Algorithm Theory

08 – Fibonacci Heaps

Dr. Alexander Souza

Winter term 11/12

Priority queues: operations

Priority queue *Q*

Operations:

Q.initialize(): initializes an empty queue *Q*

Q.isEmpty(): returns true iff *Q* is empty

Q.insert(e): inserts element *e* into *Q* and returns a pointer to the node containing *e*

Q.deletemin(): returns the element of *Q* with minimum key and deletes it

Q.min(): returns the element of Q with minimum key

Q.decreasekey(v,k): decreases the value of *v*'s key to the new value *k*

Priority queues: operations

Additional operations:

Q.delete(v) : deletes node *v* and its element from *Q*
(without searching for *v*)

Q.meld(Q´): unites *Q* and *Q´* (concatenable queue)

Q.search(k) : searches for the element with key *k* in *Q*
(searchable queue)

And many more, e.g. *predecessor, successor, max, deletemax*

Priority queues: implementations

	List	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≤n)	O(1)	O(n) or O(m log n)	O(log n)	O(1)
decr.-key	O(1)	O(log n)	O(log n)	O(1)*

*= amortized cost

Q.delete(e) = Q.decreasekey(e, -∞) + Q.deletemin( )

Fibonacci heaps

„Lazy-meld“ version of binomial queues:

The melding of trees having the same order is delayed until the next deletemin operation.

Definition

A Fibonacci heap *Q* is a collection heap-ordered trees.

Variables

Q.min: root of the tree containing the minimum key

Q.rootlist: circular, doubly linked, unordered list containing the roots of all trees

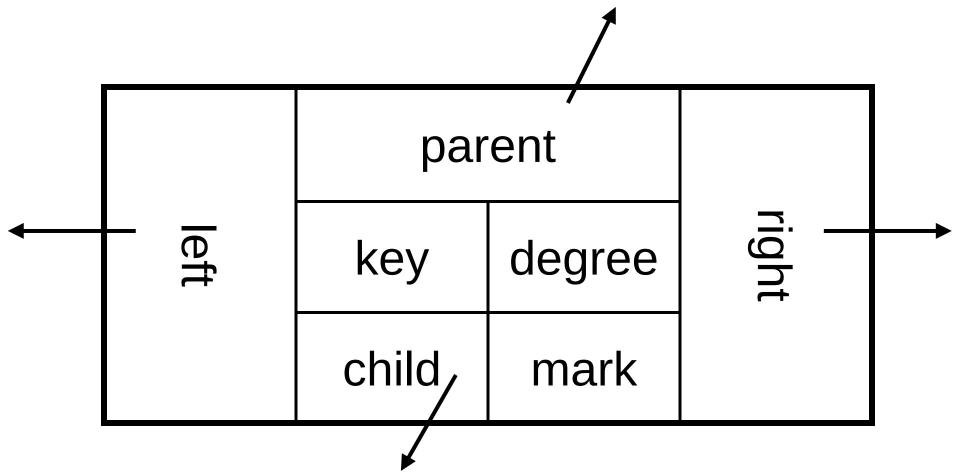
Q.size: number of nodes currently in *Q*

Trees in Fibonacci heaps

Let *B* be a heap-ordered tree in *Q.rootlist*:

B.childlist: circular, doubly linked and unordered list of the children of B

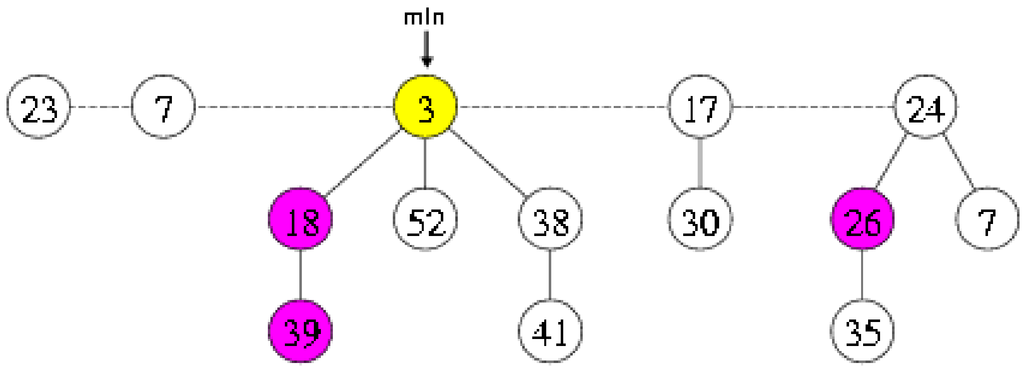
Structure of a node

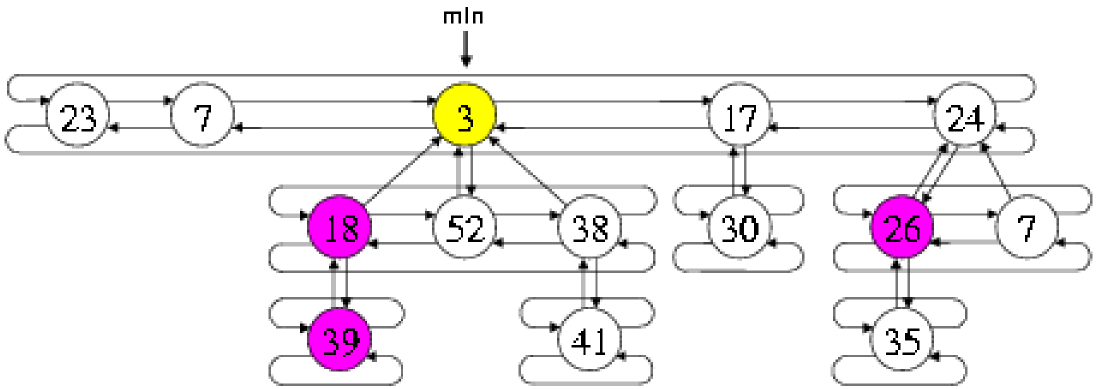


Advantages of circular, doubly linked lists:

1. Deleting an element takes constant time.
2. Concatenating two lists takes constant time.

Implementation of Fibonacci heaps: Example





Operations on Fibonacci heaps

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

Q.meld(Q´):

1. concatenate *Q.rootlist* and *Q´.rootlist*
2. update *Q.min*

Q.insert(e):

1. generate a new node with element *e* → *Q´*
2. *Q.meld(Q´)*

Q.min():

return *Q.min.key*

Fibonacci heaps: ‘deletemin’

Q.deletemin()

/*Delete the node with minimum key from *Q* and return its element.*/

```
m = Q.min()
if Q.size() > 0
      then remove Q.min() from Q.rootlist
                add Q.min.childlist to Q.rootlist
      Q.consolidate()
    /* Repeatedly meld nodes in the root list having the same
       degree. Then determine the element with minimum key. */
return m
```

Fibonacci heaps: maximum degree of a node

rank(v) = degree of node *v* in *Q*

rank(Q) = maximum degree of any node in *Q*

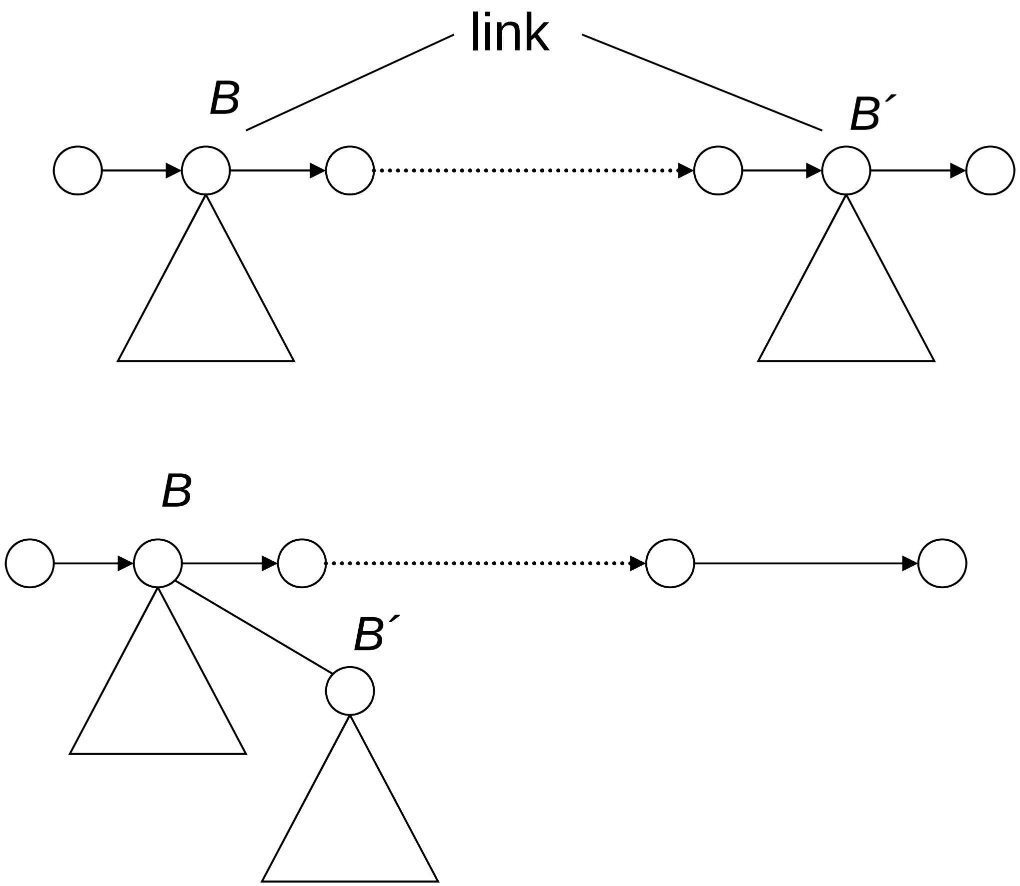
Assumption:

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

if $Q.size = n$.

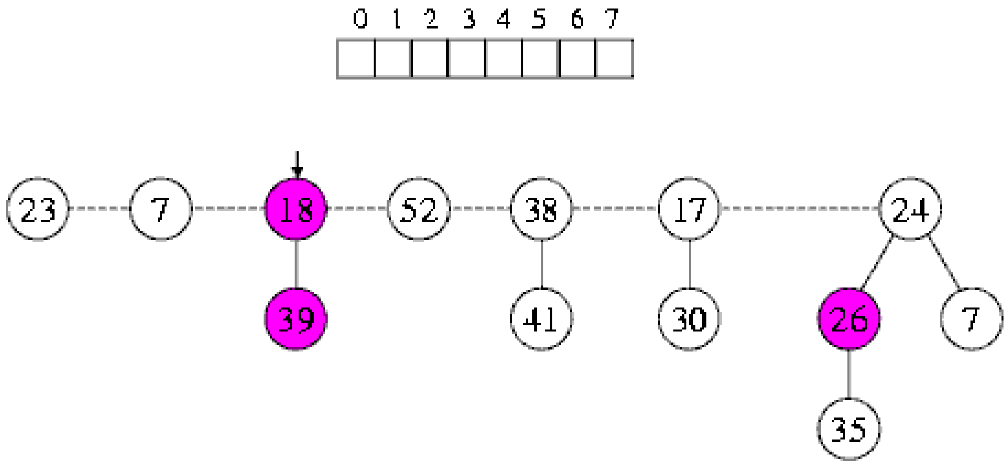
Fibonacci heaps: operation ‘link’

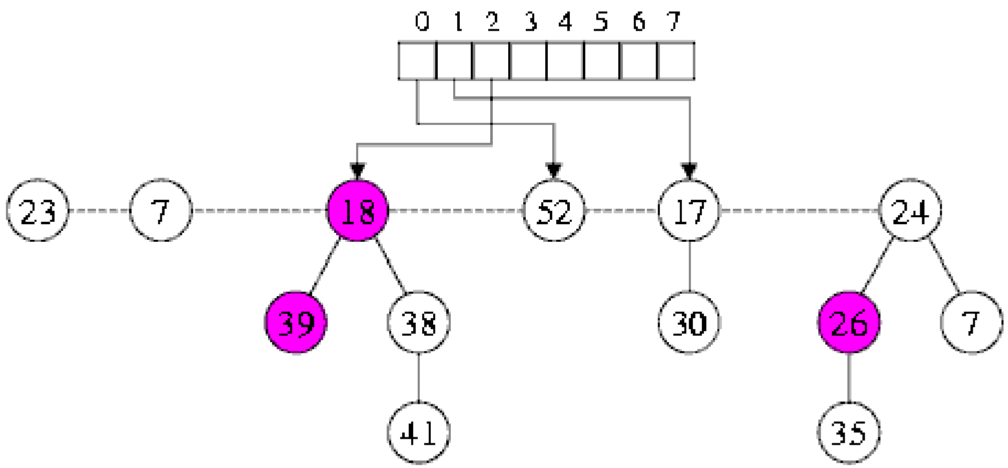
rank(B) = degree of the root of *B*
Heap-ordered trees *B,B´* with *rank(B) = rank(B´)*



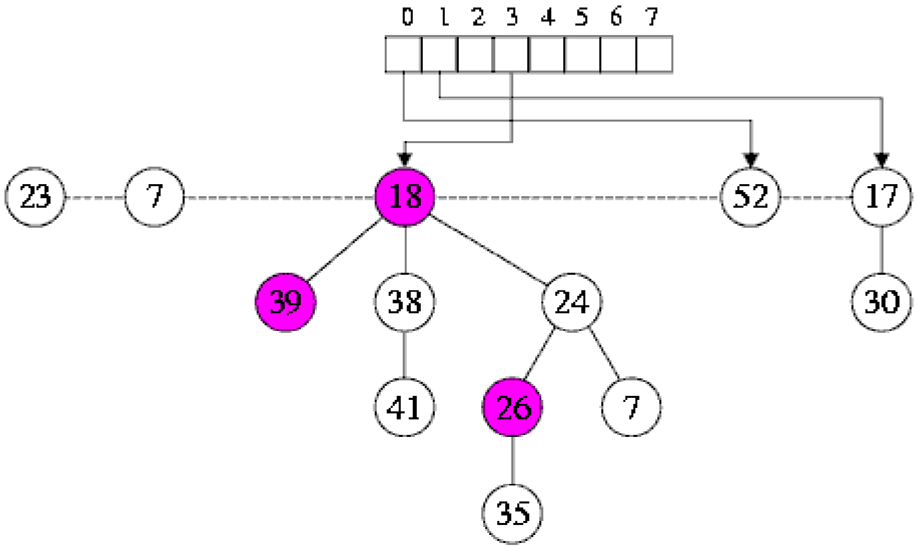
1. *rank(B)* = *rank(B)* + 1
2. *B´.mark* = *false*

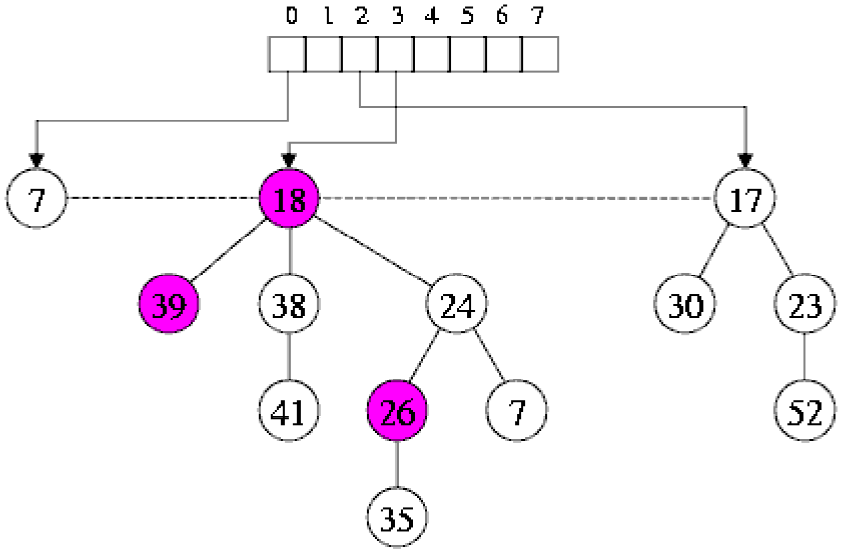
Consolidation of the root list





Consolidation of the root list





Fibonacci heaps: ‘deletemin’

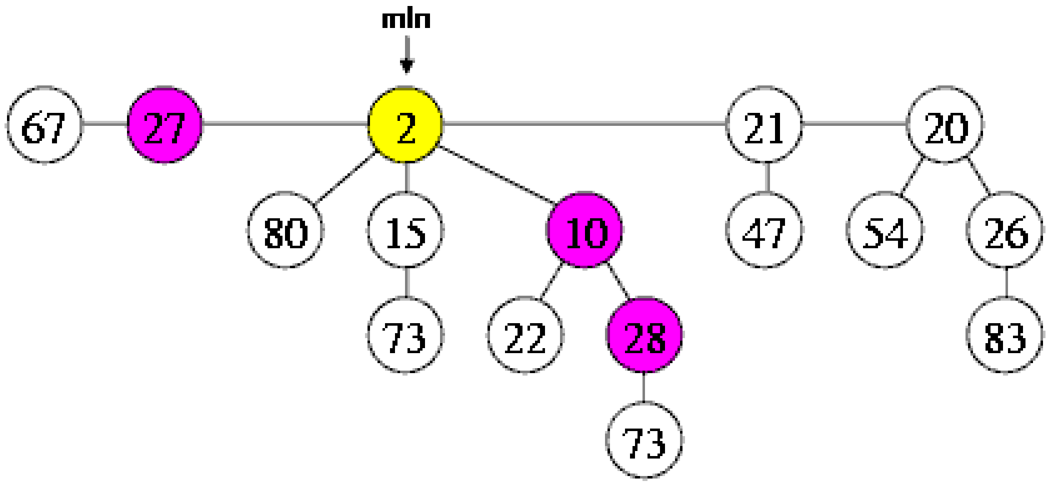
Find roots having the same rank:

Array *A*:

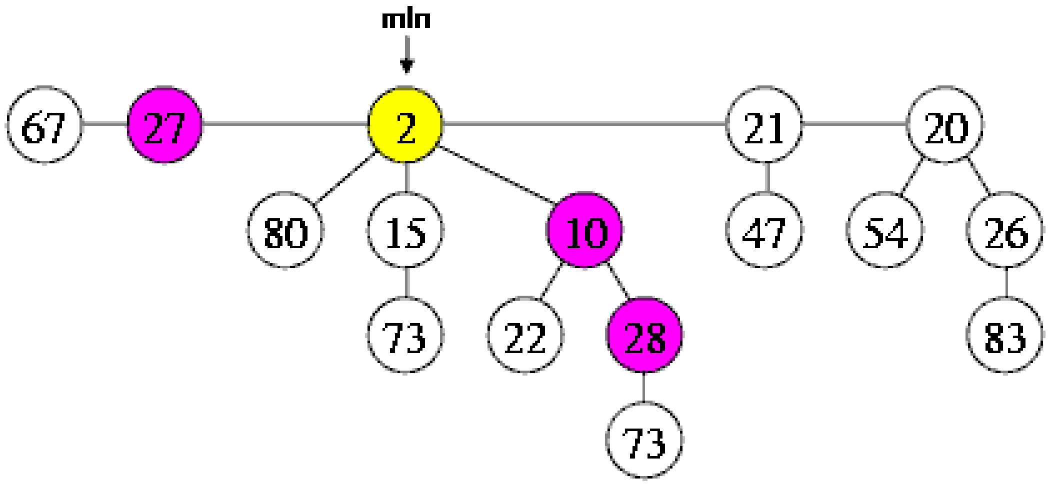
Q.consolidate()

```
A = array of length 2 log n pointing to Fibonacci heap nodes
for i = 0 to 2 log n do A[i] = null
while Q.rootlist ≠ ∅ do
        B = Q.delete-first()
        while A[rank(B)] is not null do
                B´ = A[rank(B)];   A[rank(B)] = null;   B = link(B,B´)
        end while
        A[rank(B)] = B
end while
determine Q.min
```

Fibonacci heap: Example



Fibonacci heap: Example



Fibonacci heaps: ‘decreasekey’

Q.decreasekey(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: ‘cut’

Q.cut(v)

```
if v ∉ Q.rootlist
  then  /* cut the link between v and its parent */
        rank (v.parent) = rank (v.parent) – 1
        remove v  from v.parent.childlist
        v.parent = null
        add v to Q.rootlist
```

Fibonacci heaps: marks

History of a node:

v is being linked to a node	→	*v.mark = false*
↓		
a child of *v* is cut	→	*v.mark = true*
↓		
a second child of *v* is cut	→	cut *v*

The boolean value *mark* indicates whether node *v* has lost a child since the last time *v* was made the child of another node.

Rank of the children of a node

Lemma

Let v be a node in a Fibonacci-Heap Q. Let $u_1,\ldots,u_k$ denote the children of v in the order in which they were linked to v. Then:

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

Proof:

At the time when u_i was linked to v:

children of v $(rank(v))$: $\geq i - 1$
children of u_i $(rank(u_i))$: $\geq i - 1$
children u_i may have lost: 1

Maximum rank of a node

Theorem

Let *v* be a node in a Fibonacci heap *Q*, and let *rank(v)* = *k* . Then *v* is the root of a subtree that has at least F_{k+2} nodes.

The number of descendants of a node grows exponentially in the number of children.

Implication:

The maximum rank *k* of any node *v* in a Fibonacci heap *Q* with *n* nodes satisfies:

Maximum rank of a node

Proof

S_k = minimum possible size of a subtree whose root has rank k

$S_0 = 1$

$S_1 = 2$

There is:

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

Fibonacci numbers:

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

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

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

Analysis of Fibonacci heaps

Potential method to analyze Fibonacci heap operations.

Potential Φ_Q of Fibonacci heap Q:

$$\Phi_Q = r_Q + 2\, m_Q$$

where

r_Q = number of nodes in *Q.rootlist*

m_Q = number of all marked nodes in Q, that are not in the root list.

Amortized analysis

Amortized cost a_i of the *i*-th operation:

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

Analysis of ‘insert’

insert

$t_i = 1$

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

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

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

Analysis of ‘deletemin’

deletemin:

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

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

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

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

$= O(\log n)$

Analysis of ‘decreasekey’

decreasekey:

$t_i = c + 2$

$r_i – r_{i-1} = \mathrm{c} + 1$

$m_i – m_{i-1} \leq - c + 1$

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

$\quad = O(1)$

Priority queues: comparison

	List	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≤n)	O(1)	O(n) or O(m log n)	O(log n)	O(1)
decr.-key	O(1)	O(log n)	O(log n)	O(1)*

* = amortized cost