Vorlesung

Informatik III

–

Algorithmen und Datenstrukturen

Gehalten von

Prof. Hans-Dietrich Hecker

Friedrich-Schiller-Universität Jena

Inhalt

1 – Einführung

Allgemeine Herangehensweise:

Problem – Algorithmus – Wesentliche Operationen (ADT – Menge von Operationen) – Implementierung (Datenstruktur) – Analyse (Vergleich)

Analyse: Untersuchung der Komplexität (meist: TIME)
→ Asymptotische Zeitanalyse

Problem:

INPUT – OUTPUT
INPUTSIZE – n
TIME T(n)

z.B.

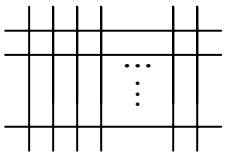
$T(n) \in O(n)$ – höchstens lineare Komplexität
$T(n) \in O(n^2)$ – höchstens quadratische Komplexität
$T(n) \in O(2^n)$ – höchstens exponentielle Komplexität
$T(n) \in \Omega(n)$ – mindestens lineare Komplexität
(z.B. Maximum einer Menge bestimmen)
$T(n) \in \Theta(n)$ – Zusammenfassung O(n) und (n)

Beispiel für Ablauf:

Problem:

ORTHO-Segment-Problem
INPUT:
H – Menge horizontaler Strecken
V – Menge vertikaler Strecken
Beide in allgemeiner Lage
Output:
Alle Schnitte
$\Omega(n^2)$ Schnitte
$O(n^2)$ Zeit
$\rightarrow \Theta(n^2)$

Worst case:

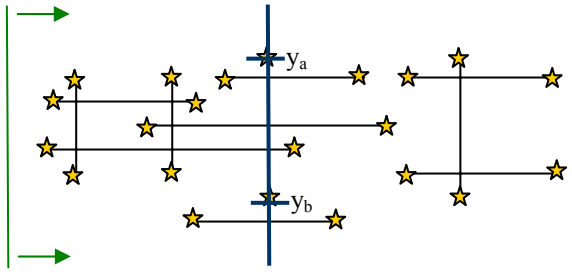


n/2 horizontale
n/2 vertikale

Outputsensitiver Algorithmus:

(Extrabehandlung der Komplexität des Outputs)

Beispiel:



plane sweep

Algorithmus:

Sortiere ☆ nach wachsende x – Laufzeit: O(n log n)
Lebende horizontale (von dem plane sweep berührte Horizontale)
Ziel: Laufzeit O(n log n +k) wobei k = Anzahl der Outputs

O(n log n + k) falls die Verarbeitung der Eventpoints (☆) jeweils in O(log n)-Time geht.

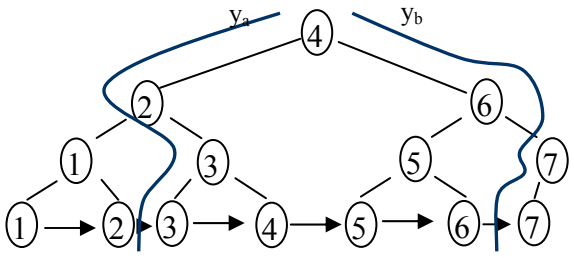
Verwalte eine Menge Y der Horizontalen Geraden (genau der y-Werte der Lebenden Horizontalen)

Start:
Y – leere Menge
Ankunft an einem linken Startpunkt einer Horizontalen
$Y := Y \cup \{y_1\}$ (INSERT)
Ankunft an einem rechten Endpunkt einer Horizontalen
$Y := Y \setminus \{y_1\}$ (DELETE)
SEARCH in $[y_a, y_b]$ nach Y (SEARCH)

ADT = {DELETE, INSERT, SEARCH}
→ Dictionary

Implementierung zur Verwaltung von Y
(→unter Beachtung nur log n Zeit zur Verfügung)

Lösung:
Höhenbalancierter Suchbaum mit verketteten Blättern
→ Höhe ist O(log n)



$3 \rightarrow 4 \rightarrow 5 \rightarrow 6$

→ Analyse
Vereinbarung:
f(n) bedeutet nicht den Wert Funktion f an der Stelle n, sondern steht für die Funktion f selbst. (Exakt wäre nur f oder λnf(n))

2 – Groß-O-Notation

Definition: (O(g(n)))

$O(g(n)) := \{f(n) \mid \exists c_2 > 0 \wedge \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : 0 \leq f(n) \leq c_2 \, g(n) \}$

Beispiel:

$\underbrace{T(n)}_{f(n)} \in O(\underbrace{n \log n + k}_{g(n)})$

$0 \leq T(n) \leq c_2 \, (n \log n + k)$

Anmerkung:

$\forall_n^\infty$ - für fast alle n gilt…

Nachteile der Definition:

1. endlich viele Ausnahmen zugelassen
2. Zulassung der Konstanten c_2

Ziel:

Vergleich 2er Algorithmen

In der Anwendung schreibt man statt $T(n) \in O(n \log n + k)$ auch z.B.
$T(n) \leq O(n \log n + k)$ oder sogar $T(n) = O(n \log n + k)$

Definition: ($\Omega(g(n))$, $\Theta(g(n))$)

$\Omega(g(n)) := \{f(n) \mid \exists c_1 > 0 \wedge \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : f(n) \geq c_1 \, g(n) \geq 0 \}$

$\Theta(g(n)) := \{f(n) \mid \exists c_2 \geq c_1 > 0 \wedge \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : 0 \leq c_1 \, g(n) \leq f(n) \leq c_2 g(n)\}$

$:= \Omega(g(n)) \cap O(g(n))$

$o(g(n)) := \{f(n) \mid \forall c > 0 \wedge \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : 0 \leq f(n) < c \, g(n) \}$

f(n) wächst wesentlich langsamer als g(n)

Beispiel: (Quicksort)

$T_{Quicksort}(n) \in O(n^2)$

$\in o(n^3)$

$\omega(g(n))$ Definition siehe Literatur

Beispiel:

$f(n) = n^2 + 1000n$

Behauptung 1: $f(n) \in O(n^2)$

Beweis:

Gesucht $c > 0$ und $n_0 \in \mathbb{N}$

Ziel: $\forall n \geq n_0 : f(n) \leq c \, n^2$

$f(n) = n^2 + 1000n \leq n^2 + 1000n^2 = 1001n^2$

$\rightarrow c = 1001, \ n_0 = 1$ q.e.d.

Behauptung 2: $f(n) \in \Omega(n^2)$

Beweis:

Gesucht $c > 0$ und $n_0 \in \mathbb{N}$

Ziel: $\forall n \geq n_0 : f(n) \geq c\, n^2$

$f(n) = n^2 + 1001n \geq 1\, n^2$

$\rightarrow c = 1,\ n_0 = 1$ q.e.d.

Behauptung 3: $f(n) \in o(n^2 \log n)$

Beweis:

Zu zeigen ist:

$\forall c>0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n)= n^2 + 1000\, n < c\, n^2 \log n$

$f(n) = n^2 + 1000n \leq 1001n^2 < c\, n^2 \log n$

Gesetze:

- $f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \rightarrow f(n) \in O(h(n))$
  (Transitivitätseigenschaft)
  Analoges gilt für Ω, Θ, o und ω
- $f(n) \in O(f(n))$
  (Reflexivitätseigenschaft)
  Analoges gilt für Ω, Θ
- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
  (Symmetrieeigenschaft)
- $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

Frage:

Wie gehen wir aus praktischer Sichtweise mit diesen Begriffen um?

Beispiel: (INSERTION SORT)

A[1..n] – Array der Länge n

```
                                                    Kosten
1  FOR j = 2 TO Länge [A] DO                        c1
2       key := A[j]                                 c2
        (*füge A[j] in die bereits
        sortierte Folge A[1] bis A[j-1] ein*)
3       i := j - 1                                  c3
4       WHILE ( i > 0 & A[i] > key) DO              c4
5                 A[i+1] = A[i]                     c5
6                 i = i - 1                         c6
7       A[i+1] := key                               c7
```

Zeile Nr.	Kosten	Anzahl der Aufrufe
1	c1	n
2	c2	(n-1)
3	c3	(n-1)
4	c4	$\sum_{j=2}^{n} t_j$
5	c5	$\sum_{j=2}^{n} (t_j - 1)$

6	c6	$\sum_{j=2}^{n}(t_j - 1)$
7	c7	(n-1)

Definition:

t_j := Anzahl der Tests in Zeile 4 für A[j]

$t_j - 1$:= Anzahl der Verschiebungen

T(n) = Summe der Kosten = $(\overbrace{c_1 + c_2 + c_3 + c_4 + c_7}^{A})n - (\overbrace{c_2 + c_3 + c_4 + c_7}^{B}) + (\overbrace{c_4 + c_5 + c_6}^{D})\sum_{j=2}^{n} t_j$

Diskussion:

1. Fall: $t_j = 1$ für alle j (best case)
   $T(n) = A\,n - B = \Theta(n)$ lineare Rechenzeit
2. Fall: $t_j = j$ für alle j (worst case)
   $T(n) = D/2\,n^2 + (A - D/2)\,n - B = \Theta(n^2)$
3. Fall:
   Vorraussetzung:
   Alle Eingabereihenfolgen sind gleichwahrscheinlich
   Man zeigt $E(t_j) = 1 + \frac{(j-1)}{2}$ und $E(\sum_{j=2}^{n}(t_j - 1)) = 0{,}25\,n\,(n-1)$
   $T(n) = D/4\,n^2 + (A - D/4)\,n - B = \Theta(n^2)$

Komplexitätstabelle

f vor g – f(n) = o(g(n))

$0 < a < b \qquad 0 < \alpha < \beta \qquad 1 < A < B$

$\alpha(n)$
log* n
log log n
$(\log n)^{\alpha}$
$(\log n)^{\beta}$
n^a
$n^a(\log n)^{\alpha}$
$n^a(\log n)^{\beta}$
n^b
A^n
$A^n n^a$
$A^n n^b$
B^n

log* n
$\log^{(0)} n = n$
$\log^{(1)} n = \log n$
$\log^{(i)} n = \log^{(i-1)} \log n$
$\log^* n := \min\{I \mid \log^{(i)} n \leq 1\}$

$\lim_{n\to\infty} \log^* (n) = \infty$

Eine Ackermannfunktion

$A(1, j) = 2^j, j \geq 1$
$A(i, 1) = A(i-1,2), i \geq 2$
$A(i, j) = A(i-1, A(i, j-1))\ i,j \geq 2$

$\alpha(m,n) := \min_i (i \geq 1 \mid A(i, \lfloor \frac{m}{n} \rfloor) > \log n) \qquad m \geq n \geq 1$

$\alpha(m,n) \leq 3$ für $n = m < 2^{16}$
$\alpha(m,n) \leq 4$ für alle "praktisch" auftretende Fälle

$\alpha(n) := \alpha(n,n), \qquad \lim_{n\to\infty} \alpha(n) = \infty$

3 – Rekurrenzen

Definition:
Eine Rekurrenz ist eine Gleichung oder Ungleichung, die eine Funktion in Termen ihrer Werte für kleinere Inputs beschreibt.

Beispiel: (Merge-Sort)

```
SORT(A[l …r])
1  IF l < r THEN
2     p = ⌊(l+r)/2⌋
3     SORT(A, l, p)
4     SORT(A, p+1, r)
5     MERGE(A, l, p, r)
        → Divide&Conquer
```

Pointer → $a_1 \le a_2 \le \ldots \le a_{n/2}$

→ $c_1 \le c_2 \le \ldots \le c_n$ $\quad c_i \in \{a_1,\ldots,a_{n/2}, b_1, \ldots, b_{n/2}\}$

Pointer → $b_1 \le b_2 \le \ldots \le b_{n/2}$

Sei $a_1 < b_1 \rightarrow a_1$

MERGE hat die Komplexität O(n)

$T_{sort}(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + O(n)$ (+ Konstante)

$= 2\,T(\frac{n}{2}) + O(n)$

3.1 Methoden zur Lösungen von Rekurrenzen

3.1.1 Vollständige Induktion (Substitutionsmethode)

a) *Raten*
b) *Induktionsbeweis*

Beispiel:

$T(n) = 2T(\frac{n}{2}) + n$

$T(1) = 1$

Ansatz: $T(n) \le c \cdot n \log n$, $\quad c$ – Konstante

$T(n) = 2T(\frac{n}{2}) + n \le 2 \cdot c \cdot \frac{n}{2} \log \frac{n}{2} + n = c \cdot n \log \frac{n}{2} + n$

$= c \cdot n \cdot \log n - c \cdot n \cdot \log 2 + n \le c \cdot n \cdot \log n - c \cdot n + n \le c \cdot n \log n$ für $c \ge 1$

→ mit $c \ge 1$: $T(n) = O(n \log n)$

Problem:

Induktionsanfang – ist in der Praxis in der Regel kein Problem

$\rightarrow T(1) = 1 \leq c \cdot 1 \cdot \log 1 = c \cdot 0 \rightarrow 1 \leq 0$ Widerspruch!

Lösung:

$T(n) \leq c \cdot n \log n$ für $n \geq 2$

Induktionsanfang:

T(2), T(3)

Wichtige Bemerkungen:

- in Zukunft lassen wir im Regelfall die Anfangswerte weg
- in Zukunft verzichten wir auf „$\lfloor\ \rfloor$“ und „$\lceil\ \rceil$“

Beispiel:

$T(n) = 2T(\frac{n}{2}) + b$, b – Konstante

Vermutung:

T(n) = O(n) ist Lösung

Ansatz:

$T(n) \leq c \cdot n$ c – Konstante

$T(n) = 2T(\frac{n}{2}) + b \leq 2 \cdot c \cdot \frac{n}{2} + b = c \cdot n + b$ ← keine Lösung

Neuer Ansatz:

$T(n) \leq c \cdot n - b$

$T(n) = 2\,T(\frac{n}{2}) + b \leq 2 \cdot (c \cdot \frac{n}{2} - b) + b = c \cdot n - b$

Lösung:

T(n) = O(n)

Beispiel:

$T(n) = 2T(\frac{n}{2}) + n$

Ansatz:

$T(n) \leq c \cdot n$

$T(n) = 2T(\frac{n}{2}) + n \leq 2 \cdot c \cdot \frac{n}{2} + n = c \cdot n + n = n \cdot (c + 1) = O(n)$ falsch!

„Beweis“ einer falschen Aussage!!!

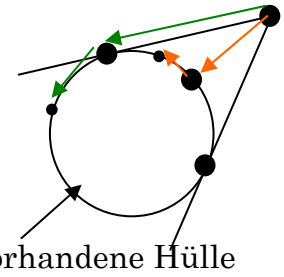
Anwendungsbeispiel:

Problem der Konvexen Hülle einer unendlichen Punktmenge

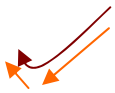
Inkrementeller Ansatz:

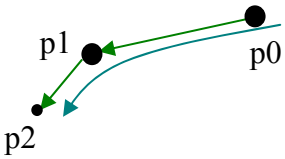
Voraussetzung:

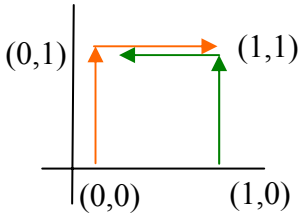
Input ist sortiert nach wachsenden x



Sei konvexe Hülle (CH) als Liste im Gegenuhrzeigersinn gegeben.







$$\det \begin{vmatrix} 1 & p_0x & p_0y \\ 1 & p_1x & p_1y \\ 1 & p_2x & p_2y \end{vmatrix} \begin{cases} > 0 \text{ wenn gegen Uhrzeigersinn} \\ < 0 \text{ wenn im Uhrzeigersinn} \end{cases}$$

$$\det \begin{vmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{vmatrix} = -1 \qquad \det \begin{vmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{vmatrix} = 1$$

$\rightarrow T(n) = T(\frac{n}{2}) + b$ (binäre Suche)

Ansatz:

Mit dem Ziel insgesamt T(n) = O(n log n) zu erhalten, brauchen wir als Lösung für $T(n) = T(\frac{n}{2}) + b$

$T(n) = O(\log n)$

Also Ansatz:

$T_{Tangente}(n) \leq c \cdot \log n$ $\rightarrow$ führt zum Erfolg

3.1.2 Methode der Variablensubstitution

Beispiel:

$T(n) = 2T(\sqrt{n}) + \log n$

Substitution:

k: $2^k = n$, $k = \log n$

$T(2^k) = 2T(2^{k/2}) + k$

$S(k) = T(2^k) \rightarrow S(k) = 2\, S(\frac{k}{2}) + k$

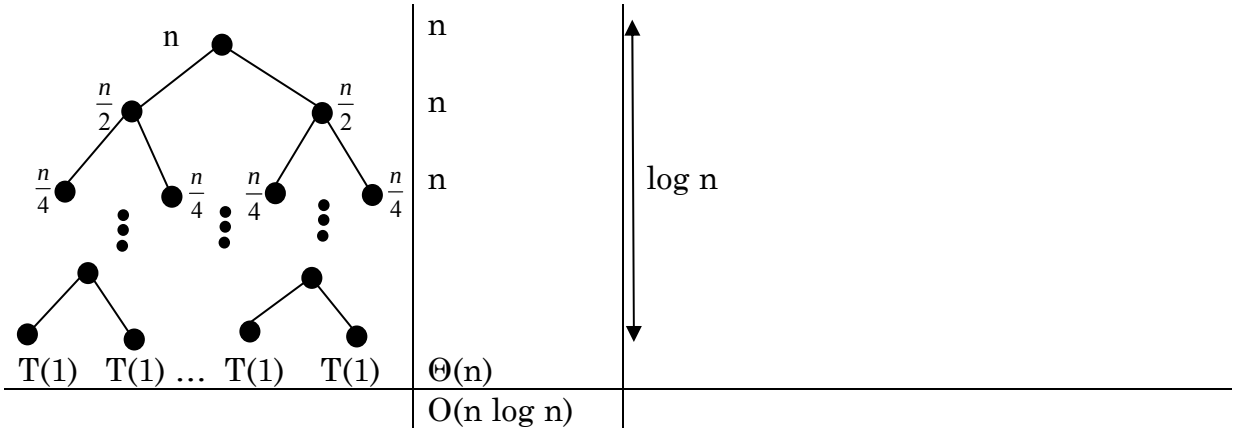
$\rightarrow S(k) = O(k \cdot \log k)$

$T(n) = O(\log \log \log n)$

3.1.3 Rekurrenzbäume

Beispiel:

$T(n) = 2T(\frac{n}{2}) + n$ $\qquad$ $T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{n}{2}$



$T(n) = n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \ldots + 2^k \cdot \frac{n}{2^k} + 2^{k+1} \cdot \frac{n}{2^{k+1}}$ $\qquad$ bis $2^{\frac{n}{k+1}} \leq 1$

$n \leq 2^{k+1}$ $\quad \log n \leq k+1$

$= n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \ldots + 2^k \cdot \frac{n}{2^k} + 2^{\log n} T(1)$

2. Beispiel:

Vorbemerkung:

$b^{\log_b n} = n$

$(b^{\log_b n})^{\log_b a} = n^{\log_b a} = (b^{\log_b a})^{\log_b b} = a^{\log b \cdot n}$

$T(n) = 3T(\frac{n}{4}) + c \cdot n^2$

$T(n) = cn^2 + \frac{3}{16}\, cn^2 + (\frac{3}{16})^2\, cn^2 + \ldots + (\frac{3}{16})^{\log n - 1}\, cn^2 + \Theta(n^{\log_4 3})$

$$= \sum_{i=0}^{\log_4(n-1)} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) = \frac{1}{1-\frac{3}{16}} cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = O(n^2)$$

Lösung:

$T(n) = O(n^2)$

3. Beispiel:

$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + cn$

Ziel: $T(n) = O(n \log n)$

Ansatz über Rekurrenzbäume führt hier wenigstens zur Vermutung
$T(n) \leq cn \log n \rightarrow$ Substitutionsmethode

3.1.4 Mastermethode

Theorem: (Mastertheorem)

Sei $a \geq 1$, $b > 1$, $f(n)$ asymptotisch positive Funktion

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

f(n) setzt sich zusammen aus dem Aufwand für das Teilen und für das Mischen

Dann gilt:

1. $f(n) = O(n^{\log_b a + \varepsilon})$ $\varepsilon > 0$ (!!!) $\rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a})$ $\rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a} + \varepsilon)$ (für ein festes) $\varepsilon > 0$ und $\exists\, c > 1$:

   $a \cdot T(\frac{n}{b}) \leq c \cdot f(n)$ $\left(\forall_n^\infty\right)$ $\rightarrow T(n) = \Theta(f(n))$

Bemerkung zum Beweis:

Methode: induktive Schlüsse, vgl. Literatur:
Raimund Seidel (Saarbrücken)
Cormen: voller Beweis

Beispiele:

1) Fall 1: $T(n) = T(\frac{n}{2}) + 1$

   ähnlich: $T(n) = T(\frac{2n}{3}) + 1$

   $a = 1$, $b = 2$: $n^{\log_b a} = n^0 = 1 = f(n)$

   Fall 2: $T(n) = \Theta(1 \cdot \log n) = \Theta(\log n)$

   $\rightarrow$ Tangenten in $O(\log n)$

2) Fall 1: $T(n) = 2T(\frac{n}{2}) + n$

$a = 2, b = 2, \; n^{\log_b a} = n^1 = n = f(n)$

Fall 2: $T(n) = \Theta(n \log n)$

3) $T(n) = T(\frac{n}{2}) + \log n$

$a = 1, b = 2, \quad n^{\log_b a} = n^0 = 1$

$n^{\log_b a} = 1 < \log n$ mit Vorsicht: $\forall_n^\infty : c \cdot n^{\log_b a + \varepsilon} \leq \log n$ geht nicht!

Mit Mastertheorem unlösbar! (Lücke zwischen Fällen 2 und 3)
Ausweg: andere Verfahren…

Vermutung bei $T(n) = T(\frac{n}{2}) + \log n$

$T(n) = O(\log^2 n)$

Ansatz: $T(n) \leq c \cdot \log^2 n$

Substitutionsmethode:

$T(n) = T(\frac{n}{2}) + \log n \leq c \cdot \log^2 \frac{n}{2} + \log n$

$= c \cdot (\log n - \log 2)(\log n - \log 2) + \log n$

$= c \cdot \log^2 n - 2c \cdot \log n + c + \log n = c \log^2 n + c + \log (1-2c)$

$c + \log n (1-2c) \leq 0 = c \log^2 n + c + \log n \cdot d, \; d < 0$ machbar

$c = 1$:

$1 + \log n (1-2) = 1 - \log n \leq 0 \quad \forall n \geq 5$

$\leq c \log^2 n \rightarrow$ Behauptung

4) $T(n) = 9T(\frac{n}{3}) + n$

$f(n) = n, a = 9, b = 2, \; n^{\log_b a} = n^2$

$\varepsilon = 1 \quad n^{\log_3 9 - 1} = n^{2-1} = n = O(f(n))$

$\rightarrow$ Fall 1:

$T(n) = \Theta(n^2)$

5) $T(n) = 2T(\frac{n}{2}) + n \log n$ (Lücke zwischen 2 und 3)

4 - Sortieren und Auswählen (Selektion)

Sort:
INPUT:
Folge von Elementen einer Menge mit linearer Ordnungsrelation (total)
OUTPUT:
Folge der Inputelemente in der Reihenfolge der Ordnungsrelation

Selektion:
(speziell für Zahlen)
INPUT:
Folge von natürlichen Zahlen (n Stück und eine Zahl k: $1 \leq k \leq n$)
OUTPUT:
Zahl vom Rang k

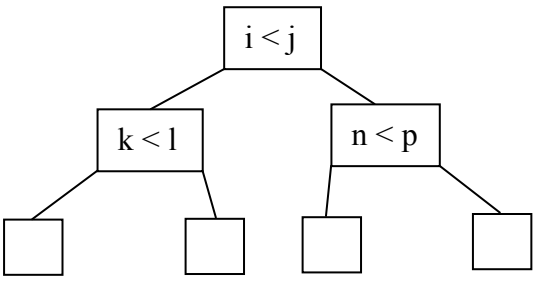
Beispiel:
INPUT: (3,5,1,6,9,7), k = 2
OUTPUT: 3

Speziell:
k = 1 – Minimum
k = n - Maximum
k= $\left\lceil \frac{n}{2} \right\rceil$ – *MEDIAN*

4.1 Hauptsatz über das Sortieren

1. Hauptsatz:
Auf der Basis von Schlüsselvergleichen kostet Sort $\Omega(n \log n)$ Zeit

Beweis: (für Zahlen)



$[a_1, a_2, \ldots, a_n]$ – INPUT-Array
Für $a_i < a_j$ schreiben wir kurz „i < j“

Sortieren abgeschlossen wenn die Reihenfolge klar ist.
Also die richtige Permutation des INPUTS ist gesucht.

Der Baum der hier entsteht heißt Entscheidungsbaum. Dieser Baum ist durch den Algorithmus bestimmt. Der Algorithmus kann nur dann immer erfolgreich sein, wenn es zu jeder Permutation ein eigenes Blatt in dem Baum gibt…

Also Baumhöhe h und 2^h Blätter.

Nötig ist:

$2^h \geq n!$ (jede Permutation braucht ein Blatt)

$h = \log 2^h \geq \log n!$

h = Modell für Rechenzeit $\geq \log n! \geq \log \frac{n}{2}^{\frac{n}{2}} = \Omega(n \log n)$

4.2 Sortieren und Auswählen – die Zweite

Eine untere Schranke:
Beispiel: MERGE
$\rightarrow O(n), \Omega(n) \rightarrow \Theta(n)$

Voraussetzung:

Die zu sortierenden Objekte sind paarweise verschieden. oBdA

INPUT: Folge $(a_1, \ldots, a_n)$ $a_i \in S$ S Menge mit $\leq$
OUTPUT: Folge $(a_{\pi(1)}, \ldots, a_{\pi(n)})$ π Permutaion von 1, …, n mit $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$
1. Hauptsatz:

Sortieren durch Vergleiche hat Komplexität $\Theta(n \log n)$

Beweis:

$O(n \log n)$ trivial (mergesort)

Zu zeigen: $\Omega(n \log n)$

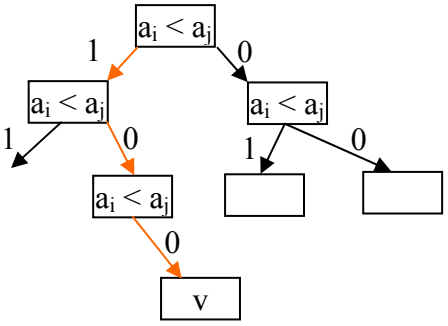
INPUT sei als Array gegeben. Wir können nur vergleichen

$$K(i,j) := \begin{cases} 1 \text{ falls } a_i < a_j \\ 0 \text{ falls } a_j < a_i \end{cases} \quad i \neq j$$

$W(i,j) := \{(a_1, \ldots, a_n) \in S^n \mid a_i < a_j\}$

Sei ein deterministisches Verfahren gegeben zur Lösung der Sortieraufgabe,
dann ist klar, welcher Vergleich der erste ist.

Etwa K(i,j) = 1?
Im 1-Ausgang ist dann klar welcher Vergleich als nächstes kommt.
Analog für den 0 Ausgang.



Alle Inputreihenfolgen beginnen bei der Wurzel.

Welche landen bei v?
Genau die Inputs der Menge : $W(i,j) \cap W(l,k) \cap (j,l)$ (*)

Fakt:
Eine Inputreihenfolge ist dann sortiert, wenn die Zugehörigkeit zu einem Blatt und der entsprechenden Menge (*) .

Eindeutiges über die gesuchte Permutation:
(*) $\hat{=} a_i < a_j < a_l < a_k$

Definition: (Entscheidungsbaum)
Ist ein Baum, der so erhalten wird.

Fakt:
Wenn der Algorithmus in allen Fällen korrekt zum Ziel führt, braucht jede Permutation ihr eigenes Blatt.
→ Zahl der Blätter muss ≥ n! sein!

Satz:
Ein Binärbaum der Höhe h hat maximal 2^h Blätter
(genau 2^h Blätter für vollständige Binärbäume)

Fakt:
Die Länge eines Pfades von der Wurzel zu einem Blatt modelliert die Rechenzeit im Sinne von Θ.

Fakt:
Die Höhe eines Binärbaumes mit m Blättern ist Θ(log m), falls dieser balanciert ist.

$2^h \geq n!$ denn jede Permutation erfordert eigenes Blatt

$\rightarrow h \geq \log n!$

$\frac{n}{2}^{\frac{n}{2}} \leq n! \leq n^n$

$\rightarrow \frac{n}{2} \log \frac{n}{2} = \log \frac{n}{2}^{\frac{n}{2}} \leq \log n! \leq \log n^n = n \log n$

$\log n! \geq \log \frac{n}{2}^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2} = \frac{n}{2} (\log n - \log 2) = \frac{n}{3} \log n + (\frac{n}{6} \log n - \frac{n}{2}) \geq \frac{n}{3} \log n$ für $n \geq 8$
$\rightarrow \log n! = \Omega(n \log n)$, $\log n! = O(n \log n)$
$\rightarrow \log n! = \Theta(n \log n)$

2. Hauptsatz:
Sortieren durch Vergleichen erfordert, wenn alle Inputreihenfolgen gleich wahrscheinlich sind im Mittel Ω(n log n) (bzw. Θ(n log n)

Beweis:

Hilfssatz 1: (Ungleichung von Kraft)

Es sei T ein Binärbaum mit m Blättern und $t_1, \ldots, t_m$ seien die Pfadlängen von der Wurzel zu diesen Blättern.

Dann gilt:

$$\sum_{i=1}^{m} 2^{-t_i} \leq 1$$

Induktiver Beweis:

m = 1 trivial

m ≥ 2 Blätter:

→ 2 Teilbäume mit m_1 und m_2 Blättern.

Pfadlänge im Teilbaum $t_{11}, \ldots, t_{1m}$ ↙ ↘ Pfadlänge im Teilbaum $t_{21}, \ldots, t_{2m}$

Nach IV gilt: $\sum_{i=1}^{m_1} 2^{-t_i} \leq 1, \ \sum_{i=1}^{m_2} 2^{-t_i} \leq 1$

$$\rightarrow \sum_{i=1}^{m_1} 2^{-(t_i+1)} + \sum_{i=1}^{m_2} 2^{-(t_i+1)} = 2^{-1} \Big(\underbrace{\sum_{i=1}^{m_1} 2^{-t_i}}_{\leq 1} + \underbrace{\sum_{i=1}^{m_2} 2^{-t_i}}_{\leq 1}\Big) \leq 1 \qquad \text{q.e.d.}$$

Hilfssatz 2:

$$\frac{1}{m}\sum_{i=1}^{m} t_i \geq \log m$$

Beweis:

$$\frac{1}{m}\sum_{i=1}^{m} t_i \geq \log m$$

$$\frac{1}{m}\sum_{i=1}^{m} 2^{-t_i} \geq \sqrt[m]{\prod_{i=1}^{m} 2^{-t_i}}$$

$$\frac{1}{m} \geq \frac{1}{m}\underbrace{\sum_{i=1}^{m} 2^{-t_i}}_{\leq 1} \geq \sqrt[m]{\prod_{i=1}^{m} 2^{-t_i}} = \sqrt[m]{2^{-(t_1+t_2+\ldots+t_m)}} = \sqrt[m]{2^{-\sum t_i}} = 2^{-\frac{1}{m}\sum t_i}$$

$$m \leq 2^{\frac{1}{m}\sum t_i} \xrightarrow{\log} \log m \leq \frac{1}{m}\sum_{i=1}^{m} t_i$$

Fakt:

$\frac{1}{m}\sum_{i=1}^{m} t_i$ - durchschnittliche Rechenzeit

Für unsere Entscheidungsbäume (Binärbäume) folgt damit m muss ≥ n! sein!

$$\frac{1}{m}\sum_{i=1}^{m} t_i \geq \log m \geq \log n! = \Omega(n \log n) \qquad \text{q.e.d.}$$

Anwendung: (indirekter Komplexitätsbeweis)

INPUT: Menge von Punkten in Ebene
OUTPUT: Der Rand der konvexen Hülle im Gegenuhrzeigersinn

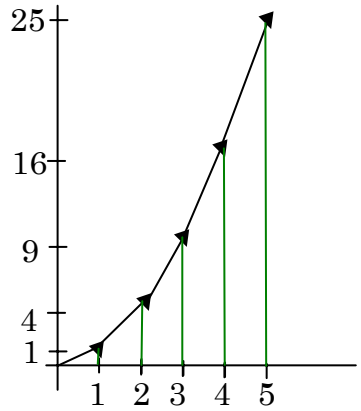
$\rightarrow O(n \log n) \rightarrow T(n) = T(\frac{n}{2}) + 1$ O(log n) für alle Punkte

Behauptung:
$\Omega(n \log n)$

INPUT: $x_1, \ldots, x_n$ Zahlen
1. Bilde Punkte $(x_1, x_1^2), \ldots, (x_n, x_n^2)$, $S := \{(x_1, x_1^2), \ldots, (x_n, x_n^2)\}$
2. CH(S)
3. Projiziere auf die x-Achse
4. OUTPUT über Ablaufen der konvexen Hülle (d.h. nach wachsenden x)

→ Schritt 1 benötigt O(n), Schritt 3 benötigt O(n), Schritt 4 benötigt O(n)
→ Schritt 2 muss $\Omega(n \log n)$ liefern

Beispiel:
INPUT: 1, 4, 3, 2, 5



OUTPUT: 1, 2, 3, 4, 5

4.3 Selektion

A – Menge paarweise verschiedener Elemente
$A = \{a_1, a_2, \ldots, a_n\}$
$\text{Rang}(a_i : A) := |\{k \in A \mid k \leq a_i\}|$
$A = \{3,1,-1,0,6,9,7\}$
→ $\text{Rang}(3 : A) = 4$

Selektion:
INPUT: $S = \{s_1, s_2, \ldots, s_n\}$ k $(1 \leq k \leq n)$
OUTPUT: $s_i \in S$, $\text{Rang}(s_i : S) = k$
(nicht notwendig paarweise verschiedene, Ausgabe ein Element mit dieser Eigenschaft)

Speziell:

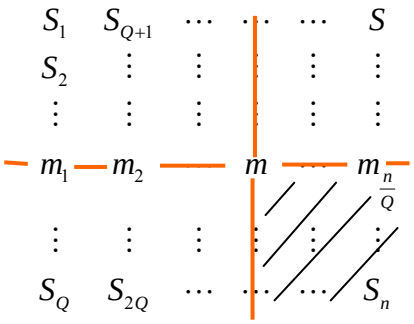
k =1, k = n, k = $\left\lceil \frac{n}{2} \right\rceil$ (Median)

k = 1 → O(n) → trivial
k = n → O(n) → trivial
k beliebig → O(n)! → nicht trivial!
→ „Median der Mediane“-Technik

Select(S, k) Parameter Q
Schritt 1:
If |S| < Q
So sortiere und entsprechender Output → O(n)
Else
Zerlege S in $\frac{|S|}{Q}$ Teilfolgen der maximalen Länge Q → O(n)

Q Zeilen $\underbrace{\begin{Bmatrix} S_1 & S_{Q+1} & \cdots & S \\ S_2 & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ S_Q & S_{2Q} & \cdots & S_n \end{Bmatrix}}_{\frac{n}{Q} Spalten}$ oBdA: n durch Q teilbar

Schritt 2:
Bestimme für jede Zeile die insgesamt $\frac{n}{Q}$ Mediane: $m_1, m_2, \ldots, m_{\frac{n}{Q}}$ → O(n)



Schritt 3:
Bestimme rekursiv: → $T(\frac{n}{Q})$

Select($\{m_1, m_2, \ldots, m_{\frac{n}{Q}}\}, \left\lceil \frac{n}{2Q} \right\rceil$)
Output: m (Median der Mediane)

Schritt 4:
Definiere:
$S_1 := \{s \in S \mid s < m\}$
$S_2 := \{s \in S \mid s = m\}$
$S_3 := \{s \in S \mid s > m\}$ Merke: $|S_1|, |S_2|, |S_3|$

Schritt 5:
If $|S_1| \geq k$
Then Select(S_1, k)
Elseif $|S_1|+|S_2| \geq k$
Then Output m
Else
Select(S_3, $k - (|S_1|+|S_2|)$)

Problem:
T( ) Schritt 5
$\rightarrow \frac{n}{2Q}$ Mediane $\geq$ m, jeder hat $\frac{Q}{2}$ Elemente $\geq$ dieser Median

$\rightarrow$ mindestens $\frac{n}{4}$ Elemente sind $\geq$ m

$\rightarrow$ höchstens $\frac{3n}{4}$ Elemente < m

$\rightarrow |S_1| < \frac{3n}{4}$, analoge Schlussweise $|S_3| < \frac{n}{4}$

$\rightarrow$ **Rekurrenz:**

$T_{Selection}(n) = O(n) + T(\frac{n}{Q}) + T(\frac{3n}{4})$ mit $\frac{n}{Q} + \frac{3n}{4} < n \rightarrow Q \geq 5$

Ansatz:
$T(n) \leq c_1 n$ (also O(n))

$T(n) \leq cn + c_1\frac{n}{5} + c_1\frac{3n}{4} = cn + c_1\left(\frac{4n+15n}{20}\right) = cn + c_1\left(\frac{19n}{20}\right)$ $c_1 := 20c$

$\rightarrow c = \frac{c_1}{20}$

$= \frac{c_1}{20} n + c_1\left(\frac{19n}{20}\right) = c_1 n$

4.4 S-Quicksort

Algorithmus:

S-Quicksort(S) $S = \{s_1,\ldots,s_n\}$
If $|S| \leq 2$ and $s_2 < s_1$
Then tausche s_2 mit s_1
Else if $|S| > 2$
Then (1) Select(S, $\left\lceil \frac{|S|}{2} \right\rceil$) $\rightarrow$ m $\rightarrow$ O(n)

(2) 2.1 $S_1 := \{s_i \mid s_i \leq m\}$ und $|S_1| = \left\lceil \frac{|S|}{2} \right\rceil$

2.2 $S_2 := \{s_i \mid s_i \geq m\}$ und $|S_2| = \left\lceil \frac{|S|}{2} \right\rceil$

(3) S-Quicksort(S_1)

(4) S-Quicksort(S_2)
(5) Merge (conquer)

Rekurrenz:

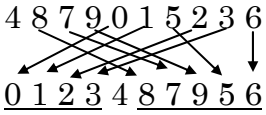
$$T(n) = 2\,T(\frac{n}{2}) + O(n)$$

4.5 Quicksort

Idee:

7 6 9 4 2 1 0 5 8 [3] ← Pivotelement
→ ←
< [3] > Rekursiv weiter….

Frei nach Schöning:



Definition:

V(n) = Anzahl der Vergleiche

Worst case:

$$V(n) = \frac{n(n-1)}{2} = \Theta(n^2)$$

Average Case:

Voraussetzung:
Alle n! Permutationen sind gleichwahrscheinlich.

INPUT: $a_1, a_2, \ldots, a_n$ - sortitert: $s_1, \ldots, s_n$ mit $s_1 \le \ldots \le s_n$

$$X_{\substack{i,j \\ i<j}} = \begin{cases} 1 & s_i \text{ mit } s_j \text{ verglichen} \\ 0 & \text{sonst} \end{cases}$$

Sei p_{ij} die Wahrscheinlichkeit dafür, dass s_i und s_j verglichen wird.
$E[X_{\substack{i,j \\ i<j}}] = 1 \cdot p_{ij} + 0 \cdot (1 - p_{ij}) = p_{ij}$

$V(n) = E[\sum_{i<j} X_{i,j}]$ (Erwartungswert für Rechenzeit)

$$= \sum_{i<j} E[X_{i,j}] = \sum_{i<j} p_{ij}$$

Wann wird s_i mit s_j verglichen?
Im Inputarray $A = (a_1, a_2, \ldots, a_n)$ sei von den
Zahlen $s_1, s_2, \ldots, s_n$ das s_μ die Zahl mit kleinsten Index in A,
d.h. $s_\mu = a_k$: k minimal
Es findet genau dann ein Vergleich zwischen s_i und s_j statt,
wenn $\mu = i$ oder $\mu = j$

gegeben i < j:
Fall: Pivotelement $x \notin \{s_i, \ldots, s_j\}$
- s_i wird nicht mit s_j verglichen

Fall: Pivotelement $x \in \{s_i, \ldots, s_j\}$
- dann ist es dasjenige, dass den kleinsten Index im Eingabearray hat.
  Sofern $x \in \{s_i, \ldots, s_j\}$ und $x \neq s_i \wedge x \neq s_j$ so kommen s_i und s_j in verschiedene Teilarrays → werden nicht mehr verglichen
  vergleichen falls $x = s_i$ oder $x = s_j$
  → 2 Möglichkeiten

$$\underbrace{s_i \underbrace{\ldots s_j}_{j-i}}_{j-i+1} \rightarrow \; p_{ij} = \frac{2}{j-i+1}$$

$$= \sum_{i<j} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{j-i+1}$$

$$\leq \sum_{i=1}^{n-1} 2(H_n - 1) \leq 2n(H_n - 1) \leq 2n \ln n \leq 1{,}39 n \log n$$

NR: (Warum $H_n - 1$ mit ln n abschätzen)

$$\frac{1}{i+1-i+1} + \frac{1}{i+2-1+1} + \ldots + \frac{1}{n-i+1} + \| \left(\ldots + \frac{1}{n} \right)$$

$$= \frac{1}{2} + \frac{1}{3} + \ldots \| \ldots + \frac{1}{n}$$

$$\Rightarrow H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

$$\int \frac{1}{x} dx = \ln x$$

$\ln n \leq H_n \leq (\ln n) + 1$
$\rightarrow H_n - 1 \leq \ln n$

4.6 Heap-Sort

Definition:
Eine Folge $F = k_1, k_2, \ldots, k_N$ heißt binary heap (also Max-Heap), wenn $k_i \leq k_{\lfloor \frac{i}{2} \rfloor}$ für $2 \leq i \leq N$ ist,

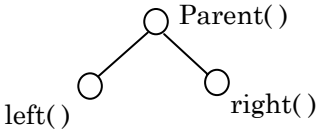
Für $2i \leq N$ und $2i + 1 \leq N$ gilt:
$k_i \geq k_{2i}$ und $k_i \geq k_{2i+1}$

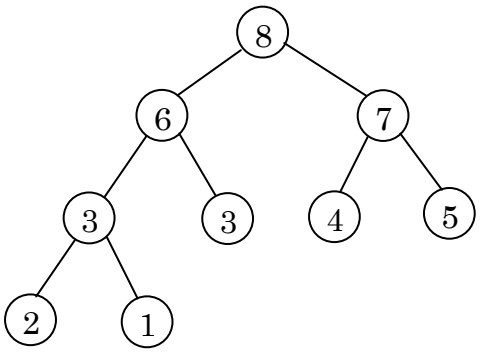
Beispiel:

	8	6	7	3	3	4	5	2	1
Index	1	2	3	4	5	6	7	8	9

$k_1 \geq k_{2\cdot1} = k_2 \rightarrow 8 \geq 6$ und $k_1 \geq k_{2\cdot1+1} = k_3 \rightarrow 8 \geq 7$
$k_2 \geq k_{2\cdot2} = k_4 \rightarrow 6 \geq 3$ und $k_2 \geq k_{2\cdot2+1} = k_5 \rightarrow 6 \geq 3$
$k_3 \geq k_{2\cdot3} = k_6 \rightarrow 7 \geq 4$ und $k_3 \geq k_{2\cdot3+1} = k_7 \rightarrow 7 \geq 5$
$k_4 \geq k_{2\cdot4} = k_8 \rightarrow 3 \geq 2$ und $k_4 \geq k_{2\cdot4+1} = k_9 \rightarrow 3 \geq 1$

Baumvorstellung (Binärbaum)





Parent(i)

Return $\left\lfloor \frac{i}{2} \right\rfloor$

Left(i)

Return 2i

Right(i)

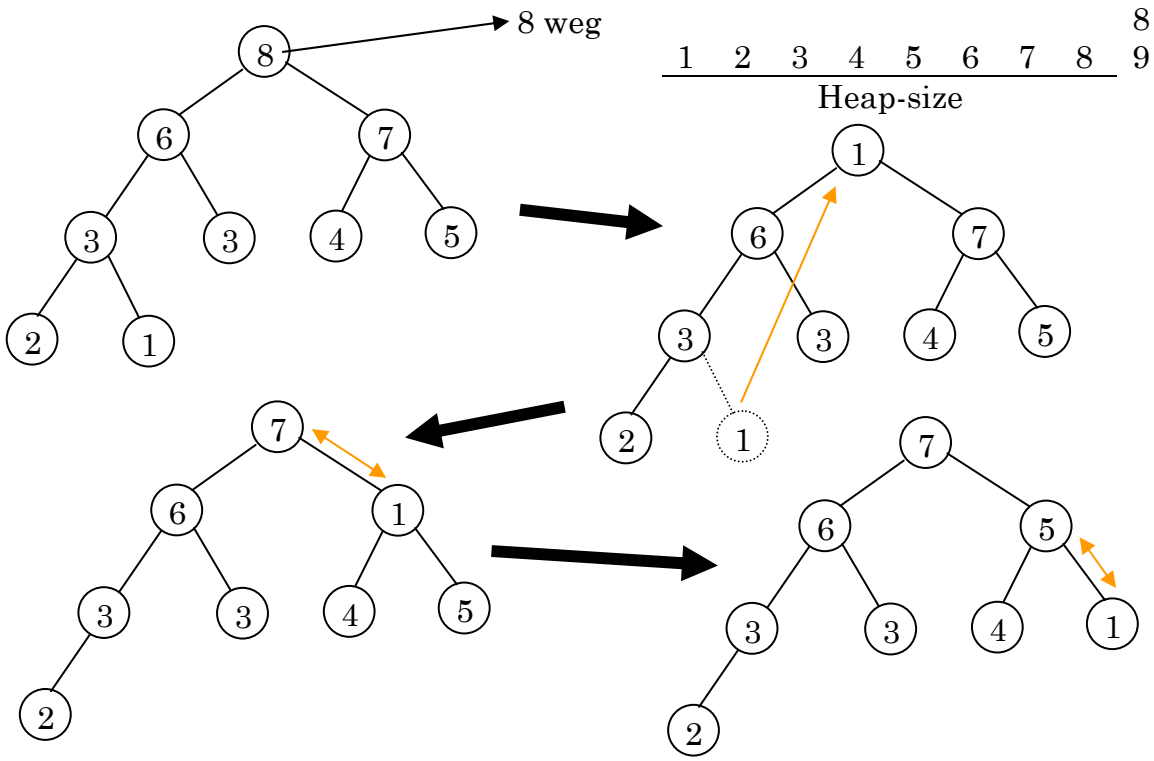
Return 2i +1

Heap-Property (des Max-Heaps)

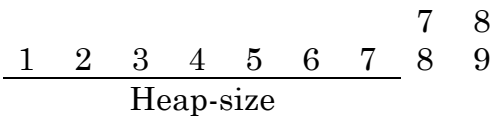
A[parent(i)] ≥ A[i]

Heapify(A,i)

```
1   l := left(i)
2   r := right(i)
3   if l ≤ heap-size[A] and A[l] > A[i]
4    then largest := l
5    else largest := i
6   if r ≤ heap-size[A] and A[r] ≥ A[largest]
7    then largest := r
8   if largest ≠ i
9    then tausche A[i] mit A[largest]
10   Heapify(A, largest)
```



Heapsort: Pflücken der Wurzeln



Algorithmus zum Heap-Bau

Build-Heap(A)

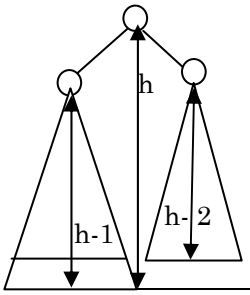
```
1  Heap-size[A] := Länge[A]
2  for i = [Länge(⌈A⌉/2)] down to 1
3     do Heapify(A,i)
```

Lemma:

Der linke Teilbaum eines Binärheaps mit n Knoten hat maximal $\frac{2}{3}$ n Knoten.

Beweis:

Fakt: Vollständiger Binärbaum der Höhe h hat maximal $2^{h+1} - 1$ Knoten



Knotenzahl:

$2^h - 1$ (linker Teilbaum)
$+ 2^{h-1} - 1$ (rechter Teilbaum
$+ 1$ (Wurzel)

$$= \frac{3}{2} \cdot 2^h - 1 \geq \frac{3}{2} \cdot (2^h - 1)$$

$$\text{Gesamt} \geq \frac{3}{2} \text{links}$$

$$\frac{2}{3} \text{gesamt} \geq \text{links}$$

→ Heapify kostet:

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

Lösung:

$T(n) = O(\log n)$

Build-Heap in O(n)

T(n) := Anzahl Vergleiche bei Build-Heap
V(t) := Summe aller Entfernungen bis zur Blattebene bei einem Baum der Tiefe t

$V(t) \leq t + V(t-1) + V(t-1) \leq t + 2\,V(t-1)$

Ansatz:

$V(t) \leq 2^{t+1} - t$

$V(t) \leq t + 2V(t-1) \leq t + 2 \cdot (2^t - t) = 2^{t+1} - t$

→ $V(t) = O(2^{t+1}) \leq 2^{t+1}$
$T(n) \leq 2\,V(\log n) \leq 2 \cdot 2^{\log n + 1} \leq 4 \cdot 2^{\log n} = 4n = O(n)$

Heap-Sort:

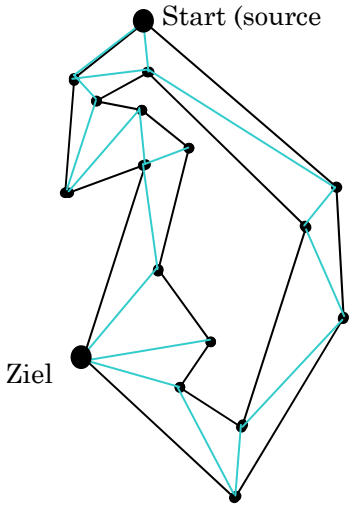
1. Build-Heap O(n)
2. Pflücke immer wieder Wurzel O(1)
3. Schreibe letztes Element in Wurzel – Heapify O(n log n)
4. Ende

→ O(n log n)

Algorithmus für HeapSort(A)

```
1  Buildheap(A)
2  for i := Länge[A] down to 2
3    do tausche A[1] mit A[i]
4    heapsize[A] := heapsize[A]-1
5    heapify(A,1)
```

Beispiel für Anwendungen der Datenstruktur



Lozano-Perez&Wesley 1979

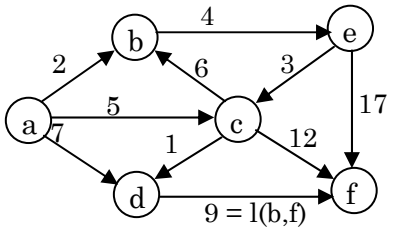
Kürzester Weg stets entlang des Sichtbarkeitsgraphs

(Seit 1979 Lee, Preparaba : O(n² log n) Konstruktion des Sichtbarkeitsgraphs, seit 198x Emo Welzl O(n²))

Sei ein solcher Graph gegeben.

Ziel:

Single-source-shortest-path (tree)



v∈V beliebig,
N(v) := {w∈V, w direkt über Pfeil erreicht}
z.B. N(a) = {c,d,b}

Dijkstra

Algorithmus shortestPath((V,E),s∈V)

```
1  for all v∈V do d(v) := +∞
2  d(s) = d(a) := 0; s := Ø
3  Initialisiere V\S = V als Priority Queue
4  while V\S ≠ Ø do
5    v := min(V\S); Extract-Min(V\S);
6    S := S∪V
7  for all w∈N(v)\S do
8    if d(v) + l(v,w) < d(w)
9    then DecreaseKey(w, d(v) + l(v,w))
10 End
```

(v, d(v)): v∈S | (v, d(v)): v∈V\S	v = min(V\S) und w∈N(v)\(S∪{v}) $\frac{(w,l(v,w))}{\min(d(w),d(v)+l(v,w)}$
Ø | (a,0),(b,∞), (c,∞),(d,∞), (e,∞), (f,∞)	(a, N(a) = {c,d,b}→ $\frac{(c,2)(d,5)(b,7)}{(c,2)(d,5)(b,7)}$
(a,0) |(b,7), (c,2), (d,5), (e,∞), (f,∞)	$\frac{(e,4)}{(e,6)}$

(a,0), (c,2) | (b,7), (d,5), (e,6), (f,∞)	$\frac{(b,1)(f,12)}{(b,6)(f,17)}$
(a,0), (c,2), (d,5) | (b,6), (e,6), (f,17)	$\frac{(f,17)}{(f,17)}$
(a,0), (c,2), (d,5), (e,6) | (b,6), (f,17)	$\frac{(f,9)}{(f,15)}$
(a,0), (c,2), (d,5), (e,6), (b,6) | (f,15)	Ende

Extract-Max(A)

```
1  if heapsize < 1
2    then error
3  max := A[1]
4  A[1] := A[heapsize[A]]
5  heapsize[A] := heapsize[A] -1
6  heapify(A,1)
7  Return max            → O(log n)
```

Heap-Max(A)

```
1  if heapsize[A] < 1 then error
2  Return A[1]           → O(1)
```

Heap-Insert(A,key)

```
1  heapsize[A] := heapsize[A] + 1
2  i := heapsize[A]
3  while i > 1 and A[parent(i)] < key do
4    A[i] := A[parent(i)}
5    i := parent(i)
6  A[i] := key
```

Heap-Increase-Key(A,I,key)

```
1  if key < A[i] then error
2  A[i] := key
3  while i > 1 and A[parent(i)] < A[i] do
4    tausche A[i] mit A[parent(i)]
5    i := parent(i)
```

2.Variante von Insert:

Heap-Insert(A,key)

```
1  heapsize[A] := heapsize[A] +1
2  A[heapsize[A]] := - ∞
4  Heap-Increase- Key(A, heapsize[A],key)
```

ADT = {Make-Heap, Extract-Min, DecreaseKey}

Allgemeiner = {Make-Heap, Extract-Min, DecreaseKey, Insert, Min}
→ Datenstruktur Priority Queue

Unschön in Literatur:

1) verschiedene Varianten
   a. kleinste Variante: {Make-Heap, Insert, Min, ExtractMin}
   b. umfangreichste Variante: { Make-Heap, Insert, Min, ExtractMin, Decreasekey, Delete, Union}

dagegen wird der Begriff (erstmals 1973) von D. Knuth oder heap für die konkrete Datenstruktur benutzt

Min-heaps ↔ Max-heaps
(DecreaseKey, <) (IncreaseKey, >)

Konkrete Heaps

Operation	Binäry Heap	BinomialHeap	FibonacciHeap
MakeHeap	O(1)	O(1)	O(1)
Max/Min	O(1)	O(log n)	O(1)*
ExtractMax ExtractMin	O(log n)	O(log n)	O(log n)*
Insert	O(log n)	O(log n)	O(1)*
DecreaseKey IncreaseKey	O(log n)	O(log n)	O(1)*
Delete	O(log n)	O(log n)	O(log n)*
Union	O(n)	O(log n)	O(1)*

* amortisiert

Eine konkrete Implementierung eines ADT ist zulässig, genau dann wenn alle Operartionen in O(log n) realisiert sind

4.7 Counting Sort

Definition:

Ein Sortierverfahren heißt stabil, wenn bei gleichen Werten die ursprüngliche Reihenfolge erhalten bleibt,

Definition:

Ein Sortierverfahren heißt inplace, wenn zum Sortieren kein weiterer Speicherplatz benötigt wird.

Rang(x) = #(Werte ≤ x)

CountingSort(A[],B[],k) A[] – InputArray, B[] – OutputArray
Werte in A[] Integer zwischen 0 und k

```
1  for i := 0 to Länge[A] do
2    C[i] := 0
3  for j := 1 to Länge[A] do
4    C[A[j]] := C[A[j]] + 1
5    //C[i] enthält #(Elemente = i)
6  for i := 1 to k do
7    C[i] := C[i] + C[i-1]
8    //C[i] enthält #(Elemente ≤ i)
```

```
9  for j := Länge[A] down to 1 do
10   B[C[A[j]]] := A [j]
11   C[A[j]] := C[A[j]] -1
```

Beispiel:
Input:

A = 2_a 5 3_a 0_a 2_b 3_b 0_b 3_c
1 2 3 4 5 6 7 8

Output:

B= 0_a 0_b 2_a 2_b 3_a 3_b 3_c 5
1 2 3 4 5 6 7 8

Algorithmus:

Nach Zeile 4:

C = 2 0 2 3 0 1
0 1 2 3 4 5

Nach Zeile 7

C= 2 2 4 7 7 8
0 1 2 3 4 5

Nach Zeile 9 – for-Schleife

j = |A| = 8

B[C[A[8]]] = B[C[3]] = B[7]

→ B = 3
1 2 3 4 5 6 7 8

→ C = 2 2 4 6 7 8

…

TIME:

Zeile 1+2: O(k)
+ Zeile 3+4: O(n)
+ Zeile 6+7: O(k)
+ Zeile 9-10: O(n)
= O(n) + O(k) = O(n +k)

4.8 Weitere Sortierverfahren

- Bucketsort
- RadixSort

5 – Elementare Datenstrukturen – Stapel, Bäume und Suchbäume

Operationen auf dynamischen Mengen

Queries	Modifizierende Operationen	Output
SEARCH(Q,k) k - Schlüssel		Zeiger auf Object
	INSERT(Q,x) x – Zeiger auf neues Objekt	
	DELETE(Q,x) (ohne suchen) x – Zeiger auf zu entfernendes Objekt	
MINIMUM(Q)		Zeiger auf Objekt
SUCCESSOR(Q,x)		Zeiger auf Nachfolger des Objekt x

Menge dieser Operationen bilden ADT's

5.1 Binärer Suchbaum

Definition: (Binärer Suchbaum)

Ein binärer Suchbaum ist entweder Ø oder er besteht aus 3 Mengen:

1. Menge, die binärer Suchbaum ist und linker Teilbaum heißt
2. Menge, die binärer Suchbaum ist und rechter Teilbaum heißt
3. EinerMenge (Knoten) aus der Wurzel

Und es gibt die binäre Suchbaumeigenschaft:

Sei x Knoten in einem solchen Baum.

Wenn y Knoten des Linken Teilbaums mit Wurzel x ist, so ist $key[y] \leq key[x]$

Wenn y Knoten des rechten Teilbaums mit Wurzel x ist, so ist $kex[y] > key[x]$

Frage:

Implementierung wofür?

Zusätzliche Forderung: höhenbalanciert (Höhe = O(log n)

Dann Implementierumg für DICTIONARY, geht

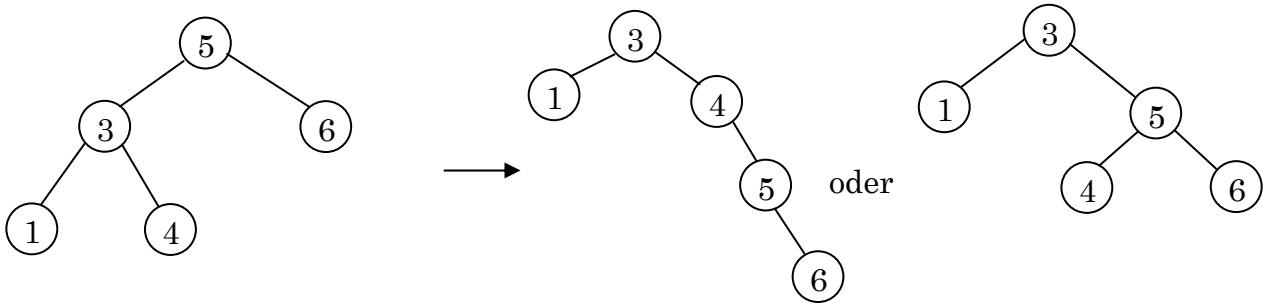
ABER: Nachweis setzt genaue Definition der Balanceeigenschaft voraus

Auch als PriorityQueue nutzbar, geht

ABER: Nachweis über genaue Definition der Balanceeigenschaft

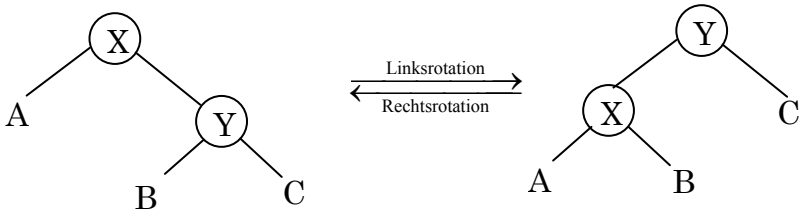
Zu einer Menge von Schlüsseln ist ein binärer Suchbaum nicht eindeutig bestimmt.

Laufzeiten der ADT-Operationen sind meist von der Höhe abhängig.



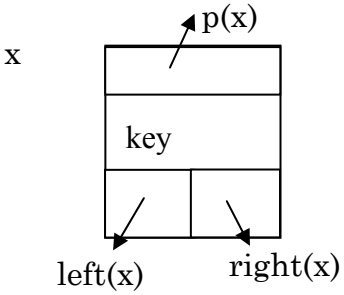
genormte Regeln: Rotationen
Satz:

Rotationen erhalten die binäre Suchbaumeigenschaft



A, B, C – Binäre Suchbäume

Knoten sind Objekte mit Attributen left (li, l,…), right (ri, r,…), p (parent, vater), key und eventuell weitere (Satellitendaten)



Traversierung von Bäumen

TREE-POST-ORDER(x)

```
1  if x != NIL
2    then      TREE-POST-ORDER(left(x))
3         TREE-POST-ORDER(right(x))
4         Print key(x)
```

TREE-PRE-ORDER(x)

```
1  if x != NIL
2    then      TREE-PRE-ORDER(left(x))
3         Print key(x)
4         TREE-PRE-ORDER(right(x))
```

TREE-IN-ORDER(x)
```
1  if x != NIL
2    then      Print key(x)
3         TREE-IN-ORDER(left(x))
4         TREE-IN-ORDER(right(x))
```

Satz:

Um eine Folge paarweise verschiedener Zahlen in einem Binärbaum zu bringen benötigt man Ω(n log n) Zeit.

Beweis:

IN-ORDER

TREE-MIN(x)
```
1  while left(x) != NIL do
2    x := left(x)
3  print x
```

TREE-SEARCH(x,k)
```
1  if x = NIL or k = key(x)
2    then return x
3  if k < key(x)
4    then TREE-SEARCH(left(x),k)
5  else TREE-SEARCH(right(x),k)
```

TREE-SUCCESSOR(x)
```
1  if right(x) != NIL
2    then return TREE-MIN(right(x))
3  y := p(x)
4  while y != NIL and x = right(y) do
5    x := y
6    y := p(y)
7  return y
```

TREE-DELETE(T,z)
```
1  if left(z) = NIL or right(z) = NIL //y wird später rausgeworfen
2    then y := z
3  else y := TREE-SUCCESSOR(z)
4  if left(z) != NIL
5    then x := left(y)
6  else x := right(y)
7  if x != NIL
8    then tausche p(x) mit p(y)
9  if x = NIL
10   then root(T) := x
11 else if y = left(p(y))
12        then left(p(y)) := x
13     else right(p(y)) := x
14 if y != z
15   then key(z) := key(y)
16 return y
```

TIME: $\Theta(h)$ wegen Zeile 3

Bemerkung:
Bei höhenbalancierten Bäumen ist die Höhe h = log n

5.2 Stack

ADT Stapel (Stack, Keller)
{MAKE-STACK, STACK-EMPTY, PUSH, POP}

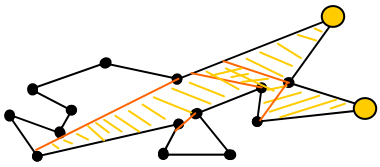
Implementierung:
Array S[1…n] - ist nicht der Stack selbst
Attribut: top[S]
→ Stack ist Teilarray S[1…top[S]] mit top(S) ≤ n

STACK-EMPTY
```
1  if top[S] = 0
2    then return true
3  else false
```

PUSH(S,x)
```
1  top[S] := top[S] + 1
2  S[top[S]] := x
```

POP(S)
```
1  if STACK-EMPTY(S)
2    then error
3  else top[S] := top[S] - 1
4    return S[top[S]+1]
```

Stack beim Algorithmenentwurf



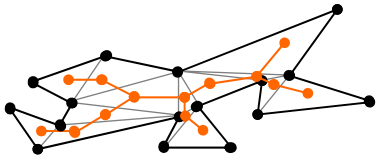
1973 Victor Klee: Art Gallery Problem

→ bei n Ecken, wie viele Lichtquellen reichen und sind im worst case auch nötig?

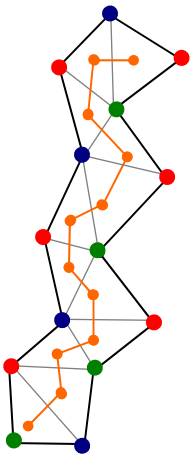
Chratal: Lösung: (5 Seiten Beweis) $\left\lfloor \frac{n}{3} \right\rfloor$ Lichtquellen

1975 Fisk:

Satz:
Jedes einfache Polygon mit n Ecken lässt sich triangulieren. (ohne Steinerpunkte)



Dualer Graph – Baum



Färben die Knoten 3-Färbung

$$\rightarrow \left\lfloor \frac{n}{3} \right\rfloor = \left\lfloor \frac{14}{3} \right\rfloor = 4$$

Definition:

Ein Polygon P heißt monoton bezüglich einer Geraden L, wenn die Projektion der Ecken von P auf L der Reihenfolge in P entspricht.

Definition:

P heiße monoton wenn eine Gerade L existiert, so dass P monoton bezüglich L ist.

Satz:

Ein Polygon P lässt sich in O(n log n) TIME triangulieren.

Beweis:

Algorithmus 1:

Zerlege P in bezüglich der y-Achse monotone Polygone.

(Lösung: in O(n log n) mit Gleitgeradenmethode)

Algorithmus 2:

Trianguliere die monotone Teilpolygone (in O(n) TIME)

1. Sortiere die Punkte $p_0, \ldots, p_n$ nach fallenden y-Werten.

(merge zweier sortierten Folgen)

o.B.d.A. allgemeine Lage: y-Werte aller Punkte paarweise verschieden

Also nach Sortierung: o.B.d.A. $p_0, \ldots, p_n$ bezüglich fallender y-Werte sortiert

Stack: $v_0, \ldots, v_t$

Die folgenden Invarianten:

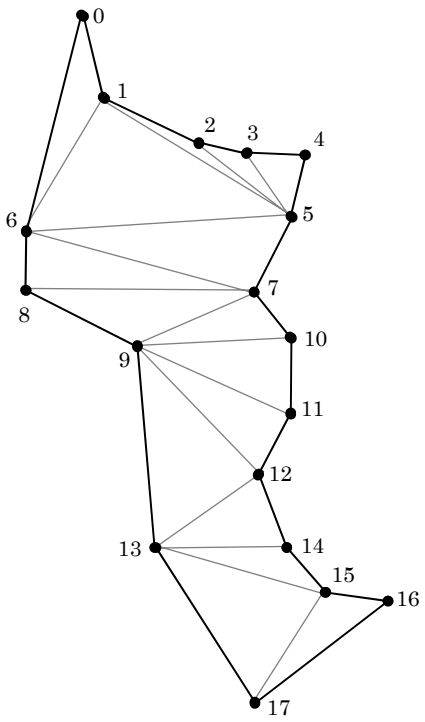
1. $v_0, \ldots, v_t$ ist fallend bezüglich y-Werte, $v_0, \ldots, v_t \in \{ p_0, \ldots, p_n\}$
2. $v_0, \ldots, v_t$ bilden Kette auf dem Rand des Polygons P_i (= P nach Schritt i)
3. $v_1, \ldots, v_{t-1}$ sind Reflexecken (Innenwinkel > 180°)
4. die folgende Ecke bei der Bearbeitung von P_i ist v_0 oder v_t

Algorithmus:

(sortiere die Eckpunkte nach fallenden y-Werten, Ergebnis: $p_0, \ldots, p_n$)

```
1  PUSH p0 (→v0)
2  PUSH p1 (→vt)
3  For i = 2 to n – 1 do
4    If pi is adjacent to v0 then
5         Begin          (sicher vt als v*)
6              While t > 0 do
7                   Begin
8                        Ziehe Diagonale [pi – vt]
9                        POP
10                  End
11             POP
12             PUSH vt ( = v*)
13             PUSH pi
14        End
15   Else if pi is adjacent to vt then
16        Begin
17             While t > 0 and vt ist nicht Reflexecke do
18                  Begin
19                       Ziehe Diagonale [pi – vt-1]
20                       POP
21                  End
22             PUSH pi
23        END
```

Beispiel:



i	Stack	Diagonale
2	0,1	
3	0,1,2	
4	0,1,2,3	
5	0,1,2,3,4	[5-3] [5-2] [5-1]
6	0,1,5	[6-5] [6-1]
7	5,6	[7-6]
8	6,7	[8-7]
9	7,8	[9,7]
…	…	…

Zu i = 6:
0,1,5 - [6-5]
POP
0,1 – [6-1]
POP
0
POP
Leer
PUSH 5
PUSH 6

Wir ersetzen die Einzel worst Case Analyse durch eine amortisierte Kostenanalyse

Für jeden Knoten gilt, er kommt maximal 2 mal in den Stack
→ Gesamtaufwand bleibt unter 2n → O(n)

Ein weiteres Beispiel zur amortisierten Kostenanalyse

Ω(n log n) für CH (convex hull)

Computational geometry seit 1973 (Dissertation M. Shamos)
1972 Graham Algorithmus für CH
1979 Andrew besseren Algorithmus

INPUT: $\{p_1,\ldots,p_n\} \subseteq \mathbb{R}^2$

Punkt durch (p_{ix},p_{iy}) charakterisiert

1. SORT berzüglich x-Koordinate $q_1,\ldots,q_{rechts}$ → Ω(n log n)
   // oBdA Algorithmus für UH
   // oBdA linkester und rechtester Punkt auf x-Achse
   // oBdA $q_1,\ldots,q_n$ neuer INPUT, sortiert bezüglich x-Wert

Algorithmus von Andrew:

Stack → Invarianten: Stack $x_0,\ldots,x_t$ s, aktueller Punkt q_{s+1}

1) $t \geq 2$, $s \geq 2$, $x_0 = q_n$, $x_1 = q_1$, $x_2 = q_s$
2) $x_1,\ldots,x_t$ ist UH (upper hull) von $q_1,\ldots, q_s$
3) $x_1,\ldots,x_t$ sind sortiert bezüglich der x-Werte, $x_i = (x_{ix},x_{iy})$ vollwertiger Punkt

Start:

```
1  PUSH q_n, PUSH q_1, PUSH q_2
2  s := 2
3  While s != n do
4    α := top-element vom Stack
5    β := second element vom Stack
6    While (q_{s+1},α,β) ist keine Linksdrehung do
7         POP
8         α := β
9         β := neues second element vom Stack
10   PUSH (q_{s+1})
11   s := s +1
12 gib Stackinhalt aus
```

Anmerkung Links- und Rechtsdrehung:

$$\det\begin{pmatrix}1 & q_{1x} & q_{1y}\\ 1 & q_{2x} & q_{2y}\\ 1 & q_{3x} & q_{3y}\end{pmatrix}\begin{matrix}>0\rightarrow \text{Linksdrehung}\\ <0\rightarrow \text{Rechtsdrehung}\end{matrix}$$

Multipop(S,k) S – Stack k – Anzahl POPs

Worst-case:

Ω(n) Arbeit für einen Punkt
Ω(n·n) = Ω(n²) für alle Punkte

5.2.1 Amortisierte Kostenanalyse:

3 Arten:

1 Aggregatsmethode
2 Acounting method (Guthabenmethode)
3 Potentialmethode

5.2.1.1 Aggregatsmethode

Man betrachtet die Folge aller n Operationen.
Berechnet die Gesamtzeit T(n) und bildet den Quotienten $\frac{T(n)}{n}$

→ Amortisierten Kosten pro Operation

Am Beispiel:

$T(n) \leq 2n$

Also pro Operation Kosten ≤ 2

5.2.1.2 Guthabenmethode

i-te Operation (=Operation bei Behandlung des i-ten Punktes)
$\hat{c}_i$ – amortisierten Kosten, c_i – aktuelle Kosten

$\sum_{i=1}^{n}\hat{c}_i$ – amortisierte Gesamtkosten, $\sum_{i=1}^{n}c_i$ – aktuelle Gesamtkosten

Ziel:

Aktuelle Gesamtkosten ≤ amortisierte Gesamtkosten ≤ O(?)

Amortisierte Kosten am Beispiel:

$$\hat{c}_i=\begin{cases}2 & \text{i-te Operation PUSH}\\ 0 & \text{i-te Operation POP, Multipop}\end{cases}$$

n-mal PUSH: $\sum_{i=1}^{n}\hat{c}_i = 2n \leq O(n)$

5.2.1.3 Potentialmethode:

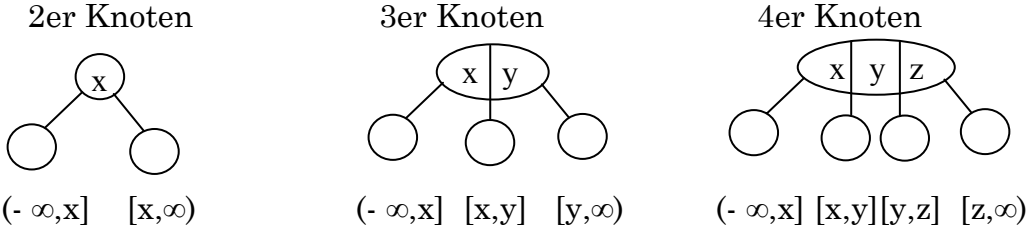
Datenstruktur bekommt Wert als Potential zugeordnet.
$\Phi(D_i)$ – Potential nach i-ter Operation

$\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$

Potential ist Anzahl der Elemente im Stack

5.3 Höhenbalancierte Suchbäume

Top-Down-2-3-4-Bäume – ideal ausbalanciert



Der Einfachheit halber: Schlüssel paarweise verschieden

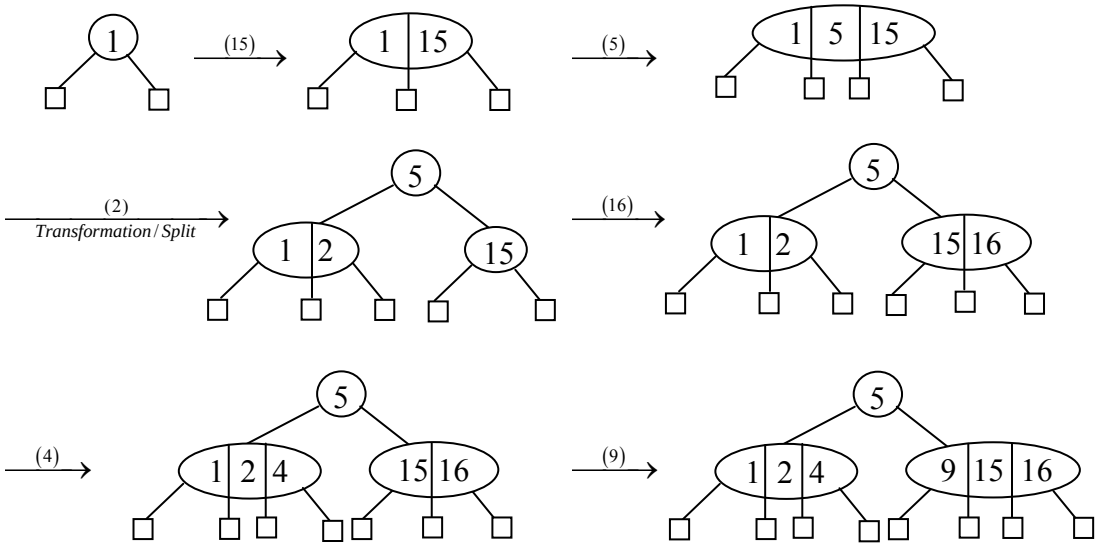
□ - dummy Elemente (Objekte) (Scheinobjekte, Markenelemente)

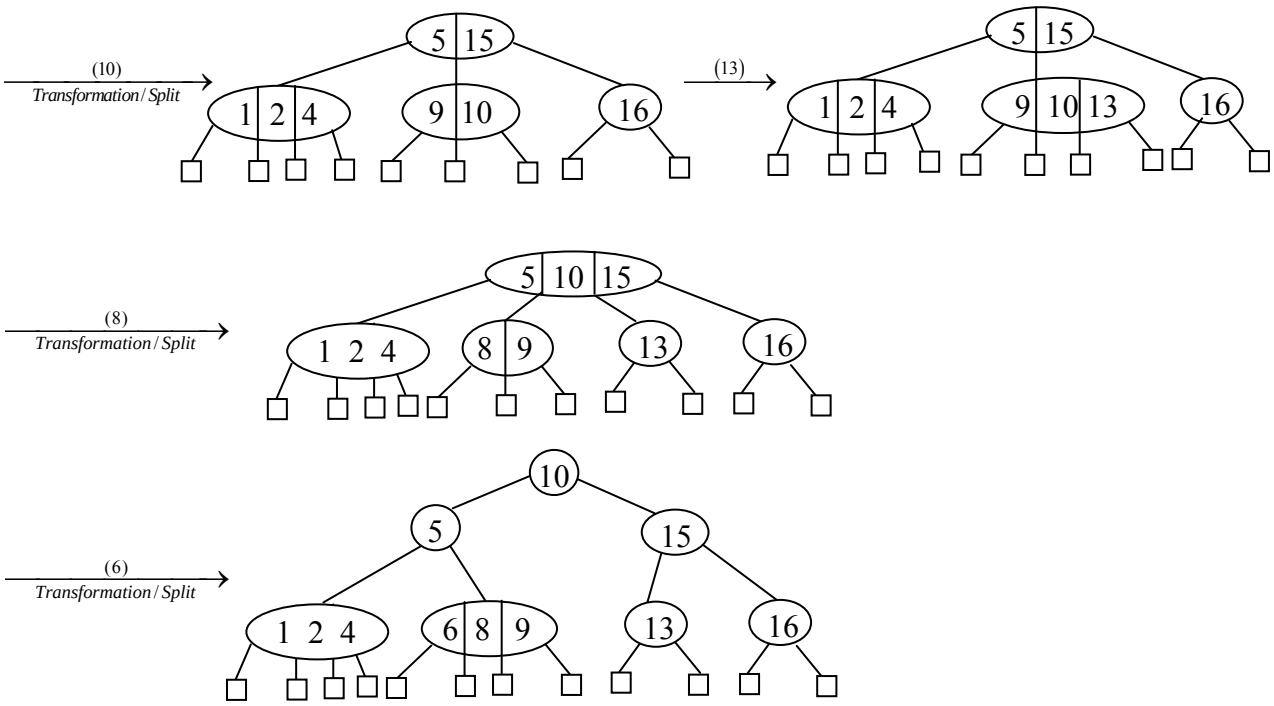
1 15 5 2 16 4 9 10 13 8 6

Regel:

Jedesmal dann, wenn bei der Suche ein 4er Knoten angetroffen wird, so wird dieser sofort aufgesplittet. („Top-Down")

Beim Einfügen wird der Wert gesucht und wenn ein NIL getroffen wird, wird eingefügt.





Satz:

Der Top-Down-2-3-4-Baum wird genau dann in der Höhe um 1 erhöht, wenn die Wurzel aufgesplittet wird.

Satz:

Die Höhe eines Top-Down-2-3-4-Baumes mit n Knoten ist stets O(log n)

Beweis:

Ideal ausbalanciert, ideal ausbalancierter Baum der Höhe h.
Ein solcher 2-3-4-Baum hat höchstens so viele Knoten wie ein solcher Binärbaum.
D.h. bei vorgegebener Knotenzahl ist die Höhe des 2-3-4-Baumes ≤ Höhe des Binärbaums. → Behauptung □

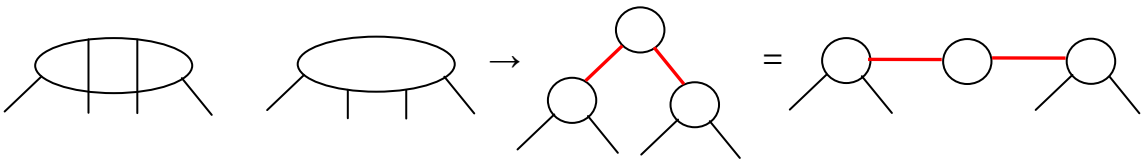
Problem:

Schwierige Implementierung

Lösung:

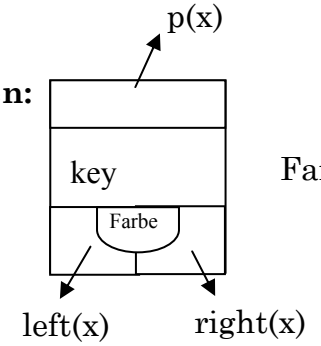
Darstellung der Top-Down-2-3-4-Bäume als RotSchwarzBaum (RB-Tree)

Idee:



5.4 Rot-Schwarz-Bäume (RB-Trees)

Knoten:



Farbe $\in$ {rot,schwarz}

Definition:

Ein binärer Suchbaum ist ein RB-Tree, falls folgende Eigenschaften gelten:

1) Jeder Knoten ist rot oder schwarz
2) Root ist schwarz
3) Blätter sind schwarz
4) Kein roter Vater darf einen roten Sohn haben
5) Die Schwarztiefe (=bh) ist eindeutig

Definition (Schwarztiefe):

Die Schwarztiefe eines Knotens x ist die Anzahl der schwarzen Knoten auf den Pfaden von x zu einem Blatt, x nicht mitgerechnet.

Hauptsatz:

Die Höhe eines RB-Tree mit n Knoten ist $\leq 2 \cdot \log (n+1) = O(\log n)$

Beweis:

Hilfssatz:

In einem RB-Tree hat jeder Knoten x einen an ihm hängenden Teilbaum mit mindestens $2^{bh(x)} - 1$ innere Knoten.

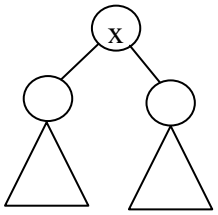
Induktiv über h (Höhe)

Induktionsanfang:

Höhe(x) = 0

$2^0 - 1 = 0$ innere Knoten

Induktion:



$$= \begin{cases} bh(x) - 1 & \text{wenn Sohn(x) = schwarz} \\ bh(x) & \text{wenn Sohn(x) = rot} \end{cases}$$

beides möglich $\geq bh(x) - 1$

Induktionsvoraussetzung:

Innere Knoten in den Teilbäumen $\geq [(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1)] + 1 = 2^{bh(x)} - 1$

Sei n Zahl der inneren Knoten

Aus (4.) folgt:
Lemma:

$\text{bh(root)} \geq \frac{h}{2}$, h – Höhe des Baumes

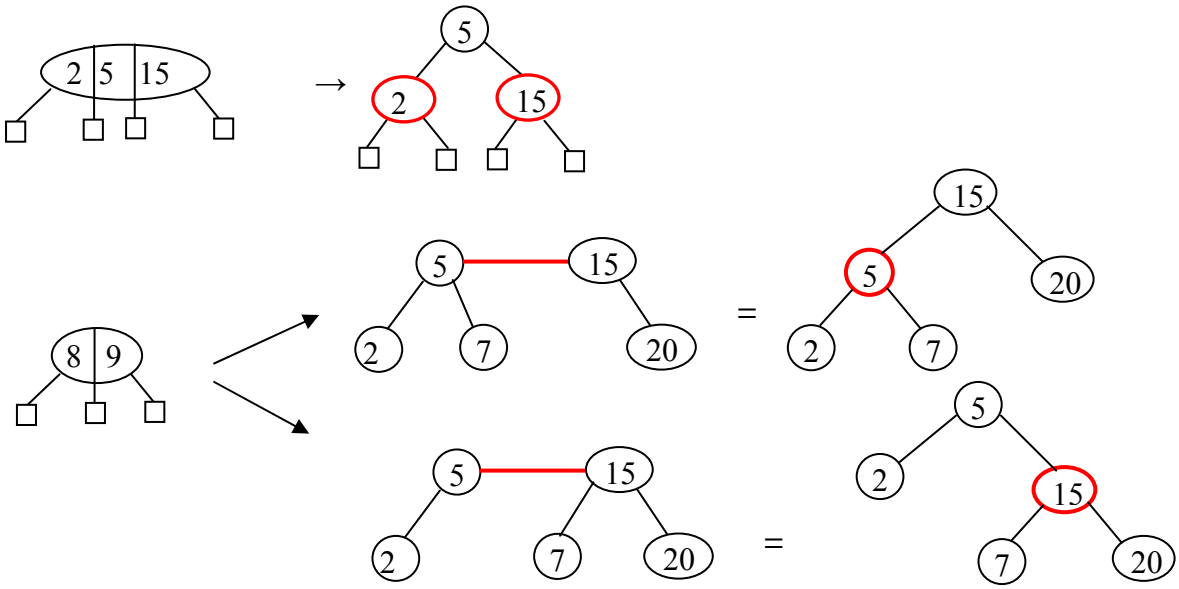
→ worst case:

Höhe Pfad-root-Blatt r s r s r s r s … s

→ Zahl der schwarzen Knoten $\geq \frac{h}{2}$

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

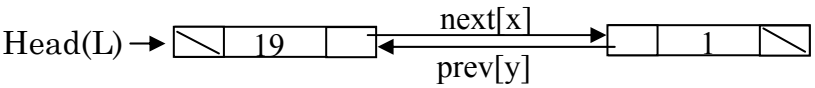
$\rightarrow n + 1 \geq 2^{\frac{h}{2}} \rightarrow \log(n+1) \geq \frac{h}{2} \rightarrow h \leq 2 \cdot \log(n+1)$ □



Beide Varianten gleichberechtigt!

5.4.1 Sentinels (Schildwache, Posten)

(Dummy Object)

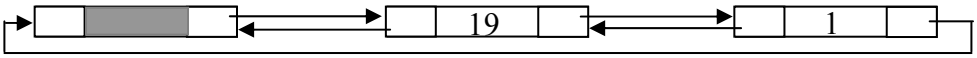


List-Delete(L,x)

```
1  if prev[x] != NIL
2    then next[prev[x]] := next[x]
3  else head[L] := next[x]
4  if next[x] != NIL
5    then prev[next[x]] := prev[x]
```

Langes Programm weil Randbedingungen zu beachten sind…

Dummy object



nil[L]

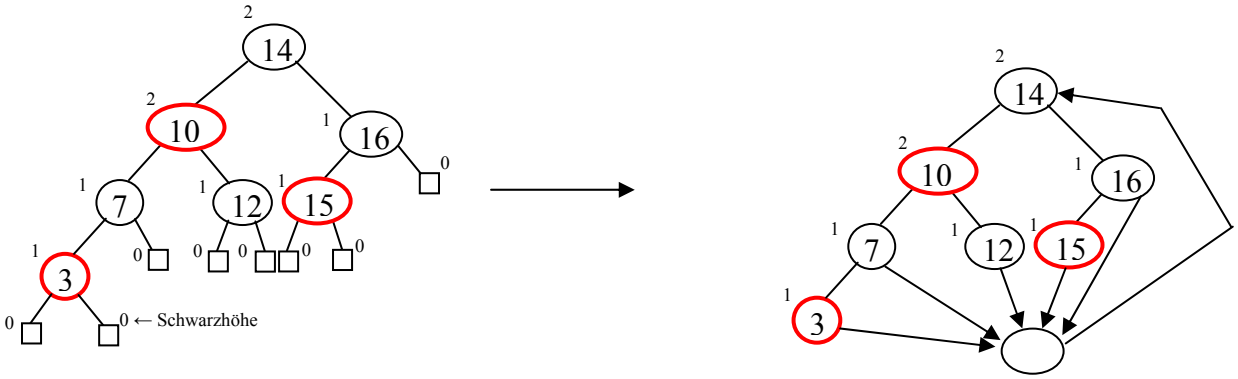
Attribut des
Objekts Liste

→ damit wird head[L] überflüssig

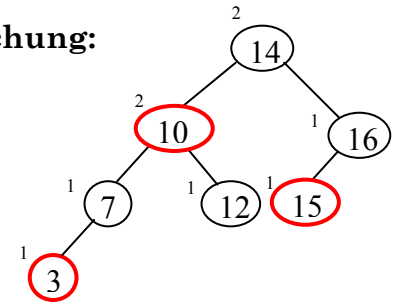
List-Delete (L,x)

```
1  next[prev[x] := next[x]
2  prev[next[x] := prev[x]
```

Darstellung von RB-Trees mit einem Sentinel nil[T]



Vereinfachung:



RB-INSERT(T,z)

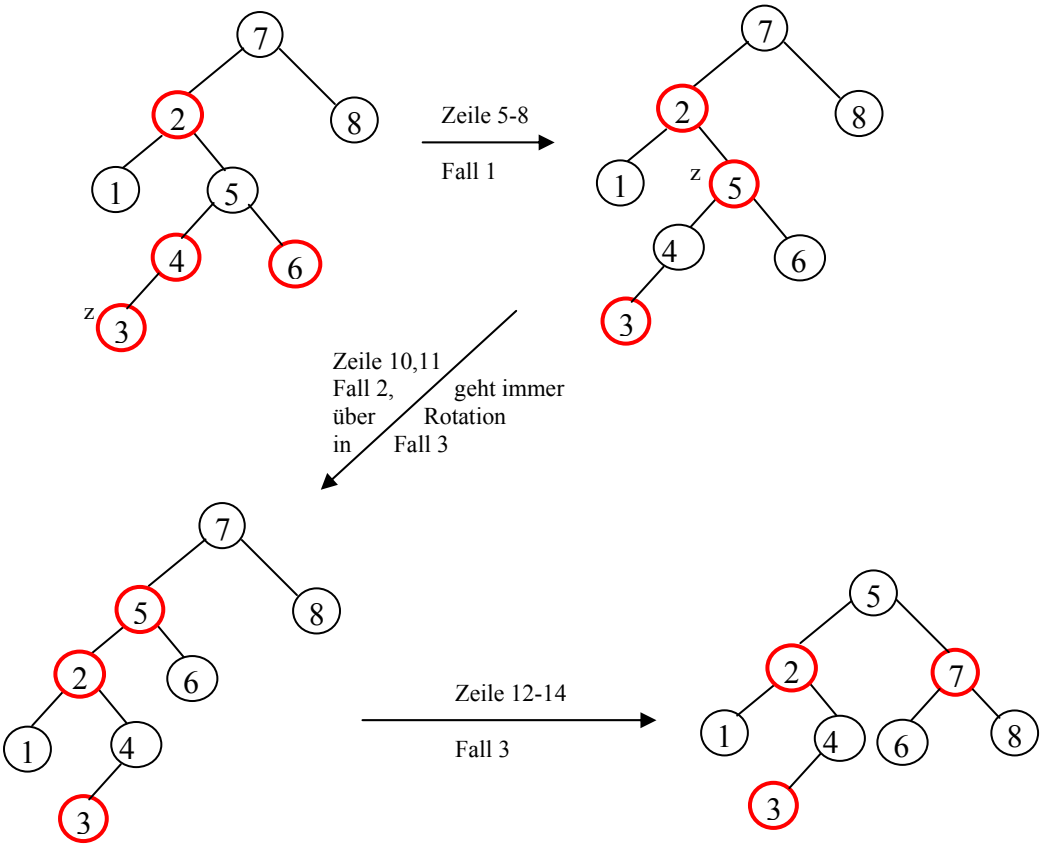
```
1   y := nil[T]
2   x := root[T]
3   while x != nil[T] do
4    y := x
5    if key[z] < key[x]
6         then x := left[x]
7    else x := right[x]
8   p[z] := y
9   if y = nil[T]
10   then root[T] := z
11 else if key[z] < key[y]
12   then left[y] := z
13 else right[y] := z
14 left[z] := nil[T]
15 right[z] := nil[T]
16 color[z] := ROT
17 KORRIGIERE(T,z)
```

KORRIGIERE(T,z)

```
1  while color[p[z]] = ROT do
2    if p[z] = left[p[p[z]]]
3         then y := right[p[p[z]]]
4         if color[y] = ROT
5              then color[p[z]] := SCH
6              color[y] := SCH
7              color[p[p[z]] := ROT
8              z := p[p[z]]
9         else if z = right[p[z]]
10             then z := p[z]
11                  leftrotate(T,z)
12             color[p[z]] := SCH
13             color[p[p[z]]] := ROT
14             rightrotate(T,p[p[z]])
15   else then y := left[p[p[z]]]
16        if color[y] = ROT
17             then color[p[z]] := SCH
18             color[y] := SCH
19             color[p[p[z]] := ROT
20             z := p[p[z]]
21        else if z = left[p[z]]
22             then z := p[z]
23                  rightrotate(T,z)
24             color[p[z]] := SCH
25             color[p[p[z]]] := ROT
26             leftrotate(T,p[p[z]])
27      color(root[T]) := SCH
```

Beispiel:

Nach RB-INSERT

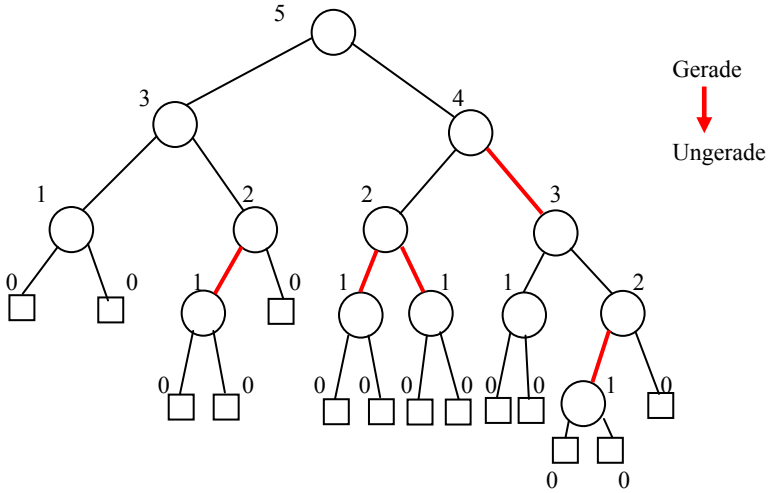


5.5 Bemerkung zu AVL-Bäumen:

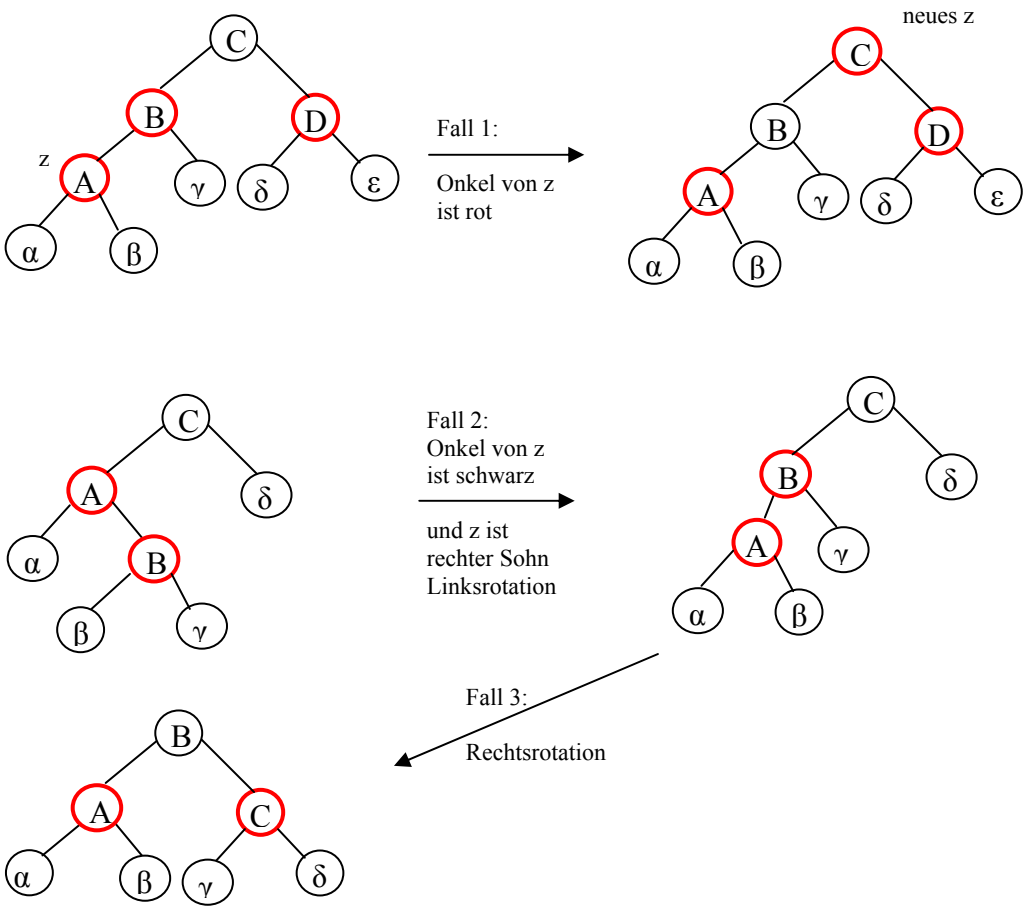
1986 Tarjan Ottmann Kantenrot RB-Tree

Behauptung:

Jeder AVL-Baum lässt sich durch eine einfache Transformation in einem Ottmann-RB-Tree (damit in einen RB-Tree) überführen, der aus einem TopDown-2-3-4-Baum hervorgeht.



Nachbemerkungen zu Insert in RB-Trees



Behauptung:

Die Schwarzhöheneigenschaft (= 5. Eigenschaft) bleibt dabei erhalten.

Die Tücken praktischer Aufgabenstellungen

(manchmal muss man eine Standardstruktur zurecht machen)

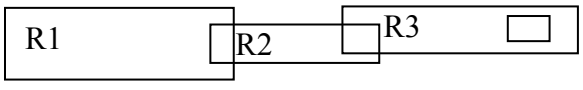
INPUT:

Menge von Rechtecken der Ebene (Ortho-Rechtecke)

OUTPUT:

Berichte alle Überschneidungen

1. Teilantwort:
Segmentschnittalgorithmus:



Start:

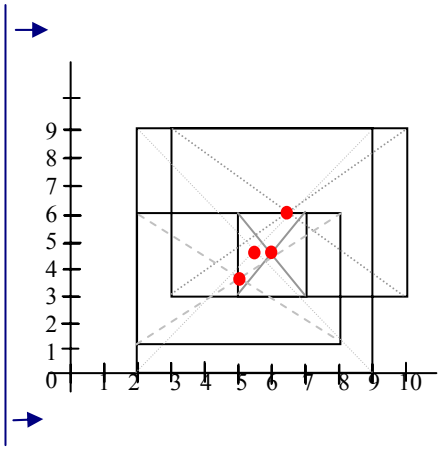
Y := ∅

Y – verwaltet aktive Rechtecke (Rest ist schlafend oder tot)

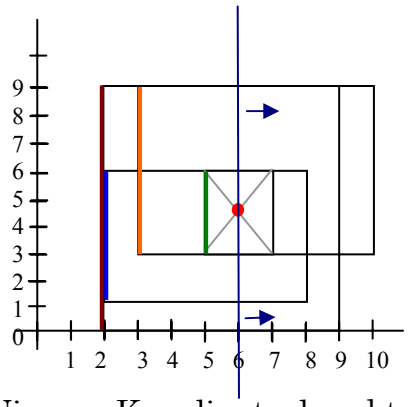
Gleitgerade:

Eventpoints:

x-Koordinate der vertikalen Reckeckkanten
x-Koordinate der roten Punkte



Fall: Linke Rechteckkante: Insert in Y
Fall: Rechte Rechteckkante: Delete in Y
Fall: x-Koordinate eines roten Punkts



Nimm y-Koordinate des aktuellen Punktes, suche y-Koordinate in der Menge der aktiven Rechtecke → Output

Problem:

Komplexität

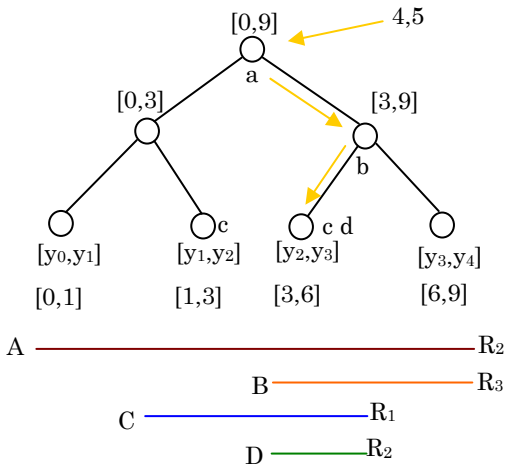
Frage:

Wie wird Y verwaltet?

→ Ziel: höhenbalancierter Suchbaum, wo Segmente verwaltet werden.

→ Bentley u.a. Segment-Trees

INPUT liefert ein Raster
y_0, y_1, y_2, y_3, y_4 Baum mit leeren Inhalt



Regel:
Eintragen so weit oben wie möglich

Suche y-Koordinate des aktuellen roten Punktes (x,y) (→4,5) im Baum und notiere alle angetrofffenen Eintragungen

Das liefert a,b,c,d
→ Output:
(d,a), (d,b), (d,c)
d.h. $R_4 \cap R_1 \neq \varnothing$
$R_4 \cap R_2 \neq \varnothing$
$R_4 \cap R_3 \neq \varnothing$

Behauptung:
Insert, Delete und Search kosten jeweils O(log n)

5.6 Prioritätswarteschlangen (mergeable heaps)

Vuillemin 1978

Verwaltung von Mengen

Beispiel:
Algorithmus von Dijkstra

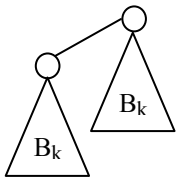
Merge, union, meld wird bei Binärheaps nicht unterstützt

Definition: (mergeable heaps)
- ADT aus den Operationen {MAKE-HEAP, INSERT(H,x), MINIMUM(H), EXTRACT-MIN(H), UNION(H_1,H_2),} $\subseteq$ ADT
- Weitere möglich: DECREASE-KEY(H,x,key), DELETE(H,x)
- SEARCH im Allgemeinen nicht unterstützt

5.7 Binomialheaps

Definition:

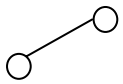
IA: ○ sei B_0 (Binomialbaum)

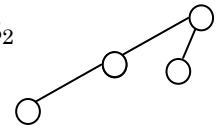
IS: 

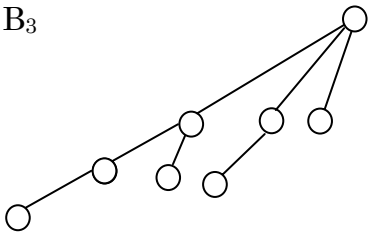
sei B_{k+1} ($k \geq 0$)

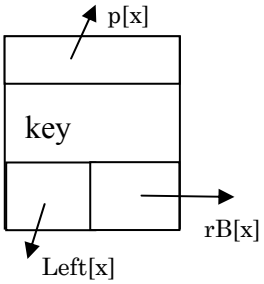
Sonst nichts!

Beispiel:

B_1 

B_2 

B_3 

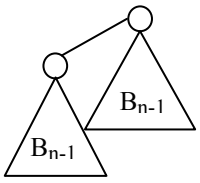


Satz:

Für einen B_n gilt:

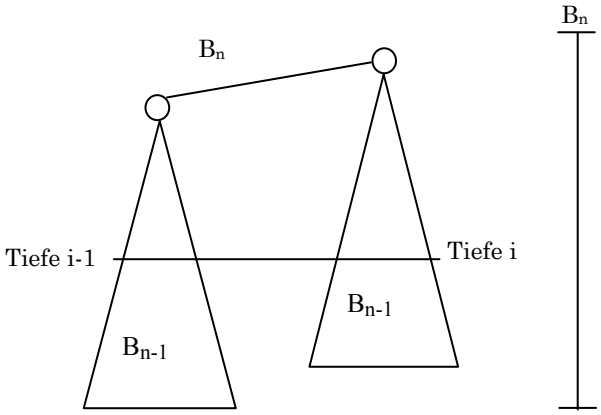
1) Er hat 2^n Knoten
2) Seine Höhe ist n
3) Es gibt genau $\binom{n}{i}$ Knoten der Tiefe i (i = 0,1,…,n)
4) Die Wurzel hat den Grad n, der größer als der aller anderen Knoten ist. Bei der Nummerierung der Söhne von Rechts nach Links durch n-1, n-2,… so ist der ite Sohn Wurzel eines B_i

Beweis:

1) B_n = 

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

3)



Definition:
d(n,i) := Anzahl der Knoten in B_n in der Tiefe i
d(n-1,i) + d(n-1,i-1)

IV: $\binom{n-1}{i}+\binom{n-1}{i-1}=\binom{n}{i}$

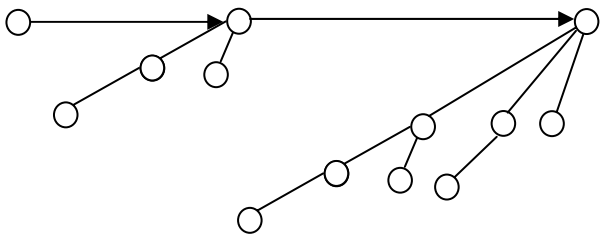
Definition: (Binomialheap)
H heiße Binomialheap wenn er ein Wald von Binomialbäumen ist:

1) für kein n existieren ≥ 2 Exemplare des B_n
2) Alle Bäume sind min-heap-geordnet

Implementierung
1) Linker-Sohn-Rechter-Bruder-Darstellung
2) Wurzeln in einer einfach verketteten Wurzelliste verwaltet, deren Knoten nach wachsenden Degree geordnet sind

Beispiel:

n = 13



$B_3B_2B_1B_0$
$13_{10} = 1101_2$

m – Knoten → Dualzahl m → eindeutige Struktur

→ Zugriff auf das Minimum dauert O(log n)
Beweis: Durchlaufen der Wurzelliste der Länge O(log n)
→ Zeit für Union ist O(log n)
Beweis: Dualzahladdition

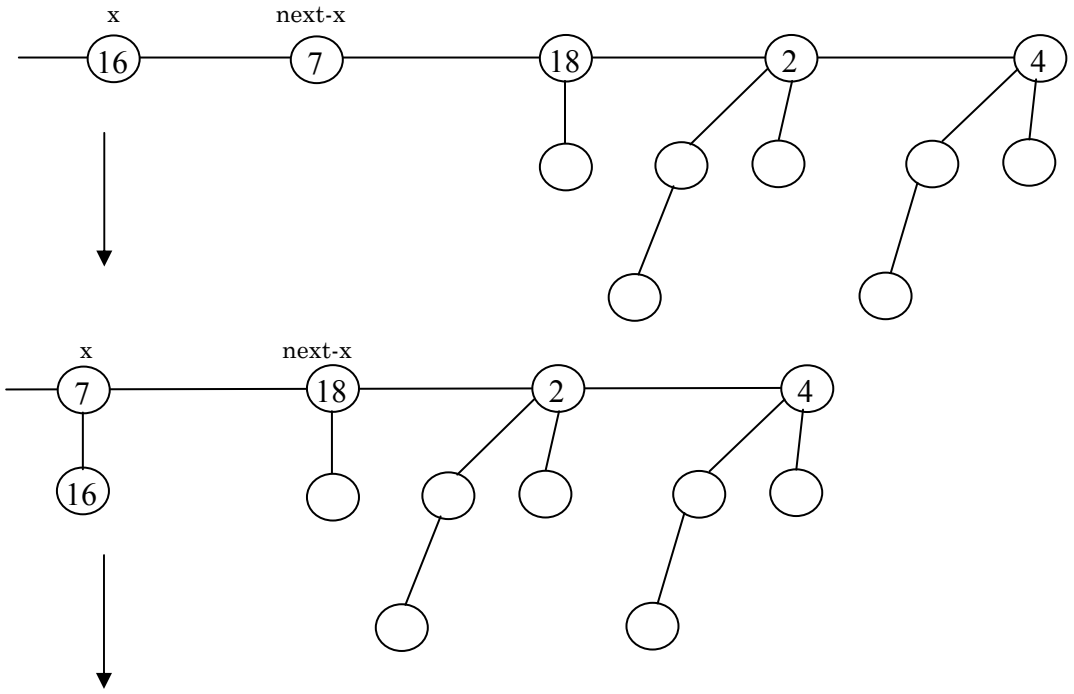
Binomialheap-UNION(H_1,H_2)

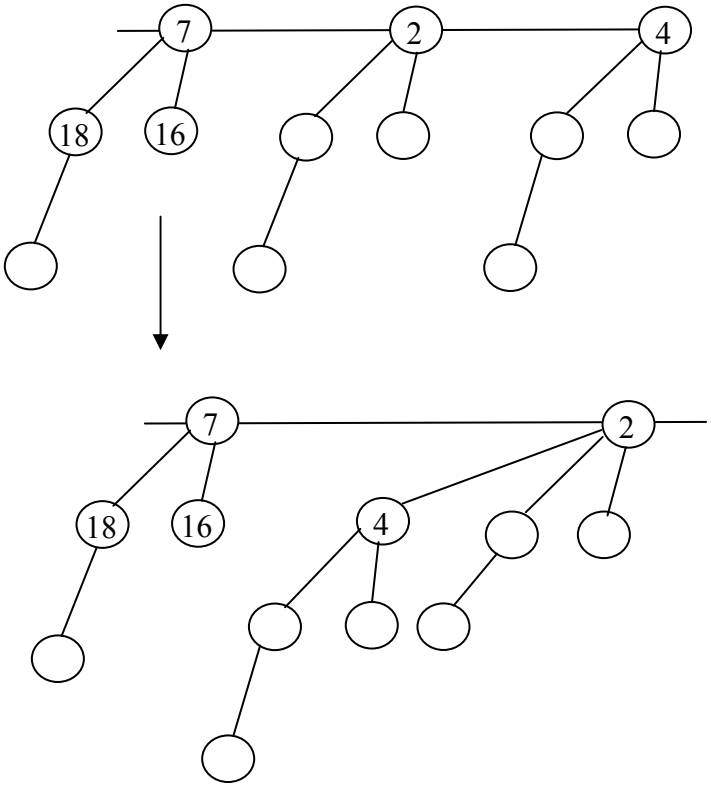
```
1  H:= MAKE-BINOMIAL-HEAP
2    head[H]  :=  Binomial-merge(H1,H2)  //gemeinsam  sortierte
Wurzelliste
3  Zerstöre H1, H2 als Objekte, aber nicht Wurzellisten
4  if head[H] = NIL
5    then return H
6  prev-x := NIL
7  x := head[H]
8  next-x := rB[x]
9  while next-x != NIL do
10   if (degree[x] != degree[next-x]) oder (rb[next-x] != NIL
                          und degree[rB[next-x] = degree[x])
11         then prev-x := x
12              x := next-x
13   else if key[x] ≤ key[next-x]
14                then rB[x] := rB[next-x]
15                   Binomial-Link(next-x,x)
16           else if prev-x = NIL
17                  then head[H] := next-x
18             else rB[prev-x] := next-x
19             Binomial-Link(x,next-x)
20              x := next-x
21   next-x := rB[x]
22 return H
```

Bemerkung:

Next-x, prev-x keine Zeiger sondern einfach Variablen

Beispiel:





Objekt Binomialheap:

Attribute: *[ ]

ADT:

mergeable heap = {MAKE-HEAP, INSERT(H,x), MINIMUM(H),
EXTRACT-MIN(H), UNION(H_1,H_2),}

Wald von Bäumen:

1) Minheap-Eigenschaft
2) Einzigkeit

→ Zugriff auf Min in O(log n)

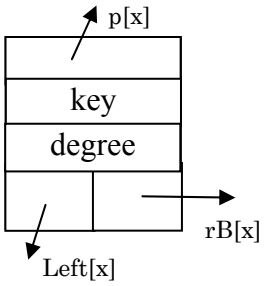
Binomialheap-MIN(H)

```
1  y := NIL
2  x := head[H]
3  min := +∞
4  while x != NIL do
5    if key[x] < min
6         then min := key[x]
7                   y := x
8    x := rB[x]
9  return y
```

Satz:

Zugriff auf Min in O(log n) Zeit

Neu: x



Binomial-Link(y,z)

```
1  p[y] := z
2  rB[y] := left[z]
3  left[z] := y
4  degree[z] :=  degree[z]+1
```

→ O(1)

MAKE-Binomial-HEAP()

```
1  head[H] := NIL
```

Binomialheap-INSERT(H,x)

```
1  H' := MAKE-Binomial-HEAP()
2  p[x] := NIL
3  left[x] := NIL
4  rB[x] := NIL
5  degree[x] := 0
6  head[H'] := x
7  H := Binomialheap-UNION(H,H')
```

→ O(log n)

Binomialheap-EXTRACTMIN(H)

1 finde die Wurzel x in Wurzelliste mit Minimalkey und entferne x aus der Wurzelliste H
2 H' := MAKE-Binomial-HEAP()
3 kehre die Ordnung der Söhne um und setze head[H'] auf Kopf der so entstandenen Liste
4 H := Binomialheap-UNION(H, H')
5 return x

5.8 Fibonacci-Heaps (F-Heaps)

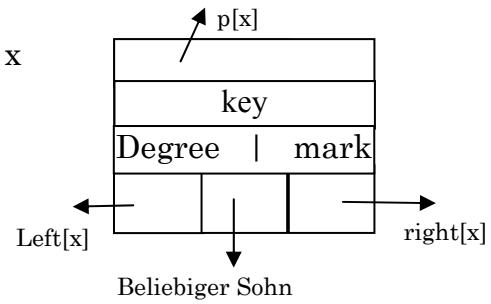
Prozedur	Binary Heap	Binomialheap	Fibonacci-Heap
MAKE-HEAP	O(1)	O(1)	O(1)*
INSERT	O(log n)	O(log n)	O(1)
MIN	O(1)	O(log n)	O(1)*
EXTRACTMIN	O(log n)	O(log n)	O(log n)
UNION	O(n)	O(log n)	O(1)*
DECREASEKEY	O(log n)	O(log n)	O(1)
DELETE	O(log n)	O(log n)	O(log n)

Potentialmethode führt zur amortisierten Kostenanalyse

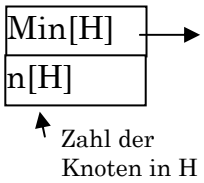
Bemerkung:

Wenn keine DECREASEKEY und keine allgemeine Delete-Operationen, so Wälder von Bäumen, die ähnlich den Binomialbäumen sind.

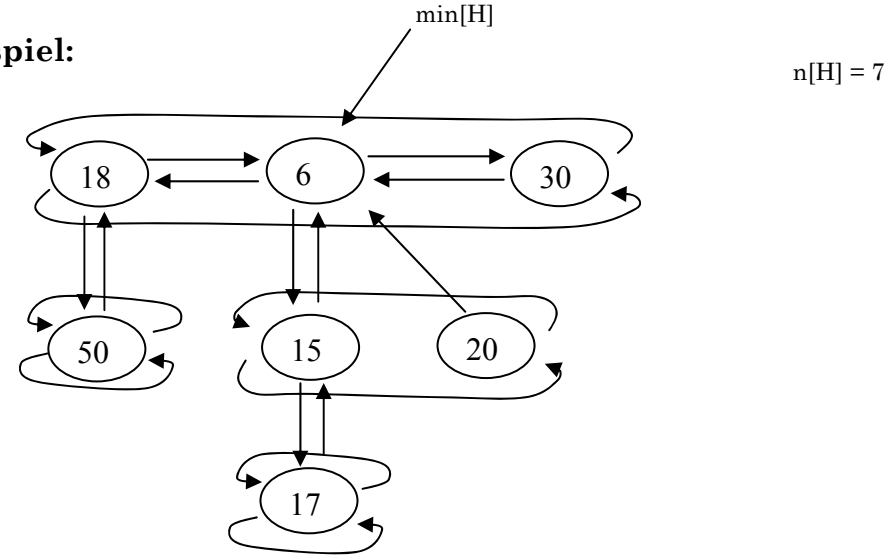
Objekte in F-Heaps



F-Heap als Objekt



Beispiel:

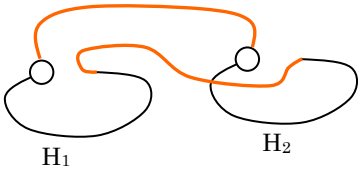


Bemerkung:

Markierung ist dazu da, damit das Abtrennen von Söhnen vom Vater gemerkt wird und die Bäume nicht zu breit werden.

n[H] ist notwendig für Minimum (EXTRACTMIN)

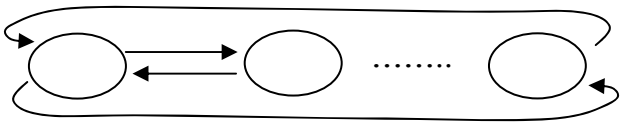
F-UNION(H_1,H_2)
1 Schneide die Wurzelliste von H_1 nach min(H_1) auf
2 Schneide H_2 entsprechend auf
3 Verknüpfe
4 aktualisiere min[H]
→ O(1)



Extract-Min:

Sowohl bei Binomialheaps als auch bei F-Heaps: O(log n)

Insert, Insert ….



Bei Extract-Min „konsolidieren“. Bezahlt mit Potential.
→ Andrew-Algorithmus: Potentialmethode Φ(Stack) := #Elemente im Stack

Definition hier:

$\Phi(H) := t(H) + 2\,m(H)$

Mit t(H) – Anzahl der Knoten in Wurzelliste und m(H) Anzahl der markierten Knoten
Ziel: Degreebeschränkung
→ Spielraum bei den Faktoren z.B. 2t(H) + 4m(H) (mit größeren Werten lassen sich Konstanten in O(n) abfedern)

Wiederholung:

Potentialmethode

Ziel:

Obere Schranke für die aktuellen Kosten $\sum_{i=1}^{n} c_i$

Weg:

Obere Schranke für die amortisierten Kosten $\sum_{i=1}^{n} \hat{c}_i$

Sinnvoll:

Forderung: $\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i$

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

D_i = Datenstruktur nach der iten Operation in einer Folge von n Operationen, i = 1, …, n

→ Vergleich $\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i$ mit $\sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$

→ $\Phi(D_n) \geq \Phi(D_0)$

Besser:

$\Phi(D_i) \geq \Phi(D_0) \quad \forall i = 1, \ldots, n$

$\Phi(D_0) = 0$

MAKE-F-HEAP()

```
1  n[H] := 0
2  min[H] := NIL
```

→ t(H) = 0, m(H) = 0

→ Φ(H) = 0

→ aktuelle Kosten = amortisierte Kosten = O(1)

F-HEAP-INSERT(H,x)

```
1  degree[x] := 0
2  p[x] := NIL
3  sohn[x] := NIL
4  left[x] := x
5  right[x] := x
6  mark[x] := false
7  concatenate die root-Liste von H mit x
8  if min[H] = NIL or key[x] < key[min[H]]
9    then min[H] := x
10 n[H] := n[H] +1
```

→ $H \xrightarrow{+x} H'$

t(H') = t(H) + 1 und m(H') = m(H)

((t(H) + 1) + 2m(H)) – (t(H) + 2m(H)) = 1

Potential danach – Potential davor

d.h. wir „überbezahlen“ das Insert

→ amortiesierte Kosten O(1) + 1 = O(1)

F-HEAP-UNION(H_1,H_2)

```
1  H := MAKE-F-HEAP()
2  min[H] := min[H1]
3  concatenate root-Liste von H2 mit der von H1
4  //Minima nun benachbart
5  if (min[H1]=NIL)or(min[H2]!=NIL & key[min[H2]]<key[min[H1]])
6    then min[H] := min[H2]
7  n[H] := n[H1] + n[H2]
8  free the objects H1, H2
9  return H
```

→ $\Phi(H) – (\Phi(H_1) – \Phi(H_2))$

$= ((t(H) + 2m(H)) – (t(H_1) + 2m(H_1) + t(H_2) + 2m(H_2)) = 0$

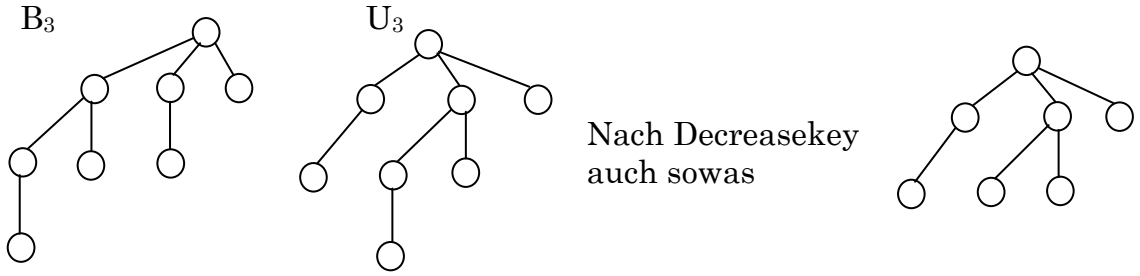
→ amortisierte Kosten = aktuelle Kosten + Potentialdifferenz = O(1) + 0 = O(1)

F-HEAP-EXTRACT-MIN(H)

```
1  z := min[H]
2  if z != NIL
3    then für jeden Sohn x von z do
4         füge ihn in die root-Liste ein
5         setze p[x] := NIL
6    entferne z aus root-Liste
7    if z = right[z]
8         then min[H] := NIL
9    else min[H] := right[z]
10        CONSOLIDATE(H)
11   n[H] := n[H] – 1
12  return z
```

Bemerkungen zu CONSOLIDATE(H):

- bei consolidate wird das nachgeholt, was bei den lazy-Operationen (Insert + Union) aufgeschoben wurde.
- Hat als Ergebnis eine Struktur für die die Einzigkeitsforderung gilt (bezogen auf den Wurzeldegree, d.h. es existiere für jeden Wurzeldegree maximal je ein Exemplar)
- Weiteres Ergebnis des Konsildieren:
  - Wald von Bäumen ähnlich wie bei den Binomialheaps:
    - 1. Unterschied: keine Ordnung in Degrees der Wurzeln
    - 2. Unterschied: keine reinen Binomialbäume, sondern ungeordnete Binomialbäume



- Realisiert über F-HEAP-LINK:

  F-HEAP-LINK(H,y,x)
  1 remove y aus der root-Liste
  2 mache y zu einem Sohn von x (sohn[x] := y)
  3 mark[y] := false
- Wie machen wir die Einzigkeitsforderung?

  Idee:

  Rang-Array (Degree-Array)

  Lemma (20.1 Cormen) sichert schliesslich, dass die Anzahl der Söhne jedes Knotens O(D(n)) = O(log n) ist

  Rang-Array:

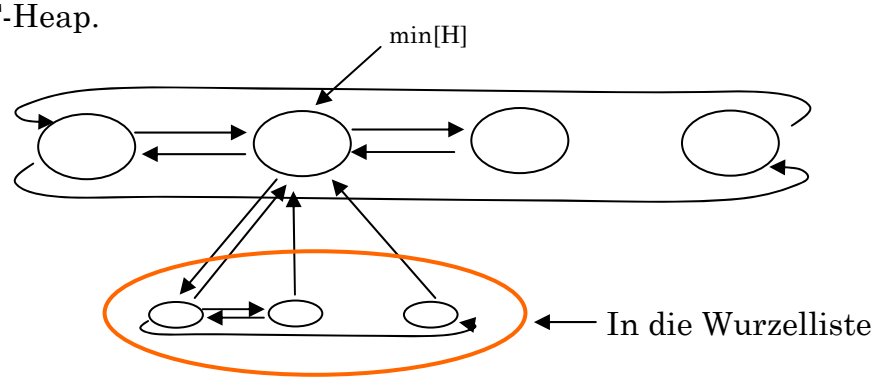
			…….	
0	1	2	…….	D_n

  → in jedes Array-Feld nur ein Wurzelknoten dem Grad entsprechend

  → wenn schon einer mit diesem Degree vorhanden ist, greift

F-HEAP-LINK(H,y,x)
→ solange bis Wurzelknoten eingefügt werden kann.

Analyse von EXTRACT-MIN:
Definition:
D(n) sei der MAximaldegree eines Baumes (eines Knotens) in einem n-F-Heap.



min[H] raus

→ Konsolidieren:
Ziel:
Keine 2 Bäume mit gleichen Wurzeldegree
→ Am Schluss:
Größe der root-Liste ≤ D(n) + t(H) – 1
Nach Konsolidieren nur noch D(n) Wurzeln

Aktuelle Arbeit:
O(D(n) + t(H))

Dazu: Potentialdifferenz:
Vorher:
t(H) + 2m(H)
Nachher:
(D(n) +1) + 2m(H)

Amortisierte Kosten:
O(D(n)) + t(H)) + ((D(n) + 1) + 2m(H)) – (t(H) + 2m(H))
= O(D(n)) – $\underbrace{t(H) + O(t(H))}$ = O(D(n))

Kann ignoriert werden

F-HEAP-DECREASEKEY(H,x,k)

```
1  if k > key[x]
2    then error
3  key[x] := k
4  y := p[x]
5  if y != NIL and key[x] < key[y]
6    then CUT(H,x,y)
7              CASCADING-CUT(H,y)
8  if key[x] < key[min[H]]
9    then min[H] := x
```

CUT(H,x,y)
```
1  Entferne x aus der Sohnliste von y
2  addiere x zur Wurzelliste von H
3  p[x] := NIL
4  mark[x] := false
```

CASCADING-CUT(H,y)
```
1  z := p[y]
2  if z != NIL
3    then if mark[y] = false
4              then mark[y] := true
5         else CUT(H,y,z)
6                 CASCADING-CUT(H,z)
```

→ Potentialdifferenz:
c – cascading cuts
$(t(H) + c) + 2 (m(H) – c + 2) – (t(H) + 2m(H)) = 4 – c$

Aktuelle Arbeit:
$O(c) + (4 – c) = O(1)$
Skizze:
$2t(H) + c – 4(m(H) – c + 2) – (2t(H) + 4m(H)) = 8 – 3c$ …

Hauptlemma:
Es sei x ein beliebiger Knoten in einem F-Heap. Degree[x] = k.
Seien $y_1,…,y_k$ alle Söhne von x in der Reihenfolge des Anhängens.
Dann ist degree[y_1] ≥ 0 und degree[y_i] ≥ i – 2 für i = 2,…,k

Beweis:
Offenbar ist degree[y_1] ≥ 0 wahr.
Für i ≥ 2 gilt als y_i an x angehängt wurde, dann wären $y_1,…,y_{i-1}$ schon angehängt.
→ degree[y_i] = i – 1
Zwischenzeitlich könnte ein Sohn verloren gegangen sein
→ degree[y_i] = i – 2 □

$$F_k = \begin{cases} 0 & \text{falls } k = 0 \\ 1 & \text{falls } k = 1 \\ F_{k-1} + F_{k-2} & \text{falls } k \geq 2 \end{cases}$$

Lemma 2:

$\forall k \geq 0$ gilt $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$

Lemma 3:

$F_{k+2} \geq \Phi^k \quad \forall k \qquad \Phi = \frac{1+\sqrt{5}}{2} = 1,618034\ldots.$

Lemma 4:

x sei Knoten in F-Heap, k = degree[x]

→ size[x] ≥ F_{k+2} ≥ Φ^k

Beweis:

Definition:

s_k := Minimalsize eines Knotens mit degree = k

$s_0 = 1$

$s_1 = 2$

$s_2 = 3$ …

Seien $y_1, \ldots, y_k$ die Söhne von x in Reihenfolge des Einhängens

→ $size[x] \geq s_k = 2 + \sum_{i=2}^{k} s_{degree[y_i]} \geq 2 + \sum_{i=2}^{k} s_{i-2}$

Induktiv:

$s_k \geq F_{k+2}$

Induktionsvoraussetzung:

$k \geq 2$, $s_i \geq F_{i+2}$ für $i = 0, 1, \ldots, k-1$

Induktionsbehauptung:

$s_k \geq F_{k+2}$

Beweis:

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=0}^{k-2} F_{i+2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$ □

Satz:

O(D(n)) = O(log n)

Beweis:

X Knoten im n-F-Heap, k = degree[x] → $n \geq size[x] \geq \Phi^k$

→ $k \leq \log_\Phi size[x] \leq \log_\Phi n$ □

6 – Datenstrukturen zur Verwaltung von Zerlegungen

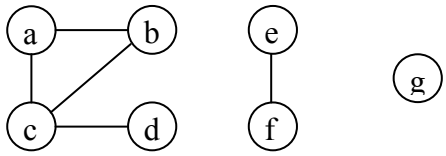
→ Zerlegung einer Menge in disjunkte Teilmengen

6.1 Union-Find-Struktur

ADT:
Union-Find-Struktur
{UNION, FIND, MAKE-SET}

	Ottmann	Cormen
MAKE-SET(e) e – Element	→ Bildung von {e}, Voraussetzung: e ist noch in keiner Menge vorhanden!	
FIND(u)	→ Ausgabe des Namens der Menge, die u enthält (in der Regel „kanonisches Element“ der Menge	
UNION	UNION(FIND(u),FIND(v)) → Vereinigung beider Mengen nach Finden der kanonischen Elementen, u_0 ist neuer Name	UNION(u,v) → Vereinigung der beiden Mengen, welche u,v enthalten ohne vorheriges Suchen der kanonischen Elemente

Beispiel 1:



CON-COMPONENT(G)

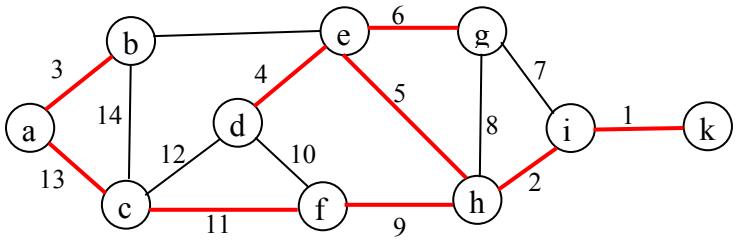
```
1  for each vertex v ∈V[G] do
2    MAKE-SET(v)
3 for each edge (u,v) ∈E[G] do
4    if (FIND(u) = u0) != (FIND(v) = v0)
5         then UNION(u0,v0)
```

SAME-COMPONENT(u,v)

```
1  if FIND(u) = FIND(v)
2    then return true
3  else return false
```

→ Ergebnis von CON-COMPONENT(G)
→{a,b,c,d}, {e,f}, {g}

Beispiel 2:



→ Kantenmarkierter Graph
Problem: MST (minimal spanning tree)

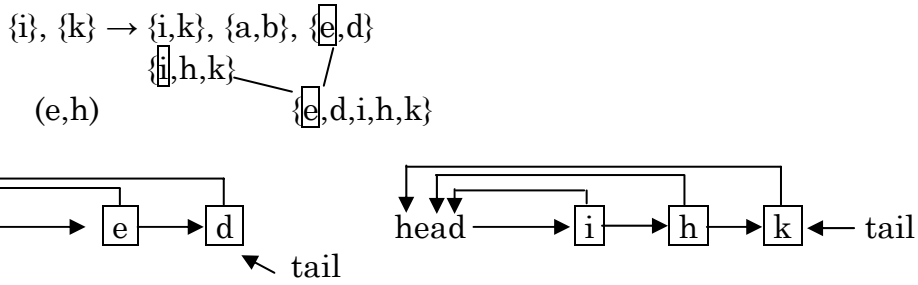
MST (Kruskal)

```
1  E* := ∅
2  for all v ∈V do
3    MAKE-SET(v)
4  Verwalte E als Prioritätswarteschlange (oder SORT)
5  for all Kanten in nicht fallender Reihenfolge (u,v) ∈E do
6    if (u0 = FIND(u)) != (v0 = FIND(v))
7         then E* := E* ∪{(u,v)}
8                  UNION(u0,v0)
9  return E*
```

6.1.1 Verwaltung im Rechner?

2 Varianten:

6.1.1.1 Variante 1: Listen

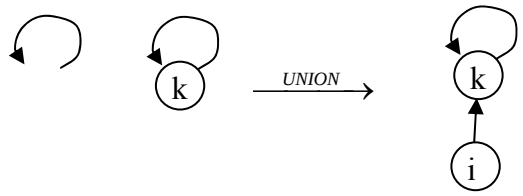


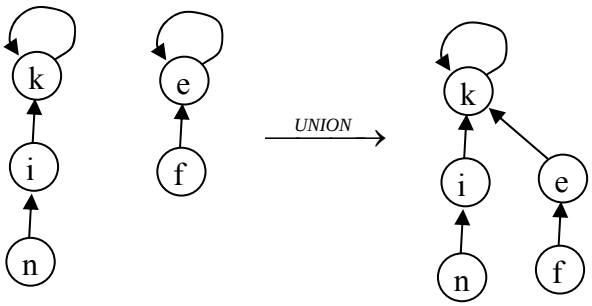
Pfeile auf head nötig wegen FIND
Kanonisches Element: head zeigt darauf
UNION erfordert Ω(n) Pointeränderungen

Satz:

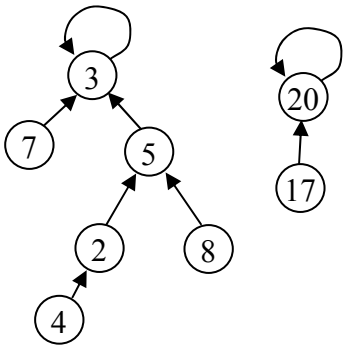
Bei m MAKE-SET, FIND und UNION Operationen, davon n MAKE-SET, brauchen wir $\Theta(n^2)$, bei gewichteten (kleinere Liste an größere Liste anhängen) UNION O(m + nlog n) TIME

6.1.1.2 Variante 2: Disjoint Set Forests (Wälder für disjunkte Mengen)



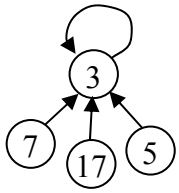


Darstellung mit Arrays



x	1	2	3	4	5	6	7	8	9	…	17	...	20
p(x)		5	3	2	3		3	5			20		20

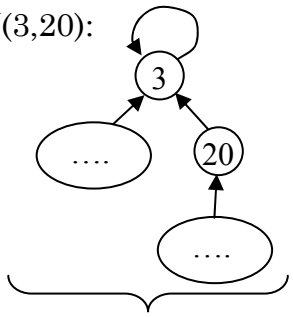
FIND, UNION?



Idealfall für FIND (da billig, O(1)) aber teuer bei UNION

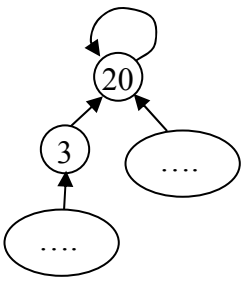
FIND und UNION können trivial implementiert werden

UNION(3,20):



Günstiger für FIND

UNION(20,3):



6.1.2 Heuristiken:

Vereinigung nach Höhe (oder nach Größe (size) oder nach Rang)
FIND mit Pfadkompression

z.B. nach size

MAKE-SET(x)
```
1  p[x] := x
2  size[x] := 1
```

UNION(e,f)
```
1  if size[e] < size[f]
2    then vertausche e und f
3  //size[e] ≥ size[f]
4  p[f] := e
5  size[e] := size[e] + size[f]
```

FIND mit Pfadkompression

FIND(x)
```
1  y := x
2  while p[y] != y do
3    y := p[y]
4  //jetzt ist y Wurzel
5  z := x
6  while p[z] != y do
7    t := z
8    z := p[z]
9    p[t] := y
10 FIND := y
```

Variante mit Rang (Obere Schranke der Höhe)

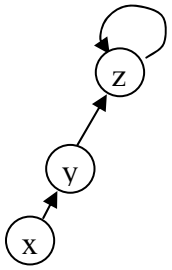
MAKE-SET(x)
```
1  p[x] := x
2  Rang[x] := 0
```

UNION(x,y)
```
1  LINK(FIND(x), FIND(y)
```
(bzw. Ohne FIND, wenn UNION nur für kanonische Elemente definiert ist)

LINK(x,y)
```
1  if Rang[x] > Rang[y]
2    then p[y] := x
3  else p[x] := y
4     if Rang[x] = Rang[y]
5         then Rang[y] := Rang[y] + 1
```

```
FIND(x)
1  x != p[x]
2    then p[x] := FIND(p[x])
3  return p[x]
```

Beispiel:



FIND(x) $x \neq p[x] \rightarrow p[x] = FIND(p[x]) = FIND(y)$
FIND(y) $y \neq p[y] \rightarrow p[y] = FIND(p[y]) = FIND(z)$
FIND(z) $z = p[z] \rightarrow$ return $p[z] = z$
FIND(z) = z
FIND(y) $y \neq p[y] \rightarrow p[y] = FIND(p[y]) = FIND(z) = z$
FIND(x) $x \neq p[x] \rightarrow p[x] = FIND(p[x]) = FIND(y) = z$
p[x] := z

Satz:

Für Vereinigung nach size (oder Höhe oder Rang) gilt:
Ein Baum mit Höhe h hat mindestens $N \geq 2^h$ Knoten.
(FIND beide Varianten)

Beweis: (size)

Induktiv über Höhe (über Komplexität UNION)

Induktionsanfang:

Satz gilt für h = 0, Beweis trivial

Induktionsschritt

Induktionsvoraussetzung:

Satz gelte für die Teilbäume T_1 und T_2 von T

o.B.d.A. sei $size[T_1] \geq size[T_2]$ (nicht etwa $h_1 \geq h_2$)

Fallunterscheidung:

Fall 1: ($h_1 > h_2$)

$h = Höhe(T) = \max(h_1,h_2) \rightarrow size[T] = size[T_1] + size[T_2]$
$\overset{IV}{\geq} 2^{h1} + 2^{h2} \geq 2^h$

Fall 2: ($h_1 \leq h_2$)

$H = Höhe(T) = \max(h_1,h_2) + 1 \rightarrow size[T] = size[T_1] + size[T_2]$
$\geq 2 \cdot size[T_2] \overset{IV}{\geq} 2 \cdot 2^{h2} \geq 2^{h2+1} \geq 2^h$ □

Folgerung:

Einfaches FIND kostet O(log N) = O(log n)

N – Anzahl Knoten eines Baumes, n – Anzahl aller Elemente im DSF

Beweis:

$n \geq N \geq 2^h \rightarrow h \leq \log N \leq \log n$ □

Bemerkung:

Gilt auch für FIND mit Pfadkompression!

Bemerkung: Ackermannfunktion

$$A_k(j) := \begin{cases} j+1 & \text{falls } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{falls } k \geq 1 \end{cases}$$

Lemma 1:

$\forall j \geq 1 : A_1(j) = 2j + 1$

Lemma 2:

$\forall j \geq 1 : A_2(j) = 2^{j+1}(j+1) - 1$

Beispiele:

$A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2047$

$A_4(1) = A_3^{(2)}(1) = A_2^{(2048)}(2047) >> A_2(2047) > 2^{2048} = 16^{512} >> 10^{80}$

$\alpha(n) := \min\{k: A_k(1) \geq n\}$

$$\alpha(n) = \begin{cases} 0 & \text{für } 0 \leq n \leq 2 \\ 1 & \text{für } n = 3 \\ 2 & \text{für } 4 \leq n \leq 7 \\ 3 & \text{für } 8 \leq n \leq 2047 \\ 4 & \text{für } 2048 \leq n \leq A_4(1) \end{cases}$$

Satz:

$$\lim_{n \to \infty} \alpha(n) = \infty$$

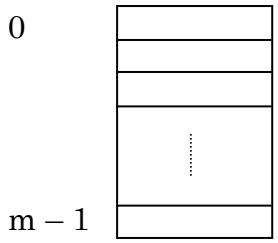
Satz: (Tarjan)

Bei Implementierung von UNION-FIND mit Vereinigung nach size (oder Höhe oder Rang) und Pfadkompression für FIND kosten m Operationen $O(m \cdot \alpha(n))$ TIME, falls Davon n Operationen MAKE-SET sind.

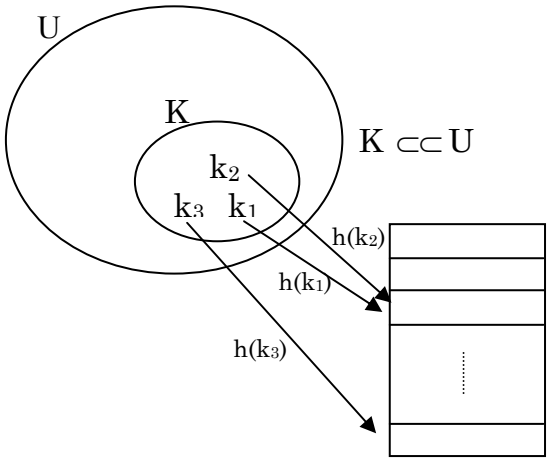
6.2 Hashing

ADT HASHING
{INSERT, DELETE, SEARCH}

Tabelle:



direkte Adressierung – untypisch für Hashing



$K \subset\subset U$ $\quad |T| = |U|$ - undenkbar

h – Hashfunktion in O(1) berechenbar
$h(k) \in \{0, \ldots, m-1\}$

$h(k_1) = h(k_2) \rightarrow$ Kollision

Definition: (simple uniform Hashing)
Annahme des einfachen uniformen Hashings:
Jedes beliebige Element aus K wird mit gleicher Wahrscheinlichkeit und unabhängig von den anderen auf jeden der m Plätze abgebildet.

6.2.1 HASHING mit Verkettung (Hashing with chaining)
In die m Zellen warden Zeiger auf Listen abgelegt.

INSERT:
Hinter den Kopf der Liste (wegen DELETE: doppelt verkettet)

Definition:

Ladefaktor $\alpha := \frac{n}{m}$ n – Gesamtzahl der gespeicherten Schlüssel

n_j := Länge der Liste, auf die der Zeiger der Zelle j zeigt

$$n = \sum_{j=o}^{m-1} n_j$$

Satz 1:

Der erwartete Wert für die Zeit bei erfolgreichen Suchen bei HASHING mit Verkettung ist unter der Annahme des einfachen uniformen Hashings gleich $\Theta(1 + \alpha)$

Beweis:

Definition:

x_i sei der i-te Wert, der eingefügt wurde

$k_i := key[x_i]$

Suche x_i: $h(k_i) \rightarrow \ldots \rightarrow x_i \rightarrow \ldots$

$$1 + E[h(k_j) = h(k_i) \text{ für } j > i] = 1 + \sum_{j=i+1}^{n} x_{ij}$$

$$X_{ij} := \begin{cases} 1 & h(k_i) = h(k_j) \\ 0 & \text{sonst} \end{cases}$$

Nebenrechnung:

$$P\{h(k_i) = r = h(k_j)\} = P\{h(k_i) = r\} \cdot P\{(k_j) = r\}$$

$$= \frac{1}{m} \cdot \frac{1}{m} = \frac{1}{m^2}$$

$$P\{h(k_i) = h(k_j)\} = \sum_{r=0}^{m-1} P\{h(k_i) = h(k_j) = r\}$$

$$= \sum_{r=0}^{m-1} \frac{1}{m^2} = \frac{1}{m^2} \sum_{r=0}^{m-1} 1 = \frac{1}{m}$$

$$\underbrace{E\left[\frac{1}{m}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} x_{ij}\right)\right]}_{\substack{\text{Erwartungswert für die Elemente} \\ \text{bei erfolgreichen Suche}}} = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} x_{ij}\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n} n - \sum_{i=1}^{n} i = 1+\left(\frac{n-1}{2m}\right)$$

$$= 1+\frac{\alpha}{2}-\frac{\alpha}{2m} = \Theta(1+\alpha)$$

Folgerung:

Ist n = O(m), d.h. α = O(1), so kostet das erfolgreiche Suchen O(1) im Erwartungswert.

Erfolgloses Suchen:

Satz:

Erfolgloses Suchen kostet auch $\Theta(1 + \alpha)$

Beweis:

$E[n_{h(k)}] = \alpha, \quad E = 1 + \alpha$

→ Wenn h die Annahme des einfachen uniformen Hashing erfüllt, gehen die ADTs in erwarteter Zeit O(1).

Implementieren mit höhenbalancierten Suchbaum funktioniert immer in O(log n)

Hashing funktioniert in der erwarteten Zeit O(1) wenn die Annahme einfache uniforme Hashing erfüllt ist.

Wie gewinnt man gute Hashfunktionen?

Bei guten Hashfunktionen gibt es gute und schlechte Inputfolgen.

2 Möglickeiten:

1. möglichst gute, geeignete Hashfunktionen
2. zufällige Auswahl der Hashfunktion (Robin-Hood-Effekt) (uniformhashing)

a) Divisionsmethode:

Seien Random-Zahlen k : 0 < k < 1 gegeben, m – Tabellengröße

$h(k) = \lfloor k \cdot m \rfloor$

→ k erfüllt die Annahme des einfachen uniformen Hashings

$\xrightarrow[\text{allgemein}]{}$ $h(k) = k \bmod m$

Wichtig: m geeignet festlegen, m sollte keine 2er-Potenz sein!

b) Multiplikationsmethode

1. Wähle festes A: 0 < A < 1

$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor = \lfloor m \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$

→ hängt stark von A ab

→ wie gut h wird liegt an A

D. Knuth

$A = 0{,}618 = \frac{\sqrt{5}-1}{2}$

6.2.2 Uniformes Hashing

Sei H eine endliche Menge von Hashfunktionen, die U nach {0,1,…,m-1} abbilden.

Definition:

Die Klasse H heiße universal, wenn jedes Paar $x \neq y \in U$ impliziert, dass die Zahl der Hashfunktionen $h \in H$: $h(x) = h(y) = \frac{|H|}{m}$ ist.

Mit anderen Worten:

H ist universal, falls für alle x,y $\in$ U: x $\neq$ y gilt P[h(x) = h(y)] = $\frac{1}{m}$ für h $\in_R$ H.

($\in_R$ – zufällige Elementauswahl)

Satz:

Sei K $\subseteq$ U mit |K| = n, sei H universal, sei h $\in_R$ H, dann ist pro Schlüssel x $\in$ K die mittlere Anzahl der Kollisionen höchstens gleich α

Beweis:

Betrachte Zufallsvariable $c_{xy} = \begin{cases} 0 & h(x) = h(y) \\ 1 & sonst \end{cases}$

→ $E[c_{xy}] = P(h(x) = h(y)) = \frac{1}{m}$

Sei C_x die Zufallsvariable, die die Anzahl der Kollisionen mit x angibt (auf K)

$$\rightarrow E\left[\sum_{y\in K, y\neq x} c_{xy}\right] = \sum_{y\in K, y\neq x} E\left[c_{xy}\right] = \sum_{y\in K, y\neq x} \frac{1}{m} \leq \frac{|K|}{m} = \frac{n}{m} = \alpha \qquad \square$$

Frage: Existenz???

1) Die Klasse aller Funktionen von U nach {0,1,…,m-1} ist universal
2) H_1, oBdA Elemente von U sind Bitstrings x gleicher Länge
   m gegeben.
   Zerlege x = $\underbrace{1011}_{x_0}\underbrace{0011}_{x_1}....\underbrace{1010}_{x_k}....\underbrace{0101}_{x_r}$ mit $0 \leq x_i < m$
   Tupel a = (a_0, a_1, …, a_r) $a_i \in \{0, 1, \ldots, m-1\}$
   Zufällige Wahl von a → h_a
   $h_a(x) = \sum_{i=0}^{r} a \cdot x_i \bmod m$ H = {h_a | a = (a_0,…,a_r) mit obigen Bedingungen}

   Satz:
   H ist universal, falls m eine Primzahl ist
   Beweis:
   Seien x, y Schlüssel: x $\neq$ y $\in$ U
   Betrachten (y_0,y_1,…,y_r) und (x_0,x_1,…,x_r)
   oBdA: $x_0 \neq y_0$

$$h_a(x) = h_a(y) \Leftrightarrow \left(\sum_{i=0}^{r} a_i \cdot x_i \equiv \sum_{i=0}^{r} a_i \cdot y_i\right) (\bmod m)$$

$$\Leftrightarrow \left(a_0 \cdot (x_0 - y_0) \equiv \sum_{i=1}^{r} a_i \cdot (y_i - x_i)\right) (\bmod m) \qquad | \cdot \frac{1}{(x_0 - y_0)}$$

$$\rightarrow \left(a_0 \equiv \frac{1}{(x_0 - y_0)} \sum_{i=1}^{r} a_i \cdot (y_i - x_i)\right) (\bmod m)$$

→ von den m^{r+1} vielen Tupeln a liefern genau m^r viele eine
Kollision $h_a(x) = h_a(y)$

→ $P(h(x) = h(y)) = \frac{m^r}{m^{r+1}} = \frac{1}{m}$

→ H_1 ist universal

3) H_2

p Prim, $H_{p,m} := \{ h_{a,b} \mid a \in Z_p^+, b \in Z_p \}$

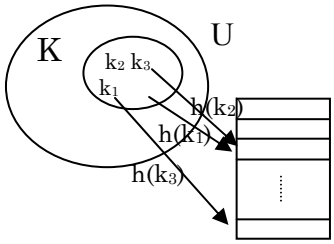
$Z_p^+ = \{1,\ldots,p-1\}$

$Z_p = \{0,\ldots,p-1\}$

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

$H_{p,m}$ ist für Primzahlen p, die hinreichend groß sind universal.

6.2.3 Open Hashing in semidynamischen Mengen (insert, search)



$h : U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$

$h(k_1,0) = h_1(k_1)$

$h(k_1,1) = ?$, $h(k_1,2) = ?$ bei $h(k_1,m-1)$ Tabelle voll

Probefolge

INSERT(T,k)

```
1  i := 0
2  repeat j := h(k,j)
3    if T[j] = NIL
4         then T[j] := k
5                   return j
6    else i := i + 1
7  until i = m
8 Fehler Tabelle voll
```

- linear probing (lineares Hashing)
  $h(k,i) := (h_1(k) + i) \bmod m$
  Problem:
  Primäre Clusterbildung

- quadratic probing
  $h(k,i) := (h_1(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$ (Sekundäre Cluster)

Ziel:

Erfülltsein der Annahme des uniformen Hashings:

Annahme:

Jeder Schlüssel k hat jeder der m! verschiedenen Permutationen von {0,1,…,m-1} mit gleicher Wahrscheinlichkeit als Reihenfolge.

Hauptsatz:

Voraussetzung: uniformes Hashing)

Sei eine open-Hashing-Tabelle T mit $\alpha = \frac{n}{m} < 1$ gegeben.

Dann ist die erwartete Anzahl von Proben bei einem weiteren Insert oder einer erfolglosen Suche höchstens $\frac{1}{1-\alpha}$

Praktische Lösung:

Doppeltes Hashing

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

Beispiel:

$h_1(k) = k \bmod 13$ $m = 13$

$h_2(k) = 1 + (k \bmod 11)$

k: 79, 69, 96, 72, 50, 14

$h_1(79) = 1$, $h_1(69) = 4$, $h_1(96) = 5$, $h_1(72) = 7$, $h_1(50) = 11$, $h_1(14) = 1$

$h(14,1) = h_1(14) + 1 \cdot (1 + 14 \bmod 11) = 5$

$h(14,2) = 1 + 2 \cdot (1+3) = 9$

7 – Dynamisches Programmieren

Ansatz:

Problemlösung über die Lösung von Teilproblemen.

Bellmann'sches Optimalitätsprinzip:

Optimale Lösungen des Gesamtproblems implizieren stets optimale Lösungen der Teilprobleme.

Suchbäume

$a_1, a_2, \ldots, a_n$ Wort a_i

$v_1, v_2, \ldots, v_n$ Knoten v_i mit Wort a_i

$t_1, t_2, \ldots, t_n$ Tiefe des Knoten v_i

Wahrscheinlichkeit $p_1, p_2, \ldots, p_n$ $\sum_{i=1}^{n} p_i = 1$

Beispiel:

1, 2, 3, 4, 5, 6 $p_1 = 0.1, p_2 = 0.2, p_3 = 0.12, p_4 = 0.05, p_5 = 0.28, p_6 = 0.25$

Lösungsansätze:

1) $\sum_{i=1}^{n} p_i t_i$ ungünstig wegen Wurzel

2) $\sum_{i=1}^{n} p_i (t_i + 1) = \min$

Definition Teilproblem (i,j) $(1 \le i,j \le n)$ $(=\{a_i, \ldots, a_j\})$

Annahme:

$a_1 < a_2 < a_3 < \ldots < a_n$

T Suchbaum für $\{a_1, \ldots, a_n\}$ T(i,j) Suchbaum für $\{a_i, \ldots, a_j\}$

$t(i,j) = \sum_{m=i}^{j} p_m (t_m + 1)$ mittlere gewichtete Suchzeit für (i,j)

Definition:

$p(i,j) := \sum_{m=i}^{j} p_m$ (Wahrscheinlichkeit für Ansprechen eines Elements aus dem Teilproblem)

$p(i,i) = p_i$

Ziel:

t(i,j) rekursiv beschreiben

$a_i, a_{i+1}, \ldots, a_j$ → Wahl von a_k als Wurzel $(i \le k \le j)$

Wie wirkt sich die Wahl von k aus?

$$t^{(k)}(i,j) = p_k + \underbrace{p(i, k-1)+t(i, k-1)}_{\text{Anteil linker Teilbaum}} + \underbrace{p(k+1, j)+t(k+1, j)}_{\text{Anteil rechter Teilbaum}}$$

$$p_k + p(i, k-1) + p(k+1, j) = \sum_{m=i}^{j} p_m = p(i, j)$$

$$t(i, j) = \begin{cases} 0 & \text{falls } i > j \\ p(i,j) + \min\limits_{i \le k \le j} (t(i, k-1) + t(k+1, j)) \end{cases}$$

t(i, j) k

	1	2	3	4	5	6
1	0.1	0.4	0.64	0.79	1.52	2.04
2		0.2	0.44			
3			0.12			
4				0.05		
5					0.28	
6						0.25

	1	2	3	4	5	6
1	2	2	2	2	5	
2		2	2	3	5	
3			3	5	5	
4				5	5	
5					5	
6						

t(1,2) = p(1,2) + min((t(1,0)+t(2,2)), (t(1,1) + t(3,2)) = 0.4
t(2,3) = p(2,3) + min((t(2,1) + t(3,3)), (t(2,2) + t(4,3)) = 0.2
t(2,4) = 0.37 + t(3,4)
t(3,5) = 0.35 + t(3,4)

Gesamtproblem = Problem (1,6)

Satz:

In einem optimalen binären Suchbaum liegt die mittlere Knotentiefe in dem Intervall $\left[\frac{H}{\log_2 3} - 1, H \right]$.

$\frac{1}{\log_2 3} \approx 0.63$, $H = H(p_1, \ldots, p_n) = \sum_{i=1}^{n} p_i \log_2 p_i$ (H – Entropie)

Beweis:

Siehe Schöning

8 – Parallele Algorithmen

Beispiel:

Sortieren seriell: Ω(n log n)
Sortieren parallel: O(log n) (R. Cole)

Genauer:

O(log n) TIME und O(n log n) WORK

WORK hat Beziehung zu Kosten $C(n) = T(n) \cdot P(n)$

Gesamtzahl der ausgeführten Operationen — TIME — Prozessorzahl

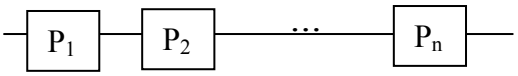
R. Cole: O(log n) TIME und O(n) Prozessoren

Ansatz:

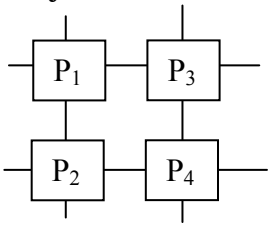
Modelle paralleler Rechner

8.1 Rechnernetze:

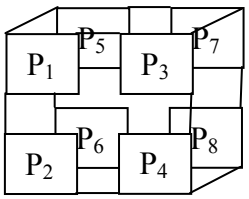
8.1.1 Lineares Modell:



8.1.2 Array-Modell



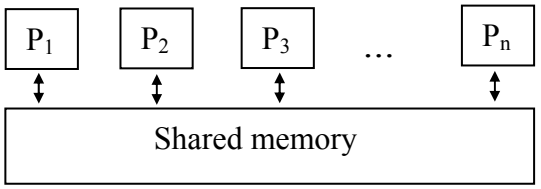
8.1.3 d-Hyperwürfel



Algorithmen für Hypercube-Modell:

In diese Kategorie fällt auch das Odd-even-sorting-Network von Batcher

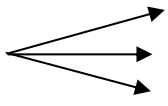
Shared-Memory-Modell



Globaler Speicher

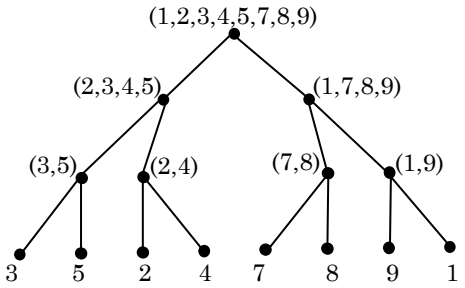
Zugriff der Prozessoren auf Speicher im Sinne des random Access (Direktzugriff) in O(1)
Prozessoren dürfen in die Zellen schreiben & lesen

Verschiedene Varianten:



E – exclusiv C – concurrent
R – Read W – Write

8.2 Binärbaum-Paradigma



→ Skizze beschreibt merge-sort

Merge-sort: Input (3,5,2,4,7,8,9,1)

Seriell: mergesort(S)
1. Zerlege S in (S_1,S_2)
2. mergesort(S_1)
3. mergesort(S_2)
4. merge

$T(n) = 2\,T\left(\frac{n}{2}\right) + O(n)$

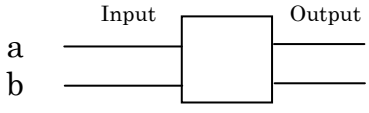
Parallel: mergesort(S)
1. Zerlege S in (S_1,S_2)
2. mergesort(S_1) } Pardo (also parallel)
3. mergesort(S_2) }
4. merge

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

Alles was auf einer Ebene passiert parallel

8.3 Paralleles mergesort:

Batcher:



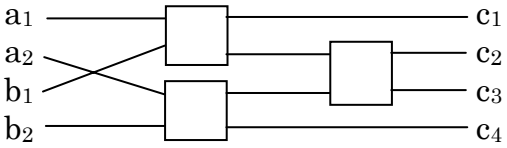
= (1,1)-Netzwerk

merging network:

$a_1 \le a_2 \le a_3 \le \ldots \le a_n$
$b_1 \le b_2 \le b_3 \le \ldots \le b_n$ $\rightarrow c_1 \le c_2 \le c_3 \le \ldots \le c_{2n}$

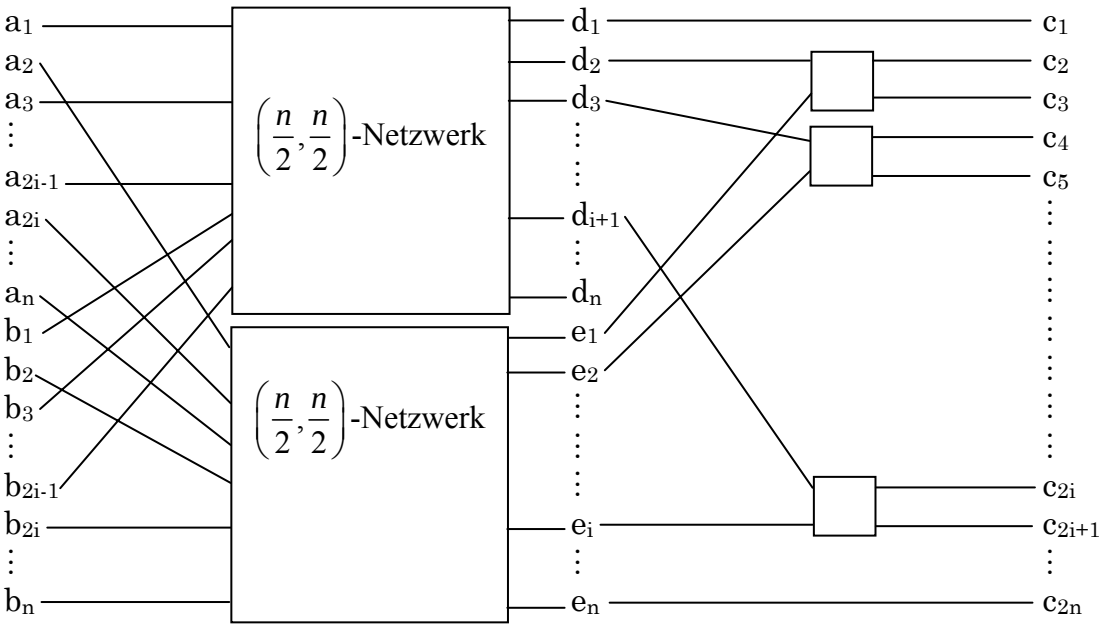
liefert über Binärbaum Paradigma sofort ein Sortiernetzwerk

(2,2)-Netzwerk



$c_1 \le c_2 \le c_3 \le c_4$

(n,n)-Netzwerk



Satz:

DiesesNetztwerk funktioniert wie angegeben.

Beweis:

(I) betrachte $(d_1, d_2, \ldots, d_{i+1})$

wenn k Elemente davon zu $(a_1, a_3, a_5, \ldots)$ gehören, so gehören $(i + 1 - k)$ Elemente zu $(b_1, b_3, b_5, \ldots)$

$\rightarrow (2k - 1)$ Elemente von $(a_1, \underline{a_2}, a_3, \underline{a_4}, \ldots)$ sind kleiner gleich d_{i+1}

analog $(2i + 1 - 2k)$ Elemente von $(b_1, \underline{b_2}, b_3, \underline{b_4}, \ldots)$ sind $\le d_{i+1}$

$\rightarrow d_{i+1} \ge c_{2i}$ (1)

analog folgt:

$\rightarrow e_i \ge c_{2i}$ (2)

(II) mögen k Elemente von $(c_1, c_2, c_3, \ldots, c_{2i+1})$ zu $(a_1, a_2, \ldots)$ gehören

$\rightarrow (2i + 1 - k)$ Elemente gehören zu $(b_1, b_2, \ldots)$

$\rightarrow c_{2i+1}$ größer gleich:

k Elemente von $(a_1, a_2, \ldots)$

d.h. $\frac{k}{2}$ Elemente von $(a_1, a_3, \ldots)$ (falls k gerade sonst $\frac{k+1}{2}$)

(2i + 1 – k) Elemente von (b_1, b_2, …)

d.h. (i + 1 – $\frac{k}{2}$) Elemente von (b_1, b_3, …) (oder $\frac{2i+1-k}{2}$ k ungerade)

→ $c_{2i+1} \geq d_{i+1}$ (3)

analog folgt:

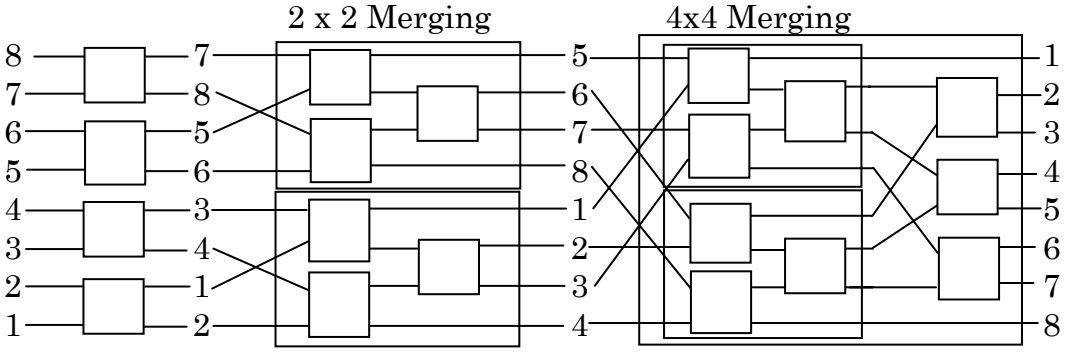
→ $c_{2i+1} \geq e_i$

(III) aus (1), (2), (3) und (4) folgt da $c_1 \leq c_2 \leq c_3 \leq \ldots$

$c_{2i} = \min(d_{i+1}, e_i)$

$c_{2i+1} = \max(d_{i+1}, e_i)$

Beispiel:



Bemerkung:

Für das Sortieren braucht man T(n) = O(log² n) TIME und O(n log² n) Prozessoren.

Definition: (Kosten eines parallelen Algorithmus)

$C(n) := T(n) \cdot P(n)$

→ Batchers Sortierverfahren

$C(n) = O(n \log^4 n)$

Satz:

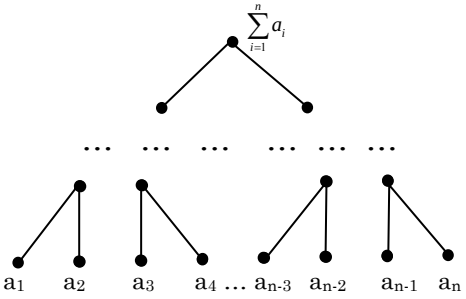
Kosten eines parallelen Verfahrens sind stets größer gleich dem seriellen Optimum.

Beweis:

Angenommen nicht, dann simulieren wir einfach den parallelen Algorithmus seriell indem wir die Prozessoren nacheinander schrittweise arbeiten lassen. Das würde dann einen schnelleren seriellen Algorithmus liefern → Widerspruch zur Annahme!

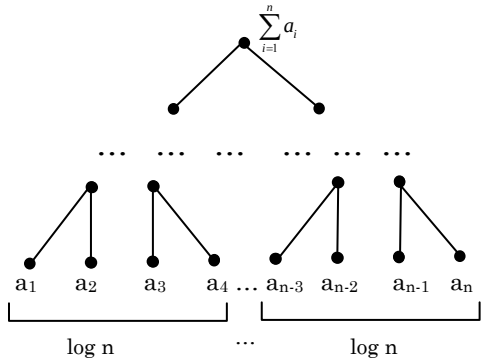
Beispiel:

Batcher ist nicht kostenoptimal.



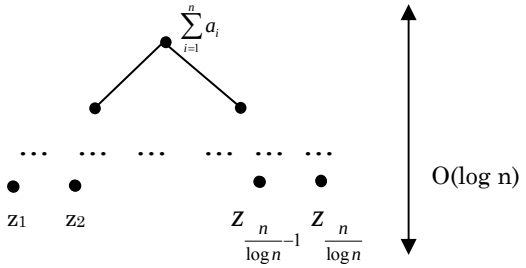
T(n) = O(log n)
P(n) = O(n)
C(n) = O(n log n)
→ nicht kostenoptimal

→ Trick: sequential subset Technik → $\frac{n}{\log n}$ Abschnitte der Länge log n



sequientiell und parallel → Summen $z_1, z_2, \ldots, z_{\frac{n}{\log n}}$

→ TIME: O(log n)
→ Prozessoren: O($\frac{n}{\log n}$)
→ Kosten: O(n)



O(log n) TIME und O($\frac{n}{\log n}$) Prozessoren

→ C(n) = O(logn $\frac{n}{\log n}$) = O(n)
→ O(n) + O(n) = O(n)
→ kostenoptimal

Bemerkung:

R. Cole:

Mergesort: O(log n) TIME mit O(n) Prozessoren
→ C(n) = O(n log n)
→ Kostenoptimal