



Algorithmtheorie

08 – Fibonacci Heaps

Prof. Dr. S. Albers

Vorrangwarteschlangen: Operationen

Vorrangwarteschlange Q

Operationen:

$Q.initialize()$: erstellt die leere Schlange Q

$Q.isEmpty()$: liefert true gdw. Q ist leer

$Q.insert(e)$: fügt Eintrag e in Q ein und gibt einen Zeiger auf den Knoten, der den Eintrag e enthält, zurück

$Q.deletemin()$: liefert den Eintrag aus Q mit minimalem Schlüssel und entfernt ihn

$Q.min()$: liefert den Eintrag aus Q mit minimalem Schlüssel

$Q.decreasekey(v,k)$: verringert den Schlüssel von Knoten v auf k

Vorrangwarteschlangen: Operationen

Zusätzliche Operationen:

Q.delete(v) : entfernt Knoten v mit Eintrag aus Q (ohne v zu suchen)

Q.meld(Q'): vereinigt Q und Q' (concatenable queue)

Q.search(k) : sucht den Eintrag mit Schlüssel k in Q (searchable queue)

u.v.a., z.B. *predecessor, successor, max, deletemax*

Vorrangwarteschlangen Implementation

	Liste	Heap	Bin. – Q.	Fib.-Hp.
insert	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
min	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
delete-min	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
meld ($m \leq n$)	$O(1)$	$O(n)$ od. $O(m \log n)$	$O(\log n)$	$O(1)$
decr.-key	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)^*$

* = amortisierte Kosten

$$Q.delete(e) = Q.decreasekey(e, -\infty) + Q.deletemin()$$

Fibonacci - Heaps

„Lazy-Meld“ - Version von Binomialqueues:

Vereinigung von Bäumen gleicher Ordnung wird bis zur nächsten **deletemin**-Operation aufgeschoben

Definition

Ein **Fibonacci-Heap** Q ist eine Kollektion heapgeordneter Bäume

Variablen

$Q.min$: Wurzel des Baumes mit kleinstem Schlüssel

$Q.rootlist$: zirkuläre, doppeltverkettete und ungeordnete Liste der Wurzeln der Bäume

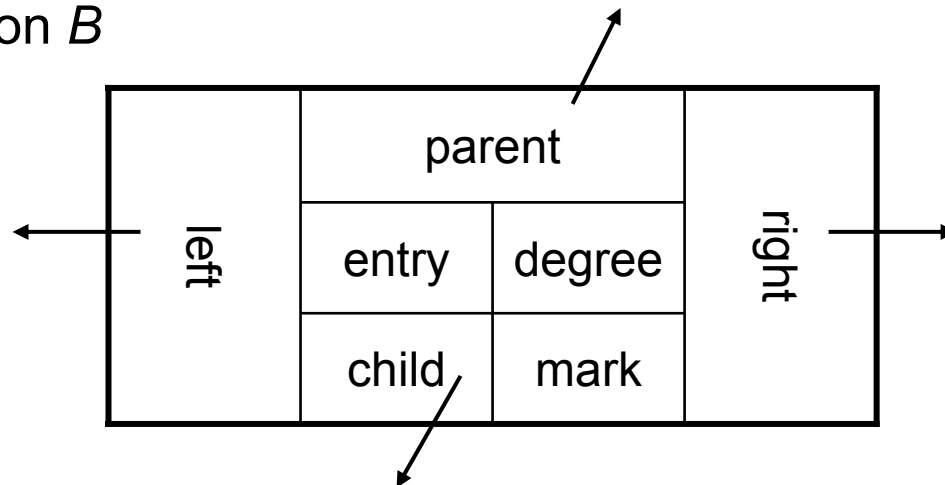
$Q.size$: Anzahl der Knoten in Q

Fibonacci-Heaps Bäume

Sei B heapgeordneter Baum in $Q.rootlist$:

B.childlist: zirkuläre, doppelverkettete und ungeordnete Liste der Söhne von B

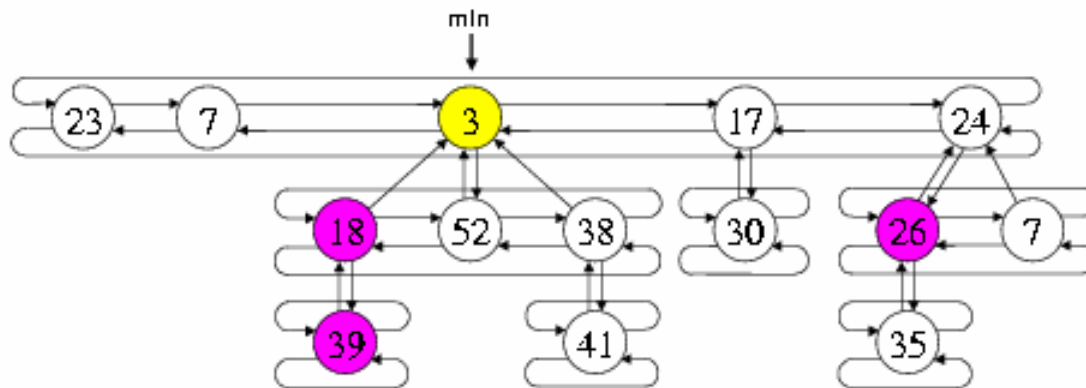
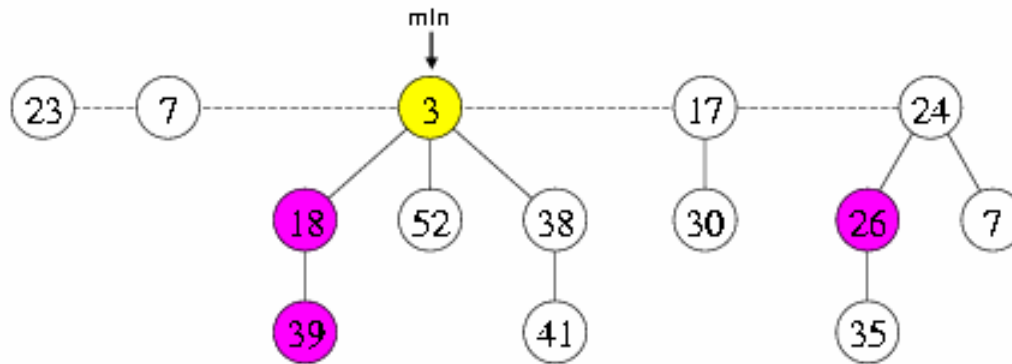
Knotenformat



Vorteile zirkulärer, doppelverketteter Listen:

1. Entfernen eines Elementes in konstanter Zeit
2. Vereinigen zweier Listen in konstanter Zeit

Fibonacci-Heaps Implementation: Beispiel



Fibonacci-Heaps Operationen

Q.initialize(): $Q.rootlist = Q.min = null$

Q.meld(Q´):

1. verkette $Q.rootlist$ und $Q´.rootlist$
2. update $Q.min$

Q.insert(e):

1. Bilde einen Knoten für einen neuen Schlüssel $\rightarrow Q´$
2. $Q.meld(Q´)$

Q.min():

Gebe $Q.min.entry$ zurück

Fibonacci-Heaps *deletemin*

Q.deletemin()

*/**Lösche den Knoten mit den kleinsten Schlüssel aus *Q* und gebe den Eintrag des Knotens zurück**/*

1 *m* = *Q.min()*

2 **if** *Q.size()* > 0

3 **then** entferne *Q.min()* aus *Q.rootlist*

4 füge *Q.min.childlist* in *Q.rootlist* ein

5 *Q consolidate*

*/** Verschmelze solange Knoten mit
gleichem Grad, bis das nicht mehr geht
und bestimme dann das minimale Element *m* **/*

6 **return** *m*

Fibonacci-Heaps: Maximaler Knotengrad

$\text{rang}(v)$ = Grad eines Knotens v in Q

$\text{rang}(Q)$ = maximaler Grad eines Knotens in Q

Annahme :

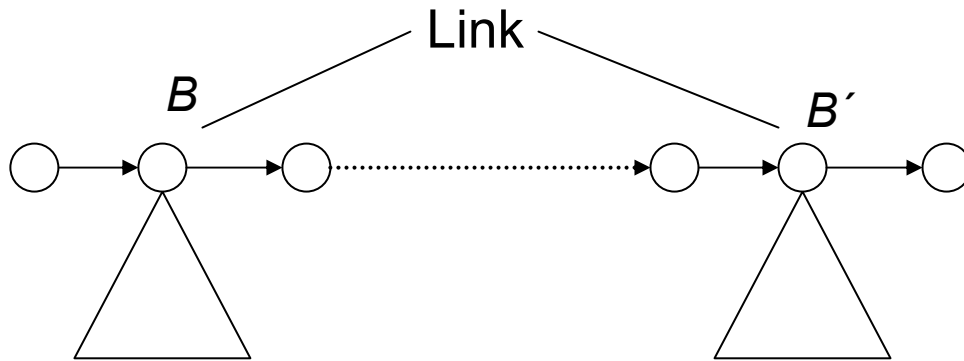
$$\text{rang}(Q) \leq 2 \log n,$$

wenn $Q.\text{size} = n$.

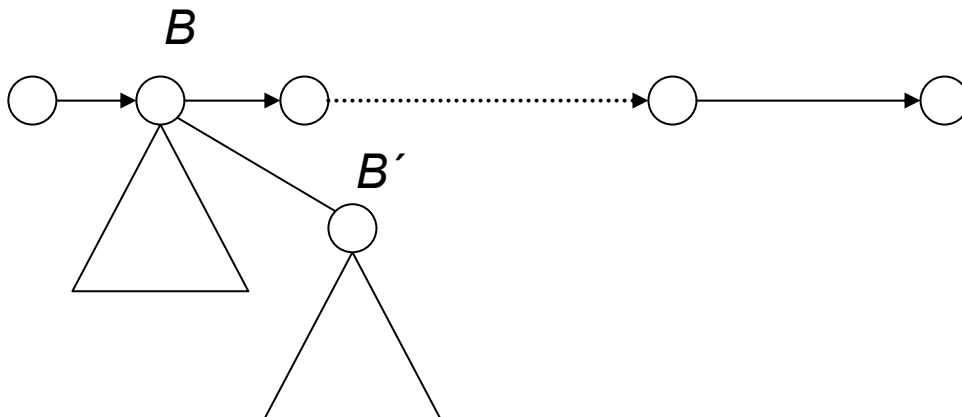
Fibonacci-Heaps: *Link-Operation*

$\text{rang}(B)$ = Grad der Wurzel von B

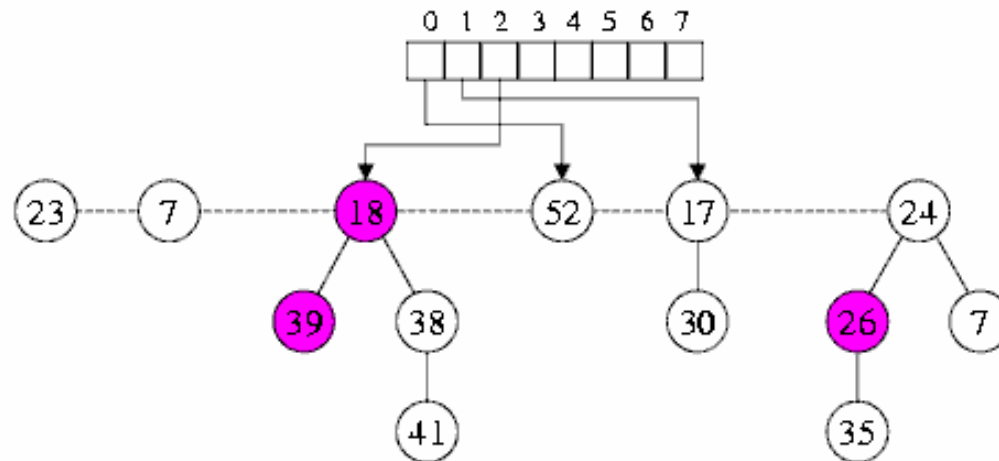
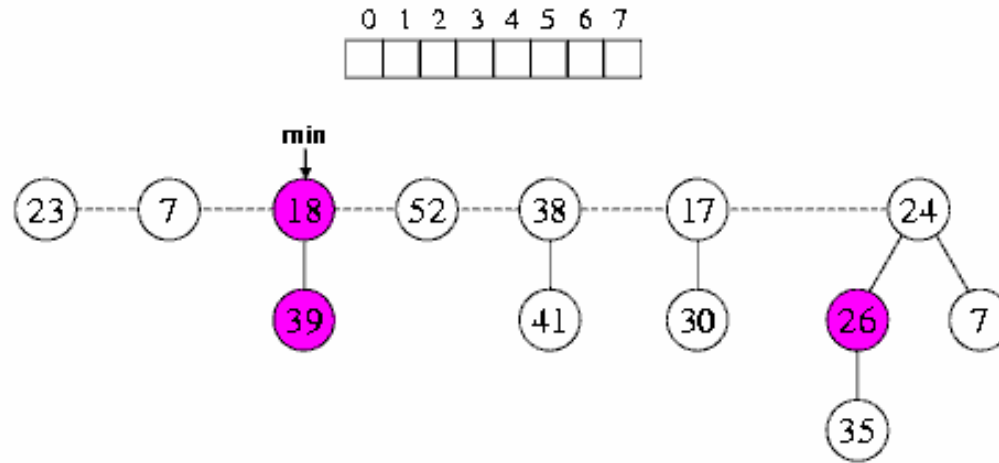
Heapgeordnete Bäume B, B' mit $\text{rang}(B) = \text{rang}(B')$



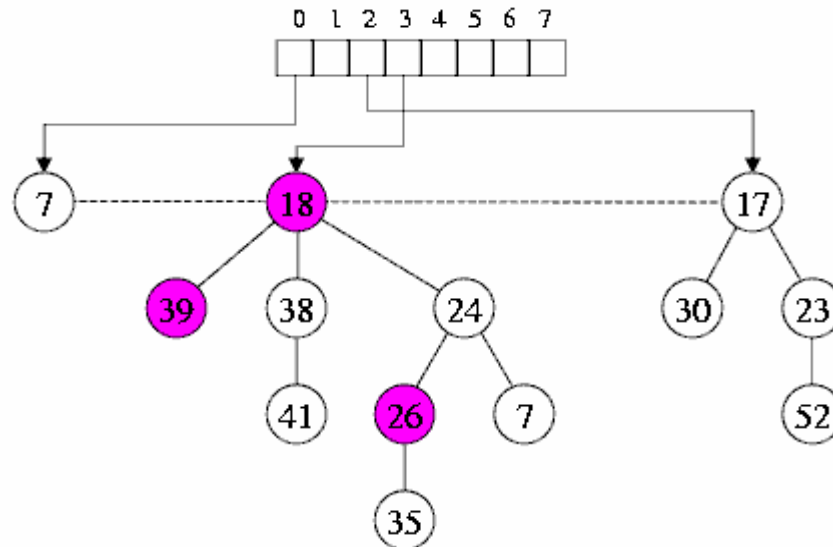
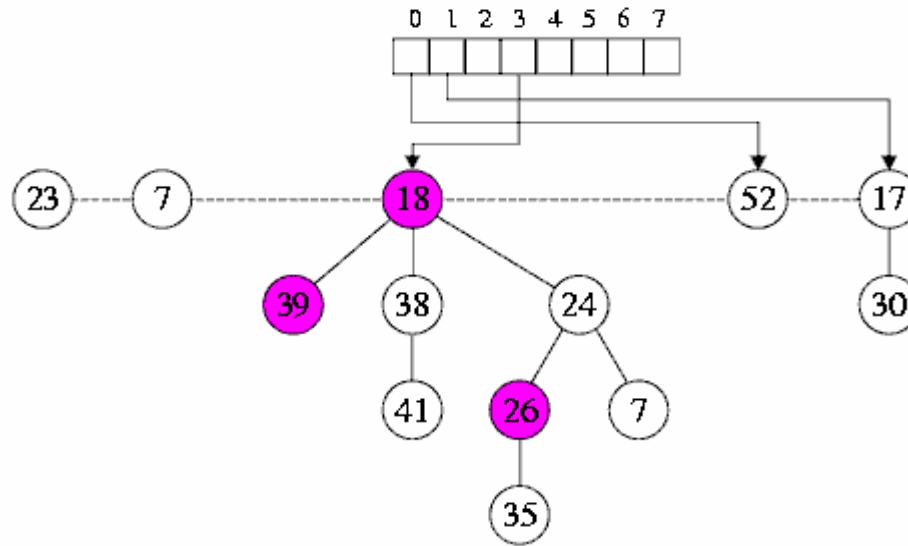
1. $\text{rang}(B) = \text{rang}(B) + 1$
2. $B'.\text{mark} = \text{false}$



Konsolidierung der Wurzelliste



Konsolidierung der Wurzelliste



Fibonacci-Heaps *deletemin*

Finde Wurzel mit gleichem Rang:

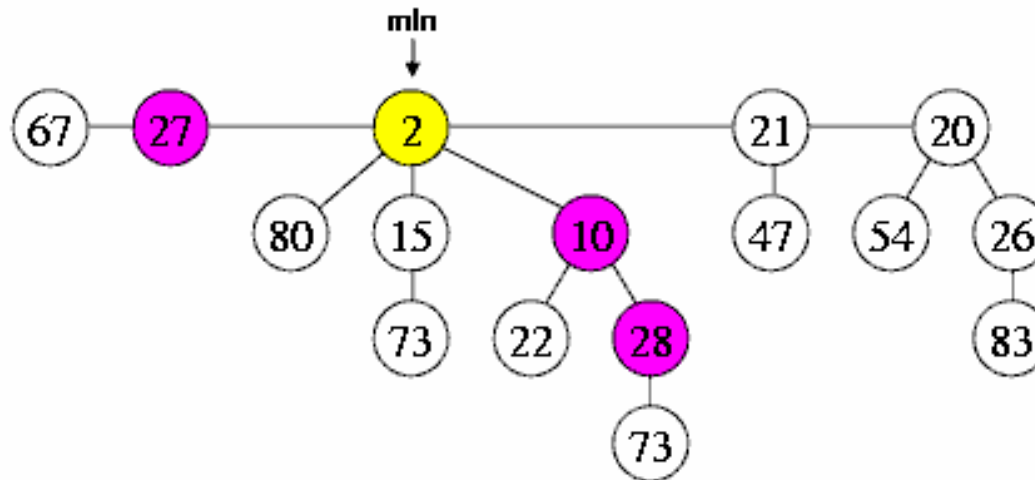
Array **A**:



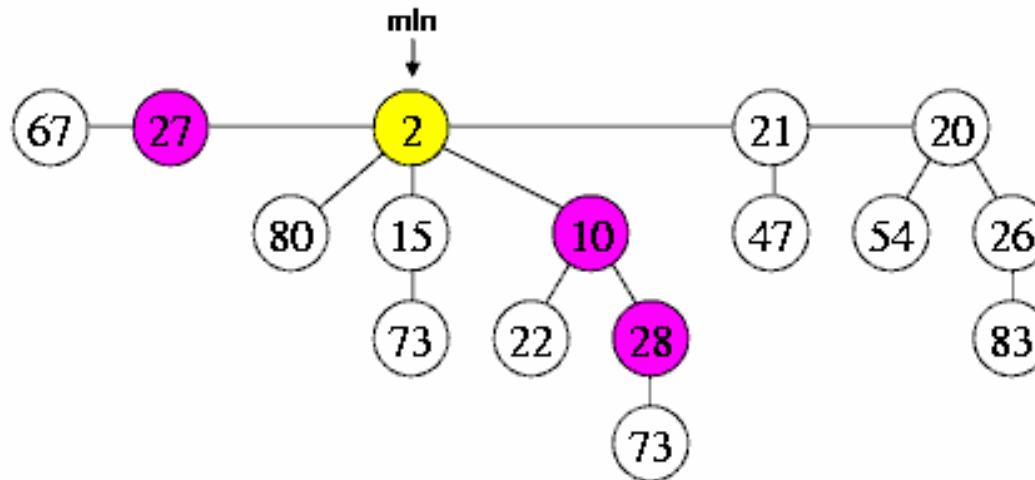
Q.consolidate()

- 1 A = Array von Fibonacci-Heap Knoten der Länge $2 \log n$
- 2 **for** $i = 0$ **to** $2 \log n$ **do** $A[i] = \text{frei}$
- 3 **while** $Q.\text{rootlist} \neq \emptyset$ **do**
- 4 $B = Q.\text{delete-first}()$
- 5 **while** $A[\text{rang}(B)]$ ist nicht frei **do**
- 6 $B' = A[\text{rang}(B)]; A[\text{rang}(B)] = \text{frei}; B = \text{Link}(B, B')$
- 7 **end while**
- 8 $A[\text{rang}(B)] = B$
- 9 **end while**
- 10 bestimme $Q.\text{min}$

Fibonacci-Heap Beispiel



Fibonacci-Heap Beispiel



Fibonacci-Heaps *decrease-key*

Q.decrease-key(v,k)

```
1  if  $k > v.key$  then return
2   $v.key = k$ 
3  update  $Q.min$ 
4  if  $v \in Q.rootlist$  or  $k \geq v.parent.key$  then return
5  do /* cascading cuts */
6      $parent = v.parent$ 
7      $Q.cut(v)$ 
8      $v = parent$ 
9  while  $v.mark$  and  $v \notin Q.rootlist$ 
10 if  $v \notin Q.rootlist$  then  $v.mark = true$ 
```

Fibonacci-Heaps: cuts

Q.cut(v)

```
1 if  $v \notin Q.rootlist$ 
2 then /* trenne  $v$  von seinem Vater */
3      $rang(v.parent) = rang(v.parent) - 1$ 
4      $v.parent = null$ 
5     entferne  $v$  aus  $v.parent.childlist$ 
6     füge  $v$  in  $Q.rootlist$  ein
```

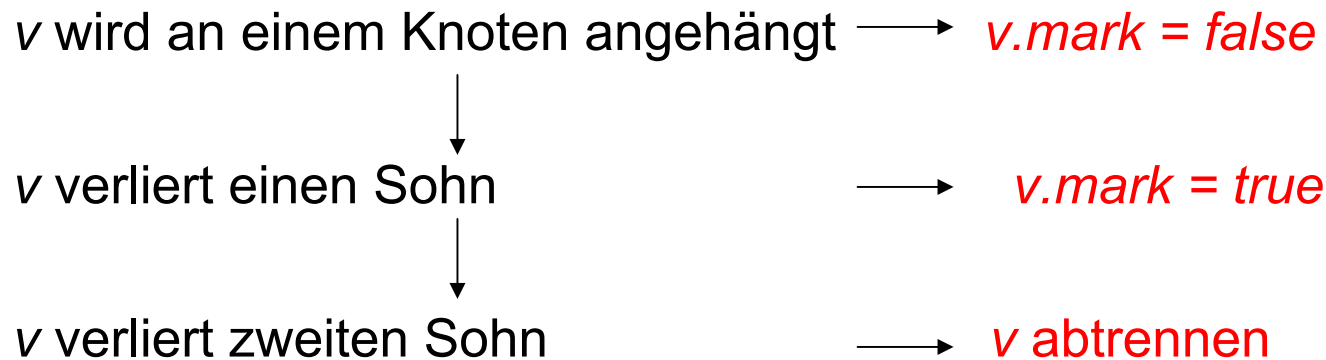
Fibonacci-Heaps: delete

Q.delete(v)

- 1 **if** $v = Q.min$
- 2 **then** return *Q.delete-min*
- 3 *Q.cut(v)* und kaskadiere
- 4 entferne v aus *Q.rootlist*
- 5 füge $v.childlist$ in *Q.rootlist* ein
- 6 **return** $v.key$

Fibonacci-Heaps Markierung

Historie



Die Markierung gibt also an, ob v bereits einen Sohn verloren hat, seitdem v zuletzt Sohn eines anderen Knoten geworden ist.

Rang der Söhne eines Knotens

Lemma

Sei v ein Knoten in dem Fibonacci-Heap Q . Ordnet man die Söhne u_1, \dots, u_k von v nach dem Zeitpunkt des Anhängens an v , dann gilt

$$\text{rang}(u_i) \geq i - 2.$$

Beweis:

Zeitpunkt des Anhängens von u_i :

Söhne von v ($\text{rang}(v)$): $\geq i - 1$

Söhne von u_i ($\text{rang}(u_i)$): $\geq i - 1$

Söhne, die u_i in der Zwischenzeit verloren hat: 1

Maximaler Rang eines Knotens

Satz

Sei v ein Knoten mit Rang k in einem Fibonacci-Heap Q . Dann ist v die Wurzel eines Teilbaumes mit mindestens F_{k+2} Knoten.

Die Anzahl der Nachkommen eines Knotens ist **exponentiell** in der Zahl seiner Söhne.

Folgerung

Für den maximalen Rang k eines Knoten v in einem Fibonacci-Heap Q mit n Knoten gilt:

Maximaler Rang eines Knotens

Beweis

S_k = Mindestgröße eines Teilbaumes, dessen Wurzel k Söhne hat

$$S_0 = 1$$

$$S_1 = 2$$

Es gilt:

$$S_k \geq 2 + \sum_{i=0}^{k-2} S_i \quad \text{für } k \geq 2 \quad (1)$$

Fibonacci-Zahlen:

$$F_{k+2} = 1 + \sum_{i=0}^k F_i \quad (2)$$

$$= 1 + F_0 + F_1 + \dots + F_k$$

$$(1) + (2) + \text{Induktion} \rightarrow S_k \geq F_{k+2}$$

Fibonacci-Heap Analyse

Potentialmethode zur Analyse der Kosten der Operationen der Fibonacci-Heaps.

Potentialfunktion Φ_Q für Fibonacci-Heap Q :

$$\Phi_Q = r_Q + 2 m_Q$$

mit:

r_Q = Anzahl der Knoten in $Q.rootlist$

m_Q = Anzahl der markierten Knoten in Q , die sich nicht in der Wurzelliste befinden.

Amortisierte Analyse

Amortisierte Kosten a_i der i -ten Operation:

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= t_i + (r_i - r_{i-1}) + 2(m_i - m_{i-1}) \end{aligned}$$

Analyse: *insert*

insert

$$t_i = 1$$

$$r_i - r_{i-1} = 1$$

$$m_i - m_{i-1} = 0$$

$$a_i = 1 + 1 + 0 = O(1)$$

Analyse: *Deletemin*

deletemin:

$$t_i = r_{i-1} + 2 \log n$$

$$r_i - r_{i-1} \leq 2 \log n - r_{i-1}$$

$$m_i - m_{i-1} \leq 0$$

$$\begin{aligned} a_i &\leq r_{i-1} + 2 \log n + 2 \log n - r_{i-1} + 0 \\ &= O(\log n) \end{aligned}$$

Analyse: *Decrease-key*

decrease-key:

$$t_i = c + 2$$

$$r_i - r_{i-1} = c + 1$$

$$m_i - m_{i-1} \leq -c + 1$$

$$a_i \leq c + 2 + c + 1 + 2(-c + 1)$$

$$= O(1)$$

Analyse: *Delete*

Q.delete(v):

Q.decrease-key(v, $-\infty$) und *Q.deletemin()*

Vorrangwarteschlangen Vergleich

	Liste	Heap	Bin. – Q.	Fib.-Hp.
insert	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
min	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
delete-min	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
meld ($m \leq n$)	$O(1)$	$O(n)$ od. $O(m \log n)$	$O(\log n)$	$O(1)$
decr.-key	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)^*$

* = amortisierte Kosten