Fibonacci Heaps

Vorrangswarteschlangen:Operationen

(Vorrangs)warteschlange *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*

Vorrangswarteschlangen: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*

Vorrangswarteschlangen 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, -∞) + 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

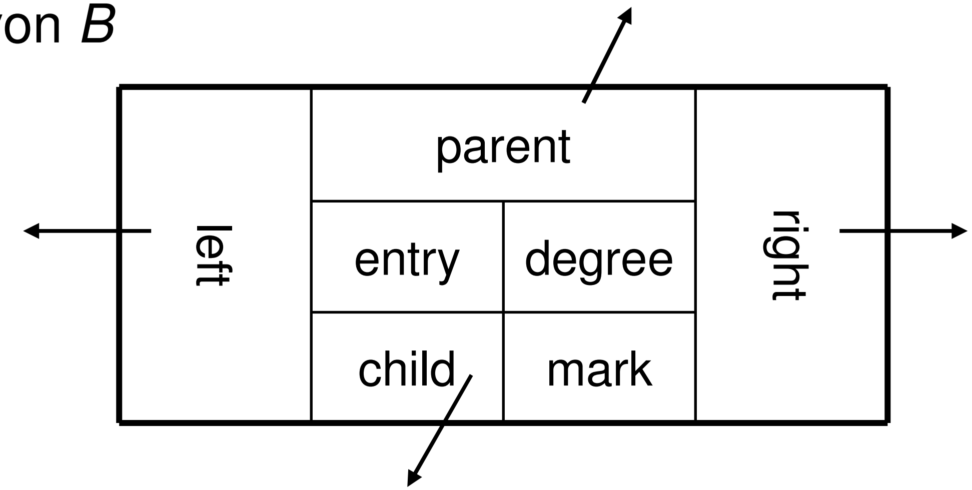
Q.size: Anzahl der Knoten in *Q*

Fibonacci-Heaps Bäume

Sei *B* heapgeordneter Baum in *Q.rootlist*:

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

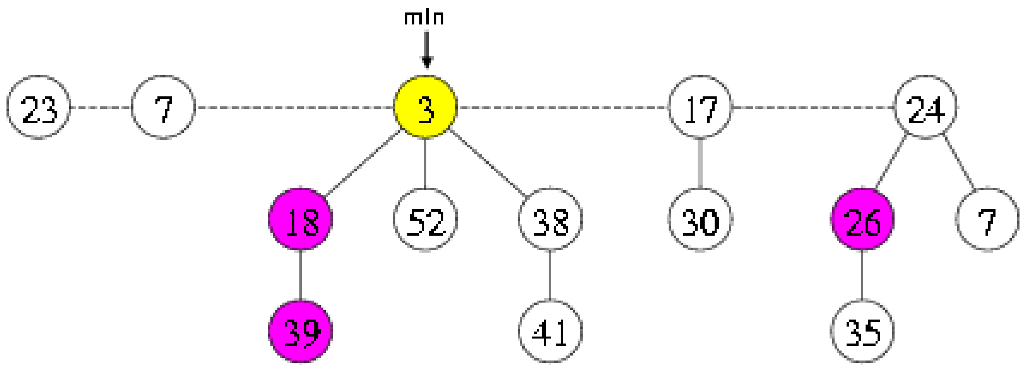
Knotenformat

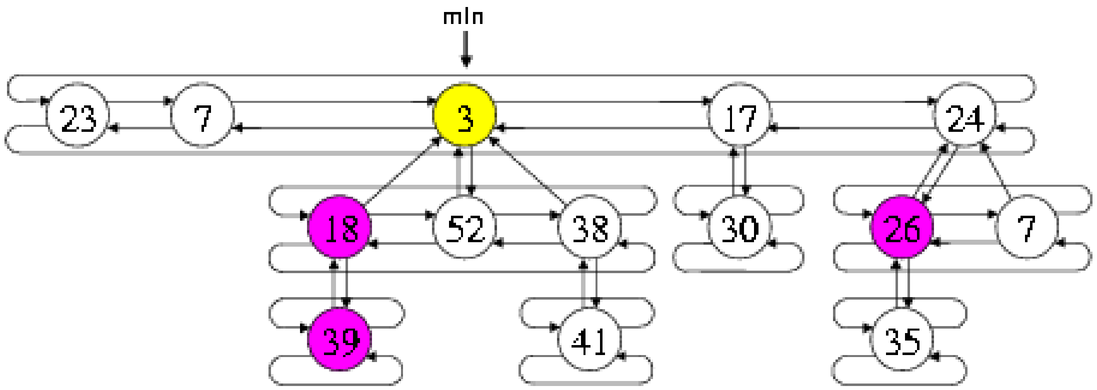


Vorteile zirkulärer, doppelverketteter Listen:

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 → *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*/

```
m = Q.min()
if Q.size() > 0
    then entferne Q.min() aus Q.rootlist
         füge Q.min.childlist in Q.rootlist ein
    Q.consolidate
  /* Verschmelze solange Knoten mit
     gleichem Grad, bis das nicht mehr geht
     und bestimme dann das minimale Element m */
return m
```

Fibonacci-Heaps: Maximaler Knotengrad

rang(v) = Grad eines Knotens v in *Q*

rang(Q) = maximaler Grad eines Knotens in *Q*

Annahme :

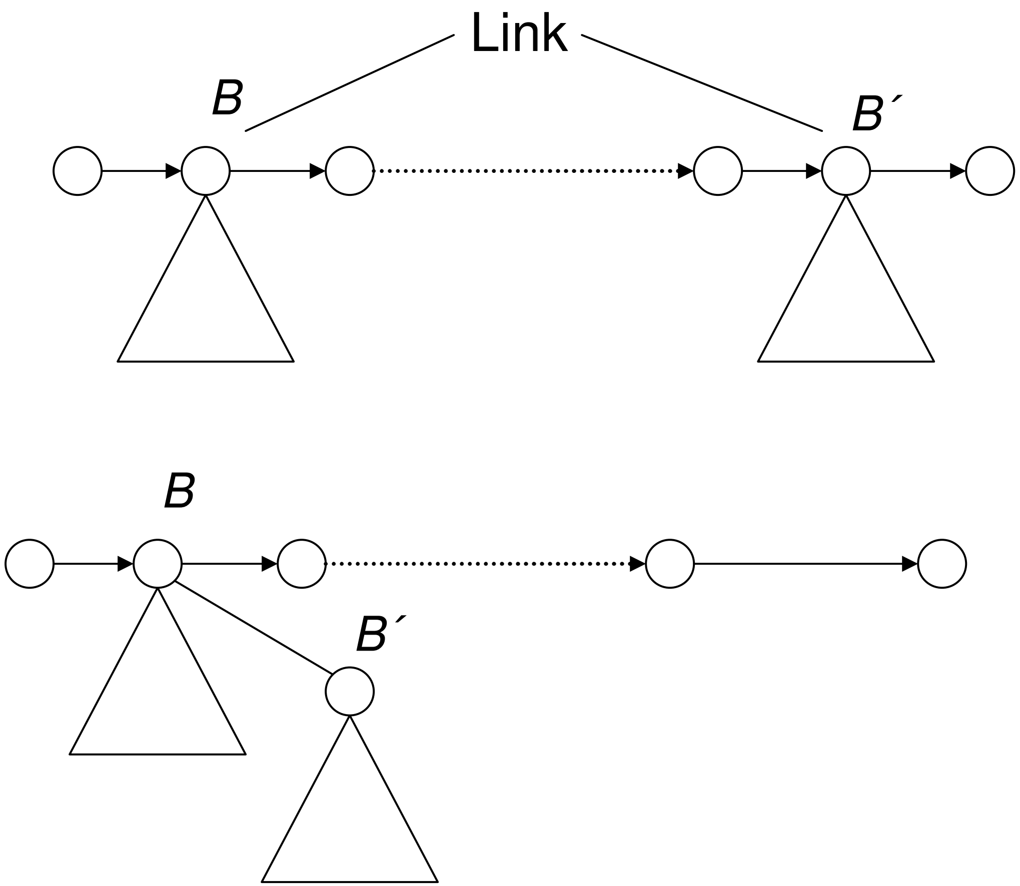
$$rang(Q) \leq 2\log n,$$

wenn $Q.size = n$.

Fibonacci-Heaps: *Link*-Operation

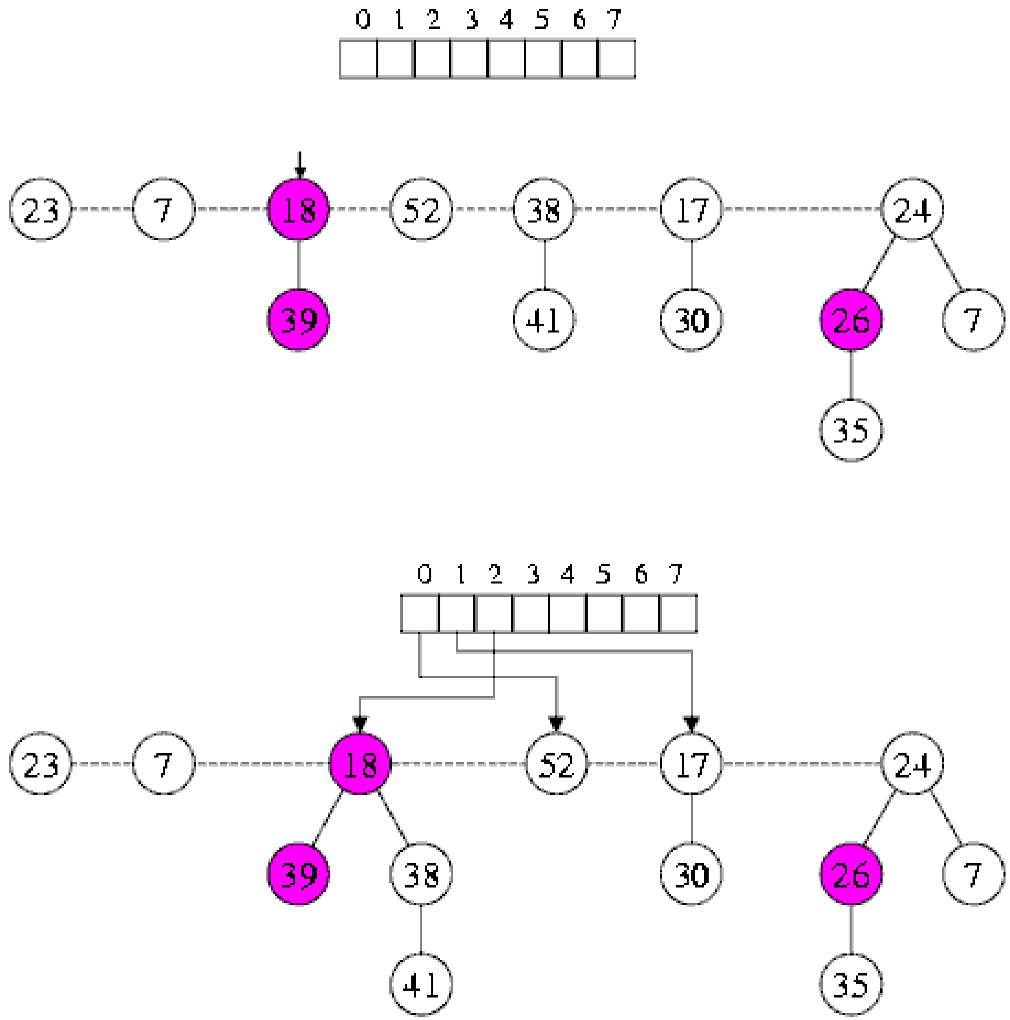
rang(B) = Grad der Wurzel von *B*

Heapgeordnete Bäume *B,B´* mit *rang(B) = rang(B´)*

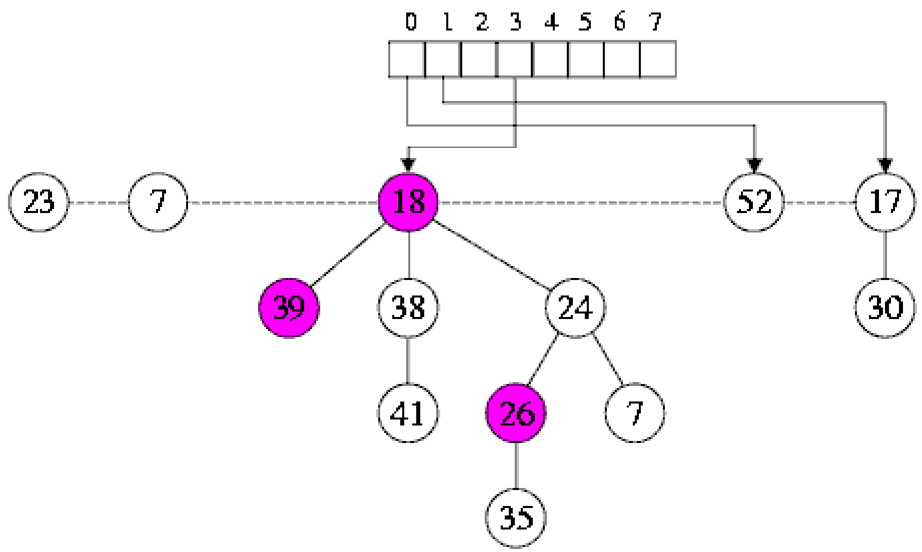


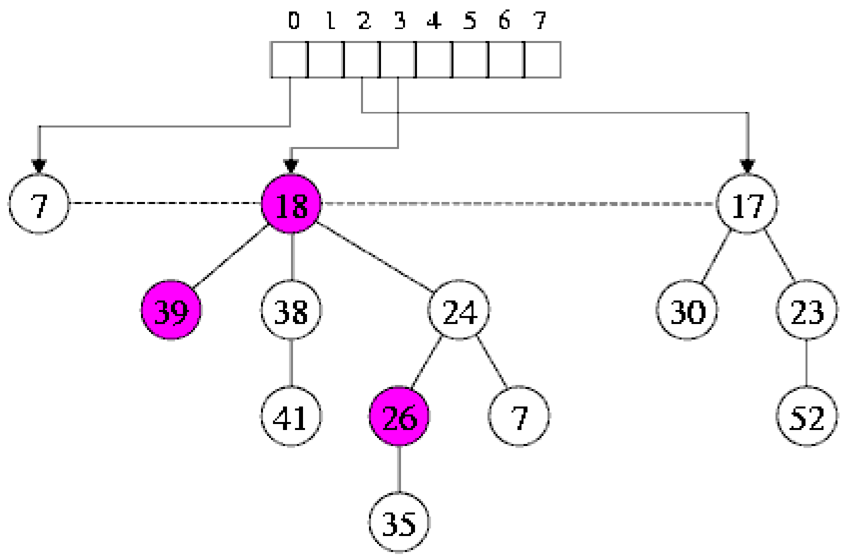
1. *rang(B) = rang(B)* + 1
2. *B´.mark = false*

Konsolidierung der Wurzelliste



Konsolidierung der Wurzelliste





Fibonacci-Heaps *deletemin*

Finde Wurzel mit gleichem Rang:

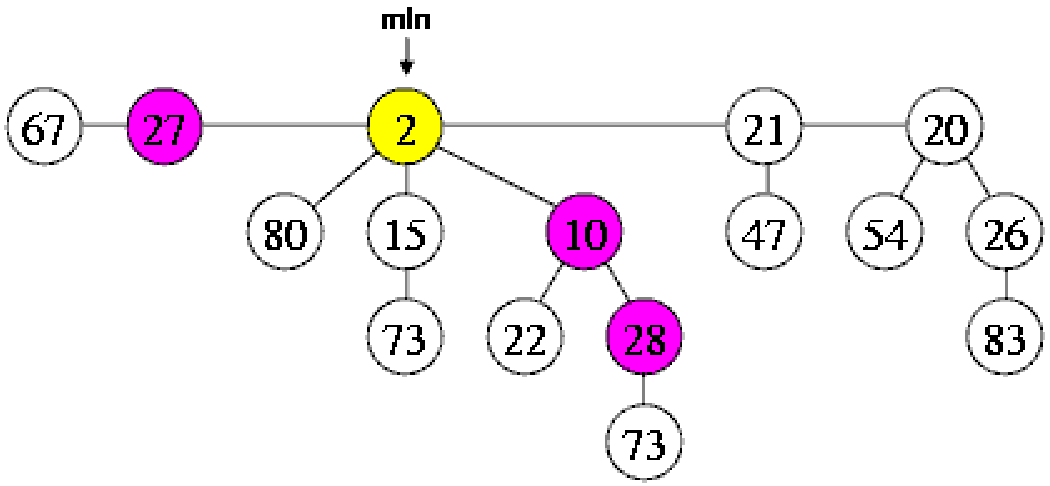
Array *A*:

0 1 2 log *n*

Q.consolidate()

```
A = Array von Fibonacci-Heap Knoten der Länge 2 log n
for i = 0 to 2 log n do A[i] = frei
while Q.rootlist ≠ ∅ do
        B = Q.delete-first()
        while A[rang(B)] ist nicht frei do
                B´ = A[rang(B)]; A[rang(B)] = frei; B = Link(B,B´)
        end while
        A[rang(B)] = B
end while
bestimme Q.min
```

Fibonacci-Heap Beispiel



Fibonacci-Heaps *decrease-key*

Q.decrease-key(v,k)

```
if k > v.key then return
v.key = k
update Q.min
if v ∈ Q.rootlist or k ≥ v.parent.key then return
do /* cascading cuts */
    parent = v.parent
    Q.cut(v)
    v = parent
while v.mark and v∉ Q.rootlist
if v ∉ Q.rootlist then v.mark = true
```

Fibonacci-Heaps: cuts

Q.cut(v)

```
if v ∉ Q.rootlist
  then  /* trenne v von seinem Vater */
        rang (v.parent) = rang (v.parent) – 1
        v.parent = null
        entferne v aus v.parent.childlist
        füge v in Q.rootlist ein
```

Fibonacci-Heaps Markierung

Historie

v wird an einem Knoten angehängt ⟶ *v.mark = false*

↓

v verliert einen Sohn ⟶ *v.mark = true*

↓

v verliert zweiten Sohn ⟶ *v* abtrennen

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 $u_1,\ldots,u_k$ von v nach dem Zeitpunkt des Anhängens an v, dann gilt

$$rang\ (u_i\ ) \geq i - 2.$$

Beweis:

Zeitpunkt des Anhängens von u_i:

Söhne von v $(rang(v))$: $\geq i - 1$
Söhne von u_i $(rang(u_i))$: $\geq i - 1$
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:

$$n \geq F_{k+2} \geq \phi^k \qquad \phi = (1+\sqrt{5})/2$$

Maximaler Rang eines Knotens

Beweis

S_k = Mindestgröße eines Teilbaumes, dessen Wurzel k Söhne hat

$S_0 = 1$

$S_1 = 2$

Es gilt:

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

Fibonacci-Zahlen:

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

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

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

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:

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

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:

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

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

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

$a_i \leq r_{i-1} + 2 \log n + 2 \log n - r_{i-1} + 0$

$= O(\log n)$

Analyse: *Decrease-key*

decrease-key:

$t_i = c + 2$

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

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

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

$\quad = O(1)$

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