Algorithmen II

Peter Sanders

Übungen:

Moritz Laupichler, Nikolai Maas

Institut für Theoretische Informatik

Web:

`algo2.iti.kit.edu/AlgorithmenII_WS23.php`

2 Fortgeschrittene Datenstrukturen

Hier am Beispiel von Prioritätslisten.

Weitere Beispiele:

- ☐ Monotone ganzzahlige Prioritätslisten — Kapitel kürzeste Wege
- ☐ perfektes Hashing — Kapitel rand. Alg.
- ☐ Suchbäume mit fortgeschrittenen Operationen — siehe Buch
- ☐ Externe Prioritätslisten — Kapitel externe Algorithmen
- ☐ Geometrische Datenstrkturen — Kapitel geom. Algorithmen

2.1 Adressierbare Prioritätslisten

Procedure build$(\{e_1,\ldots,e_n\})$ $M{:=}\ \{e_1,\ldots,e_n\}$

Function size **return** $|M|$

Procedure insert(e) $M{:=}\ M\cup\{e\}$

Function min **return** $\min M$

Function deleteMin $e{:=}\ \min M$; $M{:=}\ M\setminus\{e\}$; **return** e

Function remove$(h:\ \text{Handle})$ $e{:=}\ h$; $M{:=}\ M\setminus\{e\}$; **return** e

Procedure decreaseKey$(h:\ \text{Handle}, k:\ \text{Key})$ **assert** $\text{key}(h)\geq k$; $\text{key}(h){:=}\ k$

Procedure merge(M') $M{:=}\ M\cup M'$

Adressierbare Prioritätslisten: Anwendungen

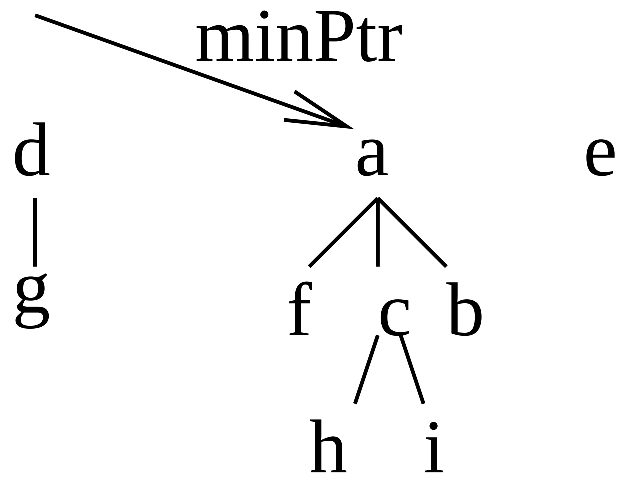
- Dijkstras Algorithmus für kürzeste Wege
- Jarník-Prim-Algorithmus für minimale Spannbäume
- Bei uns: Hierarchiekonstruktion für Routenplanung
- Bei uns: Graphpartitionierung
- Bei uns: disk scheduling

Allgemein:

Greedy-Algorithmen, bei denen sich Prioritäten (begrenzt) ändern.

Grundlegende Datenstruktur

Ein Wald heap-geordneter Bäume

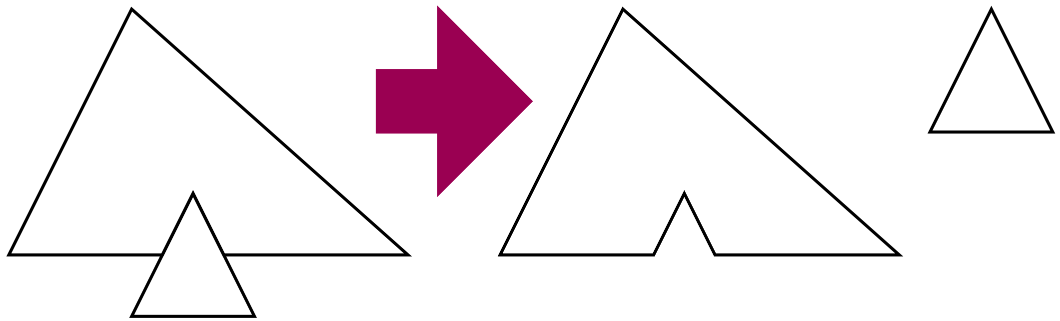


Verallgemeinerung gegenüber binary heap:

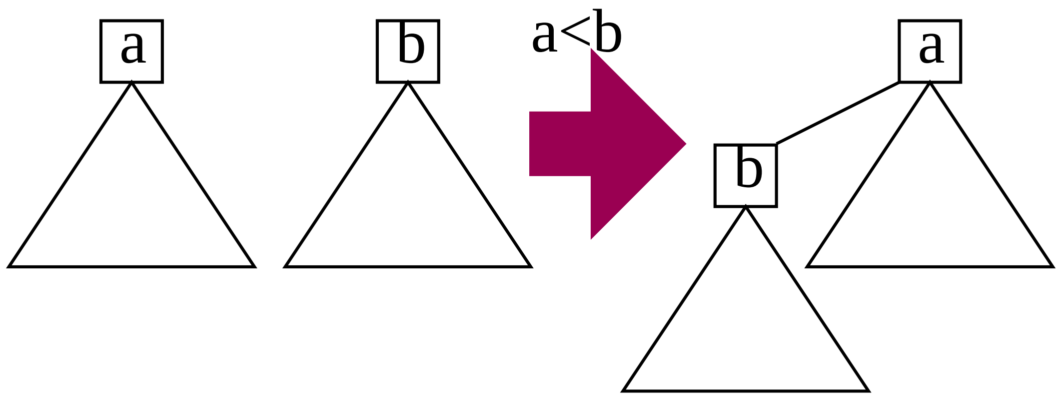
- Baum $\longrightarrow$ Wald
- binär $\longrightarrow$ beliebige Knotengrade

Wälder Bearbeiten

Cut:



Link:



union(a,b): link$(\min(a,b),\max(a,b))$

Pairing Heaps (Paarungs-Haufen??)

[Fredman Sedgewick Sleator Tarjan 1986]

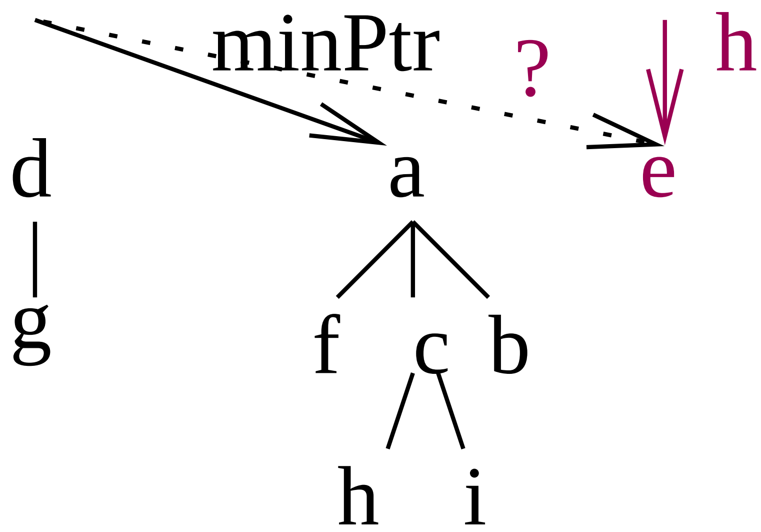
Procedure insertItem$(h$: Handle$)$

newTree(h)

Procedure newTree$(h$: Handle$)$

forest:= forest $\cup \{h\}$

if $*h < \min$ **then** minPtr:= h



Vorsicht: sehr einfache Variante aus 1. engl. Auflage.

dt. Ausgabe, 2. Auflage weichen ab

Pairing Heaps

Procedure decreaseKey$(h : \text{Handle}, k : \text{Key})$

key$(h) := k$

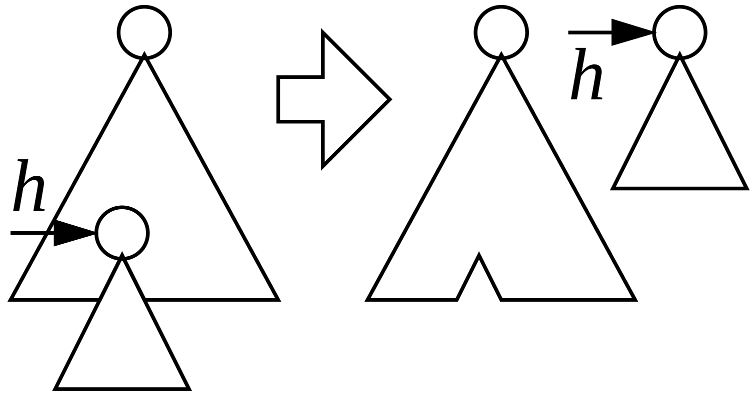
if h is not a root **then** cut(h)

else update minPtr

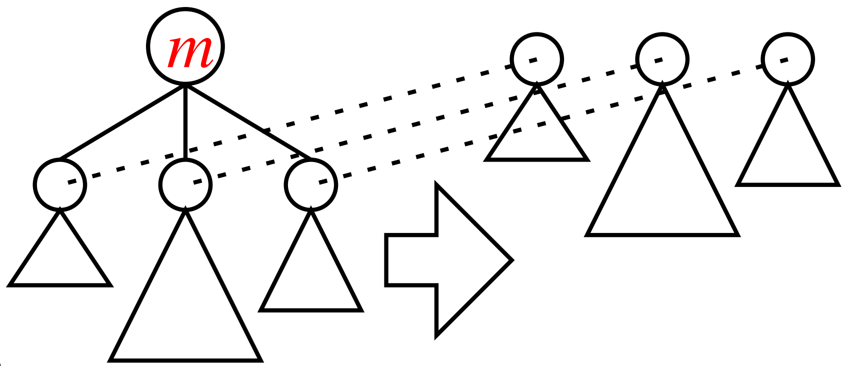
Procedure cut$(h : \text{Handle})$

remove the subtree rooted at h

newTree(h)



Pairing Heaps



Function deleteMin : Handle

m:= minPtr

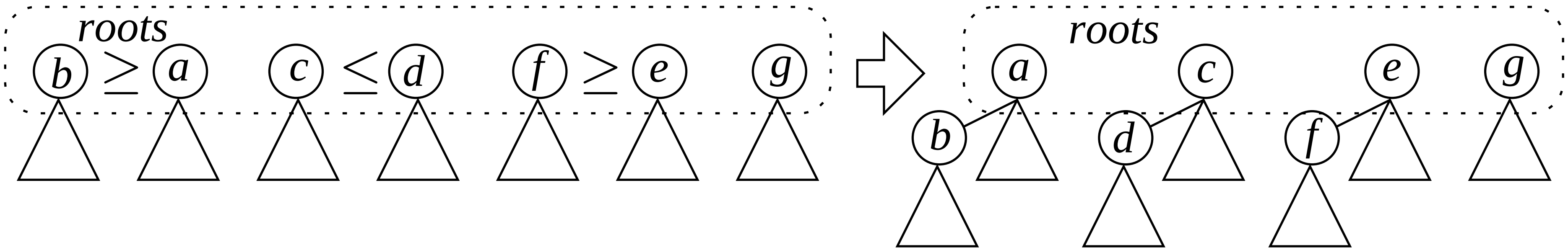
forest:= forest $\setminus \{m\}$

foreach child h of m **do** newTree(h)

perform pair-wise union operations on the roots in forest

update minPtr

return m



Pairing Heaps

Procedure merge(o : AdressablePQ)

  if *minPtr $> *(o$.minPtr) **then** minPtr:= o.minPtr

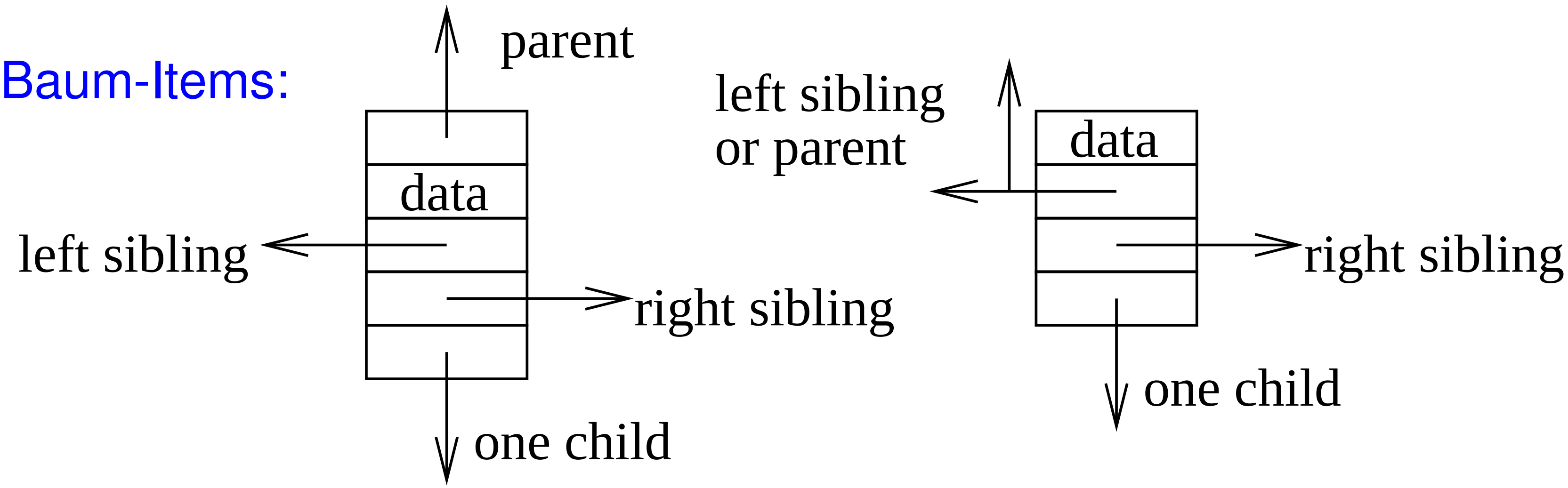
  forest:= forest $\cup$ o.forest

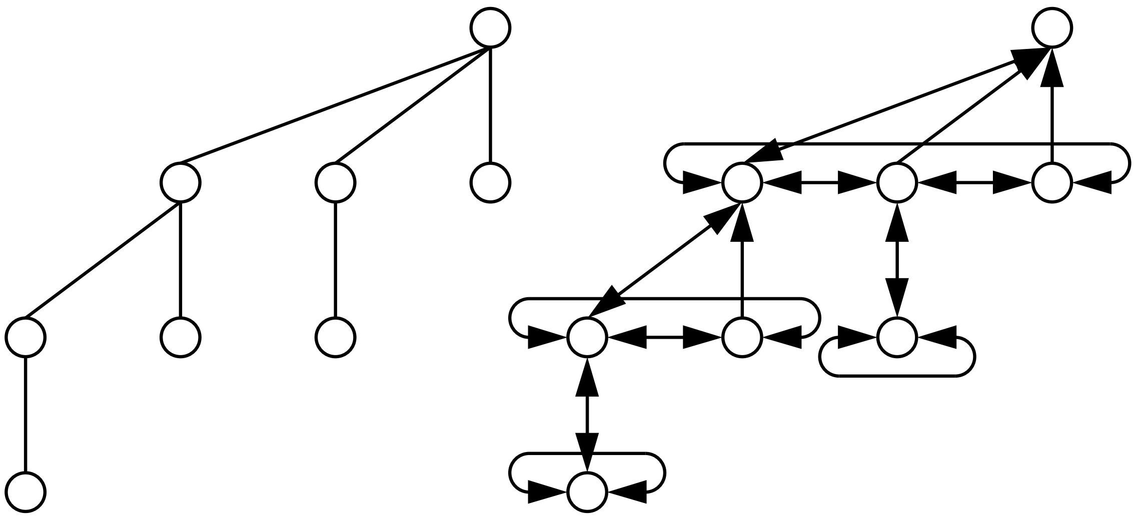
  o.forest:= $\emptyset$

Pairing Heaps – Repräsentation

Wurzeln: Doppelt verkettete Liste

Baum-Items:





Pairing Heaps – Analyse

insert, merge: $\mathrm{O}(1)$

deleteMin, remove: $\mathrm{O}(\log n)$ amortisiert

decreaseKey: amortisiert (in Verbindung mit deleteMin) unklar!

$$\mathrm{O}(\log\log n) \leq T \leq \mathrm{O}(\log n)$$

In der Praxis sehr schnell.

Beweise: nicht hier.

Fibonacci Heaps [Fredman Tarjan 1987]

Rang: Anzahl (direkter) Kinder speichern

Vereinigung nach Rang: Union nur für gleichrangige Wurzeln

Markiere Knoten, die ein Kind verloren haben

Kaskadierende Schnitte: Schneide markierte Knoten

(die also 2 Kinder verloren haben)

Satz: Amortisierte Komplexität $\mathrm{O}(\log n)$ für deleteMin, remove und $\mathrm{O}(1)$ für alle anderen Operationen

(d.h. $Gesamtzeit = \mathrm{O}(o + d\log n)$ falls

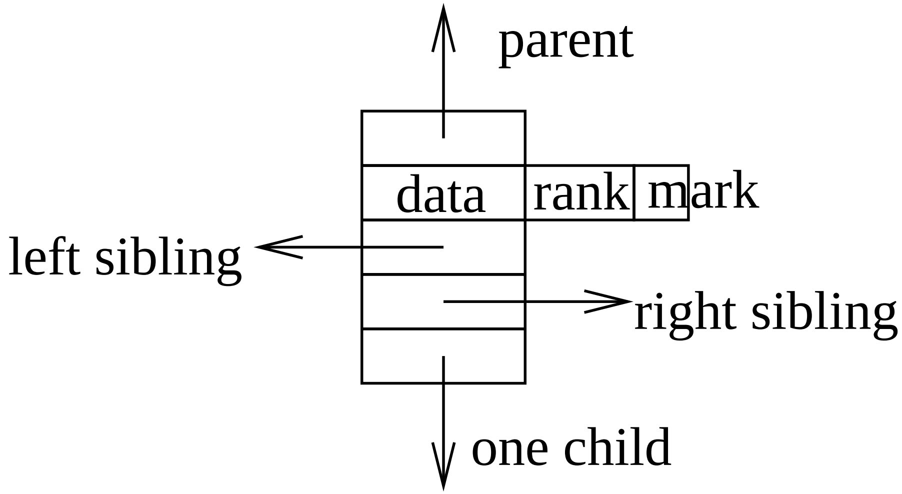
$d =$#deleteMin, $o =$#otherOps, $n = \max|M|$)

Repräsentation

Wurzeln: Doppelt verkettete Liste

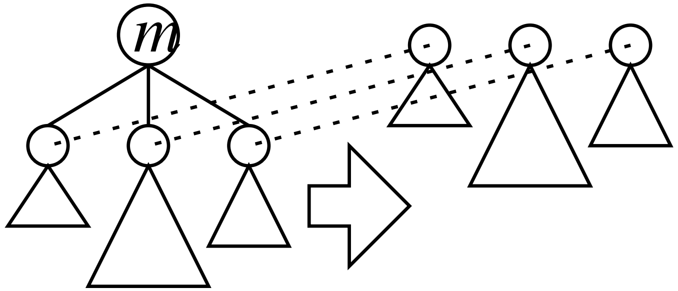
(und ein temporäres Feld für deleteMin)

Baum-Items:



insert, merge: wie gehabt. Zeit $O(1)$

deleteMin mit Union-by-Rank



Function deleteMin : Handle

m:= minPtr

forest:= forest $\setminus \{m\}$

foreach child h of m **do** newTree(h)

while $\exists a, b \in$ forest : rank$(a) =$ rank(b) **do**

union(a, b) // increments rank of surviving root

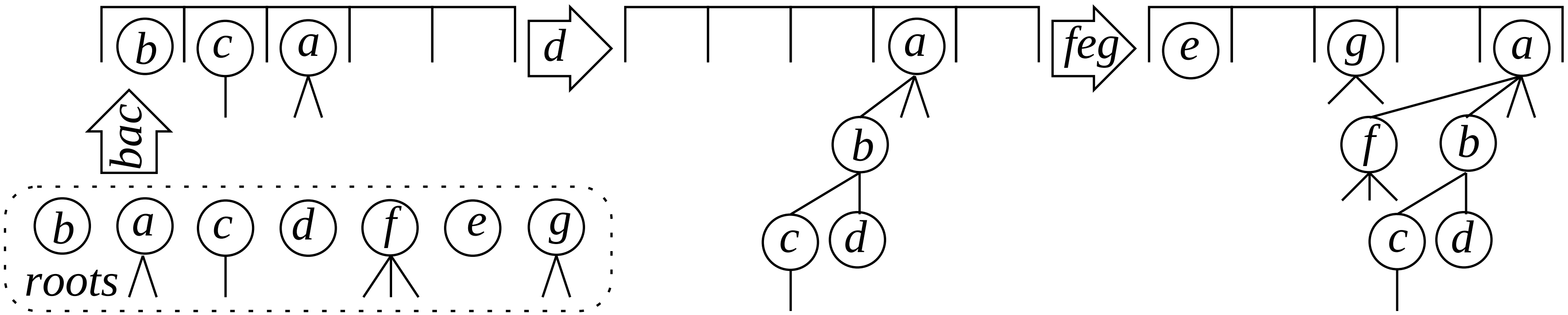
update minPtr

return m

Schnelles Union-by-Rank

Durch rank adressiertes Feld.

Solange link durchführen bis freier Eintrag gefunden.



Analyse: Zeit $\mathrm{O}(\#\text{unions} + |\text{forest}|)$

Amortisierte Analyse von deleteMin

$$\text{maxRank} := \max_{a \in \text{forest}} \text{rank}(a) \text{ (nachher)}$$

Lemma: $T_{\text{deleteMin}} = O(\text{maxRank})$

Beweis: Kontomethode. Ein Token pro Wurzel

$\text{rank}(\text{minPtr}) \leq \text{maxRank}$

$\rightsquigarrow$ Kosten $O(\text{maxRank})$ für newTrees und neue Token.

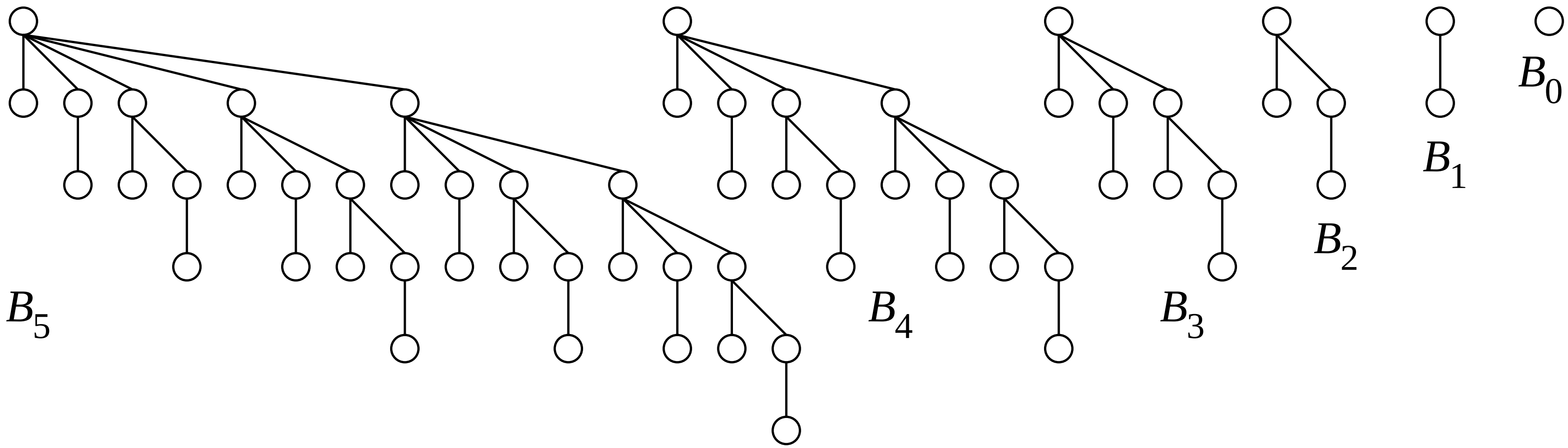
Union-by-rank: Token zahlen für

- union Operationen (ein Token wird frei) und
- durchlaufen alter und neuer Wurzeln.

Am Ende gibt es $\leq$ maxRank Wurzeln.

Warum ist maxRank logarithmisch? – Binomialbäume

$2^k + 1 \times$insert, $1 \times$deleteMin $\rightsquigarrow$ rank k



[Vuillemin 1978] PQ nur mit Binomialbäumen, $T_{\text{decreaseKey}} = \mathrm{O}(\log n)$.

Problem: Schnitte können zu kleinen hochrangigen Bäumen führen

Kaskadierende Schnitte

Procedure decreaseKey(h : Handle, k : Key)

  key(h):= k

  cascadingCut(h)

Procedure cascadingCut(h)

  if h is not a root **then**

    p:= parent(h)

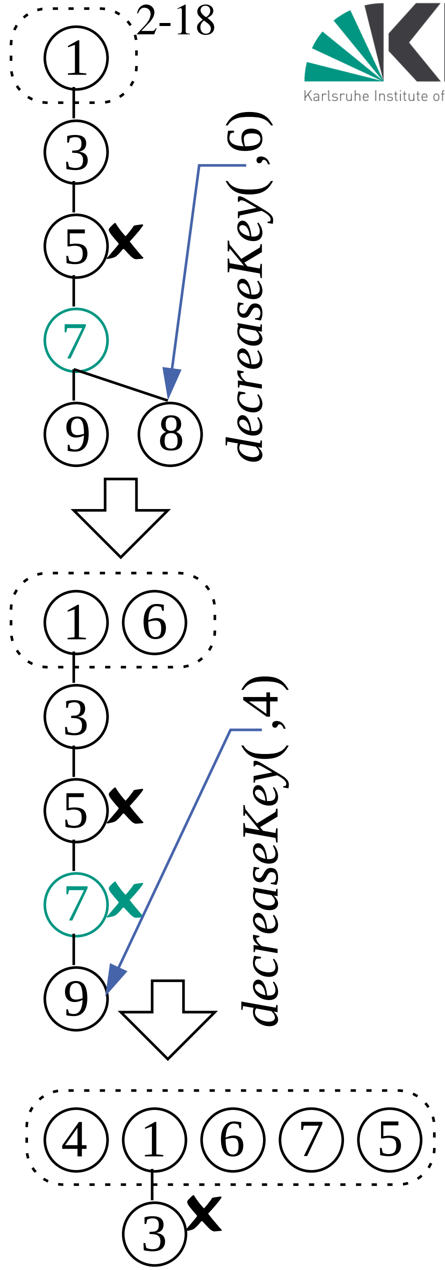
    unmark h

    cut(h)

    if p is marked **then**

      cascadingCut(p)

    else mark p



Wir werden zeigen: kaskadierende Schnitte halten maxRank logarithmisch

Lemma: decreaseKey hat amortisierte Komplexität $\mathrm{O}(1)$

Kontomethode: ($\approx$ 1 Token pro cut oder union)

1 Token für jede Wurzel

2 Token für jeden markierten Knoten

betrachte decreaseKey mit k konsekutiven markierten Vorgängern:

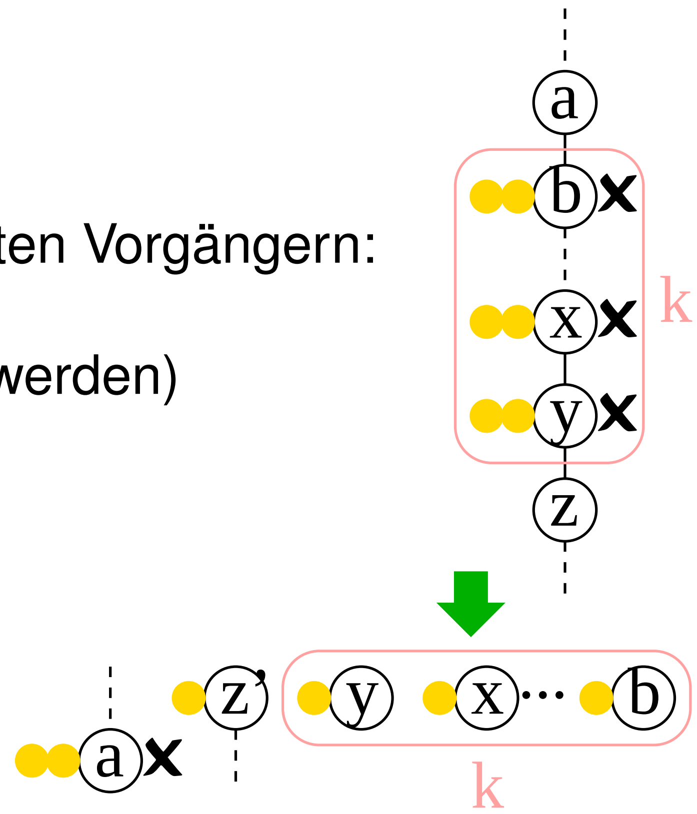
2k Token werden frei (von Knoten die entmarkiert werden)

2 Token für neue Markierung

k+1 Token für Ausstattung der neuen Wurzeln

k+1 Token für Schnitte

Bleiben 4 Token $+\mathrm{O}(1)$ Kosten für decreaseKey



Auftritt Herr Fibonacci

$$F_i := \begin{cases} 0 & \text{für i=0} \\ 1 & \text{für i=1} \\ F_{i-2} + F_{i-1} & \text{sonst} \end{cases}$$

Bekannt: $F_{i+1} \geq ((1+\sqrt{5})/2)^i \geq 1.618^i$ für alle $i \geq 0$.

Wir zeigen:

Ein Teilbaum T mit Wurzel v mit $\text{rank}(v) = i$ enthält $\geq F_{i+2}$ Elemente.

$1.618^i \leq F_{i+1} \leq F_{i+2} \leq |T| \leq n$

$\Rightarrow$

$i \leq \log_{1.618} n$

also logarithmische Zeit für deleteMin.

Beweis:

Betrachte Zeitpunkt als das j-te Kind w_j von v hinzugelinkt wurde:

w_j und v hatten gleichen Rang $\geq j-1$ (v hatte schon $j-1$ Kinder)

$\text{rank}(w_j)$ hat höchsten um eins abgenommen (cascading cuts)

$\Rightarrow$ $\text{rank}(w_j) \geq j-2$ und $\text{rank}(v) \geq j-1$

S_i:= untere Schranke für # Knoten mit Wurzel vom Rang i:

$S_0 = 1$

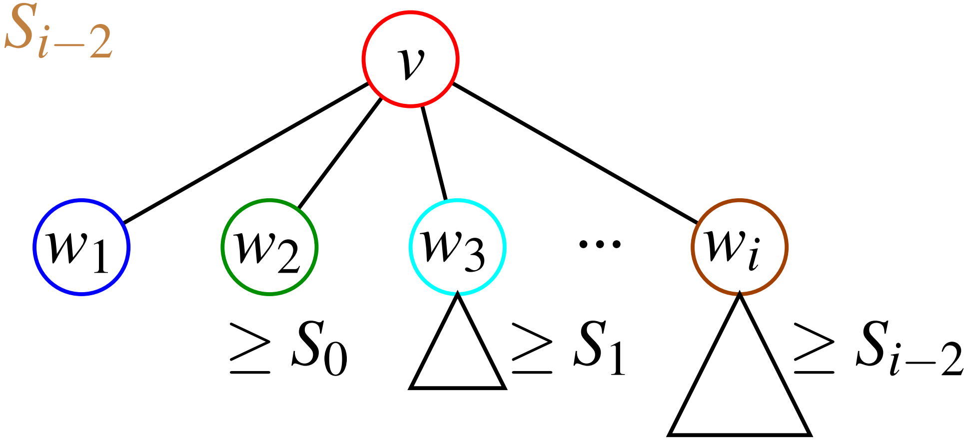
$S_1 = 2$

$S_i \geq 1 + 1 + S_0 + S_1 + \cdots + S_{i-2}$

für $i \geq 2$

Diese Rekurrenz

hat die Lösung $S_i \geq F_{i+2}$



Addressable Priority Queues: Mehr

- Untere Schranke $\Omega(\log n)$ für deleteMin, vergleichsbasiert.
  Beweis: Übung
- Worst case Schranken: nicht hier
- Monotone PQs mit ganzzahligen Schlüsseln (stay tuned)

Offene Probleme:

Analyse Pairing Heap

Zusammenfassung Datenstrukturen

- In dieser Vorlesung Fokus auf Beispiel Prioritätslisten
  (siehe auch kürzeste Wege, externe Algorithmen)
- Heapkonzept trägt weit
- Geschwisterzeiger erlauben Repräsentation beliebiger Bäume mit konstanter Zahl Zeiger pro Item.
- Fibonacci heaps als nichttriviales Beispiel für amortisierte Analyse
- Binomialbäume treten in der Algorithmik immer wieder mal auf
  (z.B. Broadcast und Reduktion in Parallelrechnern)