, dass die zu sortierenden Zahlen alle zwischen 0 und $k \in \mathbb{N}$ liegen. (Ist $k \in \mathcal{O}(n)$, so läuft COUNTINGSORT in $\mathcal{O}(n)$).

Idee:

- Für jedes Objekt den Rang bestimmen
- diese Information nutzen, um Objekte (Zahlen) dadurch ins Ausgabefeld zu platzieren
  (ist $\text{rang}(x) = k$, so gehört x an die Stelle $k + 1$)

Algorithmus

$$A[1, \ldots, n], \text{ Eingabefeld (unsortiert)}$$

$$B[1, \ldots, n], \text{ Ausgabefeld (sortiert)}$$

$$C[0, 1, \ldots, k], \ k \text{ ist größtmögliche Zahl im Feld}$$

```
COUNTINGSORT(A,B,k)
1   for i=0 to k
2       do C[i]=0
3   for j=1 to length (A)
4       do C[A[j]] = C[A[j]] + 1   // C[m] = Zahl der Eingaben = m
5   for i=1 to k
6       do C[i] = C[i] + C[i-1]    // C[m] = Zahl der Eingaben <= m
7   for j=length(A) downto 1
8       do B[C[A[j]]] := A[j]
9          C[A[j]] := C[A[j]] - 1
10  output B
```

Bemerkung Eingabe nicht notwendigerweise paarweise verschieden

Beispiel

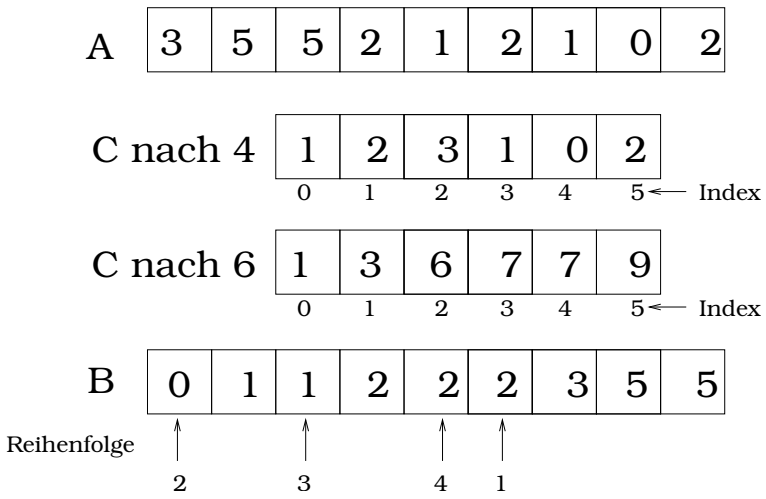


Abbildung 27: Schema COUNTINGSORT

Zeitanalyse von COUNTING-SORT

Zeilen	Laufzeit	
1-2	$\Theta(k)$	
3-4	$\Theta(n)$	
5-6	$\Theta(k)$	$\leftarrow$ jede Summation mit $\mathcal{O}(1)$ angenommen
7-9	$\Theta(n)$	

Die Laufzeit ist insgesamt $\Theta(n+k)$

Bemerkungen

- COUNTING-SORT unterbietet die untere Schranke von $\Omega(n \log n)$, da es unter anderem nicht auf Vergleichen basiert.
- Achtung: C wird riesig, wenn die Zahlen riesig werden $\Rightarrow$ Verfahren in $\Theta(n)$, wenn $k \in \mathcal{O}(n)$.
- COUNTING-SORT ist ein stabiles Sortierverfahren, d.h. gleiche Zahlen werden in ihrer ursprünglichen Reihenfolge ausgegeben.

2.5.4 RADIXSORT

Sortierverfahren ursprünglich für Lochkarten

Lochkarten

- 80 Spalten

- pro Spalte 12 Plätze um ein Loch zu Stanzen (jede Spalte hat eigene Wertigkeit (z.B. Zahl) damit können Zahlen (mit einer Länge von bis zu 80 (=Anzahl Spalten=) kodiert werden

beim Sortieren Kann vorgegebene Spalte inspizieren und Karten entsprechend in 12 Haufen sortieren

Heute ähnliche Situation z.B. Geburtsdaten: Tag.Monat.Jahr

Sortieren (intuitiv)

1. Sortiere nach Jahr
2. Sortiere nach Monat
3. Sortiere nach Tag

D.h. Sortiere nach most-significant Bit zuerst

Problem Viele Zwischenstapel nötig (wenn nach dem Jahr sortiert, muss in den Stapeln für jedes Jahr nach Monat sortiert werden, ...).

Hier: Sortiere nach dem least-significant Bit zuerst und zwar mit stabilen Sortierverfahren.

Beispiel Dreistellige Zahlen sind zu sortieren

228		123		123		123
379		333		228		228
123		444		333		333
444	sortiert nach $\xrightarrow{}$	865	sortiert nach $\xrightarrow{}$	537	sortiert nach $\xrightarrow{}$	379
865	Einern (stabil)	777	Zehnern (stabil)	444	Hundertern (stabil)	444
777		537		865		537
333		228		777		777
537		379		379		865

RADIXSORT(A,d) Wobei A das Array und d Anzahl der „Bitpositionen“ (Stellen)

```
1   for i=1 to d
2       do sortiere A bezüglich Ziffern oder (Bit)position i mit \
        einem stabilen Sortierverfahren
3   od
4   Output A
```

Zeitbedarf

$$\Theta(d * (n + k))$$

wobei:

- d – Zahl der Bitpositionen der Eingabezahlen
- k – Zahl der möglichen Werte pro Bitposition (bei Dezimalzahlen z.B. 10)

2.5.5 BUCKET-SORT

bucket=Eimer
Läuft in Linearzeit, wenn Eingabezahlen zufällig verteilt sind (gemäß Gleichverteilung (also insbesondere nicht Gausverteilung) im Intervall [0,1)

Abbildung 28: BUCKET-SORT Schema

Idee

- Zerlege $[0, 1)$ in gleichgroße Intervalle (buckets)
- sortiere die Zahlen, die in einem Eimer sind
- nimm die Zahlen der Reihe nach aus den Eimern
- $A[1, \ldots, n]$ Eingabefeld
- $B[0, \ldots, n-1]$ Feldeinträge sind Liste (linked lists)/bucket

BUCKET-SORT(A)

```
1   n := length(A)
2   for i=1 to n
3       do füge A[i] in die Liste B[⌊n * A[i]⌋] ein.
4   for i=0 to n-1
5       do sortiere Liste B[i] mit INSERTIONSORT od
6   verknüpfe die Listen B[0],B[1],... ,B[n-1] in dieser Reihenfolge
```

Wenn die Zahlen „schön gleich verteilt sind“. Dann befindet sich in jedem Eimer eine Zahl.

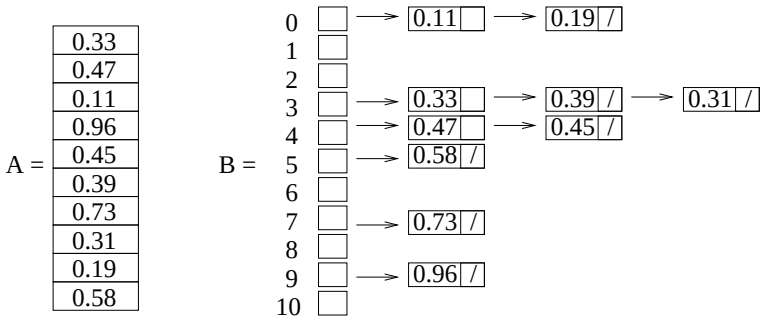


Abbildung 29: Beispiel Bucket-Sort

Beispiel Siehe Abbildung 29 auf Seite 45.

Zeitanalyse alle Zeiten bis auf Zeile 5 liegen in $\mathcal{O}(n)$. In Zeile 5:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(n_i^2)$$

Wissen über n_i:

- n_i ist Zufallsvariable
- $\sum_{i=0}^{n-1} n_i = n$

Unter Benutzung des Erwartungswertes erhält man:

$$\mathrm{E}(T(n)) = \mathrm{E}\left(\Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(n_i^2)\right)$$

$$= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}(\mathcal{O}(n_i^2))$$

$$= \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(\mathrm{E}(n_i^2))$$

Man kann nun zeigen, dass unter der Annahme dass die eingegebenen Zahlen gleich verteilt im Intervall $[0, 1)$ sind, folgendes gilt:

$$\mathrm{E}(n_i^2) = 2 - \frac{1}{n}$$

Es liegen also $\sqrt{2}$ Zahlen in jedem Eimer (Wert ist also Konstante) $\Rightarrow \mathrm{E}(T(n)) = \Theta(n)$

3 Suchbäume

Der (binäre) Suchbaum ist eine Datenstruktur, die folgende Operationen unterstützt (also effizient ermöglicht):

- SEARCH
- MINIMUM
- PREDECESSOR (Vorgänger)
- SUCCESSOR (Nachfolger)
- INSERT
- DELETE

ideal wäre, wenn sich alle Operationen in $\mathcal{O}(\log n)$ realisieren lassen, keinesfalls sollte es jedoch schlechter sein als $\mathcal{O}(n)$.

Typische Anwendungen für Suchbäume sind Wörterbücher und Prioritätsschlangen.

3.1 Grundlegende Definitionen und Operationen

Ein binärer Suchbaum ist ein binärer Baum (also eine verkettete Datenstruktur).

- neben dem Schlüssel (und anderen sekundären Daten) enthält jeder Eintrag zusätzliche Zeiger left, right, parent (für die Nachbarschaftsbeziehungen im Baum), die auf die entsprechenden Einträge (Knoten) für linkes und rechtes Kind, bzw. den Elternknoten verweisen.
- fehlen Kinder oder Eltern, so enthält das entsprechende Feld des Knoten den Wert NIL.

In einem binären Suchbaum sind die Schlüssel so gespeichert, dass die binäre-Suchbaum-Eigenschaft erfüllt ist:

> Ist x ein Knoten im Suchbaum und y ein Knoten in einem der in x gewurzelten Teilbäume, so gilt: y im linken Teilbaum $\Leftrightarrow$ $\text{key}(y) < \text{key}(x)$
> y im rechten Teilbaum $\Leftrightarrow$ $\text{key}(y) \geq \text{key}(x)$

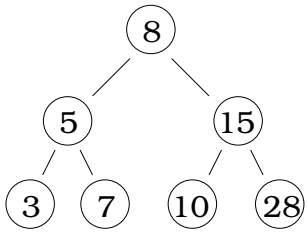
Beispiel Siehe Abbildung 30 auf Seite 47.

Die Suchbaumeigenschaft erlaubt es, die im Baum gespeicherten Schlüssel in sortierter Reihenfolge auszugeben (INORDER-TREEWALK(root,(T)) – also in der Wurzel des Baumes gestartet).

INORDER-TREEWALK(x)

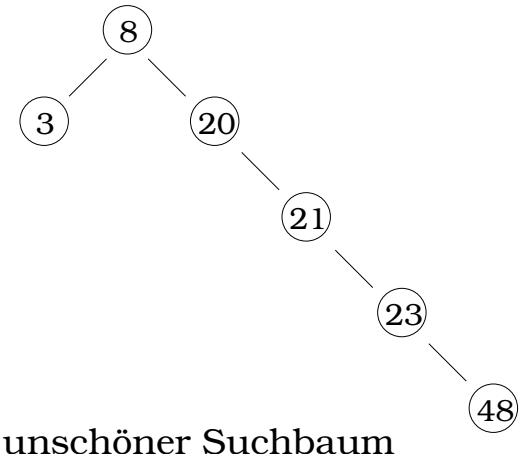
```
1   if x != NIL then do
2       INORDER-TREE-WALK(left(x))
3       print(key(x))
4       INORDER-TREE-WALK(right(x))
```

a)



Schöner Suchbaum

b)



unschöner Suchbaum

Abbildung 30: Beispiele für Suchbäume

Laufzeit

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

$$T(n) = T(n-1) + T(n-1) + \mathcal{O}(1)$$
$$= 2T(n-1) + \mathcal{O}(1)$$

Kann nicht $\in \mathcal{O}(n)$ sein

Die Rekurrenz ist schlecht (bzw. falsch), denn es können nicht beide worst-cases ($T(n-1)$ links und rechts) gleichzeitig auftreten.

$$T(n) = T(l) + T(n-l-1) + \mathcal{O}(1)$$

Annahme: $T(n) \leq c * n$

$$\Rightarrow T(n) \leq \underbrace{c * l + c(n-l-1) + k}_{c*n-c+k} \overset{\text{soll}}{\underset{\text{sein}}{\leq}} c * n$$

gilt für $c \geq k$

3.1.1 Satz – INORDER-TREEWALK

Ist x die Wurzel eines Suchbaumes mit n Knoten, so liefert INORDER-TREEEWALK(x) die Folge der sortierten Schlüssel in Zeit $\Theta(n)$.

Beweis:

- sortierte Folge – klar!
- $\mathcal{O}(n)$ – klar!
- $\Omega(n)$ – einfach

a) SEARCH

gegeben: Schlüssel k, Zeiger auf Wurzel des Suchbaumes

gesucht: Zeiger auf einen Knoten (im Suchbaum) mit Schlüssel k
Binärsuche im Baum:

```
TREE-SEARCH(x,k)
1   if x = NIL or k = key[x]  // falls x = NIL, wird NIL zurückgegeben
2       then return x
3   if k < key(x) then TREE-SEARCH(left(x),k)
4                 else TREE-SEARCH(right(x),k)
```

Selber Algorithmus ohne rekursive Aufrufe:

```
ITERATIVE-TREE-SEARCH(x,k)
1   while x != NIL and k != key(x) do
2   {   if k < key(x) then x:=left(x)
3                     else x:=right(x)
4   }
5   return x
```

Beim Start einer Suche im Suchbaum T ist $x =$ root (T).

Laufzeit

$\mathcal{O}$(Höhe des Suchbaumes)

b) MINIMUM/MAXIMUM

```
TREE-MINIMUM(x)
1   while left(x) != NIL do
2       x := left(x)
3   return key(x)    // Wenn man den INHALT des kleinsten Schlüssels sucht
//  return x         // Alternativ, wenn Schlüssel gesucht wird
```

```
TREE-MAXIMUM(x)
1   while right(x) != NIL do
2       x := right(x)
3   return key(x)
```

Laufzeit:

$\mathcal{O}$(Höhe des Suchbaumes)

c) Nachfolger und Vorgänger

gegeben: Knoten x

gesucht: Knoten mit kleinstem Schlüssel $>$ key(x) (Nachfolger, Successor) bzw. Knoten mit größtem Schlüssel $<$ key(x) (Vorgänger, Predecessor)
Vorgänger: siehe Abbildung 31 auf Seite 49

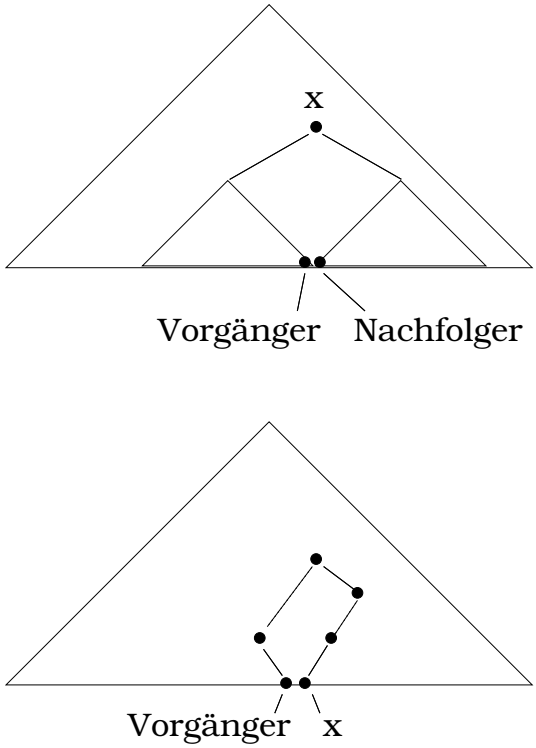


Abbildung 31: Position von Vorgänger im günstigen und im ungünstigen Fall

Nachfolger Schöner Fall siehe Abbildung 32 auf Seite 50

```
TREE-SUCCESSOR(x)
1   if right (x) != NIL then return TREE-MINIMUM(right(x))
```

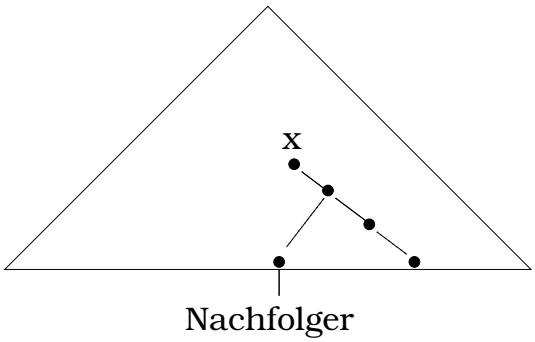
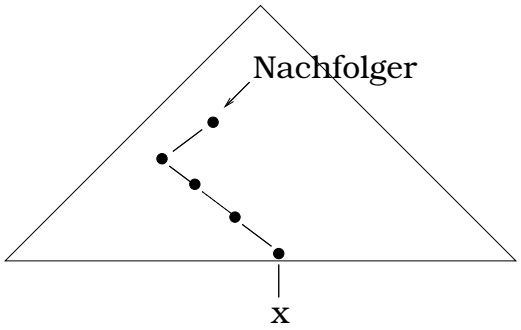


Abbildung 32: Nachfolger, günstiger Fall

```
2   y := parent(x)
3   while x != NIL and x = right(y) do  // x ist rechter Kindkonten
4       x := y
5       y := parent(x)
6   return y   // Wenn x Maximum ist, wird NIL zurückgegeben
```

siehe Abbildung 33 auf Seite 50.



solange absteigen, bis man an Knoten angelangt ist,
der selbst linkes Kind ist

Abbildung 33: Nachfolger, ungünstiger Fall

Laufzeit:

$\mathcal{O}$(Höhe des Suchbaumes)

Bemerkung

- Vorgänger analog
- bei nicht paarweise verschiedenen Schlüsseln ist der Nachfolger eines Knoten x per Definition derjenige Knoten, den TREE-SUCCESSOR(x) liefert

3.1.2 Satz – Zeitbedarf von Operationen im Suchbaum

In Suchbäumen können die Operationen SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR in Zeit $\mathcal{O}$(Höhe des Suchbaumes) ausgeführt werden.

d) Einfügen und Löschen

- beide Operationen verändern die Datenstruktur

Einfügen:

gegeben:

- Suchbaum T (Zeiger auf root(T))
- Knoten z (mit key(z), right(z) = left(z) = parent(z) = NIL)

gesucht:

- Suchbaum T', der alle Knoten von T und zusätzlich z enthält

```
TREE-INSERT(T,z)
1   y := NIL
2   x := root(T)
3   while x != NIL do
4       y := x
5       if key(z) < key(x) then x := left(x)
6                          else x := right(x)   // y ist nun parent(x)
7   parent(z) := y
8   if y = NIL then root(T) := z   // in diesem Fall war T leer
9   else if key(z) < key(y) then left(y) := z
10                           else right(y) := z
11  return T
```

Beispiel Siehe Abbildung 34 auf Seite 52.

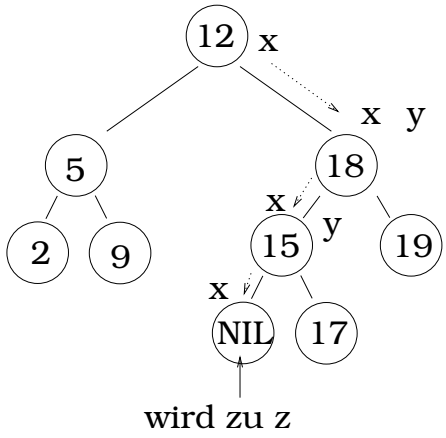
Laufzeit:

$\mathcal{O}$(Höhe des Suchbaumes)

Löschen

gegeben

- Suchbaum T
- Knoten z aus T



y ist Wurzelknoten des Baumes

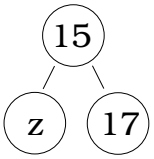


Abbildung 34: Beispiel für Hinzufügen eines Elementes

gesucht:

- Suchbaum T', der genau die Knoten T bis auf z enthält.

Fälle

1.Fall: z hat keine Kinder	2. Fall: z hat ein Kind	3. Fall: z hat zwei Kinder
z entfernen, Elternknoten modifizieren	z herauslösen, parent(z) ist jetzt parent von child(z)	z's Nachfolger aus dem Baum herauslösen und an z's Stelle einsetzen

Siehe Abbildung 35 auf Seite 53 und Abbildung 36 auf Seite 53.

```
TREE-DELETE(T,z)
1   if left(z) = NIL or right(z) = NIL then y := z  // Fälle 1 und 2
2                                       else y := TREE-SUCCESSOR(z)
3   if left(y) != NIL then x := left(y)
4                     else x := right(y)
5   if x != NIL then parent(x) := parent(y)
6   if parent(y) = NIL then root(T) := x
7   else if y = left(parent(y)) then left(parent(y)) := x
8   else right(parent(y)) := x
9   if y!=z then key(z) := key(y) (und überschreibe z's Daten mit y's Daten)
10  return T
```

3. Fall:

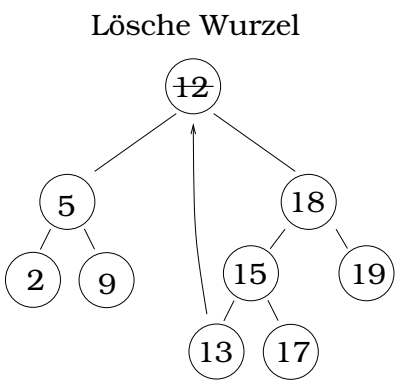


Abbildung 35: 3. Fall, z hat zwei Kinder

2. Fall:

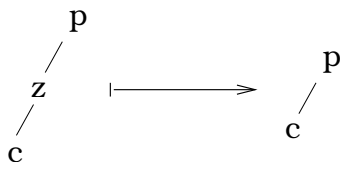


Abbildung 36: 2. Fall, z hat ein Kind

Kommentare

Zeile	Kommentar
1,2	bestimme Knoten y, der herausgelöst werden kann $y = z$ falls z höchstens ein Kind hat
3,4	setzen x auf nicht-NIL-Kind von y, bzw. $x =$ NIL falls y kinderlos
5	y herauslösen
6,7,8	Zeiger umsetzen
9	Daten umschreiben

Laufzeit

$\mathcal{O}$(Höhe des Suchbaumes)

3.1.3 Satz – Zeitbedarf von INSERT und DELETE

Die Operationen INSERT und DELETE benötigen in Suchbäumen der Höhe h nicht mehr als $\mathcal{O}(h)$ Zeit.

Bemerkungen

- zufällig entstandende Suchbäume (viele INSERT/DELETE-Operationen) haben

eine erwartete Höhe von $\mathcal{O}(\log n)$, wobei n die Knotenanzahl ist $\Rightarrow$ average case ist dichter am best case als am worst case

3.2 Rot-Schwarz-Bäume

- rot-schwarz-Bäume sind gefärbte Suchbäume, die bei n Knoten stets balanciert sind und eine Höhe von $\mathcal{O}(\log n)$ haben
  $\Rightarrow$ alle wichtigen Operationen laufen in $\mathcal{O}(\log n)$

3.2.1 Definition – Rot-Schwarz-Baum

Ein rot-schwarz-Baum ist ein binärer Suchbaum mit folgenden zusätzlichen Eigenschaften:

1. Jeder Knoten des Baumes besitzt genau eine der Farben rot oder schwarz. (Informationen eines Knoten: left, right, parent, colour)
2. Die Wurzel des Baumes ist schwarz.
3. Alle NIL-Blätter sind schwarz.
4. Ist ein Knoten rot gefärbt, so sind *beide* Kinderknoten schwarz.
5. Für jeden Knoten v gilt: alle Pfade von v zu einem (von v abstammenden) Blatt enthalten die gleiche Anzahl von schwarzen Knoten.

Bemerkung

- zu 4.) jeder Knoten hat *genau* zwei Kinder, das sind unter Umständen NIL-Knoten.

Beispiel

$$17, 28, 36, 59, 92, 45$$

Erster Versuch einen rot-schwarz-Baum zu erstellen ist in Abbildung 37 auf Seite 55 zu sehen.

Dies kann offensichtlich kein rot-schwarz-Baum sein. Unternehme neuen Versuch das Feld in einen rot-schwarz-Baum zu bringen Das Resultat ist in Abbildung 38 auf Seite 55 zu sehen. Dieser Baum ist tatsächlich ein rot-schwarz-Baum.

Bemerkungen

- wir haben in Zukunft nur einen NIL-Knoten, dieser hat folgende Eigenschaften: colour = schwarz; parent, left und right sind beliebig
- in Beispielen werden wir den NIL-Knoten häufig weglassen

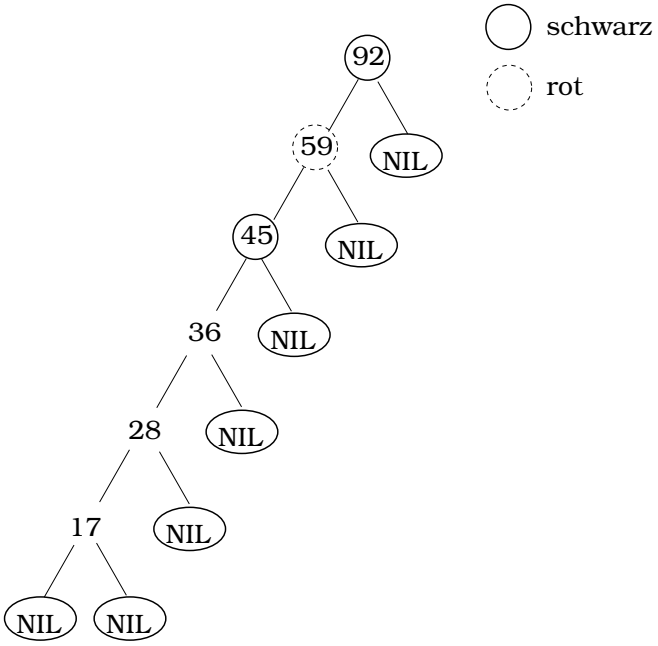


Abbildung 37: Erster Versuch – Erstellen eins rot-schwarz-Baums

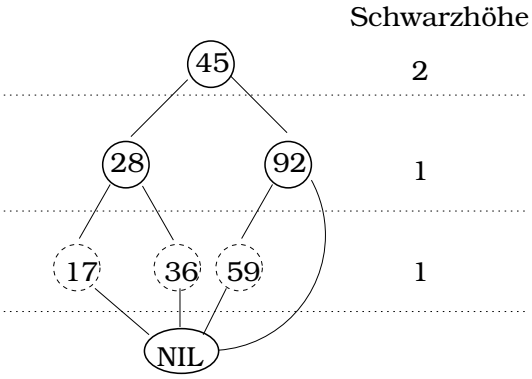


Abbildung 38: Korrekter rot-schwarz-Baum

3.2.2 Definition – Schwarzhöhe

Sei T ein rot-schwarz-Baum und x ein innerer Knoten (jeder Knoten bis auf NIL) von T. Die Schwarzhöhe des Knoten x, $\mathrm{bh}(x)$, ist die Anzahl der schwarzen Knoten auf einem Pfad von x – x wird dabei nicht mitgezählt – zu einem (von x abstammenden) NIL-Blatt (dieses wird mitgezählt).

Bemerkung

- Die Definition Schwarzhöhe ist sinnvoll aufgrund Eigenschaft 5 bei der Definition von rot-schwarz-Bäumen.

Als erstes zeigen wir, dass rot-schwarz-Bäume gute Suchbäume sind.

3.2.3 Satz – Höhe von Rot-Schwarz-Bäumen

Jeder rot-schwarz-Baum mit n inneren Knoten hat eine Höhe von höchstens $2\log(n+1)$.

Beweis Wir zeigen zunächst folgende Hilfsaussage:

> (*) Für jeden Knoten x gilt: Die Zahl der inneren Knoten $n(x)$, des in x gewurzelten Teilbaumes ist durch $2^{\mathrm{bh}(x)} - 1$ von unten beschränkt:
>
> $$2^{\mathrm{bh}(x)} - 1 \leq n(x)$$

Beweis von (*): Induktion über $h(x)$ – Höhe von unten her

Induktionsanfang: $h(x) = 0$, x ist Knoten unmittelbar über NIL-Blatt.

$$1 \text{ innerer Knoten } = 2^{\mathrm{bh}(x)} - 1$$

$$(\text{nämlich } x) = 2^1 - 1 = 1$$

Induktionsschritt, - Voraussetzung Für alle Knoten x mit Höhe $h(x) \leq k$, $k \geq 0$, gelte $n(x) \geq 2^{\mathrm{bh}(x)} - 1$

Induktionsbehauptung Dann gilt auch für alle Knoten x mit $h(x) = h + 1 : n(x) \geq 2^{\mathrm{bh}(x)} - 1$.

Induktionsbeweis Sei x ein Knoten mit $h(x) = k + 1$. Wegen $k \geq 0$ folgt $k + 1 \geq 1$, folglich x ist ein innerer Knoten mit zwei (nicht NIL-Kindern) x_l (linkes Kind) und x_r (rechtes Kind)

Für $\mathrm{bh}(x_r)$ und $\mathrm{bh}(x_l)$ kommen nur die Werte $\mathrm{bh}(x)$ und $\mathrm{bh}(x) - 1$ in Frage, je nachdem, ob sie selbst rot oder schwarz sind.

Da $h(x_r) < h(x)$ und $h(x_l) < h(x)$ können wir auf x_l und x_r die Induktionsvoraussetzung anwenden:

$$2^{\mathrm{bh}(x_r} - 1 \leq n(x_r) \text{ und } 2^{\mathrm{bh}(x_l)} - 1 \leq n(x_l)$$

Wegen $\mathrm{bh}(x_r) \geq \mathrm{bh}(x) - 1$ und $\mathrm{bh}(x_l) \geq \mathrm{bh}(x) - 1$:

$$\begin{aligned} 2^{\mathrm{bh}(x)} - 1 &= 2^{\mathrm{bh}(x)-1} + 2^{\mathrm{bh}(x)-1} - 1 \\ &\leq 2^{\mathrm{bh}(x_l)} + 2^{\mathrm{bh}(x_r)} - 1 \\ &\leq n(x_r) + n(x_l) + 1 = n(x) \end{aligned}$$

Beweis von Satz 3.2.3 Sei T ein rot-schwarz-Baum mit n inneren Knoten. Sei h die Höhe von T. aus Eigenschaft 4 von rot-schwarz-Bäumen folgt, dass auf jedem Pfad von der Wurzel zu einem Blatt höchstens die Hälfte der Knoten rot sein kann.

Damit muss auf einem solchen Pfad mindestens die Hälfte der Knoten schwarz sein.

$$\Rightarrow \mathrm{bh}(root(T)) \geq \frac{h}{2}$$

Gemäß (*) folgt nun:

$$n = n(root(T)) \geq 2^{\mathrm{bh}(root(T))} - 1 \geq 2^{\frac{h}{2}} - 1$$

$$\Rightarrow n \geq 2^{\frac{h}{2}} - 1$$

$$\Rightarrow 2\log(n+1) \geq h$$

□

Es gilt also, dass die Höhe des Baumes in $\mathcal{O}(\log n)$ liegt.

3.2.4 Folgerung – Zeitaufwand von Operationen in RS-Bäumen

Die Operationen SEARCH, MINIMUM, MAXIMUM, SUCCESSOR und PREDECESSOR können auf rot-schwarz-Bäumen in Zeit $\mathcal{O}(\log n)$ implementiert werden.

Beweis nach Abschnitt 3.1 laufen diese Operationen in $\mathcal{O}(h)$. Wegen 3.2.3 ist $h \in \mathcal{O}(\log n) \Rightarrow$ Aussage gilt.

□

Das Einfügen eines neuen Elements mit dem Wert 24 in den Baum aus Abbildung 38 auf Seite 55: ist in Abbildung 39 auf Seite 58 dargestellt. Das Löschen des Schlüssels mit dem Wert 28: ist in Abbildung 40 auf Seite 58 dargestellt.

Im Allgemeinen klappt das Löschen allerdings nicht so einfach. In Vorbereitung auf die Algorithmen zum Einfügen und Löschen in rot-schwarz-Bäumen, behandeln wir so genannte Rotationen, die die Zeiger und Farbstruktur im Baum verändern: siehe Abbildung 41 auf Seite 59.

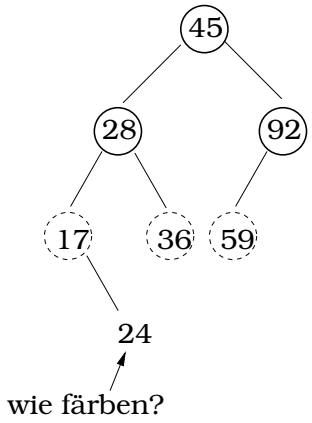


Abbildung 39: Insert 24

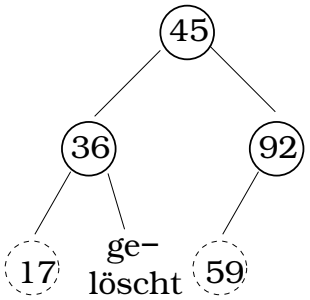


Abbildung 40: Del 28

x, y, z sind Suchbäume, die bei der Rotation nicht angetastet werden. Die Rotationen sind lokale Operationen, die die Suchbaumeigenschaft nicht beeinflussen. Im folgenden sind right(u) $\neq$ NIL(T) und parent(root(T))=NIL(T).

```
LEFT-ROTATE(T,u)
1   v := right(u)
2   right(u) := left(v)
3   parent(left(v)) := u
4   parent(v) := parent(u)
5   if parent(u) = NIL(T) then root(T) := v
6   else if u = left(parent(u)) then left(parent(u)) := v
7                               else right(parent(u)) := v
8   left(v) := u
9   parent(u) := v
```

Bemerkung RIGHT-ROTATE analog

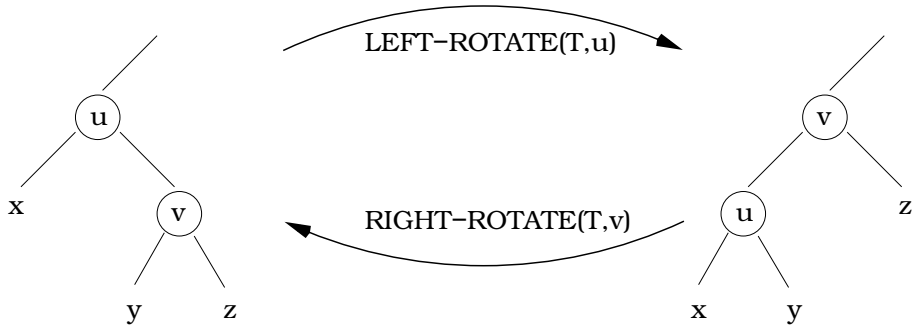


Abbildung 41: Rotationen

Laufzeit

$$\mathcal{O}(1)$$

Beispiel
siehe Abbildung 42 auf Seite 60

Bemerkung

- Rotationen erhalten die Suchbaumeigenschaft
- bleibt die rot-schwarz-Baum-Eigenschaft erhalten? – NEIN!
- Wir werden uns zu gegebener Zeit darum kümmern!

Einfügen in rot-schwarz-Bäumen Im Prinzip funktioniert das genauso wie in normalen Suchbäumen, wobei der neu eingefügte Knoten zunächst rot gefärbt wird und danach, wenn nötig die Färbung korrigiert wird.

```
RS-INSERT(T,z)
1   --|
.     |
.     |-   wie TREE-INSERT(T,z), aber ersetze NIL durch NIL(T)
.     |    (es gibt nur noch einen einzigen NIL-Knoten)
10   -|
11   left(z) := NIL(T)
12   right(z) := NIL(T)
13   colour(z) := red
14   RS-FIXUP(T,z)
```

siehe Abbildung 43 auf Seite 61. Dieser Baum ist kein r-s-Baum $\Rightarrow$ RS-FIXUP(T,z)
Ab sofort gilt: p(x) $\widehat{=}$ parent(x).

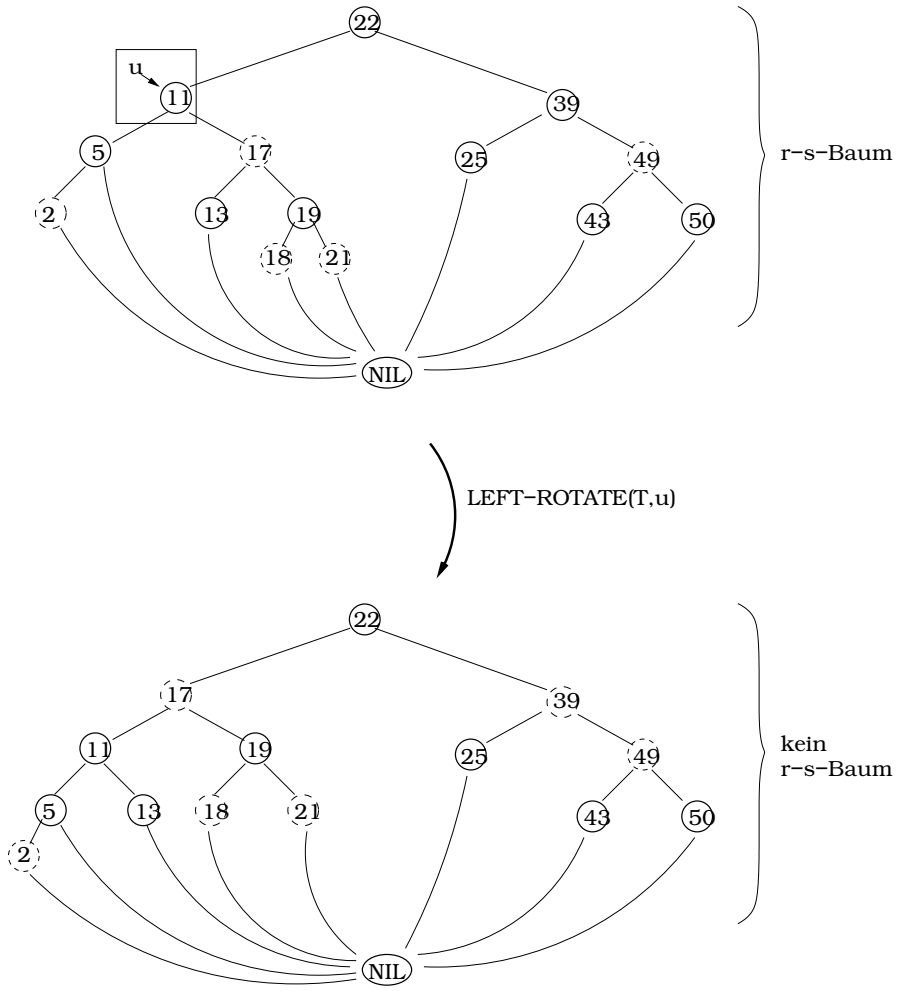


Abbildung 42: Beispiel für LEFTROTATE(T,u)

```
RS-FIXUP(T,z)
1   while color(p(z)) = red do
2   if p(z) = left(p(p(z)) then do
3       y := right(p(p(z)))
4       if color(y) = red then { color(p(z)) := black
5                                color(y) := black
6                                color(p(p(z))) := red
7                                z := p(p(z))
        }
8       else do   if z = right(p(z)) then do {
9                    z := p(z)
10                   LEFTROTATE(T,z)
```

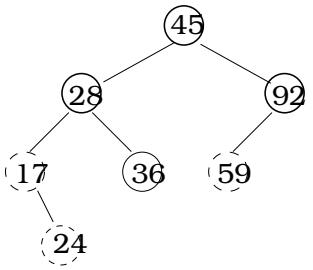


Abbildung 43: Baum nach Insert(T,24)

```
                }
11
12              color(p(z)) := black
13              color(p(p(z))) := red
14              RIGHT-ROTATE(T,p(p(z)))
15-|
.  |
.  |--  else do (selber Teil, wie nach dem then (ab Zeile 3)
.  |    allerdings left und right vertauschen)
25-|
26  color(root(T)) := black
27  od
```

siehe Abbildung 44 auf Seite 62.

Analyse

1) Welche Arten der Verletzung der RS-Baumeigenschaften können bei TREE-INSERT auftreten?

Eigenschaft des RS-Baums	Verletzung
1	nicht verletzt, da color(z) := red
2	möglich, wenn z Wurzel wird – Abhilfe schafft Zeile 25 in RS-Fixup
3	nicht möglich, da z innerer Knoten ($\neq$ NIL) des RS-Baumes wird
4	möglich, wenn color($p(z)$) = red – Zeile 1 ff. von RS-Fixup sollte Abhilfe schaffen
5	nicht möglich, da color(z):=red

2) zur Korrektheit von RS-FIXUP (am Ende liegt ein RS-Baum vor) Während eines jeden Durchlaufs der while-Schleife werden folgende drei Eigenschaften aufrecht erhalten.

- color(z)=red

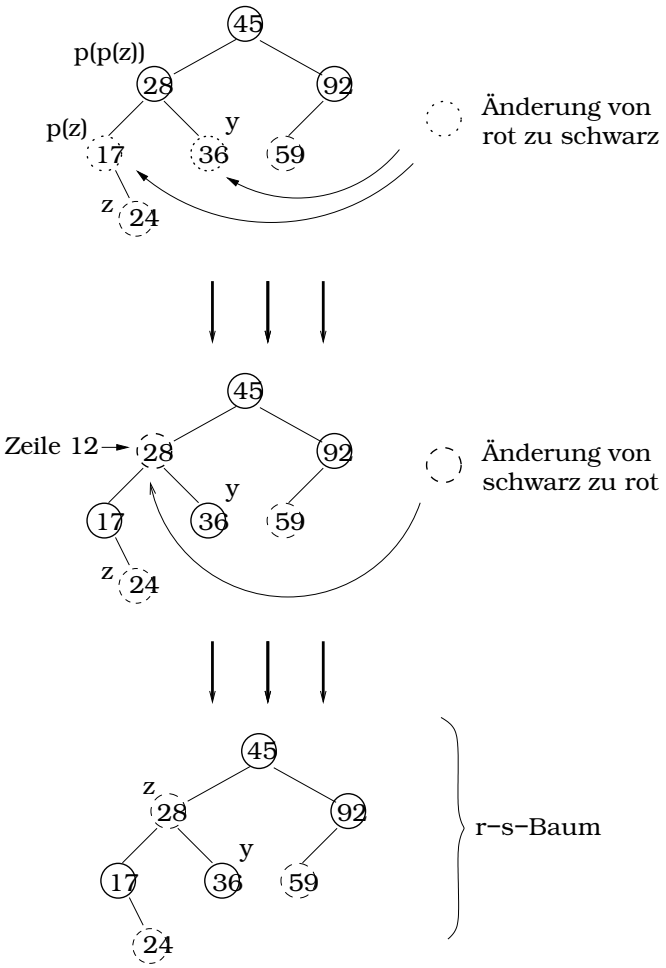


Abbildung 44: Algorithmus RS-Fixup

- Ist $p(z)$ die Wurzel, so ist $p(z)$ schwarz
- sofern Verletzungen der RS-Baumeigenschaften vorliegen, so wird stets höchstens eine der fünf Eigenschaften verletzt und zwar entweder Eigenschaft 2 oder Eigenschaft 4
  Gibt es eine Verletzung der Eigenschaft 2, so ist es deshalb, weil z die Wurzel und rot ist. Gibt es eine Verletzung der Eigenschaft 4, dann nur deshalb, weil sowohl z als auch $p(z)$ rot sind.

Zwar ist 3. die zentrale Forderung, um zu zeigen, dass RS-FIXUP korrekt ist, aber um das zu zeigen benötigt man auch 1. und 2.

Noch zu zeigen 1.,2. und 3.:

i gelten vor dem ersten Durchlauf der while-Schleife

ii werden während des Durchlaufs der while-Schleife aufrecht erhalten

iii (1.,2. und 3.) sichern die Korrektheit von RS-Fixup nach der Terminierung der while-Schleife

zu i) Eigenschaften erfüllt?

1. ja
2. ja, da sofern $p(z)$ existiert, er bisher nicht umgefärbt wurde
3. ja, denn Eigenschaft 2 (des RS-Baumes) ist verletzt, wenn z Wurzel ist, damit ist z der einzige Knoten im Baum und hat somit einen NIL-Vater und NIL-Kinder → keine Verletzung von 4;
   ist hingegen Eigenschaft 4 verletzt, so kann das nur vorliegen, weil z und $p(z)$ rot sind ⇒ z ist nicht Wurzel

zu iii) Nach Beendigung eines Durchlaufs der while-Schleife:
Wenn Bedingung für den Eintritt in die while-Schleife nicht mehr gilt, muss gelten: color($p(z)$)=black. Ist z Wurzel, so ist $p(z) = \text{NIL}(T)$ und somit auch schwarz ⇒ Eigenschaft 4 gilt im ganzen Baum, denn Eigenschaft 4 kann nur bei Knoten z verletzt sein und dort ist sie nicht verletzt.

Da 1.,2. und 3. während des Durchlaufs der while-Schleife stets vorliegen, kann höchstens noch eine Verletzung der Eigenschaft 2 vorhanden sein. Diese wird mit Zeile 25 aber repariert.

⇒ RS-FIXUP ist korrekt (sofern ii gilt), da Eigenschaften 1-5 des RS-Baums nicht verletzt sind

zu ii) zu zeigen: Eigenschaften 1.,2.,3. werden während des Durchlaufs nicht gestört:

Beispiel
siehe Abbildung 45 auf Seite 64 und Abbildung 46 auf Seite 65.

zu Korrektheitsbeweis, Punkt iii) während eines Durchlaufes der while-Schleife werden a,b,c nicht zerstört. Eigentlich müssten 6 Fälle betrachtet werden:

$p(z) = \textbf{left}(p(p(z)))$

- 1. Fall: color(right(p(p(z))))=red
- 2. Fall: color(right(p(p(z))))=black und z=right(p(z))
- 3. Fall: color(right(p(p(z))))=black und z=left(p(z))

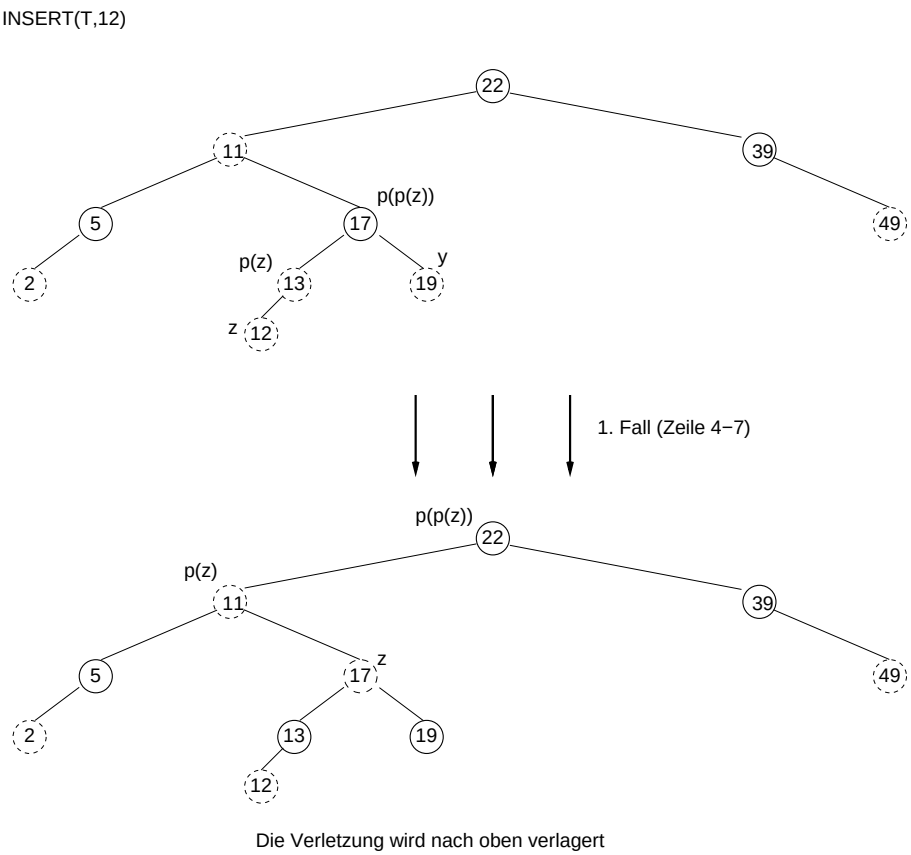


Abbildung 45: Beispiel RS-Fixup Teil 1

$p(z) =$ **right**$(p(p(z)))$

- 4. Fall: color(left(p(p(z))))=red
- 5. Fall: color(left(p(p(z))))=black und z=right(p(z))
- 6. Fall: color(left(p(p(z))))=black und z=left(p(z))

Fälle 1,2,3 sind symmetrisch zu den Fällen 4,5,6. Wir betrachten daher nur 1,2,3.

zum 1. Fall p(z) ist linkes Kind von p(p(z)) und Onkel von z ist rot. Damit muss gelten p(p(z)) ist schwarz, sonst lägen 2 Verletzungen der r-s-Baumeigenschaft vor → Verletzung der Invariante, siehe Abbildung 47

unser Ziel: siehe Abbildung 48 auf Seite 67.

liegt der 1. Fall vor, so werden die Programmzeilen 4-7 ausgeführt. Da sowohl p(z) als auch y (= Onkel) rot sind, können wir p(z) und y schwarz färben und gleichzeitig p(p(z)) rot färben und bewahren so die Eigenschaft 5. z wandert zwei Etagen höher im Baum.

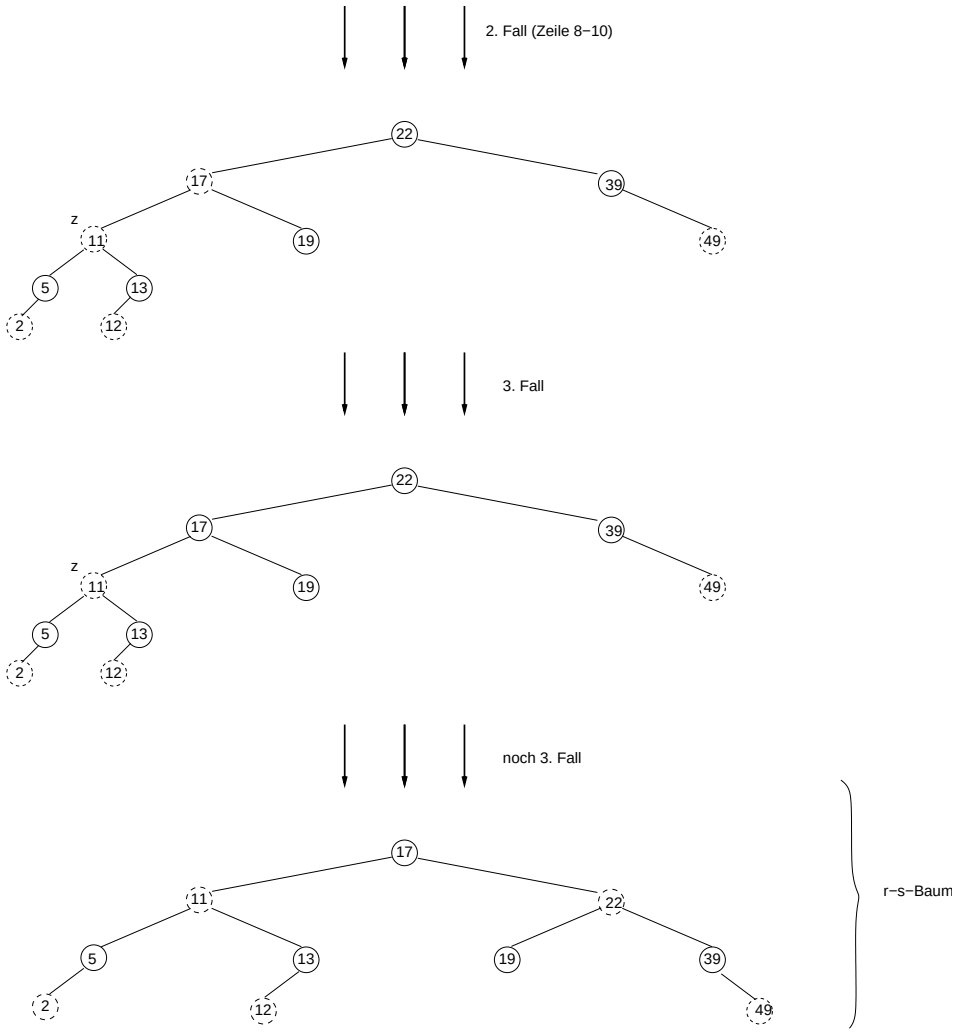


Abbildung 46: Beispiel RS-Fixup Teil 2

n.z.z. a,b und c gelten nach Umfärbung

- zu Invariante a) „z ist rot“ – gilt dank der gerade vorgenommenen Umfärbungen und Umbenennung.
- zu Invariante b) „ist p(z) Wurzel, so ist p(z) schwarz“ – p(z) war eben noch p(p(p(z))). Dessen Farbe wurde im aktuellen Durchlauf der while-Schleife nicht verändert. Da c) vor dem Durchlauf galt und da wir uns im 1. Fall befinden, wurde die Farbe der Wurzel nicht verändert und diese ist unverändert schwarz.

  (Wenn p(z) Wurzel ist, war diese vor dem Durchlauf unseres Algorithmus auch schon Wurzel, also schwarz.)
- zu Invariante c) „Nur Verletzung entweder der Eigenschaften 2 oder 4“ – Betrachte die 5 Eigenschaften des rot-schwarz Baumes:

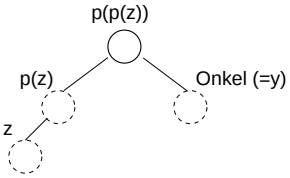


Abbildung 47: Ausgangssituation

1. bleibt erhalten
2. kann durchaus verletzt sein: Ist z nach der Umbenennung Wurzel, dann hat der 1. Fall (Zeilen 4-7) jede Verletzung der Eigenschaft 4 ausgemerzt. Da z nun rot ist, so ist, falls eine Verletzung der Eigenschaft 2 vorliegt,, dies die einzige Verletzung der r-s-Baumeigenschaft.
3. bleibt erhalten
4. Ist z nach Umbenennung nicht die Wurzel, so ist Eigenschaft 2 nicht verletzt (da c vor Durchlauf der while-Schleife galt). Die Umfärbung hatte die „alte“ Verletzung der Eigenschaft 4 repariert, hat aber unter Umständen eine neue solche verursacht und zwar am „neuen“ und seinem Elternknoten.
5. dank unserer geschickten Umformung, bleibt dies erhalten

2. Fall p(z) ist linkes Kind von p(p(z)) und Onkel von z ist schwarz und z ist rechtes Kind. Siehe Abbildung 49 auf Seite 68.

Damit liegt nun der 3. Fall vor. Achtung: Eigenschaft 5 wird bei dieser Rotation nicht beeinflusst (nur weil p(z) und z beide rot sind).

3. Fall p(z) ist linkes Kind von p(p(z)) und Onkel von z ist schwarz und z ist linkes Kind.

Egal wie wir in den 3. Fall (mittels Fall 2 oder direkt) gelangt sind, der Onkel von z ist schwarz, da sonst noch 1. Fall vorliegen würde. Außerdem existiert p(p(z)), da wir zwar z vor der Rotation eine Etage anheben, es aber in der Rotation wieder eine Etage absenken.

Im 3. Fall werden ein paar Umfärbungen und eine Rechtsrotation ausgeführt. Dabei erhalten wir Eigenschaft 5 und außerdem, da nun nicht mehr zwei aufeinanderfolgende Knoten rot sind, p(z) nun insbesondere schwarz ist, wird die while-Schleife verlassen. Siehe Abbildung 50 auf Seite 68.

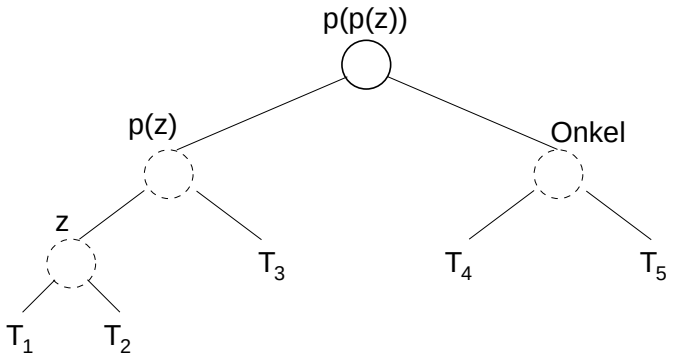
n.z.z. Invarianten gelten nach Ausführung der Zeilen 11-13

a) „z ist rot“ – ist erfüllt

b) „Ist p(z) Wurzel, so ist p(z) schwarz“ – ist erfüllt

c) – Eigenschaften 1,3,5 – mehr oder weniger einfach

aus:



mache:

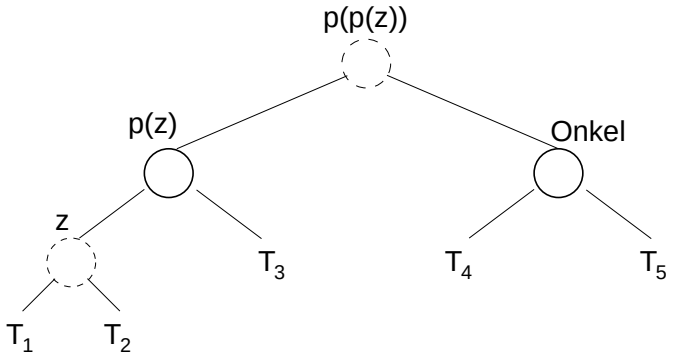


Abbildung 48: Arbeitsschritt von RS-Fixup

- Eigenschaft 2 – OK (z ist weder im 2. noch im 3. Fall die Wurzel)
- Eigenschaft 4 – nach Umfärbung und Rotation OK

$\Rightarrow$ Haben nun korrekt gefärbten RS-Baum $\rightarrow$ while-Schliefe wird verlassen

Laufzeit von RS-Fixup

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

$\Rightarrow$ RS-INSERT in $\mathcal{O}(\log n)$ möglich

Löschen in RS-Bäumen Wie TREE-DELETE(T,z), wobei NIL durch NIL(T) ersetzt wird. Unterschiede zu TREE-DELETE:

```
RS-DELETE(T,z)
1   |
2   |-- Wie TREE-DELETE
```

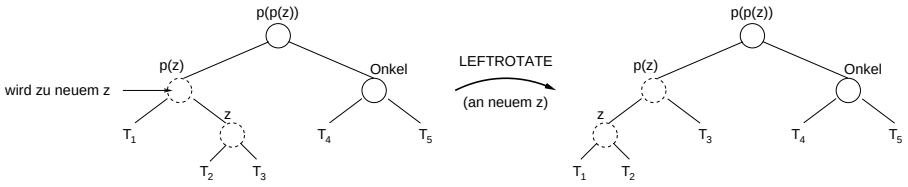


Abbildung 49: Leftrotate in RS-FIXUP

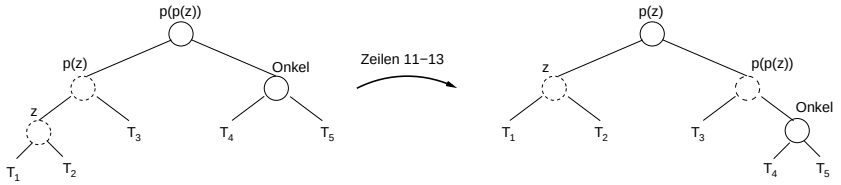


Abbildung 50: Ausführung von Zeilen 11-13 in RS-Fixup

```
3   |
4   |
5   p(x) := p(y)
6   |
7   |
8   |-- Wie TREE-DELETE
9   |
10  if color(y)=black then RS-DELETE-FIXUP(T,x)
```

Erinnerung: Vorgehen bei TREE-DELETE (Fall zu Löschender Knote hat 1 Kind)

- z – zu löschender Knoten
- y – ist Knoten mit nur einem Kind
- x – ist das eine Kind von y
- y nimmt den Platz von z ein
- x nimmt den Platz von y ein

Bemerkung Wann wird FIXUP nötig? Siehe Abbildung 51 auf Seite 69.
Löschen von Knoten:

- 2 kann einfach gelöscht werden
- 13 kann auch einfach gelöscht werden, allerdings wird 12 dabei an Position von 13 gebracht und schwarz gefärbt (RS-FIXUP führt hier nicht zum korrekten Ergebnis!)
- Löschen die 19 → Probleme

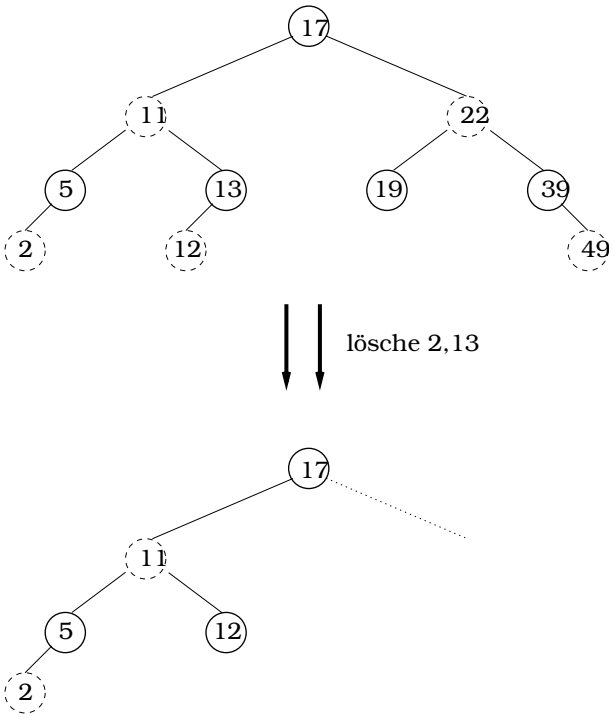


Abbildung 51: Löschen von 2 und 13

FIXUP wird nötig:

- ist der gelöschte Knoten rot, d.h. genauer ist der Knoten und Schlüssel y rot, so ist der Baum nach der Löschoperation immer noch ein RS-Baum
- ist der Knoten y schwarz, treten Probleme auf!

RS-Baum–Eigenschaften:

1. nicht beeinflusst
2. y könnte Wurzel sein → evt. Problem
3. nicht beeinflusst
4. Problem, falls p(y) und x beide rot sind
5. Problem, jeder Pfad, der den Knoten y enthält hat nach dem Löschen einen schwarzen Knoten weniger

Von jetzt an sei color(y)=black. Korrekturversuch:

- die schwarze Farbe von y an x übergeben.

⇒ Problem, wenn x bereits schwarz ist (dann müsste color(x)=schwarz-schwarz sein)

Lösungsvorschlag:

- „schwarz-schwarz“ zur Wurzel hoch reichen

Fall 0.0 x hat keine Geschwister → reiche 1 Schwarz an p(x) weiter (wenn dieser vorher schwarz war, ist dieser jetzt schwarz-schwarz) → x:=p(x)

Fall 0.1 x ist Wurzel: 1 Schwarz verschwinden lassen

Fall 1 x hat ein Geschwister w

Fall 1.1 color(w)=red → Überführe Situation, so dass sie in Fall 1.2 fällt, siehe Abbildung 52

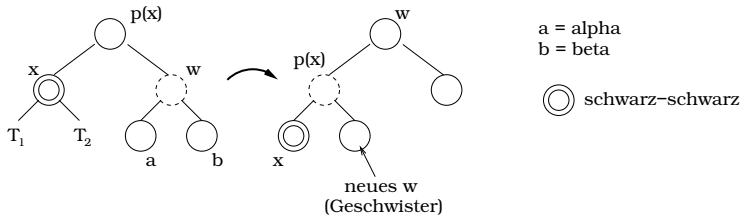


Abbildung 52: Überführe Situation in Fall 1.2

Fall 1.2 color(w)=black

Fall 1.2.1 color(α) = color(β)=black siehe Abbildung 53 auf Seite 70

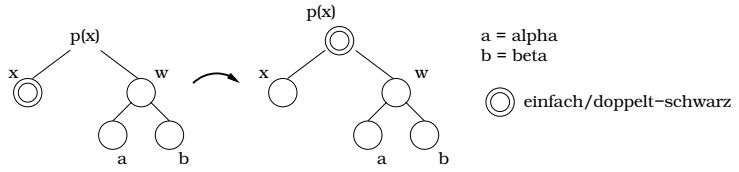


Abbildung 53: Reiche ein Schwarz nach oben

Fall 1.2.2 color(α)=red und color(β)=black siehe Abbildung 54 auf Seite 71; Eine Schwarzeinheit nach oben schieben nicht möglich, da color(w)=red; ⇒ überführt in Fall 1.2.3

Fall 1.2.3 color(α)=egal und color(β)=red siehe Abbildung 55 auf Seite 71

```
RS-DELETE-FIXUP(T,x)
1   while x != root(T) and color(x) = black
2   do {
```

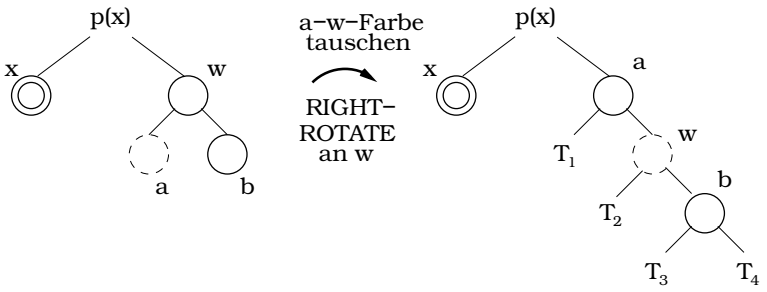


Abbildung 54:

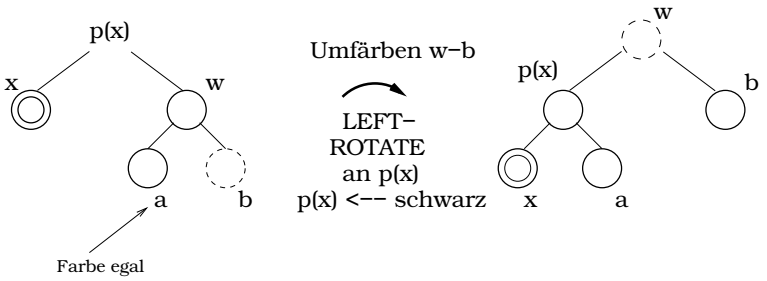


Abbildung 55:

```
        if x = left(p(x)) then do {
    w:=right(p(x))
3 if color(w)=red then do{
    color(w):=black
4               color(p(x)):=red
5               LEFT-ROTATE(T,p(x))
6               w:=right(p(x))
             }
7       if color(left(w))=black and color(right(w))=black then do {
            color(w):=red
8           x:=p(x)
9       } else do {
            if color(right(w)=black then do {
                color(left(w)):=black
10              color(w):=red
11              RIGHT-ROTATE(T,u)
12              w:=right(p(x))
            }
13          color(w=:=color(p(x))
14          color(p(x):=black
15          color(right(w)):=black
16          LEFT-ROTATE(T,p(x))
        }
```

```
17      x:=root(T) // Erlaubt die while-Schleife zu verlassen
18      (unter if aus 2. Zeile) else (wie then-Teil, nur left
        und right vertauscht)
.
.
.
25    color(x):=black
```

Laufzeit von RS-DELETE-FIXUP

$$\mathcal{O}(1) \quad \text{pro Fall}$$

wie wird zwischen den Fällen hin und her gesprungen und was passiert mit x?

Fall a: es gilt: x ist linkes Kind von $p(x)$

- x wandert eine Etage nach unten
- danach kommt Fall b1 – x eins hoch
- danach kommt nochmal Fall b1 – x eins hoch

$\Rightarrow$ insgesamt wandert x eins hoch

Fall b1 x eins hoch

Fall b2 x unverändert, aber danach gleich Fall b3

Fall b3 x auf Wurzel gesetzt

$$\Rightarrow \text{Laufzeit } \mathcal{O}(\log n)$$

$\Rightarrow$ Laufzeit von RS-DELETE auch in $\mathcal{O}(\log n)$

4 Hash-Tabellen

Sind eine effiziente Datenstruktur zur Implementierung von Wörterbüchern, d.h.

- INSERT
- DELETE
- SEARCH

sind die wichtigsten Operationen. Sie sind eine Verallgemeinerung von Arrays für folgende Situation:

> nur wenige von sehr vielen möglichen Schlüsseln müssen zu jedem Zeitpunkt verwaltet werden

Siehe Abbildung 56.

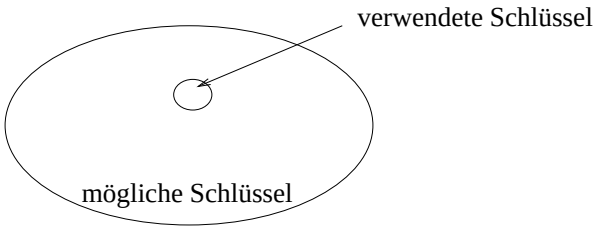


Abbildung 56: Hash-Werte

array:	Feldindex $\widehat{=}$ Schlüssel
Hash-Tabelle:	Feldindex $\widehat{=}$ f (Schlüssel)^2

4.1 Grundbegriffe

Direkte Adressierung günstig, wenn das Universum der Schlüssel eine vernünftige Größe hat.
$U = \{0, 1, \ldots, l-1\}$ Universum der Schlüssel
wir verwalten eine dynamische Menge, jedes Element besitzt einen Schlüssel aus U, keine zwei Elemente haben den selben Schlüssel

geeignete Datenstruktur:

- array (direkte Adressierungstabelle)
- jeder Feldindex entspricht einem möglichen Schlüssel
- $A[k]$ zeigt auf Element mit Schlüssel k (oder auf NIL, wenn k nicht existiert)

Effizienz Die Wörterbuchoperation sind sehr leicht zu implementieren:

```
DIRECT-ADDRESS-SEARCH(A,k)
    return A[k]
```

Laufzeit: $\mathcal{O}(1)$

```
DIRECT-ADDRESS-INSERT(A,x)
    A[key(x)]:=x
```

Laufzeit: $\mathcal{O}(1)$

```
DIRECT-ADDRESS-DELETE(A,x)
    A[key(x)]:= NIL
```

Laufzeit: $\mathcal{O}(1)$

Bemerkungen

- es gibt zwei Möglichkeiten mit Speicherplatz bei direkter Adressierung effizient umzugehen:
  - Zeiger in A[key(x)], der auf x zeigt
  - in A[key(x)] wird das komplette Datenpaket x gespeichert, bis auf key(x) (key(x) ist in Form des Feldindexes gespeichert)
- unpraktisch, wenn Menge der wirklich genutzten Schlüssel K sehr viel kleiner ist, als U

  $$\text{card } (K) \ll \text{card } (U)$$

  array: Platzbedarf $\mathcal{O}(\text{card } (U))$
  Hash-Tabelle: Platzbedarf $\mathcal{O}(\text{card } (K))$

  Dabei sollen die Operationen INSERT, DELETE, SEARCH auch in der Hash-Tabelle in $\mathcal{O}(1)$ (im average case) sein.
- direkte Adressierung vs Hash-Tabelle siehe Abbildung 57 auf Seite 74.

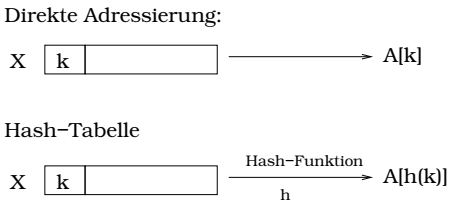


Abbildung 57: Direkte Adressierung vs. Hash-Tabelle

Hash-Funktion:

$$h : U \to \{0, 1, \ldots, m-1\} \quad h(k) \text{ — Hashwert von } k$$
$$\Rightarrow \text{Hash-Tabelle } T[0, \ldots, m-1] \quad T[j] \text{ — slot}$$

Problem beim Hashing bei card $(K) \ll$ card (U) ist h nicht injektiv, d.h. es gibt (sehr viele) Paare $k_1, k_2 \in U, k_1 \neq k_2$, aber $h(k_1) = h(k_2)$. Dies nennt man eine Kollision.

Problemlösung in Hash-Tabellen werden verkettete Listen in jeder Feldposition angelegt, sobald Kollisionen auftreten (Wenn ein Eintrag an eine Stelle geschrieben werden soll, an der schon einer steht, so wird dieser an die nächste freie Stelle der verketteten Liste geschrieben).

slot j enthält einen Zeiger auf den Kopf der Liste, in der diejenigen Elemente stehen, die auf j gehasht werden (gibt es keine solchen Elemente, so enthält slot j nur NIL)

Eine Methode, die Werte mit gleichem Hash-Wert verkettet:

```
CHAINED-HASH-INSERT(T,x)
    füge x am Kopf der Liste T[h(key(x))] ein
```

Löschen:

```
CHAINED-HASH-DELETE(T,x)
   lösche x aus der Liste T[h(key(x))]
```

Suchen:

```
CHAINED-HASH-SEARCH(T,k)
    suche in T[h(k)] nach einem Dateineintrag mit Schlüssel k
```

Zeitanalyse

- INSERT: $\mathcal{O}(1)$
- SEARCH/DELETE:
  - worst-case: $\mathcal{O}(n)$, wobei $n =$ card K
  - average case:

$$m\text{– Zahl der slots}$$

$$n\text{– Zahl der gespeicherten Elemente}$$

$$\alpha =_{\text{def}} \frac{n}{m}$$

α ist die mittlere Zahl der Elemente pro slot (Liste) $\widehat{=}$ Auslastungsfaktor

Analyse mit Hilfe von α

4.1.1 Satz – Zeitaufwand von SEARCH

In einer Hash-Tabelle, in der Kollisionen durch Verkettung aufgelöst werden, benötigt SEARCH im Erwartungswert $\mathcal{O}(1 + \alpha)$ viele Rechenschritte, wenn man davon ausgeht, dass

* jedes Element unabhängig von den bisher einsortierten Elementen mit gleichverteilter Wahrscheinlichkeit in jeden der m slots gehasht wird.

Bemerkung

- ist $n \in \mathcal{O}(m)$, so ist SEARCH im average case in $\mathcal{O}(1)$ möglich
- ebenso DELETE in $\mathcal{O}(1)$

4.2 Hash-Funktionen

- ideal: Hash-Funktionen, die * erfüllen
- Aber: * ist „unmöglich“ zu prüfen

In Praxis mit Heuristiken Hash-Funktionen erzeugen (unter Benutzung von Informationen über die zu erwartenden Daten)

Jetzt ein paar konkrete Hash-Funktionen

1. Divisionsmethode

$$h(k) := k \bmod m$$

- schnell, sicher
- günstig, m keine Zweierpotenz
- m sollte Primzahl sein, nicht zu dicht an Zweierpotenz

2. Multiplikationsmethode

$$h(k) := \lfloor m * (k * A - \lfloor k * A \rfloor) \rfloor \qquad \text{wobei } 0 < A < 1$$

- günstig: Wahl von m unkritisch
- üblicherweise nimmt man $m = 2^p$
- Man erhält immer einen Wert zwischen 0 und $m - 1$

bisher: jede statische Hash-Funktion ist anfällig gegen ungünstige Schlüssel

3. Universelles Hashing

Idee Hash-Funktion zufällig wählen und zwar so geschickt, dass die Hash-Funktion unabhängig von den Schlüsseln ist

Man hat Familie von Hash-Funktionen, aus denen zufällig gewählt werden

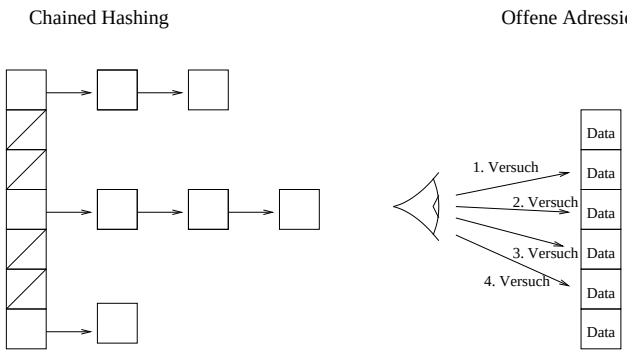


Abbildung 58: Chained hashing vs. offene Adressierung

Beispiel zu universellem Hashing wähle p eine Primzahl

$$h_{a,b}(k) =_{\text{def}} ((ak + b) \bmod p) \bmod m$$

$a \in \{1, 2, \ldots, p-1\}$, $b \in \{0, 1, 2, \ldots, p-1\}$

$$H_p = \{h_{a,b} : a \in \{1, \ldots p-1\} \wedge b \in \{0, \ldots, p-1\}\}$$

Offene Adressierung

siehe Abbildung 58 auf Seite 77

hash-Funktion liefert Folge von Slots, die nacheinander dahingehend überprüft werden, ob sie frei sind. In den ersten freien solchen Slot wird das Datenpaket hineingehasht.

Bemerkungen

- Hash-Table kann „voll“ sein → hash-table-overflow
- Auslastungsfaktor $\alpha \leq 1$

  formal:

  $$h : \underbrace{U}_{\text{Schlüssel}} \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{Versuchsnummer}} \to \underbrace{\{0, 1, \ldots, m-1\}}_{\text{Slotnummer}}$$

  wobei U das Universum und m die Größe der Hash-Tabelle

Beispiel

- eine schlechte Hash-Funktion: ??? nächste Zeile unklar

  $$h_1(k, i) = 1 \text{ für alle } \in \text{ und } k$$

  Idealerweise ist für alle $k \in U$ die Folge $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ eine Permutation von $0, 1, \ldots, m-1$

- eine weitere schlechte Hash-Funktion:

$$h_2(k, 1) = i \text{ für alle } k \text{ und } i$$

  Nachteil: Clusterbildung, Einfügezeit zu hoch

Algorithmen

```
HASH-INSERT(T,k)
1   i := 0
2   repeat j := h(k,i)
3          if T(j) = NIL then T(j) := k (+ Satellitendaten)
4                              return j
5                         else i := i+1
6   until i = m
7   error "hash-table-overflow"
```

```
HASH-SEARCH(T,k)
1   i := 0
2   repeat j := h(k,i)
3          if T(j) = k then return j -> Ende
4          else i := i+1
5   until T(j) = NIL or i = m
6   return NIL
```

Bemerkung Da bei HASH-SEARCH genau die gleiche Reihenfolge angenommen wird wie bei HASH-INSERT, folgt, dass wenn man auf ein NIL-Element stößt, kann auch bei weiterem Durchlaufen der Liste kein weiteres Elemente gefunden werden → Zeile 5 in Hash-Search(T,k)

Daraus folgt allerdings: beim Löschen muss man sorgfältig agieren: nicht einfach NIL in einen Slot schreiben, sondern DELETED

INSERT muss entsprechend modifiziert werden: Zeile 3:

```
if T(j) = NIL or T(j) = DELETED then T(j) := k
```

Ein paar Hash-Funktionen für offene Adressierung

i) Lineares Sondieren Sei $h' : U \to \{0, 1, \ldots m-1\}$ eine normale Hash-Funktion. Wir definieren

$$h(k, i) =_{\text{def}} (h'(k) + i) \bmod m \quad \text{für } m \in U \text{ und } i \in \{0, \ldots, m-1\}$$

siehe Abbildung 59 auf Seite 79

Die Slots werden in der Reihenfolge $T(h'(k)), T(h'(k)+1), T(h'(k+2)), \ldots$ ausprobiert.

- Vorteil: leicht zu implementieren

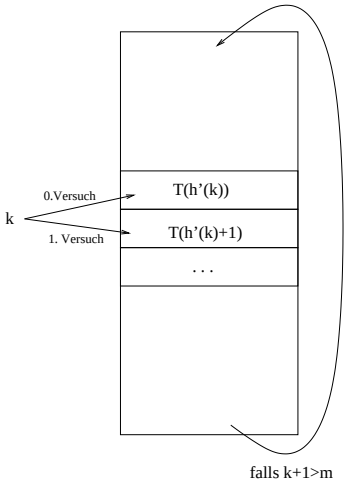


Abbildung 59: Lineares Sondieren

- Nachteil: wenn $h'(k) = h'(\hat{k})$, dann wird bei Einfügen der Schlüssel k und $\hat{k}$ die Hash-Tabelle in der selben Weise sondiert
  $\Rightarrow$ Clusterbildung (Lange zusammenhängende, volle Blöcke)

ii) quadratisches Sondieren Sei h' wie eben eine normale Hash-Funktion. Wir definieren:

$$h(k,i) = (h'(k) + c_1 * i + c_2 * i^2) \bmod m$$

Dabei sind c_1 und c_2 Konstanten und es gelte $c_2 \neq 0$.

- Vorteile:
  - einfach zu implementieren
  - funktioniert besser als lineares Sondieren (kaum Cluster)
- Nachteile:
  - Wahl von c_1, c_2 und m gewinnt zunehmend an Bedeutung (sie müssen sorgfältiger gewählt werden).
  - Für zwei Schlüssel k und $\hat{k}$ mit $h'(k) = h'(\hat{k})$ sind die Folgen der sondierten Slots gleich.
    $\Rightarrow$ schwache Form der Clusterbildung

siehe Abbildung 60 auf Seite 80.

iii) doppeltes Hashing Seien h_1 und h_2 zwei normale Hash-Funktionen. Wir definieren:

$$h(k,i) := (h_1(k) + i * h_2(k)) \bmod m$$

Im Gegensatz zu i) und ii) hängt die Folge der sondierten Slots zweifach von k ab, siehe Abbildung 61 auf Seite 80

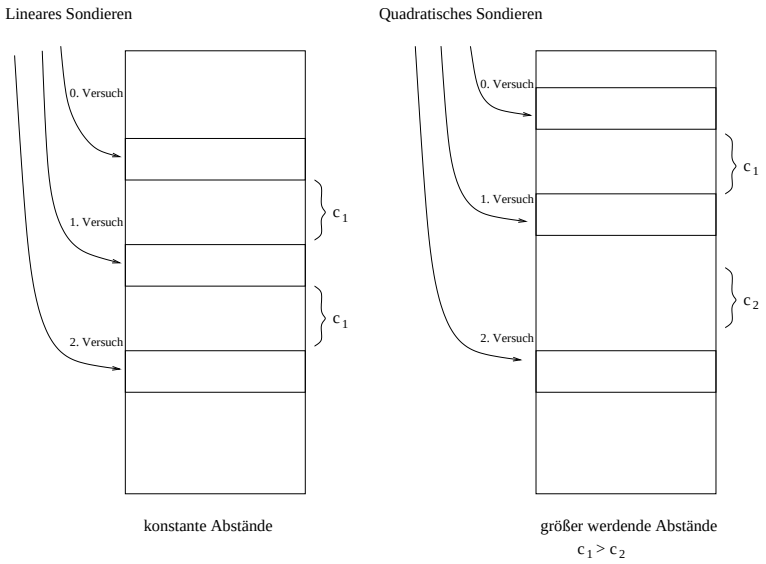


Abbildung 60: Lineares Sondieren vs. quadratisches Sondieren

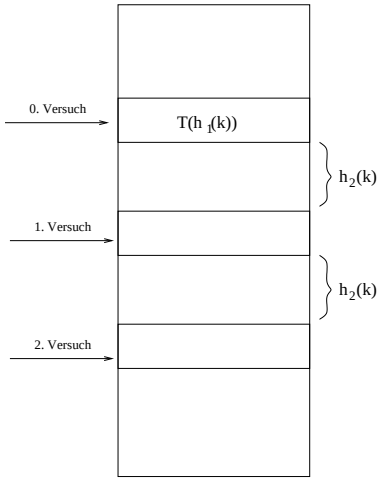


Abbildung 61: Doppelte Abhängigkeit von k

Bemerkung $h_2(k)$ und m sollten stets teilerfremd sein

Beispiel

$$m = 15$$
$$h_2(k) = 5$$
$$h_1(k) = 7$$

sondierte Slots:

$$h(k,0) = 7$$
$$h(k,1) = 12$$
$$h(k,2) = 2$$
$$h(k,3) = 7$$
$$h(k,4) = 12$$

Wie erreicht man $\text{ggT}(h_2(k), m) = 1$ einfach?

- eine Möglichkeit:
  - m Primzahl
  - $h_2(k)$ positive ganze Zahl ($< m$)
- weitere Möglichkeit:
  - $m = 2^t$ für ein t
  - $h_2(k)$ ungerade für alle k

Um die Qualität eines Hash-Verfahrens zu beurteilen, kann man die Zahl der verschiedenen Sondierungssequenzen heranziehen:

- lineares Sondieren: $h'(k) + i$:

$$\left.\begin{array}{c} 0, 1, 2, 3, \ldots \\ 1, 2, 3, 4, \ldots \\ \vdots \\ m-1, 0, 1, 2, \ldots \end{array}\right\} m \text{ Stück}$$

$\mathcal{O}(m)$ viele Sondierungsfolgen

- quadratisches Sondieren: $h'(k)$ definiert die Folge vollständig

$\mathcal{O} = (m)$ viele Sondierungfolgen

- doppeltes Hashing:

$\mathcal{O}(m^2)$ viele Sondierungsfolgen

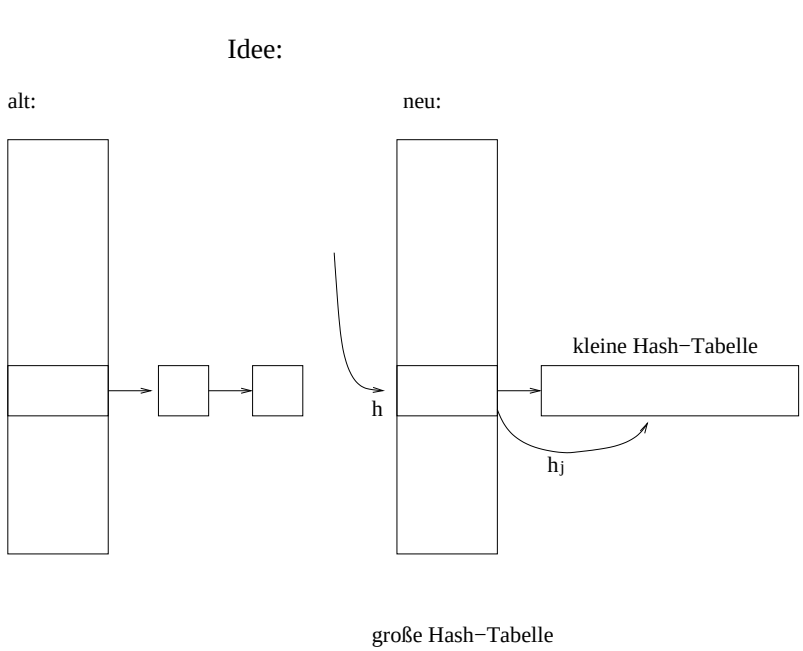


Abbildung 62: Neuer Ansatz, Hash-Tabelle in der Hash-Tabelle

Nachtrag zu chainded hashing Statt in einem Slot j bei Kollision eine Liste zu kreieren, nutzen wir eine zweite kleine Hash-Tabelle S_j mit einer Hash-Funktion h_j. Siehe Abbildung 62 auf Seite 82.

Wählt man die h_j sehr sorgfältig, so kann man Kollisionen in der 2. Ebene völlig vermeiden. Dazu muss man die Größe von S_j quadratisch in n_j wählen. Dabei ist n_j, die Zahl der Schlüssel, die in den Slot $T(j)$ gehasht werden. Zwar erscheint der Speicherplatzbedarf nun immens groß, wenn man die Hash-Funktion h gut wählt, ist der Gesamtspeicherbedarf in lediglich $\mathcal{O}(n)$.

4.2.1 Satz – Zahl der inspizierten Zellen bei doppeltem Hashing

Ist T eine Hash-Tabelle basierend auf doppeltem Hashing mit Auslastungsfaktor $\alpha = \frac{n}{m} < 1$, so ist die erwartete Anzahl der inspizierten Zellen

- bei erfolgloser Suche: $\leq \frac{1}{1-\alpha}$
- bei erfolgreicher Suche: $\leq \frac{1}{\alpha} * \ln \frac{1}{1-\alpha}$

wenn man davon ausgeht, dass das Hashing uniform ist, d.h. * erfüllt.

4.3 Perfektes Hashing

- funktioniert auch im worst case sehr gut (effizient)
- ideal, wenn Menge der Schlüssel statisch ist (Wörter in Programmiersprache, Filename auf CD-ROM)

Idee: zweistufiges Hashing Schema mit universellem Hashing in jeder Stufe

1. Stufe: ähnlich wie Hashing mit Verkettung. Hash-Funktion wird sorgfältig aus Familie universeller Hash-Funktionen ausgewählt.
$\Rightarrow$ Hash-Tabelle T, m Slots

2. Stufe in jedem Slot $T(j)$ wird eine Hash-Tabelle S_j angelegt und die Schlüssel $k_1, \ldots k_j$, die in der 1. Stufe nach $T(j)$ gehasht wurden, werden nun mit Hilfe einer Hash-Funktion h_j in die Slots von S_j gehasht. Dabei werden die Funktionen h_j wiederum aus einer Familie universeller Hash-Funktionen ausgewählt.

Beispiel Folgende Werte sollen gehasht werden: $K = \{10, 22, 37, 40, 60, 70, 75\}$

- 1. Stufe: $h(k) =_{\text{def}} ((ak + b) \bmod p) \bmod m$, $a = 3$, $b = 42$, $p = 101$, $m = 9$
- 2. Stufe: $h_j(k) =_{\text{def}} ((a_j k + b_j) \bmod p) \bmod m_j$

$$\begin{array}{ccc} m_0 = 8 & \ldots & m_7 = 8 \\ a_0 = 1 & \ldots & a_7 = 23 \\ b_0 = 1 & \ldots & b_7 = 88 \end{array}$$

siehe Abbildung 63 auf Seite 83.

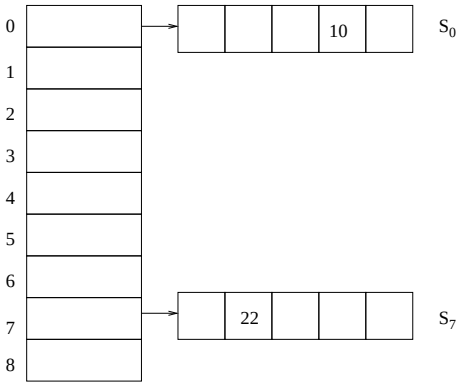


Abbildung 63: Beispiel für Hash-Tabelle in Hash-Tabelle

4.3.1 Satz – universelles Hashing

Werden n Schlüssel in einer Hash-Tabelle der Größe $m = n^2$ mit einer zufällig gewählten Hash-Funktion aus einer universellen Familie von Hash-Funktionen eingefügt, so gelten

1. Die Wahrscheinlichkeit, dass es eine oder mehrere Kollisionen gibt ist kleiner als $\frac{1}{2}$.

2. $E[\sum_{j=0}^{n-1} n_j] < 2n$ ($E \;\widehat{=}$ Erwartungswert), wobei n_j die Zahl der in den Slot j gehashten Schlüssel ist.

(ohne Beweis)

5 Graphenalgorithmen

5.1 Graphen und ihre Darstellungen

a) Adjazensmatrix (Nachbarschaftsmatrix)

5.1.1 Definition – Adjazensmatrix

Sei $G = (V, E)$ ein gerichteter Graph (ohne Mehrfachknoten) mit $V = \{v_1, \ldots, v_n\}$. Die Adjazensmatrix $A_g(a_{ij})_{i,j=1}^n$ von G ist definiert durch

$$a_{ij} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$

für alle $i, j \in \{1, \ldots, n\}$

Beispiel Siehe Abbildung 64 auf Seite 84.

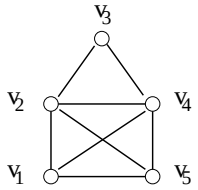


Abbildung 64: Beispielgraph

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	0	1	1
v_2	1	0	1	1	1
v_3	0	1	0	1	0
v_4	1	1	1	0	1
v_5	1	1	0	0	1

Bemerkungen

- Speicherbedarf $\Theta(n^2)$, egal wie viele Kanten der Graph hat
- für ungerichtete Graphen ist

$$a_{ij} = \begin{cases} 1 & \text{falls } \{v_i, v_j\} \in E \\ 0 & \text{sonst} \end{cases}$$

$\Rightarrow$ Adjazensmatrix ist symmetrisch
$\Rightarrow$ eigentlich nur eine der beiden Dreiecksmatrizen interessant. Trotzdem immer noch $\Theta(n^2)$ Einträge. gerichtet: $E \subseteq V \times V$, ungerichtet: $E \subseteq \mathcal{P}_2(v)$

- bei gewichteten Graphen kann statt 0 oder 1 der das Gewicht der Kante $(v_i, v_j$ $\omega(v_i, v_j)$ in a_{ij} gespeichert werden. Falls $(v_i, v_j) \notin E \rightarrow$ NIL (um keine Verwechslung mit Kanten der Wertigkeit 0 zu haben).
- uneffizient, wenn Graphen wenig Kanten haben (so genannte dünne Graphen, d.h. $m \in \mathcal{O}(n)$, wobei m Zahl der Kanten, z.B. Bäume, Pfade, Kreise). Uneffizient weil: Größe der Datenstruktur kann auch negative Auswirkung auf die Rechenzeit haben. Z.B. nehme man einen Algorithmus mit folgender Rechenzeit an: $\mathcal{O}(m +$ Größe der Datenstruktur)
- einfach zu implementieren

b) Adjazensliste (Nachbarschaftsliste)

5.1.2 Definition – Adjazensliste

Sei $G = (V, E)$ ein gerichteter Graph (ohne Mehrfachkanten) mit $V = \{v_1, \ldots v_n\}$. Die Adjazensliste AL_g von G ist ein Feld $AL_g[1, \ldots, n]$ bestehend aus n Listen, eine für jeden Knoten von G. Dabei enthält eine jede Liste $AL_g[i]$ alle Knoten v_j, für die $(v_i, v_j) \in E$.

Beispiel Siehe Abbildung 65

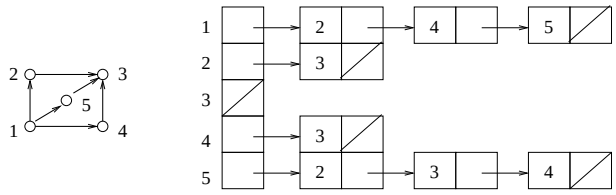


Abbildung 65: Beispieladjazensliste

Bemerkungen

- die in $AL_g[i]$ gespeicherten Knoten bilden gerade die Nachbarschaft von v_i
- die Knoten in $AL_g[i]$ sind in beliebiger Reihenfolge gespeichert
- effiziente Form der Speicherung, denn Summe der Längen der Listen ist gerade card (E)
- bei ungerichteten Graphen ist die Summe der Länge der Listen $2 *$ card (E)
- es können auch Gewichte gespeichert werden, indem zusätzlich zu dem Zeiger auf den nächsten Knoten der Eintrag Gewicht gespeichert wird

Nachteil Wie schwierig ist es herauszufinden, ob für gegebenes i, j $(v_i, v_j) \in E$?

Adjazensmatrix	Adjazensliste
$\mathcal{O}(1)$	$\mathcal{O}(n)$ (worst case)

5.2 Breitensuche (breath-first-search, bfs)

Siehe Abbildung 61 auf Seite 80.

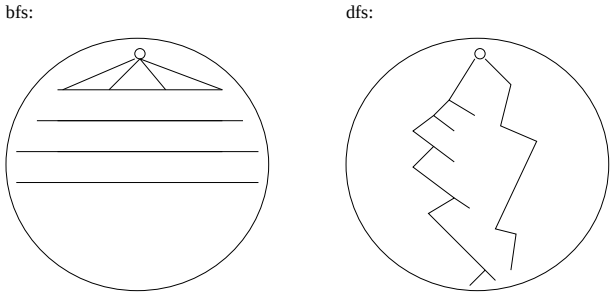


Abbildung 66: Vergleich Breitensuche vs. Tiefensuche

Bemerkungen

- eines der einfachsten Suchverfahren in Graphen
- Grundbaustein vieler Graphenalgorithmen
- durchsucht alle Knoten des Graphen ausgehend von einem Startknoten gemäß der Regel: erst in der Breite, dann in der Tiefe suchen
- Algorithmus erzeugt einen Breitensuchbaum, der alle vom Startknoten aus besuchten Knoten enthält
- dabei entspricht der im Breitensuchbaum gespeicherte Pfad zwischen zwei Knoten s und v (s ist Startknoten!) immer einem kürzesten Pfad zwischen den Knoten im Ausgangsgraphen
- Knoten werden gefärbt gemäß ihrer Position relativ zum aktuellen Stand der Suche: siehe Abbildung 67 auf Seite 86.

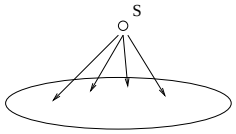


Abbildung 67: Breitensuche (Schema)

Bedeutung der Farben:

  - weiß: Knoten, die noch nicht besucht wurden
  - grau: Knoten, die bereits besucht wurden, deren Nachbarschaft jedoch noch nicht vollständig „abgegrast“ wurde
  - schwarz: Knoten, deren komplette Nachbarschaft schon besucht wurde

  Graue Knoten haben noch weiße Nachbarn, schwarze Knoten haben nur graue oder schwarze Nachbarn.

- bfs-Baum:
  - Wurzel s (Startknoten)
  - wird während der Breitensuche ein weißer Knoten v entdeckt, so wird v in den bfs-Baum aufgenommen und auch die Kante (u, v) auf der ich den Knoten v entdeckt habe (u heißt Vorgänger von v im bfs-Baum).
- G ist in Form einer Adjazensliste gegeben
- während einer Breitensuche werden folgende Informationen gespeichert: für jeden Knoten u:

color(u)	Farbe
$\pi(n)$	Vorgänger
$d(u)$	Entfernung vom Startknoten s

- benutzen FIFO-Datenstruktur (Schlange), um graue Knoten zu speichern. Operationen auf der Datenstruktur:

head	Kopf der Schlange
Enqueue	Fügt in Schlange ein
Dequeue	Entfernt aus Schlange

BFS(y, s)

```
1   für jeden Knoten  u ∈ V\{s}  do color (u) :=  weiß
2                                                  d(u) := ∞
3                                                  π(u) :=  NIL
4   color(s) :=  grau
5   d(s) :=  0
6   π(s) :=  NIL
7   Q := < s >    // Q ist queue der noch abzuarbeitenden Knoten
8   while Q ≠ ∅  do
9                 u :=  head(Q)
10                für jeden Knoten v ∈ AL_g(u)  do
11                                  if color(v) = weiß then do
12                                                  color(v) :=  grau
13                                                  d(v) := d(u) + 1
14                                                  π(v) := u
15                                                  Enqueue(Q, v)
```

```
16                  Dequeue(Q)
17                  color(u) := schwarz
```

Bemerkung

- mit leichten Modifikationen kann im Algorithmus BFS der BFS-Baum mit konstruiert werden

Beispiel Siehe Abbildung 68 auf Seite 88.

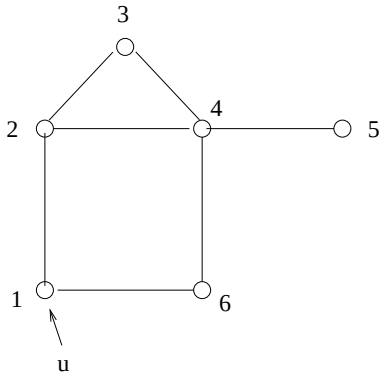


Abbildung 68: Beispielgraph

Folgende Abkürzungen werden verwendet: s: schwarz, g: grau, w: weiß

	1	2	3	4	5	6
color	g	w	w	w	w	w
d	∞	∞	∞	∞	∞	∞
π	NIL	NIL	NIL	NIL	NIL	NIL

$$Q =< 1 > \text{ (Liste)}$$

$$\Downarrow$$

	1	2	3	4	5	6
color	g	g	w	w	w	w
d	0	1	∞	∞	∞	∞
π	NIL	1	NIL	NIL	NIL	NIL

$$Q =< 1, 2 >$$

$$\Downarrow$$

	1	2	3	4	5	6
color	g	g	w	w	w	g
d	0	1	∞	∞	∞	1
π	NIL	1	NIL	NIL	NIL	1

$\Downarrow$

	1	2	3	4	5	6
color	s	g	g	w	w	g
d	0	1	2	∞	∞	1
π	NIL	1	2	NIL	NIL	1

$Q =< 6, 3, 4 >$

$\Downarrow$

	1	2	3	4	5	6
color	s	g	g	w	w	s
d	0	1	2	∞	∞	1
π	NIL	1	2	0	0	1

$Q =< 3, 4 >$

$\Downarrow$

$\vdots$

$\Downarrow$

	1	2	3	4	5	6
color	s	s	s	s	s	s
d	0	1	2	2	3	1
π	NIL	1	2	2	5	1

Annahme für unseren Durchlauf: Knoten in Adjazensliste jeweils sortiert gemäß $\leq$ auf $\mathbb{N}$. Der BFS-Baum würde bei diesem Beispiel wie Abbildung 69 auf Seite 90 aussehen.

Laufzeit des Algorithmus $G = (V, E)$, $n = \text{card } (v)$, $m = \text{card } (E)$

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

Denn Zeilen 1-3 im Algorithmus $\in \mathcal{O}(n)$ und Zeilen 9-15 $\in \mathcal{O}(m)$.

5.3 Kürzeste Wege

shortest path

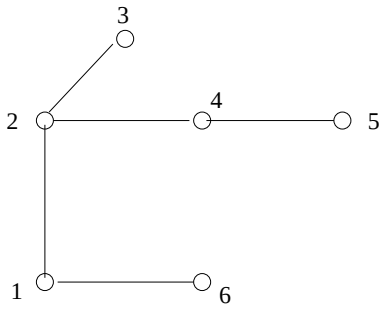


Abbildung 69: Graph, der durch Breitensuche ermittelt wurde

Problem:

gegeben: Zwei Knoten $u, v \in V$ eines Graphen $G = (V, E)$

gesucht: Länge eines kürzesten Pfades von u nach v in G.

$$\delta(u, v) = \text{ Zahl der Kanten auf einem kürzesten } u - v - \text{Pfad}$$

Wir betrachten hier das Problem nur für ungerichtete Graphen.

Wir wollen zeigen, dass für die im Algorithmus BFS(G, s) berechneten d-Werte gilt: Für alle Konten $u \in V$ ist $d(u) = \delta(s, u)$

$\Rightarrow$ Das kürzeste-Wege-Problem kann mit einem Aufruf BFS(G,u) sehr einfach gelöst werden.

Zunächst ein paar vorbereitende Lemmata.

5.3.1 Lemma

Seien $G = (V, E)$ ein (einfacher, ungerichteter) Graph, $s \in V$ und $\{u, v\} \in E$. Dann gilt:

$$\delta(s, v) \leq \delta(s, u) + 1$$

Beweis Siehe Abbildung 70 auf Seite 91.

5.3.2 Lemma

Sei $G = (V, E)$ ein Graph (ungerichtet, einfach). Für jeden Knoten $u \in V$, sei $d(u)$ der im Algorithmus BFS(G, s) berechnete d-Wert. Für alle Knoten $u \in V$ gilt dann:

$$d(u) \geq \delta(s, u)$$

Beweis $d(u)$ gibt die Länge *eines* Pfades von s nach u an. $\delta(s, u)$ ist die Länge des kürzesten Pfades. Damit muss die Ungleichung gelten.

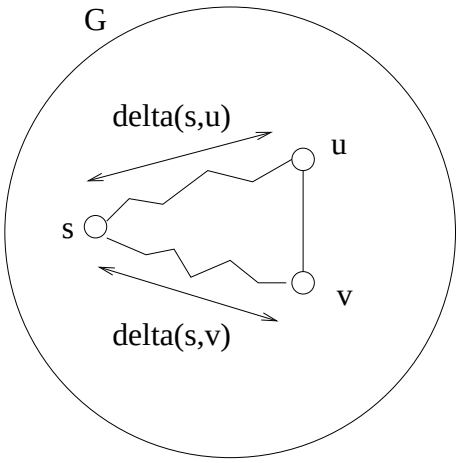


Abbildung 70: Pfadlängen

5.3.3 Lemma

Während des Durchlaufs von BFS(G, s) möge die Schlange Q die Knoten $v_1, v_2, \ldots, v_t$ in dieser Reihenfolge enthalten, v_1 ist head und v_t tail von Q.

Es gilt:

$$d(v_t) \leq d(v_1) + 1$$

$$d(v_i) \leq d(v_{i+1}) \text{ für alle } i = 1, 2, \ldots, t$$

Bemerkung

$$Q =< v_1\, v_2\, v_3\, \ldots\, v_t >$$

$$d(v_1) \leq d(v_2) \leq d(v_3) \leq \cdots \leq d(v_t) \leq d(v_1) + 1$$

Beweis Induktion über die Zahl der Queue-Operationen

Induktionsanfang:

$$Q :=< s >$$

Aussage gilt.

Induktionsschritt: zeige, dass wenn Ungleichungen vor Queue-Operationen (Enqueue, Dequeue) galten, die dann auch danach noch gelten.

- zu Dequeue:

  vorher: $Q =< v_1, v_2, v_3, \ldots, v_t >$ $\quad d(v_1) \leq d(v_2) \leq d(v_3) \leq \cdots \leq d(v_t) \leq d(v_1) + 1$

  nachher: $Q =< v_2, v_3, \ldots, v_t >$ $\quad d(v_2) \leq d(v_3) \leq \cdots \leq d(v_t) \leq d(v_1)+1 \leq d(v_2)+1$

- zu Enqueue:

  $$\text{vorher: } Q =< v_1, v_2, \ldots, v_t > \quad d(v_1) \leq d(v_2) \leq \cdots \leq d(v_t) \leq d(v_1) + 1$$

  $$\text{nachher: } Q =< v_1, v_2, \ldots, v_t, u > \quad d(v_1) \leq d(v_2) \leq \cdots \leq d(v_t) \overset{?}{\leq} d(u) \overset{?}{\leq} d(v_1) + 1$$

  Untersuche Ungleichung: $d(v_t) \leq d(u)$: im Algorithmus wird $d(u)$ gerade auf $d(v_1)+1$) gesetzt. Damit gilt ebenfalls die Ungleichung $d(u) \leq d(v_1) + 1$

5.3.4 Satz – Eigenschaften von BFS

Sei $G = (V, E)$ ein ungerichteter Graph und sei $v \in V$. Führt man BFS(G, v) aus, so gilt:

1. Genau die Knoten, die von v aus erreichbar sind, werden schwarz oder grau markiert.
2. Für alle Knoten $u \in V$ gilt am Ende des Algorithmus $d(u) = \delta(v, u)$.
3. Für alle Knoten $u \in V$ mit $u \neq v$, die von v aus erreichbar sind, gilt:

   Es gibt einen kürzesten Pfad von v nach u, der den Knoten $\pi(u)$ und die Kante $\{\pi(u), u\}$ enthält

Beweis

1. Fall: u ist von v aus nicht erreichbar

$$\Rightarrow \delta(v, u) = \infty$$

$$\Rightarrow d(u) = \infty$$

Dies folgt aus Lemma 5.3.2 ($d(u) \geq \delta(v, u)$). Dieses Lemma gilt gdw. $d(u)$ im Verlauf von BFS(G, v) nie auf einen endlichen Wert gesetzt wird. Dies gilt gdw. u nie grau oder schwarz gefärbt wird.
$\Rightarrow$ 1.,2.,3. für diesen Fall erledigt.

2. Fall: u ist von v aus erreichbar Sei $V_k =_{\text{def}} \{u \in V : \delta(v, u) = k\}$ für alle $k \geq 0$. Wir beweisen 1., 2. und 3. mittels Induktion über k.

Insbesondere zeigen wir induktiv, dass für jeden Knoten $u \in V_k$ genau ein Zeitpunkt im Verlauf von BFS(G, v) existiert zudem:

1. u grau gefärbt wird
2. $d(u) = k$ gesetzt wird
3. für $u \neq v$ der Wert $\pi(u)$ auf $\tilde{u}$ für ein $\tilde{u} \in V_{k-1}$ gesetzt wird
4. u in die Schlange Q aufgenommen wird.

Induktionsanfang $k = 0$ ($V_0 = \{v\}$)
$\Rightarrow$ während der Initialisierung (Zeilen 4-7) wird v grau gefärbt, $d(v) = 0$ gesetzt und v in Schlange Q aufgenommen.

Man sieht sofort, dass kein Knoten mehr als einmal in die Schlange aufgenommen wird.

Induktionsschritt Beobachtungen:

- Q niemals leer, bis Algorithmus endet
- wird Knoten u in Schlange Q aufgenommen, so werden „in dem Moment“ auch die d und die π Werte gesetzt und nicht mehr verändert

Wissen: werden die Knoten $u_1, u_2, \dots u_r$ in dieser Reihenfolge in die Schlange Q aufgenommen, so gilt $d(u_1) \leq d(u_2) \leq \cdots \leq d(u_r)$ (Lemma 5.3.3).

Sei $u \in V_k,\ k \geq 1$
$\Rightarrow$ u wird erst nach *allen* Knoten aus V_{k-1} in Q aufgenommen.

- wegen $\delta(v, u) = k$ gibt es einen Pfad von v nach u mit k Kanten, d.h. es existiert $\tilde{u} \in V_{k-1}$ mit $\{\tilde{u}, u\} \in E$
- Sei $\tilde{u}$ der erste solche Knoten, der im Verlauf von BFS(G, v) grau gefärbt wird.
- $\tilde{u}$ wird irgendwann Kopf von Q, wird die Nachbarschaft von $\tilde{u}$ durchsucht und u entdeckt

$\Rightarrow$ Zeile 12 färbt u grau

Zeile 13 setzt $d(u) = d(\tilde{u} + 1) = k$

Zeile 14 setzt $\pi(u) = \tilde{u}$

Zeile 15 fügt u in Q ein.

$\Rightarrow$ Induktionsbehauptung gilt!

□

Bemerkungen zu BFS-Bäume Baum mit Knotenmenge

$$V_\pi = \text{def}\{v \in V : \pi(v) \neq \text{NIL}\} \cup \{s\}$$

und Kantenmenge

$$E_\pi =_{\text{def}} \{\{\pi(v), v\} : v \in V_\pi \backslash \{s\}\}$$

- das (V_π, E_π) ein Baum ist, kann man sich leicht überlegen
- der BFS-Baum enthält nur kürzeste Pfade: für jeden Knoten $v \in V_\pi \backslash \{s\}$ ist der $s - v$-Pfad in (V_π, E_π) ein kürzester Pfad in (V, E) von s nach v.

5.4 Tiefensuche (depth first search, DFS)

- ähnlich BFS: $\pi(v)$ und Färbung
- anders als BFS:
  - die begangenen Kanten formen keinen Baum mehr, sondern Wald G_i (Menge von Bäumen) $G_\pi = (V, \{\{\pi(u), u\} : u \in V \wedge \pi(u) \neq \text{NIL})$
  - pro Knoten u haben wir zwei Zeitpunkte
    - $d(u)$ – Entdeckungszeitpunkt
    - $f(u)$ – Zeitpunkt, zu dem $\text{AL}_g(u)$ komplett durchforstet

    diese beiden Zeitpunkte sind nützlich für diverse Graph-Algorithmen

DFS(G)

time – globale Variable

```
1   für jeden Knoten u aus V do color(u) := weiß
2                                π(u) := NIL
3   time := 0
4   für jeden Knoten u aus V do if color(u) = weiß then DFS-VISIT(G,u)
```

DFS-VISIT(G, u)

```
1   color(u) := grau
2   d(u) := time + 1
3   time := time + 1
4   für jeden Knoten v ∈ AL_g(u) do
                    if color(v) = weiß then do π(v) := u
5                                              DFS-VISIT(G,v)
6   color(u) := schwarz
7   f(u) := time +1
8   time := time +1
```

Beispiel Endzustand von DFS(G) bezogen auf Beispielgraph aus Abbildung 68 auf Seite 88:

	1	2	3	4	5	6
color	s	s	s	s	s	s
π	NIL	1	2	3	4	4
d	1	2	3	4	5	7
f	12	11	10	9	6	8

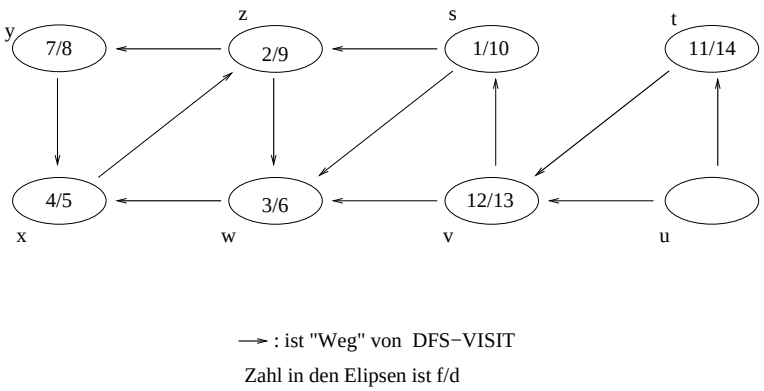


Abbildung 71: Beispiel für DFS-Visit Durchlauf

Laufzeit

- Zeilen 1-3: $\mathcal{O}(n)$
- für jeden Knoten wird DFS-VISIT genau einmal aufgerufen
- während eines Aufrufs wird Schleife card $(N(u))$ (Zahl der Nachbarn) mal durchlaufen.

$$\sum_{u \in V} \text{card } (N(u)) \in \Theta(m)$$

$$\Rightarrow \text{ Gesamtlaufzeit } \Theta(n+m)$$

Bemerkungen

- G_π ist ein Wald
- die d- und f-Werte haben Klammerstruktur (d steht für eine öffnende Klammer, f für eine Schließende)

$$\underset{(}{d(v_1)} \leq \underset{(}{d(v_2)} \leq \cdots \leq \underset{)}{f(v_n)} \leq \underset{)}{f(v_{n-1})} \leq \cdots \leq \underset{)}{f(1)}$$

5.4.1 Satz – Klammertheorem

Nach jedem Durchlauf von DFS gilt für beliebige zwei Knoten u und v des Graphen genau eine der drei folgenden Bedingungen:

1. Die Intervalle $[d(u), f(u)]$ und $[d(v), f(v)]$ sind disjunkt und weder u noch v ist Abkömmling des jeweils anderen Knoten s im DFS-Wald
2. Das Intervall $[d(u), f(u)]$ ist vollständig im Intervall $[d(v), f(v)]$ enthalten und u ist ein Abkömmling von v im DFS-Wald
3. wie 2. mit u und v vertauscht

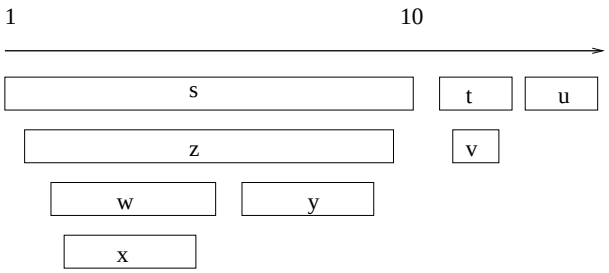


Abbildung 72: Klammerstruktur von DFS

Beispiel siehe Abbildung 72 auf Seite 96.

Beweis

1. Fall $d(v) < d(u)$

Fall 1a) $d(u) < d(f(v))$

$\Rightarrow$ u wird entdeckt, als v noch grau ist

$\Rightarrow$ u ist ein Abkömmling von v.

$\Rightarrow$ da u später als v entdeckt wurde, wird die Nachbarschaft von u eher abgegrast, als die von v

$$\Rightarrow f(u) < f(v)$$

$\Rightarrow$ Intervall $[d(u), f(u)]$ ist vollständig im Intervall $[d(v), f(v)]$ enthalten.

Fall 1b) $f(v) < d(u)$

$$\Rightarrow d(v) < f(v) < d(u) < f(u)$$

$\Rightarrow$ die Intervalle $[d(v), f(v)]$ und $[d(u), f(u)]$ sind disjunkt

$\Rightarrow$ (keiner der Knoten u bzw. v ist Abkömmling der anderen)

2. Fall $d(u) < d(v)$ analog zu Fall 1, nur dass u und v vertauscht werden müssen

5.4.2 Satz – Weißer-Pfad-Theorem

Ein Knoten v ist ein Abkömmling eines Knoten u im DFS-Wald eines Graphen G genau dann, wenn zum Zeitpunkt $d(u)$ der Knoten v von u aus auf einem Pfad mit ausschließlich weißen Knoten erreicht werden kann

Beweis Übungsserie!

Klassifikation der Kanten (in gerichteten Graphen)

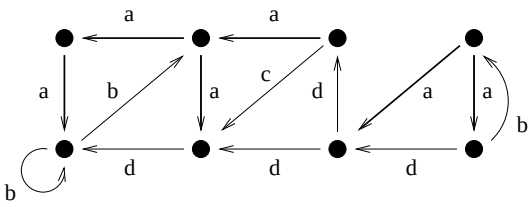
a) Baumkanten sind Kanten im DFS-Wald G_π (hervorgehobene Linien in Abbildung 73 auf Seite 97)

b) Rückkanten sind Kanten, die von einem Knoten u zu einem Vorfahren v von u verlaufen (Loops sind Rückkanten)

c) Vorwärtskanten sind Kanten, die keine Baumkanten sind, aber von einem Knoten u zu einem seiner Abkömmlinge v verläuft

d) Querkanten sind alle verbleibenden Kanten

- zwischen Knoten in ein und dem selben DFS-Baum, wobei kein Knoten Abkömmling des anderen ist
- zwischen Knoten in verschiedenen DFS-Bäumen



a) Baumkante
b) Rückkante
c) Vorkante
d) Querkante

Abbildung 73: Klassifikation der Kanten

Bemerkung Die Tiefensuche kann genutzt werden, um die Klassifizierung der Kanten mit Hilfe einer Färbung (weiß, grau, schwarz) vorzunehmen:

- Farbe einer Kante $(u, v) =_{\text{def}}$ Farbe des Knoten v, wenn die Kante (u, v) im DFS-Algo zum ersten Mal benutzt wird.
  - weiße Kanten = Baumkanten
  - graue Kanten = Rückkanten (denn Nachbarschaft von u noch nicht vollständig abgegrast, aber es gibt eine Kante zu v)
  - schwarze Kanten = Vorwärts- oder Querkanten
- In ungerichteten Graphen ($\{u, v\} \;\widehat{=}(u, v)$ und (v, u)) wird eine Kante dem ersten Farbtyp in der Liste (weiß, grau, schwarz) zugeordnet, der zutrifft (klassifiziert gemäß, welche der Kanten (u, v),(v, u) zuerst betreten wurde)

5.4.3 Satz – Kantenart bei Tiefensuche

Bei der Tiefensuche in einem *ungerichteten* Graphen ist jede Kante entweder Baum- oder Rückkante.

Beweis: Sei $\{u, v\} \in E$ o.B.d.A.: $d(u) < d(v)$

$\Rightarrow$ v wird entdeckt und vollständig bearbeitet, bevor u vollständig bearbeitet wird (u ist in dieser Zeit grau)

- Wird (u, v) zuerst bearbeitet, so ist v bis zu diesem Zeitpunkt unentdeckt (also weiß) $\Rightarrow \{u, v\}$ wird Baumkante
- Wird (v, u) zuerst bearbeitet, so ist u zu diesem Zeitpunkt grau und $\{u, v\}$ wird somit Rückkante

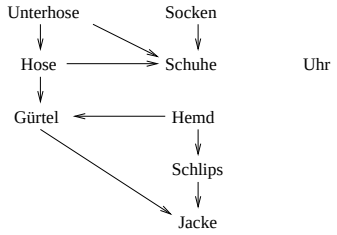
□

5.5 Topologisches Sortieren

Gegeben ist ein gerichteter, kreisfreier Graph $G = (V, E)$ $V = \{v_1, v_2, v_3, \ldots, v_n\}$. Wir suchen eine Permutation der Knoten $v_{i_1}, v_{i_2}, v_{i_3}, \ldots, v_{i_n}$. so dass für alle j, j' gilt:

$$(v_{i_j}, v_{i_{j'}}) \in E \rightarrow j < j'$$

Bemerkungen / Beispiel Siehe Abbildung 74 auf Seite 98.



Sortiert: Unterhose Socken Hose Uhr Schuhe Hemd Gürtel Schlips Jacke

Abbildung 74: Beispiel für Topologisches Sortieren

- Aufgabe: Knoten so umsortieren, dass alle Kanten in eine „Richtung“ (links nach rechts) zeigen.
- enthält Graph einen Kreis, so existiert keine topologische Sortierung (im Beispiel mit der Kleidung hieße dies, dass sich zwei Dinge gegenseitig voraussetzen)
- gerichteter kreisfreier Graph wird auch directed acyclic graph (= dag) genannt

G sei dag

```
TOPOLOGICAL SORT(G)
1   DFS(G)    [zur Berechnung der f-Werte]
2   sobald in 1 ein Knoten v vollständig bearbeitet wurde, \
       füge ihn an den Anfang einer verketteten Liste ein
3   gib verkettete Liste aus
```

An Beispiel: starten mit Gürtel:
entdecken:

1. Jacke
2. Gürtel

Suche weiter bei Socken

1. Schuhe
2. Socken

usw.
Wir schreiben die entdeckten Knoten so auf, dass der zuletzt entdeckte am weitesten links steht.

Laufzeit

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

5.5.1 Lemma

Ein gerichteter Graph G ist genau dann kreisfrei, wenn bei DFS(G) keine Rückkante entsteht.

Beweis des Lemmas

$\Rightarrow$ zu zeigen: wenn G kreisfrei, dann keine Rückkante
Sei G also kreisfrei.

Annahme: es gibt eine Rückkante $(u, v) \in E$.

$\Rightarrow$ u ist Abkömmling von v im DFS-Baum

$\Rightarrow$ es gibt einen Pfad $v \to u_1 \to u_2 \to \cdots \to u_k \to u$ in G

$\Rightarrow$ $v \to u_1 \to u_2 \to \cdots \to u_k \to u$ ist ein Kreis in G.

Widerspruch zu G kreisfrei!

$\Leftarrow$ zu zeigen: wenn keine Rückkante, dann kreisfrei. Genügt zu zeigen: Wenn Kreis, dann Rückkante (Kontraposition).

Sei C ein Kreis in G und sei v der erste Knoten aus C, der bei DFS(G) entdeckt wird. Sei (u, v) die in C in den Knoten einlaufende Kante.

Zum Zeitpunkt $d(v)$ bilden die Knoten von C einen weißen $v \rightarrow u$ Pfad.

$\Rightarrow$ u wird Abkömmling von v im DFS-Wald

$\Rightarrow$ (u, v) wird Rückkante

□

5.5.2 Satz – Korrektheit von TOPOLOGICAL SORT

TOPOLOGICAL SORT(G) erzeugt eine topologische Sortierung der Knoten eines dags G.

Beweis Es genügt zu zeigen: Für jedes Paar von Knoten u, v, $u \neq v$ mit $(u, v) \in E$ gilt: $f(v) < f(u)$

Sei $(u, v) \in E$ die gerade vom Algorithmus DFS(G) benutzte Kante.

1. Fall v ist grau

$\Rightarrow$ V ist Vorgänger von u

$\Rightarrow$ (u, v) wäre Rückkante $\rightarrow$ Widerspruch zu Lemma 5.5.1 und Kreisfreiheit von G

2. Fall v ist weiß

$\Rightarrow$ v ist Abkömmling von u

$\Rightarrow$ $f(v) < f(u)$

3. Fall v ist Schwarz

$\Rightarrow$ v wurde bereits vollständig bearbeitet, d.h. $f(v)$ ist bereits definiert

$\Rightarrow$ da wir (u, v) gerade „erforschen“ kann $f(u)$ noch nicht definiert sein

$\Rightarrow$ $f(v) < f(u)$

□

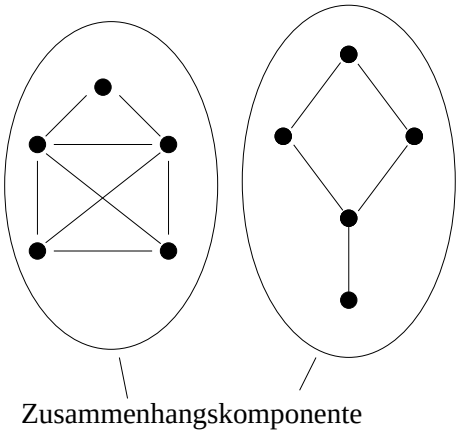


Abbildung 75: Zusammenhangskomponenten

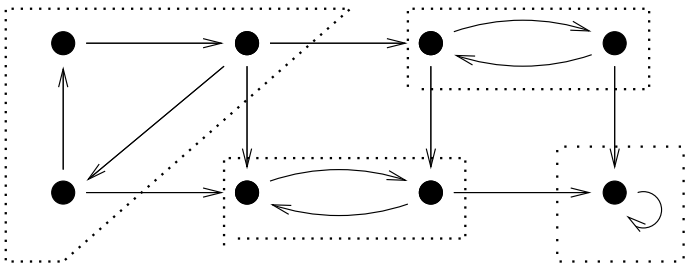
5.6 Stark zusammenhängende Komponenten

- ungerichtete Graphen:
  - Zusammenhangskomponente, siehe Abbildung 75 auf Seite 101.
  - Zusammenhangskomponenten $\widehat{=}$ maximal großer Teilgraph, so dass zwischen je zwei Knoten u und v ein $u-v$-Pfad existiert
  - 2-fach zusammenhängende Komponenten $\widehat{=}$ maximal großer Teilgraph, so dass mindestens zwei Kanten gelöscht werden müssen, damit der Teilgraph in zwei (oder mehr) Zusammenhangskomponenten zerfällt; zwischen je zwei Knoten gibt es mindestens zwei Kantendisjunkte Pfade
- gerichtete Graphen:
  - starke Zusammenhangskomponente $\widehat{=}$ maximale Knotenmenge $U \subseteq V$, so dass für je zwei Knoten $u, v \in U$ sowohl ein $u \to v$-Pfad, als auch ein $v \to u$-Pfad existiert; Beispiel in Abbildung 76 auf Seite 102.

Wie berechnet man die starken Zusammenhangskomponenten?

Idee:

- nutzen $G^T = (V, E^T)$ mit $E^T = (E^{-1}) = \{(v, u) : (u, v) \in E\}$ (gemäß Übungsserie kann G^T aus G in Zeit $\mathcal{O}(n + m)$ berechnet werden)
- G und G^T haben die gleiche stark zusammenhängenden Komponenten.



starke Zusammenhangskomponente

Abbildung 76: Starke Zusammenhangskomponente für gerichtete Graphen

STRONGLY-CONNECTED-COMPONENTS(G)

```
1   DFS(G) und dabei alle Knoten entsprechend ihrer f-Werte sortieren
2   Berechne G^T
3   DFS(G^T) aber besuche in Hauptschleife von DFS (Zeile 4) die Kno- \
        ten in der Reihenfolge der absteigenden f-Werte
4   Gib Knoten eines jeden DFS-Baumes im DFS-Wald von DFS(G^T) als \
        eigenständige stark zusammenhängende Komponente aus (Einzelknoten \
        bilden nur dann eine stark zusammenhängende Komponente, wenn sie \
        einen Loop haben)
```

Laufzeit

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

5.6.1 Lemma

Sind zwei Knoten u und v in der selben stark zusammenhängenden Komponente, so verlässt kein Pfad zwischen u und v diese Komponente. (Beweis in Übungsserie)

5.6.2 Satz – Zusammenhangskomponenten in DFS-Baum

Bei jeder Tiefensuche werden alle Knoten aus ein und derselben stark zusammenhängenden Komponente in ein und demselben DFS-Baum platziert. (Beweis in Übungsserie)

Bemerkung

- Für Korrektheit fehlt noch folgende Aussage:
  Bei DFS(G^T) (wie im Algo) werden Knoten aus verschiedenen stark zusammenhängenden Komponenten auch in verschiedenen DFS-Bäumen platziert.

- dazu benötigt man Hilfsbegriff: forefather $\phi(n)$ = derjenige Knoten w, für den ein $u \to w$ - Pfad existiert und dessen f-Wert maximal unter derartigen Knoten ist

$$\phi(u) = u \text{ möglich}$$

$$\phi(\phi(u)) = \phi(u)$$

- man kann zeigen:
  - in jeder stark zusammenhängenden Komponente gibt es einen Knoten, der forefather aller Knoten der stark zusammenhängenden Komponenten ist. Diesen kann man als Repräsentant der stark zusammenhängenden Komponente auffassen
  - in DFS(G) ist das der zuerst entdeckte und (damit auch) als letztes abgegraste Knoten der stark zusammenhängenden Komponenten
  - in DFS(G^T) ist dieser Knoten dann Wurzel des DFS-Baumes der stark zusammenhängenden Komponente
  - $\phi(u)$ ist stets Vorfahre von u (u ist Abkömmling von $\phi(u)$) in jedem DFS-Durchlauf
  - u, v liegen genau dann in der selben stark zusammenhängenden Komponente, wenn $\phi(u) = \phi(v)$ gilt

5.6.3 Satz – Korrektheit von Algo SCC

STRONGLY-CONNECTED-COMPONENTS(G) berechnet die stark zusammenhängenden Komponenten von G (ohne Beweis).

6 Greedy-Algorithmen

- greed(engl) – Gier
- greedy – gierig

TSP – Traveling Salesman Problem

gegeben:

- gerichteter Graph $G = (V, E)$
- $d : E \to \mathbb{N}$

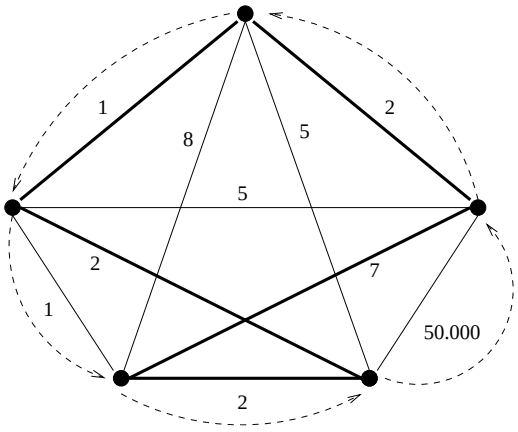


Abbildung 77: Beispiellauf für Greedy-Algorithmus im K_5

gesucht:

- Hamiltonkreis mit möglichst geringem Gewicht

$$\sum_{(u,v)\in K_5} d(u,v) \to A_{\min}$$

siehe Abbildung 77 auf Seite 104.

Problem TSP ist *nicht* effizient lösbar $\sim \mathcal{O}(n^2 * 2^n)$

6.1 Minimum Spanning Tree

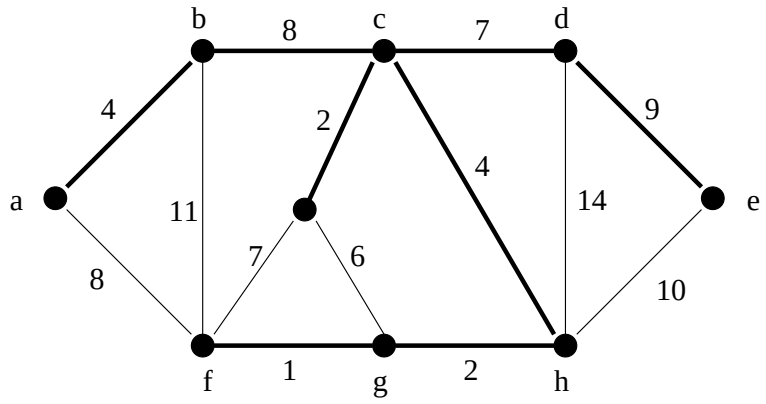
Spannbaum (Spanningtree) Teilgraph $T = (V', E')$ heißt Spannbaum von $G = (V, E)$ gdw. $V' = V$ und T ist ein Baum (ungerichtete Graphen). D.h. alle Knoten sind noch verbunden, aber es gibt nur noch minimal viele Kanten.

MIN SPANNING TREE

- gegeben:
  - ungerichteter Graph $G = (V, E)$
  - $d : E \to \mathbb{N}$
- gesucht:
  - Spannbaum T von G mit minimalem Gewicht

Suchen Spannbaum

für Graph in Abbildung 78 auf Seite 105.



— Linien, die zum Spannbaum gehören

Abbildung 78: Spannbaum durch Tiefensuche + Greed

1. Versuch: Tiefensuche + Greed (d.h. erzeuge einen Tiefensuchbaum, wenn Wahl zwischen 2 Kanten, die als nächstes abgelaufen werden können nimm die mit dem geringsten Gewicht).

Beispiel: Start in h, DFS-Baum hat Gewicht von 37

Gibt es untere Schranken für das Gewicht eines minimalen Spannbaumes?
Summe der Gewichte der 8 leichtesten Kanten $\leq$ Gewicht min. Spannbaum ≤ 37. Hier: $1+2+2+4+4+6+7+7=33$

Klappt zufällig hier; i.A. aber nicht! Siehe auch 13. Übungsserie.

Kenntnis über Minimum und Maximum (das bis dahin gefunden ist) ist bei so genannten branch-and-bound-Algorithmen von Nutzen. Hierbei wird versucht alle Strukturen durchzuprobieren und diejenigen, die nicht als Lösung in Frage kommen schnell auszuschließen.

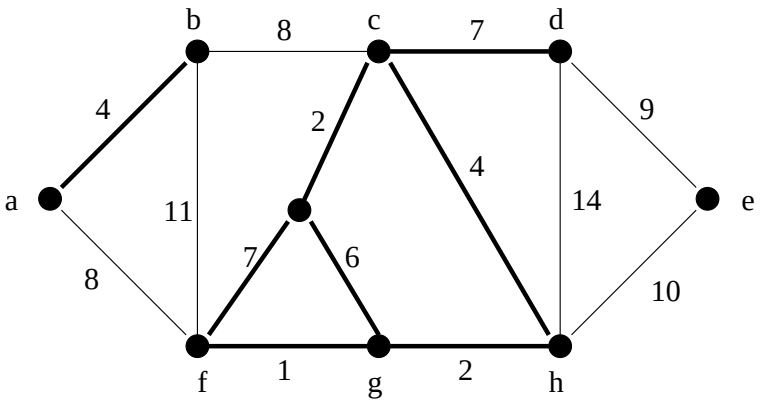
2. Versuch (naiv)

1. Wählen die $n-1$ leichtesten Kanten (Sortiere Kantengewichte mit z.B. Merge-Sort). Siehe Abbildung 79 auf Seite 106
2. Test, ob generierter Teilgraph ein Baum ist (Breitensuche)

Falls kein Baum:

- Fügen bisher nicht im Teilgraph enthaltenen Knoten durch Hinzunahme der billigsten Kanten hinzu
- in jedem Kreis die teuerste Kante entfernen

- Fügen Zusammenhangskomponenten zusammen (mit billigen Kanten)



— Zwischenergebnis von Algorithmus 2

Abbildung 79: Algorithmus 2

Prinzipielle Konstruktion eines Spannbaumes

Sei $G = (V, E)$ ein ungerichteter zusammenhängender gewichteter Graph. Die Gewichtsfunktion sei $w : E \to \mathbb{R}_+$. Wir wollen den minimalen Spannbaum Kante für Kante aufbauen. Unser Algorithmus soll dabei eine Kantenmenge A verwalten, für die folgende Eigenschaft gilt:

> (*) (vor jeder Iteration): A ist Teilmenge der Kantenmenge eines minimalen Spannbaums (MST).

In jedem Schritt wird eine Kante $\{u, v\}$ bestimmt, die zu A hinzugefügt werden kann, ohne (*) zu verletzen, d.h. so wie A ist auch $A \cup \{u, v\}$ Teilmenge der Kantenmenge eines MST. Eine derartige Kante nennen wir *sichere Kante*.

Siehe Abbildung 80 auf Seite 107.

GENERIC-MST(G,w)

```
1   A := ∅
2   while A  bildet noch keinen Spannbaum do
3       finde eine Kante u,v, die bzgl. A sicher ist
4       A := A ∪ {{u, v}}
5   return A
```

Wie prüft man, ob A bereits einen Spannbaum bildet

- sichere Kanten $\to$ A ist kreisfrei (*)
- Hat A $n - 1$ Kanten, so muss A zusammenhängend sein.

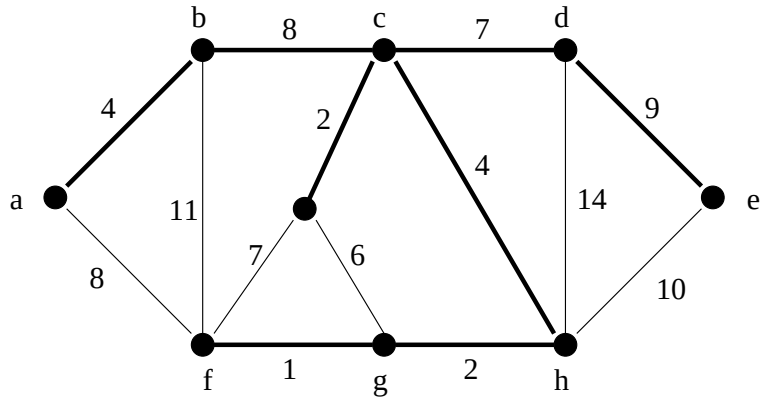


Abbildung 80: Prinzipielle Konstruktion eines Spannbaums

Wie finde ich eine sichere Kante

- Existenz einer sicheren Kante ist klar
- Dazu ein paar neue Begriffe

6.1.1 Definitionen im zusammenhängenden ungerichteten Graphen

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph.

1. Ein Schnitt (cut) von G, $(S, V \backslash S)$, ist eine Zerlegung von V.
2. Eine Kante $\{u, v\}$ überquert / kreuzt den Schnitt $(S, V \backslash S)$ genau dann, wenn $(u \in S \wedge v \in V \backslash S) \vee (v \in S \wedge u \in V \backslash S)$
3. Eine Menge von Kanten A respektiert den Schnitt $(S, V \backslash S)$ genau dann, wenn keine Kante aus A den Schnitt kreuzt
4. Eine Kante $\{u, v\}$ heißt leichte Kante eines Schnittes $(S, V \backslash S)$ genau dann, wenn

   $$w(\{u, v\}) = \min \{w(e) : e \text{ kreuzt den Schnitt } (S, V \backslash S)\}$$

   wobei w das Gewicht der Kante angibt.

6.1.2 Satz – Korrektheit der Spannbaumalgorithmen

Sei $G = (V, E)$ ein zusammenhängender ungerichteter Graph mit Gewichtsfunktion w. Sei $A \subseteq E$ ein Teilmenge der Kantenmengen eines MST von G. Sei $(S, V \backslash S)$ ein Schnitt, der von A respektiert wird und sei $\{u, v\}$ eine leichte Kante des Schnittes. Dann ist $\{u, v\}$ eine sichere Kante für A.

Beweis Sei $T = (V, E)$ ein MST, so dass $A \subseteq E$.

1. Fall $\{u, v\} \in E$
$\Rightarrow A \cup \{\{u, v\}\} \subseteq E$, d.h. $\{u, v\}$ ist sicher für A

2. Fall $\{u, v\} \notin E$
Da T ein Baum ist (d.h. zusammenhängend und kreisfrei), gibt es in T einen Pfad von u nach v. Da u und v in verschiedenen Teilen des Schnitts liegen, muss dieser Pfad eine Kante $\{x, y\}$ enthalten, die den Schnitt kreuzt. Offenbar $\{x, y\} \notin A$, da A den Schnitt $(S, V \backslash S)$ respektiert.

Entfernt man $\{x, y\}$ aus T und fügt $\{u, v\}$ hinzu, so erhält man einen anderen Spannbaum T'. Wir zeigen nun, dass T' auch ein MST ist.

$T' = (V, E')$ mit $E' = ((E \backslash \{\{x, y\}\}) \cup \{\{u, v\}\}$.

Wissen

- $w(\{u, v\} \leq w(\{x, y\})$ Da $\{u, v\}$ eine leichte Kante bezüglich des Schnittes ist.
- $w(T') = w(T) \underbrace{-w(\{x, y\}) + w(\{u, v\})}_{\leq 0} \leq w(T)$
  Da aber T ein MST war, gilt sogar $w(T') = w(T)$ und somit ist auch T' ein MST.
  Damit ist auch $A \cup \{u, v\} \subseteq E'$ und da T' ein MST ist, ist $\{u, v\}$ sicher für A.

6.2 Die Algorithmen von Prim und Kruskal

Grundlegende Fragestellungen

- Wie bestimmt man eine sichere Kante?
- Wie bestimmt man eine leichte Kante, die einen Schnitt kreuzt?
- Wie sieht der Schnitt $(S, V \backslash S)$ aus?

Wir behandeln zwei Varianten:

- Prim siehe Abbildung 81 auf Seite 109. Vorgehen:
  - Baum wachsen lassen durch Hinzunehmen billiger Kanten
  - Der Schnitt liegt immer um die bereits gefundenen Knoten herum.
- Kruskal siehe Abbildung 82 auf Seite 109. Vorgehen:
  - Billigste Kanten hinzunehmen und Kreisfreiheit erhalten.

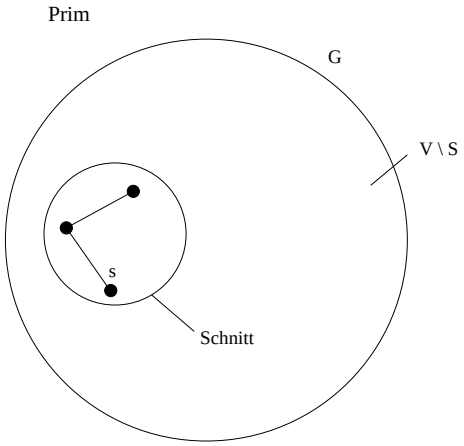


Abbildung 81: Vorgehensweise des Algorithmus von Prim

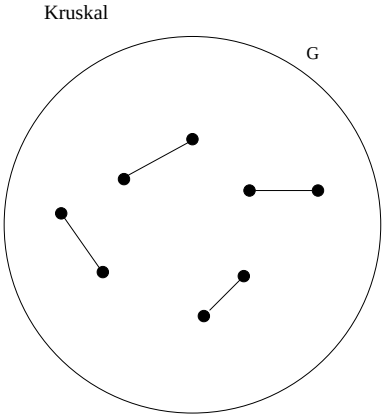


Abbildung 82: Vorgehensweise des Algorithmus von Kruskal

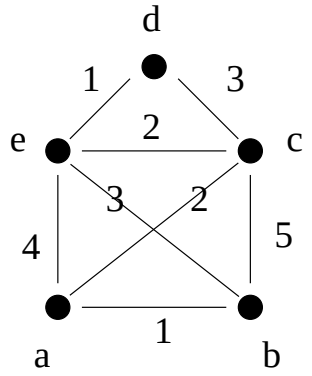
a) Der Algorithmus von Prim

```
MST-PRIM(G,w,r) // wobei r der Startknoten
1   Q := V      // Q ⇐ alle Knoten
2   for each v ∈ Q do key(v)=∞
3   key(r) := 0
4   π(r) := NIL
5   while Q ≠ ∅ do
6         u := EXTRACT-MIN(Q) // Minimum bzgl. key
7         for each v ∈ AL_G(u) do \
                  if v ∈ Q and w({u,v}) < key(v) then
                      π(v) := u
8                     key(v) := w(u,v)
```

Bemerkungen

- $V \backslash Q$ ist Menge der Knoten im wachsenden Baum. Q ist die Menge der Knoten, die noch nicht im Baum sind.
- key(v) – kleinstes Kantengewicht einer Kante, die von einem Knoten v zu einem Knoten in $V \backslash Q$ verläuft $\widehat{=}$ Anfügekosten des Knoten v an den Baum.
- Q (zwei Operationen EXTRACT-MIN, DECREASE-KEY) $\Rightarrow$ Prioritätsschlange z.B. MIN-HEAP. Die Laufzeit beträgt für EXTRACT-MIN $\mathcal{O}(1)$ und für DECREASE-KEY $\mathcal{O}(\log n)$.

Beispiel siehe Abbildung 83 auf Seite 110.



Startknoten: e

Abbildung 83: Beispiel

	a	b	c	d	e
key	∞	∞	∞	∞	∞
π					NIL

$\Downarrow$

	a	b	c	d	e
key	4	∞	∞	∞	∞
π	e				NIL

$\Downarrow$

	a	b	c	d	e
key	4	3	2	1	0
π	e	e	e	e	NIL

$\Downarrow$

	a	b	c	d	e
key	2	3	2	1	0
π	c	e	e	e	NIL

$\Downarrow$

	a	b	c	d	e
key	2	1	2	1	0
π	c	a	e	e	NIL

Die π-Werte definieren den MST

Laufzeit

- Initialisierung (Zeilen 1-4): $\mathcal{O}(n)$
- n mal EXTRACT-MIN: $n * \mathcal{O}(\log n)$ (mit MIN-HEAP) also $\mathcal{O}(n)$
- m mal DECREASE-KEY: $m * \mathcal{O}(\log n)$ also $\mathcal{O}(m \log n)$

Der Algorithmus von Kruskal

```
MST-KRUSKAL(G,w)
1   A := ∅
2   Sortiere die Kanten in E mit aufsteigenden Gewichten \
        e_1, e_2, ..., e_m für die gilt w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m)
3   for i = 1 to m do
4           if für die Kante e_i = {u,v} die Knoten u und v in verschiedenen \
                Zusammenhangskomponenten bezüglich der Kantenmenge A liegen
5              then füge e_i zu A hinzu
6   return A
```

Bemerkungen

- A ist Kantenmenge des MST
- Laufzeit (ohne Zeilen 3-5): $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$
- Laufzeit von Zeilen 3-5: $\mathcal{O}(m * n)$

Nachtrag Breitensuche

- Können BFS ja statt auf G auf A (Menge der Kanten im Spannbaum) laufen lassen

A wächst, untere Schranke für den Zeitbedarf ist $\sum_{i=1}^{n} \mathcal{O}(i) \in \mathcal{O}(n^2)$.

4. Versuch mit UNION-FIND-Datenstruktur

- erlaubt effiziente Handhabung disjunkter Mengen (z.B. Zusammenhangskomponenten)
- unterstützt drei Operationen:
  - MAKEUNIONFIND – für eine Menge S wird Datenstruktur initialisiert, so dass jedes Element von S in einer eigenen Menge ist (Zielzeit: $\mathcal{O}(n)$, $n = \text{card}\ (S)$)
  - FIND(u) – liefert für ein $u \in S$ den Namen der Menge, zu der u gehört (Zielzeit: $\mathcal{O}(\log n)$)
  - UNION(A, B) – Für Mengen A, B in Datenstruktur, wird diese so modifiziert, dass in ihr A und B vereinigt werden (Zielzeit: $\mathcal{O}(\log n)$?)

unter Einsatz dieser Datenstruktur ergibt sich für MST-KRUSKAL:

Laufzeit	Schritt
$\mathcal{O}(m \log n)$	Sortieren
$\mathcal{O}(n)$	Initialisierung von UNION-FIND
$\mathcal{O}(m \log n)$	FIND(u) $\neq$ FIND(v) Abfrage
$\mathcal{O}(n \log n)$	+ UNION-OPERATIONEN
$\Downarrow$	
$\mathcal{O}(m \log n)$	Gesamtaufwand

6.3 UNION-FIND

Grundmenge $S = \{1, \ldots, n\}$

Operationen

- MAKEUNIONFIND
- FIND
- UNION

Variante 1

Idee:

- Array COMPONENT der Größe n
- COMPONENT[i] = Name der Menge zu der $i \in S$ gehört

```
MAKEUNIONFIND(S)
1   für alle s ∈ S setze COMPONENT(s)  = s
```

Laufzeit $\mathcal{O}(n)$

```
FIND(u)
1   return COMPONENT(u)
```

Laufzeit $\mathcal{O}(1)$

```
UNION(A, B)
1   für alle a ∈ A setze COMPONENT(a) := B
```

Laufzeit $\mathcal{O}(n)$ (denn man muss das gesamte Array durchlaufen um alle Elemente von A zu finden)

Verbesserung der Laufzeit

- führen Liste aller Elemente einer Menge
- Namen für $A \cup B$ so wählen, dass für möglichst wenige Elemente der COMPONENT-EINTRAG geändert werden muss (führe dazu einen Parameter `size` der Liste, die die Zahl der Elemente einer Menge speichert, mit)

Laufzeiten: Im worst case: UNION benötigt immer noch $\mathcal{O}(n)$ – wenn card (A) = card $(B) = \frac{n}{2}$

Aber: Das kann nicht allzuoft passieren!

Behauptung: In der Array-Implementierung der UNION-FIND-Datenstruktur einer Grundmenge S mit n Elementen, in der die UNION-Operationen den Namen der größeren der beteiligten Mengen übernimmt benötigen

1. die MAKEUNIONFIND-Operation $\mathcal{O}(n)$
2. eine FIND-Operation $\mathcal{O}(1)$
3. eine Folge von k UNION-Operationen $\mathcal{O}(k \log k)$ (ausgehend von Startsituation)

Zeit.

Begründung 1. und 2. klar.
zu 3.:

- Kosten werden durch den notwendigen Update verursacht

- statt Kosten einer einzelnen UNION-Operation zu beschränken, beschränken wir die Kosten um COMPONENT(v), $v \in S$, zu aktualisieren im Verlauf der k UNION-Operationen
- die k UNION-Operationen beeinflussen höchstens $2k$ der n original einelementigen Mengen

$\Rightarrow$ höchstens $2k$ Elemente von S werden durch die k-UNION-Operationen berührt.

Sei $v \in S$. Immer wenn COMPONENT(v) aktualisiert wird, verdoppelt sich auch die Größe der Menge, zu der v gehört (da immer die kleinere Menge aktualisiert wird). Am Anfang ist die Menge zu der v gehört nur 1 groß, d.h. am Ende höchstens $2k$.

$\Rightarrow$ COMPONENT(v) wird höchstens $\log_2(2k)$ mal aktualisiert. Da aber höchstens $2k$-Elemente überhaupt berührt werden, müssen von höchstens $2k$-Elementen die COMPONENT-Werte aktualisiert werden.

$$\Rightarrow \mathcal{O}(k \log k) \text{ Aktualisierungen}$$

□

Bemerkung

- genügt für $\mathcal{O}(m \log n)$-Laufzeit von MST-KRUSKAL:

$\mathcal{O}(m)$	m mal je 2 FIND-Abfragen
$n-1$	UNION-Operationen

$\Rightarrow$ Laufzeit MST-KRUSKAL: $\mathcal{O}(m \log n)$

- ABER: UNION-FIND geht noch besser!

Variante 2

Idee:

- mit Zeigern
- jedes Element von S erhält Zeiger auf die Menge, zu der das Element gehört
- Mengennamen werden wieder Elemente von S sein (Repräsentanten)

```
MAKEUNIONFIND
1   für alle s ∈ S do Zeiger auf sich selbst
```

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

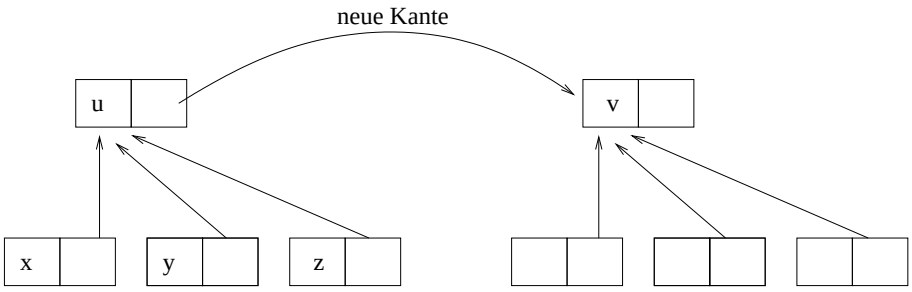


Abbildung 84: Vorgehen von UNION

`UNION(`A, B`)` `// gleich UNION(`u, v`), wobei` $u \in A$, $v \in B$
`Nutzen für` $A \cup B$ u `oder` v `als neuen Namen` (Vereinbarung: immer der Name der zweiten Menge wählen), 'd.h. setzen dem u sein Zeiger'auf v

Siehe Abbildung 84 auf Seite 115.

- UNION kostet $\mathcal{O}(1)$
- FIND-Kosten steigen, muss für FIND(x) eine Kette von Zeigern verfolgen, um den Namen der Menge zu erfahren, zu der $x \in S$ gehört.

Aber: diese Zeigerketten können nicht allzulang sein, wenn Zeiger stets auf die größere der beiden an UNION beteiligten Mengen gesetzt wird.

Behauptung In der Zeiger-Implementierung der UNION-FIND-Datenstruktur einer Grundmenge S mit n Elementen, in der bei jeder UNION-Operation der Name der größeren Menge übernommen wird, benötigen

1. die MAKEUNIONFIND-Operation $\mathcal{O}(n)$
2. eine FIND-Operation $\mathcal{O}(\log n)$
3. eine UNION-Operation $\mathcal{O}(1)$

Zeit.

Begründung 1. und 3. klar.
zu 2.: Zeit für FIND(v) = Anzahl der Namensänderungen der Menge , zu der v gehört
Da bei UNION der Name der größeren Menge übernommen wird, geht eine Namensänderung der Menge, die v enthält einher mit einer (mindestens) Verdoppelung der Größe dieser Menge. Die Menge hat anfänglich die Größe 1 und kann die Größe n nicht überschreiten. $\Rightarrow$ höchsten $\log n$ Namensänderungen

□

Bemerkung

- auch mit dieser Implementierung läuft MST-KRUSKAL in $\mathcal{O}(m \log n)$

3. Versuch Unbefriedigend bei Zeigerimplementierung ist folgende Situation: langer Weg in Abbildung 85 auf Seite 116.

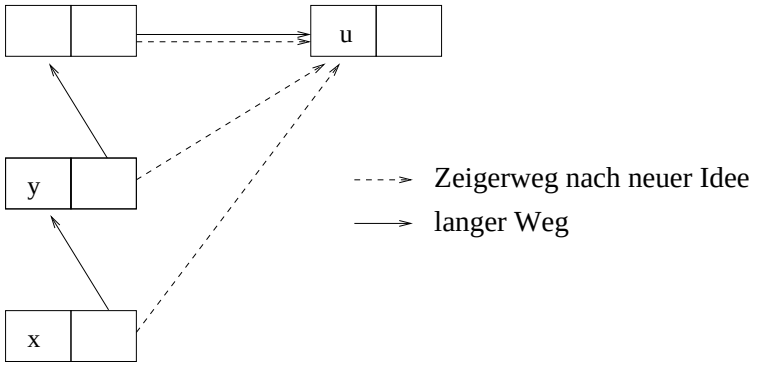


Abbildung 85: Verschiedene Zeigerimplementierungen

- FIND(x) kostet im worst case $\mathcal{O}(\log n)$
- FIND(y) „alles muss noch einmal abgelatscht werden“

Besser: nach FIND(x), den Pfad von x zu dem Element u, welches die Menge repräsentiert, zu der x gehört, noch einmal ablaufen und alle Zeiger auf u umsetzen. Beim nächsten Aufruf liegt die Laufzeit in $\mathcal{O}(1)$. Dies nennt man Pfadkompression.

Vorteil

- nachfolgende FIND-Operationen werden wesentlich beschleunigt

Nachteil

- Zusätzliche Kosten durch Pfadkompression

Die erhöhten Kosten, die durch die Pfadkompression entstehen amortisieren sich im Laufe der nachfolgenden FIND-Operationen. Man kann zeigen, dass bei Pfadkompression für eine jede Folge von n FIND-Operationen nicht mehr als $\mathcal{O}(n * \alpha(n))$ Zeit benötigt wird.

Bemerkungen

- α ist inverse Ackermann-Funktion
- α wächst sehr langsam, $\alpha(n) \leq 4$ für alle praktischen Werte von n
- Definition von A:

$$A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$A(0, y) =_{\text{def}} y + 1$$

$$A(x + 1, 0) =_{\text{def}} A(x, 1)$$

$$A(x + 1, y + 1) =_{\text{def}} A(x, A(x + 1, y))$$

Definition

$$\alpha(n) =_{\text{def}} \min\{i : A(i, 1) > \log n\}$$

- $\alpha(n) \leq 3$ für alle $n < 2^{13}$
- $\alpha(n) \leq 4$ für alle $n < 2^{65533}$

7 Dynamische Programmierung

- Prinzipielle Herangehensweise zur Lösung eines algorithmischen Problems:

divide-and-conquer	dynamische Programmierung
Ausgangsproblem in unabhängige Teilprobleme zerlegen, diese lösen und Gesamtlösung erstellen	Ausgangsproblem in abhängige Teilprobleme zerlegen (Teilprobleme haben gemeinsame Unterprobleme. Jedes Unterproblem lösen und Ergebnisse in Tabelle speichern (table-look-up))

- dient zur Lösung von Optimierungsproblemen. Suche nach *einer* optimalen Lösung
- Herangehen in Phasen:
  1. Charakterisierung der optimalen Lösung
  2. Wert einer optimalen Lösung rekursiv definieren
  3. Wert einer optimalen Lösung berechnen – bottom up (von unten nach oben)
  4. Eine optimale Lösung aus der berechneten Information rekonstruieren

7.1 Fließbandplanung

Haben Autofabrik, zwei Fließbandfertigungslinien

- Fahrgestell, Rahmen, Bodengruppe am Anfang
- fertiges Auto steht am Ende
- n Fertigungsstationen pro Linie ($S_{1,j}$, $S_{2,j}$, $j = 1, \ldots, n$) mit Fertigungszeiten $a_{i,j}$, $i \in \{1, 2\}, j = 1, \ldots, n$ (zum Beispiel: $a_{1,j} \neq a_{2,j}$ für alle j)
- Eintrittszeiten e_1, e_2, Austrittszeiten x_1, x_2
- im Normalfall arbeiten die Fertigungslinien unabhängig
- die Übergangszeiten von $S_{i,j}$ zu $S_{i,j+1}$ sind vernachlässigbar
- bei Eilaufträgen ist es vielleicht günstiger, die Fertigung zwischen den Linien hin und her springen zu lassen
- $t_{1,j}$: Zeit für Wechsel von $S_{1,j}$ zu $S_{2,j+1}$
- $t_{2,j}$: Zeit für Wechsel von $S_{2,j}$ zu $S_{1,j+1}$

Siehe Abbildung 86 auf Seite 118.

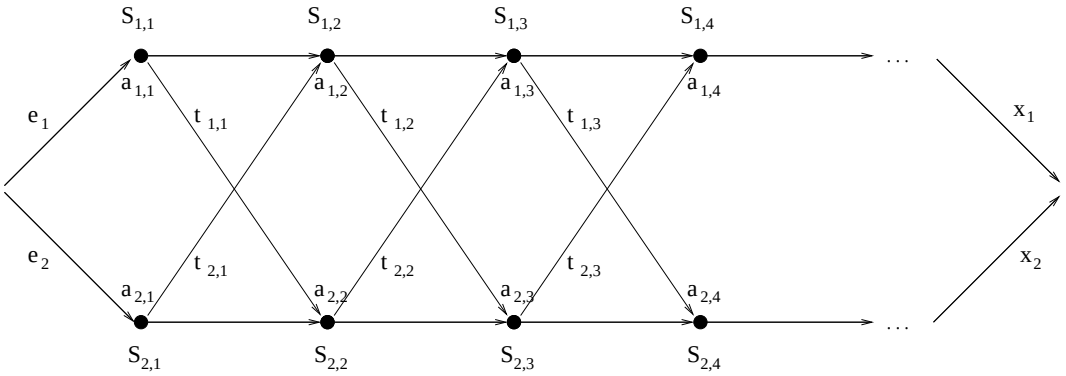


Abbildung 86: Fließbandfertigungslinien

Wie produziert man ein Auto am schnellsten?

Naiver Algorithmus

- alle Varianten durchmustern, 2^n viele Varianten

Dynamische Programmierung

Phase 1: Charakterisierung einer optimalen Lösung schnellste Fertigungszeit = schnellste Art e_i, $S_{i',j}$ und $x_{i''}$ zu durchlaufen.

Definieren:

$f_i(j) =_{\text{def}}$ kürzeste Zeit, die Werkstück vom Anfang, bis zum Durchlauf der Station $S_{i,j}$ (inklusive) benötigt.

Beispiel:

$$f_1(1) = e_1 + a_{1,1}$$
$$f_1(2) = \min\{\underbrace{e_1 + a_{1,1}}_{f_1(1)} + a_{1,2}, \underbrace{e_2 + a_{2,1}}_{f_2(1)} + t_{2,1} + a_{1,2}\}$$
$$f_2(2) = \min\{\underbrace{e_1 + a_{1,1}}_{f_1(1)} + t_{1,2} + a_{2,2}, \underbrace{e_2 + a_{2,1}}_{f_2(1)} + a_{2,2}\}$$

usw.

Allgemein

- $f_1(1), f_2(1)$ sind einfach abzulesen
- für $f_i(j),\ (j \geq 2)$ ergibt sich folgendes Bild:
  $f_1(j)$:
  - von $S_{1,j-1}$ direkt zu $S_{1,j}$
  - von $S_{2,j-1}$ zu $S_{1,j}$

$$f_1(j) = \min\{f_1(j-1) + a_{1,j}, f_2(j-1) + t_{2,j-1} + a_{1,j}\}$$

Fazit schnellster Weg durch $S_{i,j}$ enthält als Teilprobleme die schnellsten Wege durch $S_{1,j-1}$ und $S_{2,j-1}$.

Lösungswert

$$\text{opt} = \min\{f_1(n) + x_1, f_2(n) + x_2\}$$

Phase 2 Rekursive Definition der Lösung

$$\text{opt} = \min\{f_1(n) + x_1, f_2(n) + x_2\}$$

mit

$$f_1(j) = \begin{cases} e_1 + a_{1,1} & \text{falls } j = 1 \\ \min\{f_1(j-1) + a_{1,j}, f_2(j-1) + t_{2,j-1} + a_{1,j}\} & \text{falls } j = 2 \end{cases}$$

analog $f_2(j)$.

Um einen optimalen Weg rekonstruieren zu können, definiert man: $l_i(j)$ = Fertigungsname (1 oder 2), dessen Station $j-1$ in einem schnellsten Weg durch $S_{i,j}$ genutzt wurde, für $j \geq 2$.

Phase 3: Berechnung des optimalen Wertes

- können die Werte $f_i(j)$ rekursiv berechnen. Frage, ist das empfehlenswert?

Nein: Sei $r_i(j)$ die Zahl der Aufrufe von $f_i(j)$. Es gilt:

$$r_1(n) = r_2(n) = 1$$

$$r_1(j) = r_1(j+1) + r_2(j+1) = r_2(j)$$

	1	...	$n-3$	$n-2$	$n-1$	n
$r_1(j)$	2^{n-1}		8	4	2	1
$r_2(j)$	2^{n-1}		8	4	2	1

$\Rightarrow$ Top-down Berechnung ist zu zeitaufwändig ($\mathcal{O}(2^n)$). Bottom-up funktioniert besser, die $f_i(j)$ der Reihe nach wachsendem j berechnen. $\Rightarrow$ $\Theta(n)$

Eingaben:

$$a = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \end{pmatrix}$$

$$e = (e_1, e_2)$$

$$t = \begin{pmatrix} t_{11} & t_{12} & t_{13} & \dots & t_{1n} \\ t_{21} & t_{22} & t_{23} & \dots & t_{2n} \end{pmatrix}$$

$$x = (x_1, x_2)$$

```
FASTESTWAY(a, t, e, x, n)
1   f_1(1) := e_1 + a_11
2   f_2(1) := e_2 + a_21
3   for j = 2 to n do
        // Berechnung von f_1(j)
4       if f_1(j-1) + a_1j ≤ f_2(j-1) + t_2,j-1 + a_1j then
            f_1(j) := f_1(j-1) + a_1j
5           l_1(j) := 1
6       else f_1(j) := f_2(j-1) + t_2,j-1 + a_1j
7            l_1(j) := 2
        // Berechnung von f_2(j)=
8       if f_2(j-1) + a_2j ≤ f_1(j-1) + t_1,j-1 + a_2j then ...
            // Berechnung von f_2(j) analog zu der von f_1(j)
9
10
11
12  if f_1(n) + x_1 ≤ f_2(n) + x_2 then f* := f_1(n) + x_1
13      l* := 1
14  else f* := f_2(n) + x_2
15      l* := 2
```

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

Bemerkungen

- jede der Zeilen 1-2, 12-15 benötigt Zeit in $\mathcal{O}(1)$
- $n-1$ Durchläufe der for-Schleife, je $\mathcal{O}(1)$

Phase 4 Rekonstruktion einer optimalen Lösung

$$\text{Eingabe} : \begin{pmatrix} l_{11} & l_{12} & l_{13} & \dots & l_{1n} \\ l_{21} & l_{22} & l_{23} & \dots & l_{2n} \end{pmatrix}$$

```
  PRINTSTATIONS(l, n)
1   i := l*
2   print "S" i "g" n      (print S_{i,n})
3   for j = n downto 2 do i := l_i(j)
4       print "S" i "g" j − 1
```

7.2 Longest Common Subsequence

- Problemstellung aus Biologie (Genetik): DNA-Stränge vergleichen.
- DNA-Strang = String über {A,C,G,T}

Wie ähnlich sind zwei DNA-Stränge?

Variante 1 Hamming Distanz

A	C	C	T	T	G
A	C	**G**	T	T	**A**

Variante 2: Alignment Score

Variante 3: Längste gemeinsame Teilsequenz (unzusammenhängend). Im obigen Beispiel wäre dies A C T T.

Beispiel
S_1 = `ACCGGTCGAGTGCGCGGAAGCCGGCCGA`
S_2 = `GTCGTTCGGAATGCCGTTGCTCTGTAAA`

Variante 1 18 w.o.G

Variante 2
`GGTCG-A`
`GTTCGGA`
interessantes algorithmisches Problem!

Variante 3 Zu langem Beispiel oben: 16 ist Länge der Teilsequenz der gemeinsamen Basen.

16 ist jedoch nicht optimal.

Ist längste gemeinsame Teilsequenz immer eindeutig bestimmt? Nein: Beispiel

```
AG
GA
```

7.2.1 Definition Teilwort

Sei $x = x_1 x_2 \dots x_n$ ein Wort über dem Alphabet Σ, $x_i \in \Sigma$.

Ein Wort $z = z_1 \dots z_k$ heißt Teilwort von x genau dann, wenn es eine Folge $(i_1, i_2, \dots i_k)$ von Indizes gibt, für die gilt: $i_1, i_2, \dots, i_k \in \{1, \dots, n\}$, so dass gilt

1. $i_1 < i_2 < \dots < i_k$
2. Für alle $j \in \{1, \dots, k\}$ gilt $x_{i_j} = z_j$

Beispiel

$$\Sigma = \{A, C, G, T\}$$

GCT ist Teilwort von `AGGTCAAT`

7.2.2 Definition gemeinsames Teilwort

Ein Wort z heißt gemeinsames Teilwort zweier Wörter x und y genau dann, wenn z sowohl Teilwort von x, als auch Teilwort von y ist.

LONGEST COMMON SUBSEQUENCE PROBLEM (LCS)

Eingabe: Zwei Wörter x, y über ein Alphabet Σ.

Gesucht: gemeinsames Teilwort von x und y mit maximum Länge

Wie können wir das Problem LCS algorithmisch lösen?

1. Versuch: Erzeuge der Reihe nach alle Teilworte w von x und überprüfen ob w auch ein Teilwort von y ist. Laufzeit: $2^{|x|} = 2^n$

2. Versuch: Dynamische Programmierung

$$x = x_1\, x_2\, x_3\, \dots\, x_{m-1}\, x_m$$

$$y = y_1\, y_2\, y_3\, \cdots\, y_{m-1}\, y_m\, y_{m+1}\, \cdots\, y_{n-1}\, y_n$$

Sei z ein gemeinsames Teilwort von x und y mit maximaler Länge, d.h. z ist ein LCS von x und y.

$$z = z_1\, z_2\, z_3\, \ldots\, z_{k-1}\, z_k$$

Offenbar gilt:

1. Ist $x_m = y_n$ so gilt $z_k = x_m = y_n$ und $z_1 \ldots z_{k-1}$ ist LCS von $x_1 \ldots x_{m-1}$ und $y_1 \ldots y_{n-1}$.

   Begründung ($z_k = x_m = y_n$): ist $z_k \neq x_m$, so wäre $z\, x_m$ auch ein gemeinsames Teilwort von x und y

   $$|z\, x_m| > |z|$$

   Widerspruch!

   Begründung, dass $z_1 \ldots z_{k-1}$ LCS ist: Offenbar ist $z_1 \ldots z_{k-1}$ ein Teilwort von $x_1 \ldots x_{m-1}$ und $y_1 \ldots y_{n-1}$. $|z_1 \ldots z_{k-1}| = k-1$

   Annahme: es gibt LCS von $x_1 \ldots x_{m-1}$ und $y_1 \ldots y_{n-1}$ namens foo mit $|\text{foo}| > k-1$. $\Rightarrow$ $\text{foo}\, x_m$ ist Teilwort von x und y und $|\text{foo}\, x_m| > |z|$
   Widerspruch!

2. Ist $x_m \neq y_n$ dann folgt aus $z_k \neq x_m$ dass z in LCS von $x_1 \ldots x_{m-1}$ und y ist.

3. Ist $x_m \neq y_n$ dann folgt aus $z_k \neq y_n$ dass z ein LCS von x und $y_1 \ldots y_{n-1}$ ist. Begründung elementar!

Phase 1 im Prinzip beendet, Phase 2 kann beginnen. Schematische Darstellung in Abbildung 87 auf Seite 126.

Definieren $c(i,j)$= Länge eines lcs von $x_1 \ldots x_i$ und $y_1 \ldots y_j$. Damit gilt dann gemäß unseren bisherigen Beobachtungen:

$$c(i,j) = \begin{cases} 0 & \text{falls } i = 0 \text{ oder } j = 0 \\ c(i-1, j-1) + 1 & \text{falls } i,j > 0 \text{ und } x_i = y_j \\ \max(c(i, j-1), c(i-1, j)) & \text{falls } i,j > 0 \text{ und } x_i \neq y_j \end{cases}$$

Bemerkung

- im Gegensatz zum Fließbandbeispiel ist hier die Menge der zu betrachtenden Teilprobleme abhängig von der Eingabe
- rekursive Algo für $c(m,n)$ hätte exponentielle Laufzeit
- es gibt nur $\mathcal{O}(m * n)$ viele Teilprobleme $\text{lcs}(x_1 \ldots x_i, y_1 \ldots y_i)$

Eingabe

$$x = x_1 \dots x_m$$

$$y = y_1 \dots y_n$$

im Programm:

$$c[0 \dots m; 0 \dots n] = \begin{matrix} c(0,0) & \dots & c(0,n) \\ \vdots & & \vdots \\ c(m,0) & \dots & c(m,n) \end{matrix}$$

Benötigen zusätzlich eine Tabelle zur Rekonstruktion von $\text{lcs}(x, y)$

$$b[1 \dots m; 1 \dots n] = \begin{matrix} b(1,1) & \dots & b(1,n) \\ \vdots & & \vdots \\ b(m,1) & \dots & b(m,n) \end{matrix}$$

$b(i, j)$ zeigt auf den c-Eintrag, der bei der Berechnung von $c(i, j)$ bestimmend war

Ausgabe

- b und c
- $c(m, n)$ enthält den optimalen Wert

```
  LCS-length(x, y)
1   m := length(x)
2   n := length(y)
3   for i = 0 to m do c(i,0) := 0
4   for j = 0 to n do c(0,j) := 0
5   for i = 1 to m do
6       for j = 1 to n do
7           if x_i = y_j then c(i,j) := c(i-1,j-1) + 1
8                             b(i,j) :=↖
            // >> der bestimmende Wert ist der Wert, der diagonal über
            // dem Wert liegt
9           else if c(i-1,j) ≥ c(i,j-1) then c(i,j) := c(i-1,j)
10                                           b(i,j) :=↑
11          else c(i,j) := c(i,j-1)
12               b(i,j) :=←
13  return b und c
```

Beispiel

x=ABCBDAB

y=BDCABA

	y	B	D	C	A	B	A
x	0	0	0	0	0	0	0
A	0	0↑	0↑	0↑	1↖	1←	1↖
B	0	1↖	1←	1←	1↑	2↖	2←
C	0	1↑	1↑	2↖	2←	2↑	2↑
B	0	1↖	1↑	2↑	2↑	3↖	3←
D	0	1↑	2↖	2↑	2↑	3↑	3↑
A	0	1↑	2↑	2↑	3↖	3↑	4↖
B	0	1↖	2↑	2↑	3↑	4↖	4↑

$\Rightarrow$ lcs(x, y) = `BCBA`

Laufzeit

$$\mathcal{O}(m * n)$$

```
  LCS-print(b, x, y, i, j)
1   if i = 0 or j = 0 then return
2   if b(i, j) =↖ then LCS-print(b, x, y, i − 1, j − 1)
3       print x_i
4   else if b(i, j) =↑ then LCS-print(b, x, y, i − 1, j)
5   else LCS-print(b, x, y, i, j − 1)
```

Mit LCS-print(b, x, y, m, n) wird die Lösung rekonstruiert

Laufzeit

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

Verbesserungen

1. b ist eigentlich überflüssig. $c(i, j)$ hängt nur von $c(i-1, j-1)$, $c(i-1, j)$, $c(i, j-1)$ und x_i, y_j ab
   $\Rightarrow$ $c(i, j)$ in $\mathcal{O}(1)$ berechenbar und ebenso auch der entsprechende Buchstabe des lcs$(x,\ y)$
   $\Rightarrow$ ähnlich $\mathcal{O}(m + n)$ Algo für Rekonstruktion von lcs$(x,\ y)$
   $\Rightarrow$ Laufzeit von LCS-length bleibt aber dennoch $\mathcal{O}(m * n)$

2. c Tabelle zu groß. zu jedem Zeitpunkt werden lediglich die aktuelle c-Zeile und die darüberliegende c-Zeile benötigt.
   $\Rightarrow$ zur Berechnung von $c(m, n)$ genügt ein Speicherplatzbedarf von $\mathcal{O}(\min(m, n))$
   Achtung: jetzt lcs(x, y) nicht mehr so leicht rekonstruierbar

Abbildung 87: Schema: dynamische Programmierung