

6 Prioritätslisten



Prioritätslisten

Verwalte Menge M von Elementen mit Schlüsseln

Insert(e): $M := M \cup e$

DeleteMin: return and remove $\min M$

Anwendungen (ohne zusätzl. Operationen):

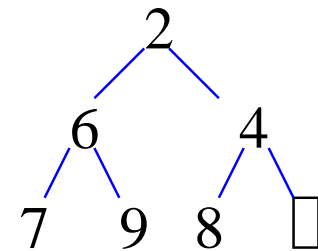
- Mehrwegemischen (klein)
- Greedy Algorithmen (z.B., Scheduling) (klein–mittel)
- Simulation diskreter Ereignisse (mittel–groß)
- Branch-and-Bound (groß)
- run formation (groß)
- Time forward processing (riesig)

6.1 Binäre Heaps

Heap-Eigenschaft: Bäume (oder Wälder) mit $\forall v : \text{parent}(v) \leq v$

Binärer Heap: Binärbaum, Höhe $\lfloor \log n \rfloor$, fehlende Blätter rechts unten.

insert, deleteMin brauchen **Zeit $O(\log n)$**



Sekundärspeicher-Prioritätslisten

$$\approx \frac{2n}{B} \left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right)$$

I/Os für n Einfügeoperationen

$O(n \log n)$ Arbeit.

[Sanders 1999].

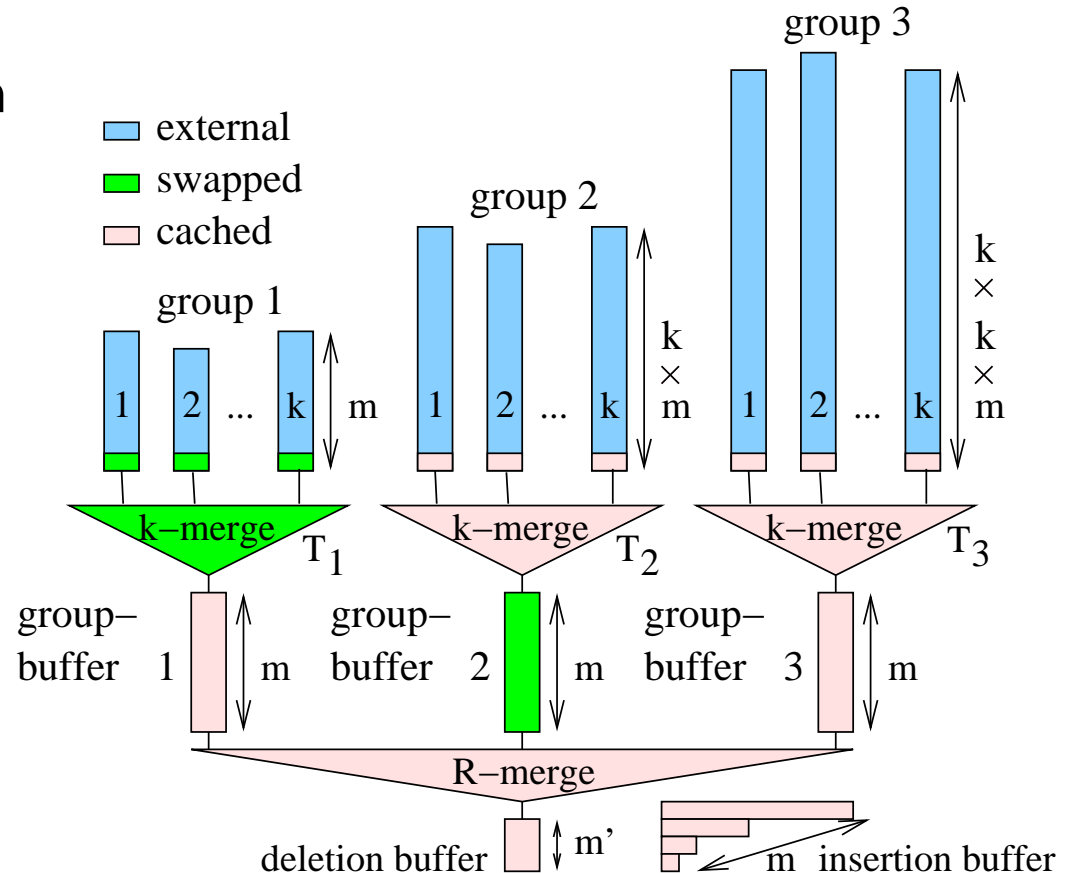
deleteMin:

“amortisiert umsonst”.

Details:

Vorlesung

Algorithm Engineering.





6.2 Adressierbare Prioritätslisten

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

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

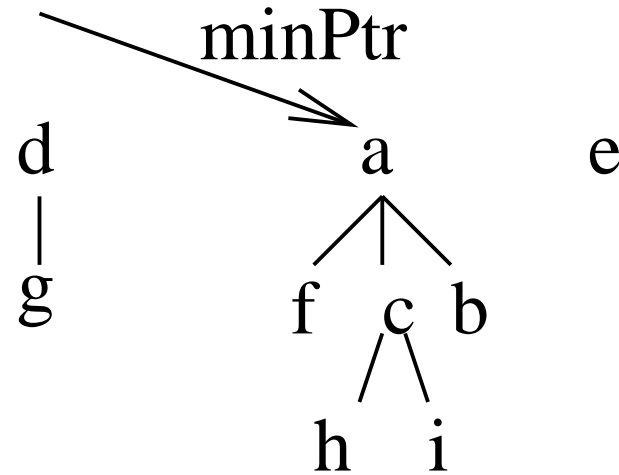
- Dijkstras Algorithmus für **kürzeste Wege**
- Jarník-Prim Algorithmus für **minimale Spannbäume**
- DA Robert Geisberger: Hierarchiekonstruktion für **Routenplanung**

Allgemein:

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

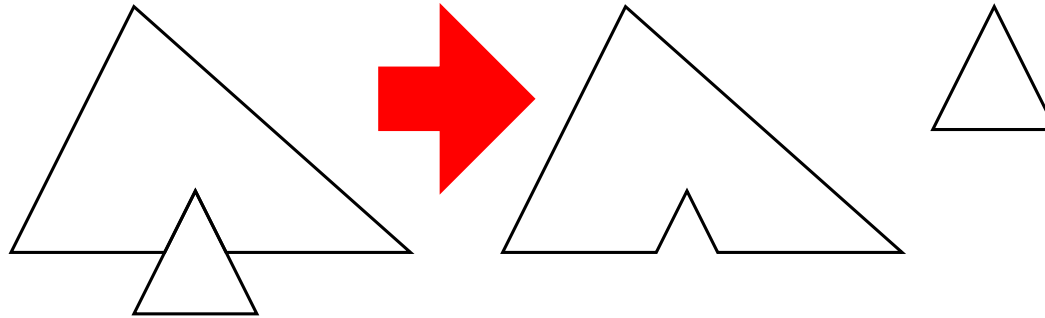
Grundlegende Datenstruktur

Ein **Wald heap-geordneter** Bäume

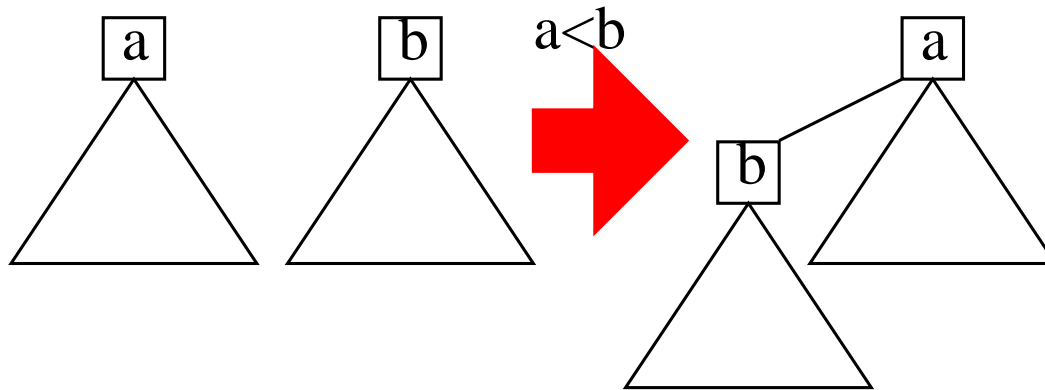


Wälder Bearbeiten

Cut:



Link:



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

Pairing Heaps (Paarungs-Haufen??)

[Fredman Sedgewick Sleator Tarjan 1986]

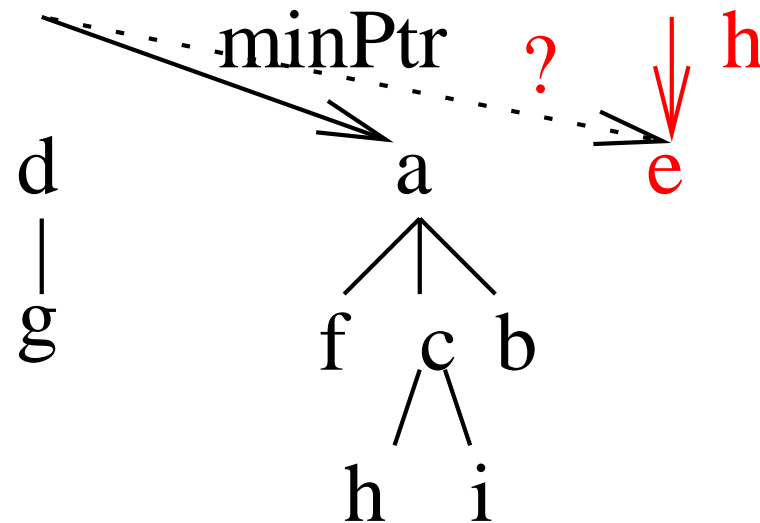
Procedure insertItem(h : Handle)

newTree(h)

Procedure newTree(h : Handle)

forest := forest \cup { h }

if $*h < \text{min}$ **then** minPtr := h

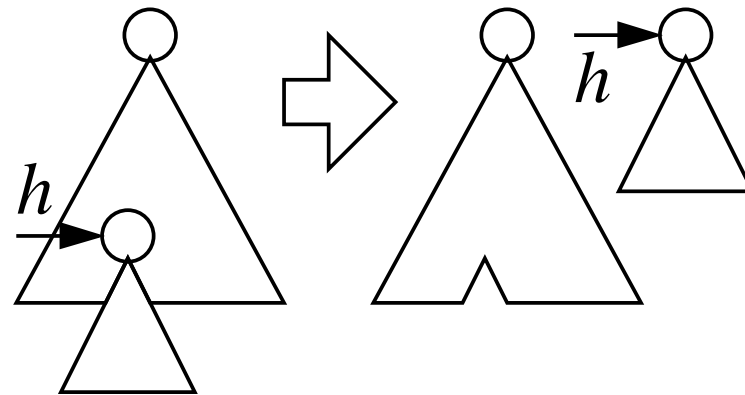


Pairing Heaps

Procedure decreaseKey(h : Handle, k : Key)

key(h) := k

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





Pairing Heaps

Function deleteMin : Handle

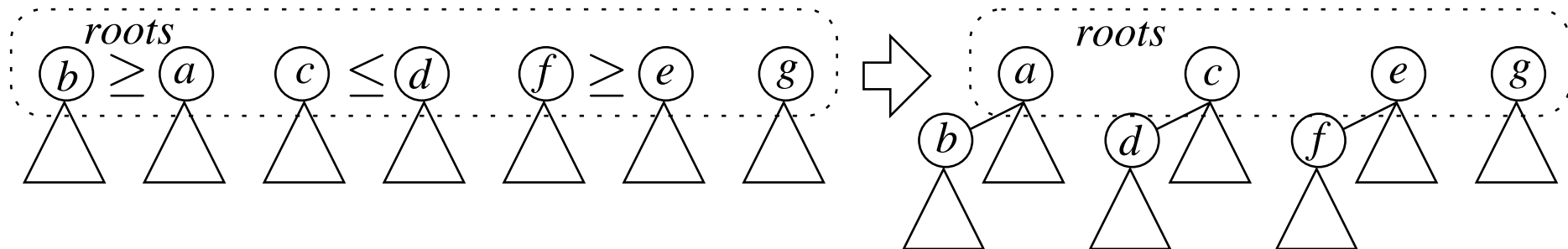
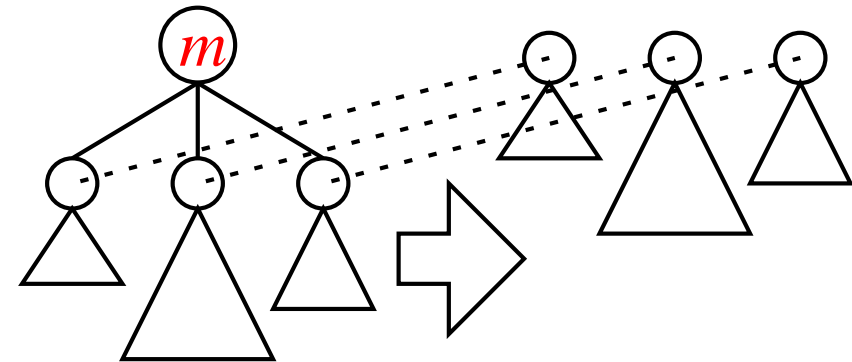
$m := \text{minPtr}$

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

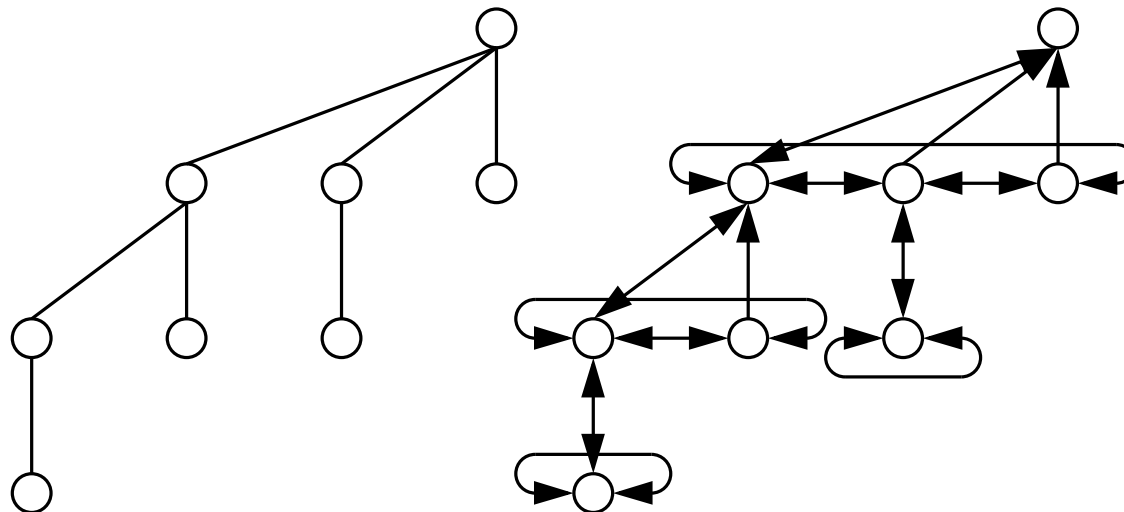
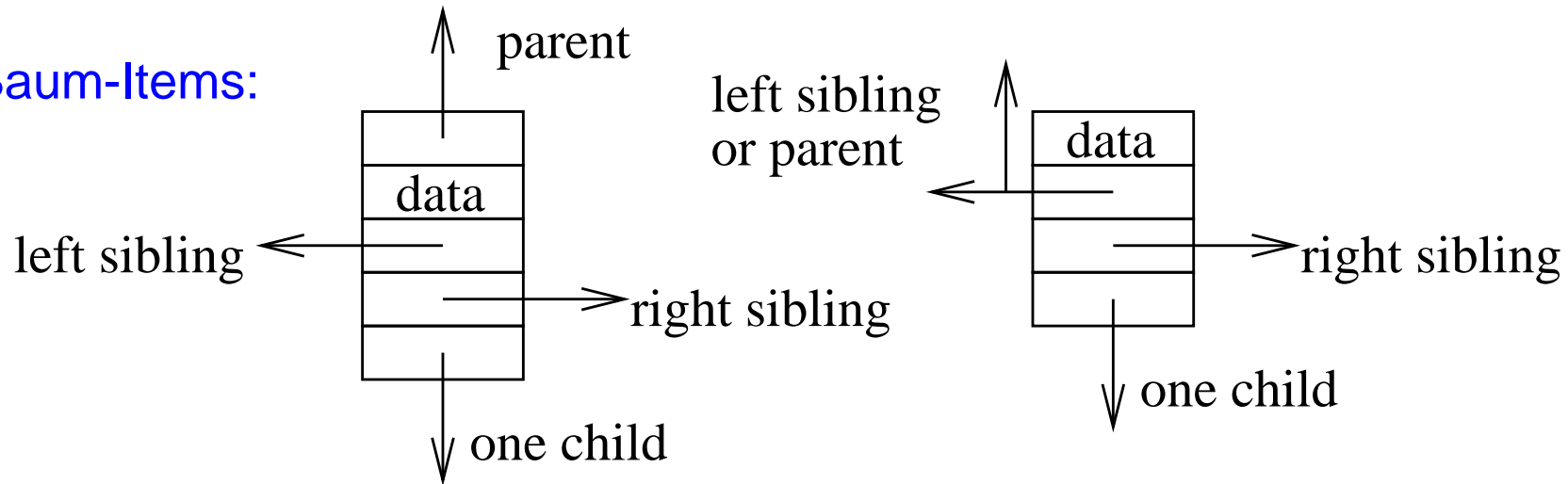
forest := forest \cup o .forest

o .forest := \emptyset

Pairing Heaps – Repräsentation

Wurzeln: Doppelt verkettete Liste

Baum-Items:



Pairing Heaps – Analyse

insert,merge: $O(1)$

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

decreaseKey: **unklar!** $O(\log \log n) \leq T \leq O(\log n)$ amortisiert.

In der Praxis sehr schnell.

Beweise: nicht hier.



Fibonacci Heaps [Fredman Tarjan 1987]

Rang: Anzahl 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 $O(\log n)$ for deleteMin und

$O(1)$ für alle anderen Operationen

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

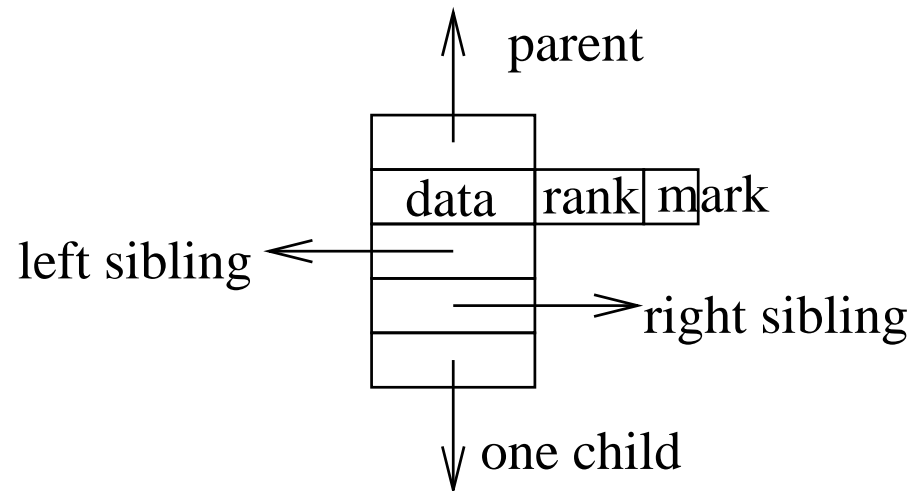
$d = \#deleteMin$, $o = \#otherOps$, $n = \max |M|$)

Repräsentation

Wurzeln: Doppelt verkettete Liste

(und ein tempräres Feld für deleteMin)

Baum-Items:



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

deleteMin mit Union-by-Rank

Function deleteMin : Handle

$m := \text{minPtr}$

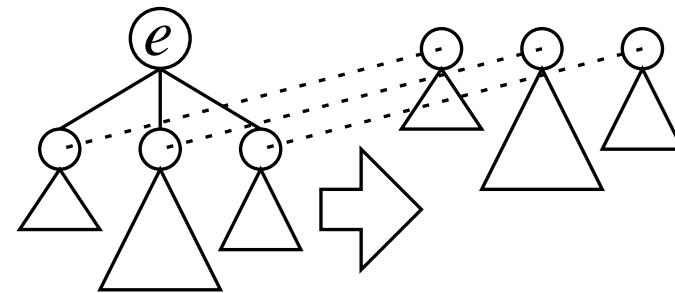
forest := forest \setminus { m }

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

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

update minPtr

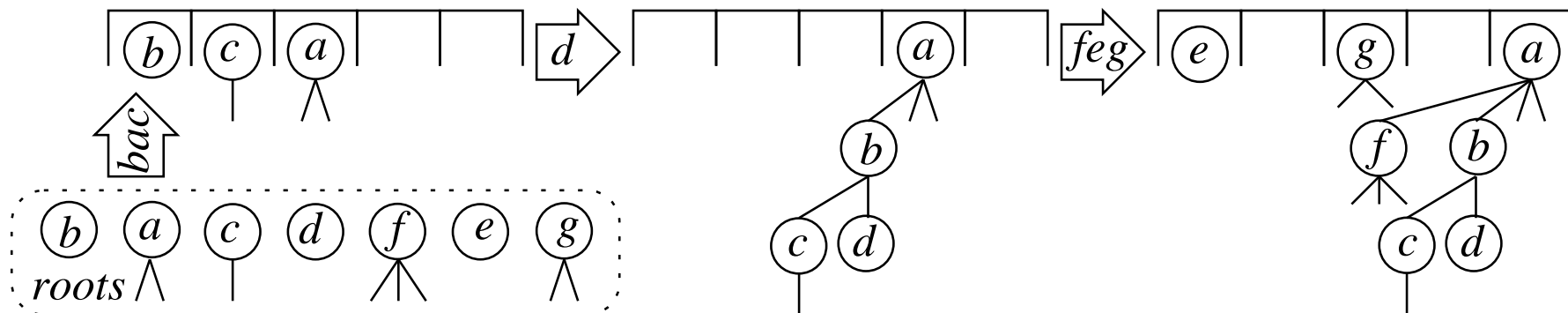
return m



Schnelles Union-by-Rank

Durch rank adressiertes Feld.

Solange link durchführen bis freier Eintrag gefunden.



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



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.

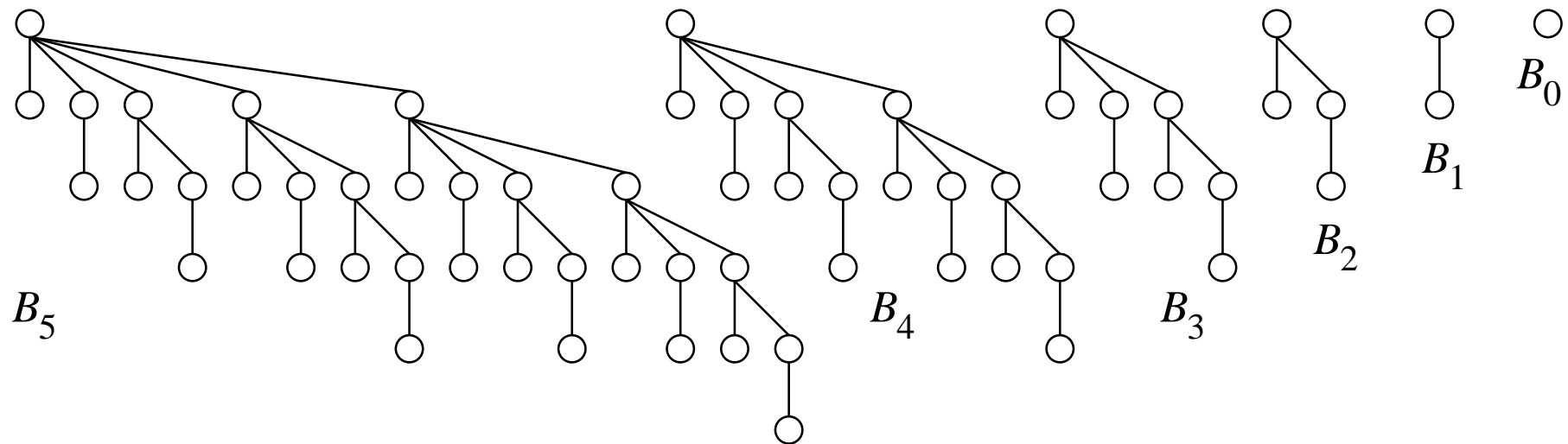
Am Ende gibt es $\leq \text{maxRank}$ Wurzeln,
die wir mit neuen Token ausstatten.



Warum ist maxRank logarithmisch? –

Binomialbäume

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



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

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)

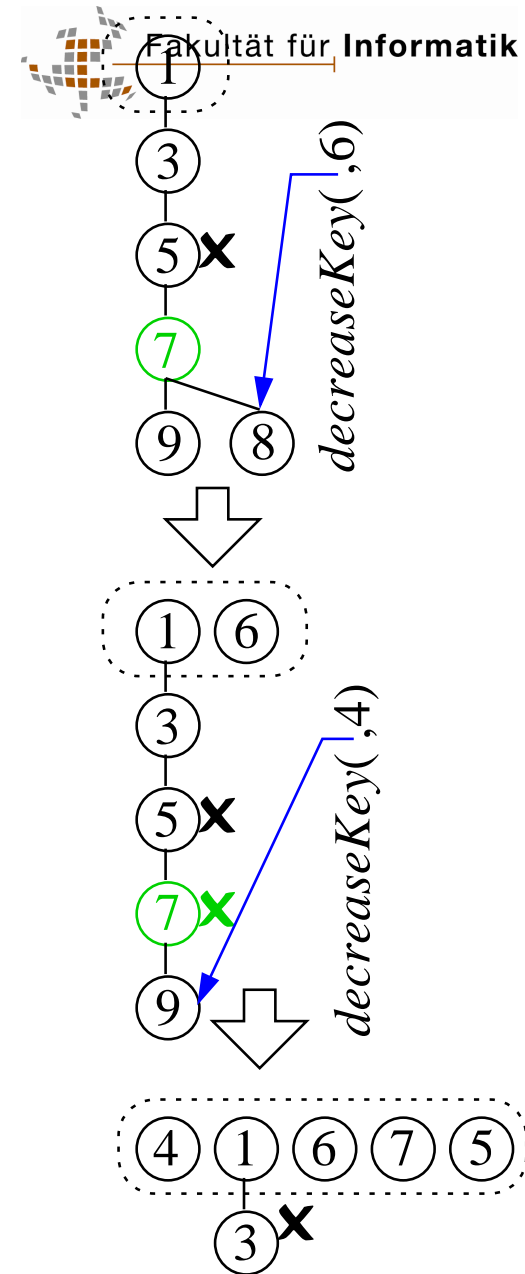
unmark h

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 $O(1)$

Kontomethode: (≈ 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 (unmarked nodes)

2 Token für neue Markierung

$k+1$ Token für Ausstattung der neuen Wurzeln

$k+1$ Token für Schnitte

Bleiben 4 Token $+O(1)$ 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: $F_{i+1} \geq (1 + \sqrt{5}/2)^i \geq 1.618^i$ for all $i \geq 0$.

Wir zeigen:

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

\Rightarrow

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 **höchstens 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 :

$$S_0 = 1$$

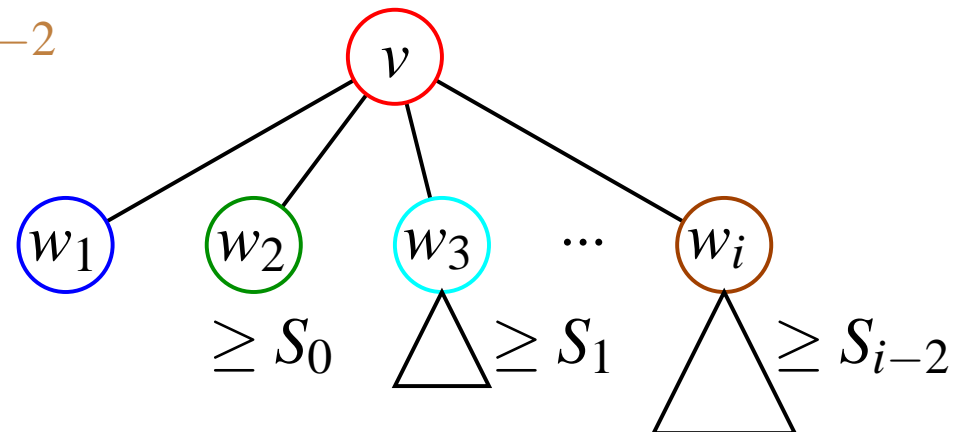
$$S_1 = 2$$

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

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, Vereinfachung Fibonacci Heap.