ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Algorithm Theory

16 Fibonacci Heaps

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08

CoNe
Freiburg

IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Priority Queues: Operations

- **Priority queue Q**
  - Data structure for maintaining a set of elements, each having an associated priority
- **Operations:**
  - Q.initialize():
    - creates empty queue Q
  - Q.isEmpty():
    - returns true iff Q is empty
  - Q.insert(e):
    - inserts element e in to Q and returns a pointer to a 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

Priority Queues: Operations

- **Additional Operations:**
  - Q.delete(v):
    - deletes node v and its elements from Q
  - Q.meld(Q´):
    - unites Q and Q´ (concatenable queue)
  - Q.search(k):
    - searches for the element with key k in Q (searchable queue)
- **possibly many more,**
  - e.g. predecessor, successor, max, deletemax

Priority Queues: Implementations

	List	Heap	Binomial Queue	Fibonacci Heap
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) or O(m log n)	O(log n)	O(1)
decrease-key	O(1)	O(log n)	O(log n)	O(1)*

* = amortized cost

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

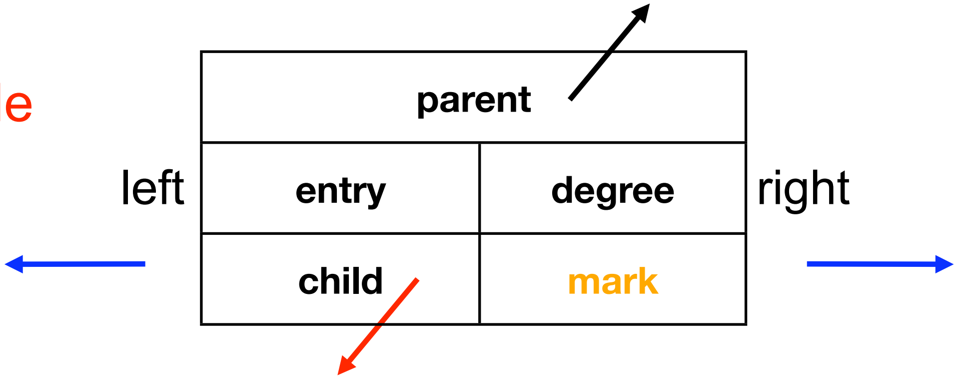
Fibonacci Heaps

- **„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 of 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

Structure of Fibonacci Heaps

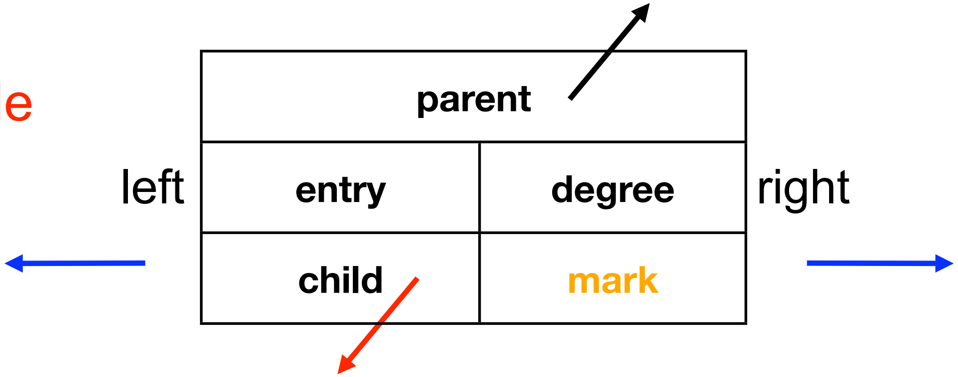
- **Let B be a heap-ordered tree in Q.rootlist**
  - B.childlist: circular, doubly linked and unordered list of the children of B
- **Advantages of circular, doubly linked lists**
  - Deleting an element takes constant time
  - Concatenating two lists takes constant time

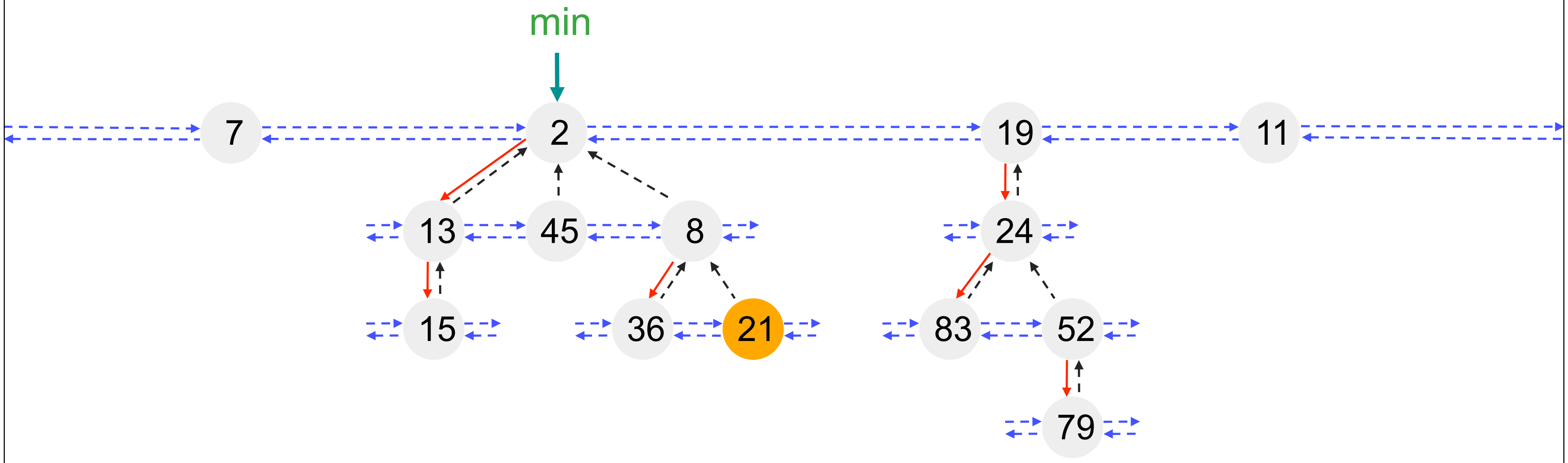
Structure of a node



Trees in Fibonacci Heaps

Structure of a node


parent
left
entry
degree
right
child
mark



Fibonacci-Trees: Meld (Link)

Meld operation of trees *B*, *B´* of same degree *k*

Link-Operation:

Can be computed in constant time: *O*(1)

Resulting tree has degree *k*+1

Difference compared to Binomial Queues:

Trees do not need to have the binomial form.

Operations on Fibonaccis Heps

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

Q.meld(F-Heap F):

```
/* concatenate root lists */
Q.min.right.left = F.min.left
F.min.left.right = Q.min.right
Q.min.right = F.min
F.min.left = Q.min
Q.min = min { F.min, Q.min }
/* no cleaning up - delayed until next
   deletemin */
```

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
```

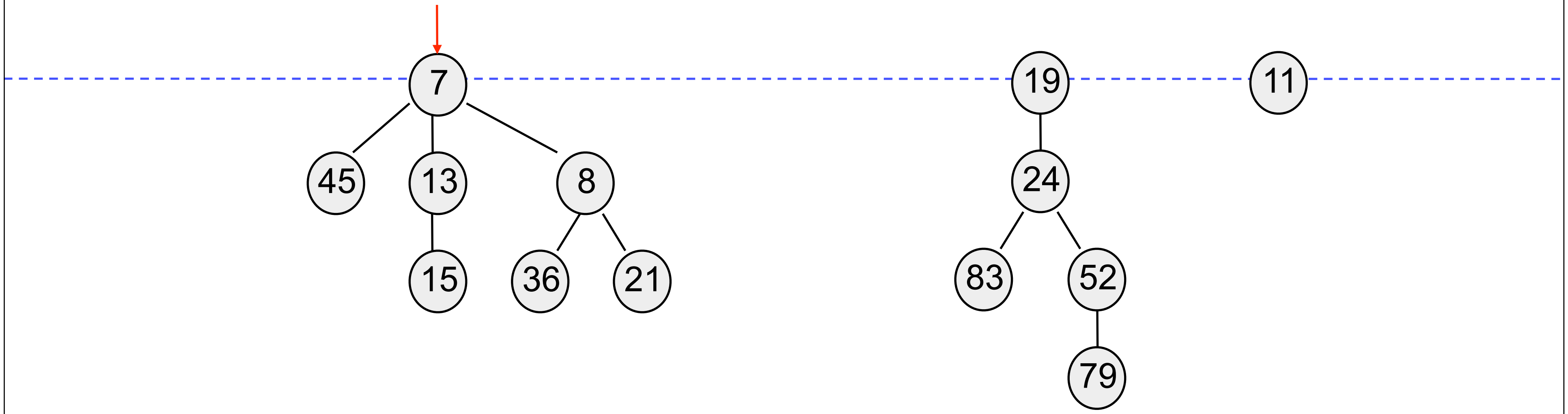
Fibonacci Heaps Maximum Degree of a Node

- **Definition**
  - rank(v) = degree of a node v in Q
  - rank(Q) = maximum degree of any node in Q
- **Assumption:**
  - rank(Q) ≤ 2 log n
    - where n = Q.size

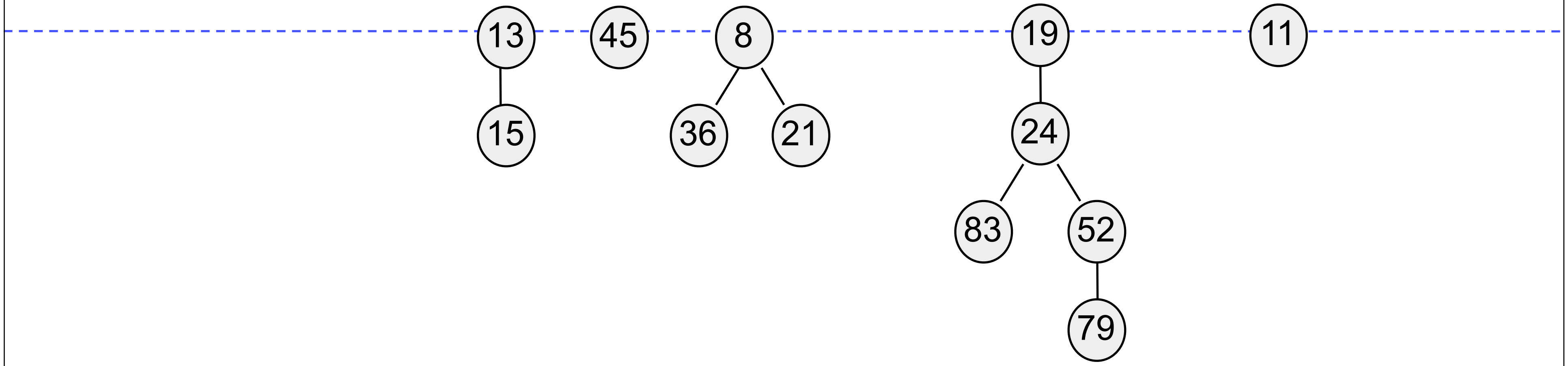
Cost of *deletemin*

- **Deleting has has constant time cost *O*(1)**
- **Time cost results from the consolidate operation**
  - i.e. the length of the root list and the number of necessary link-operations
- **How to efficiently perform the consolidate operation?**
  - Observation:
    - Every root must be considered at least once
    - For every possible rank there is at most one node

deletemin: **Example**



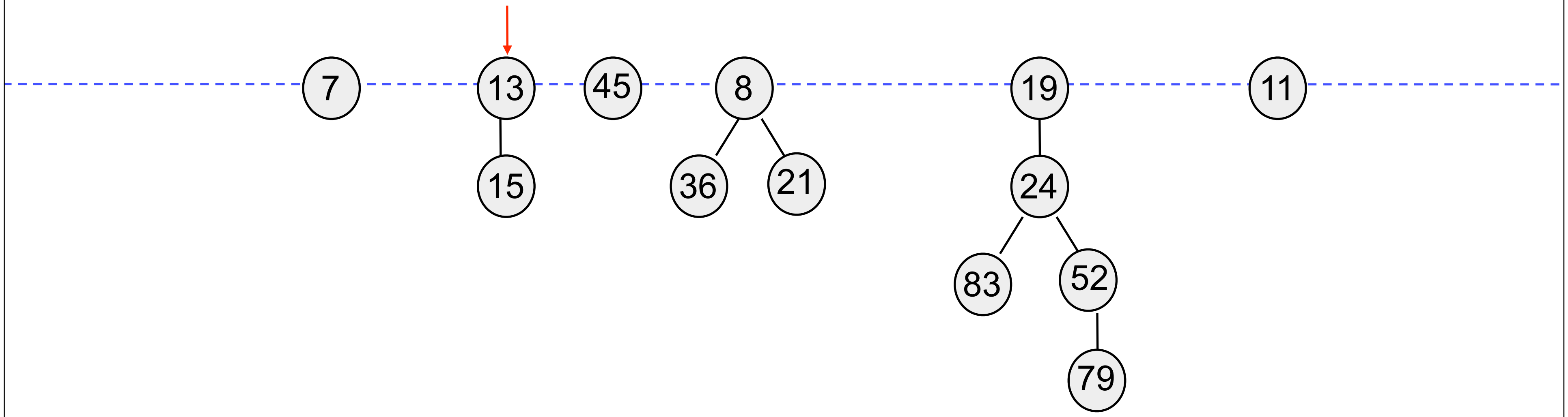
deletemin: **Example**



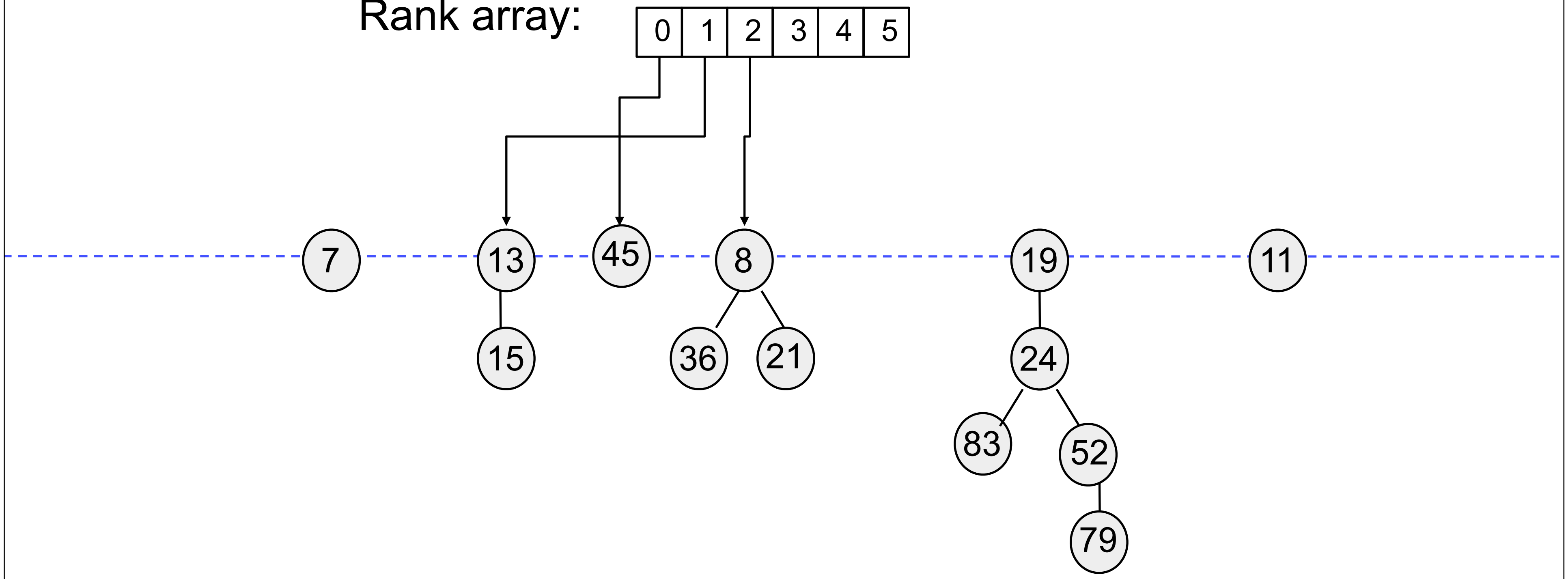
deletemin: **Example**

Rank array:

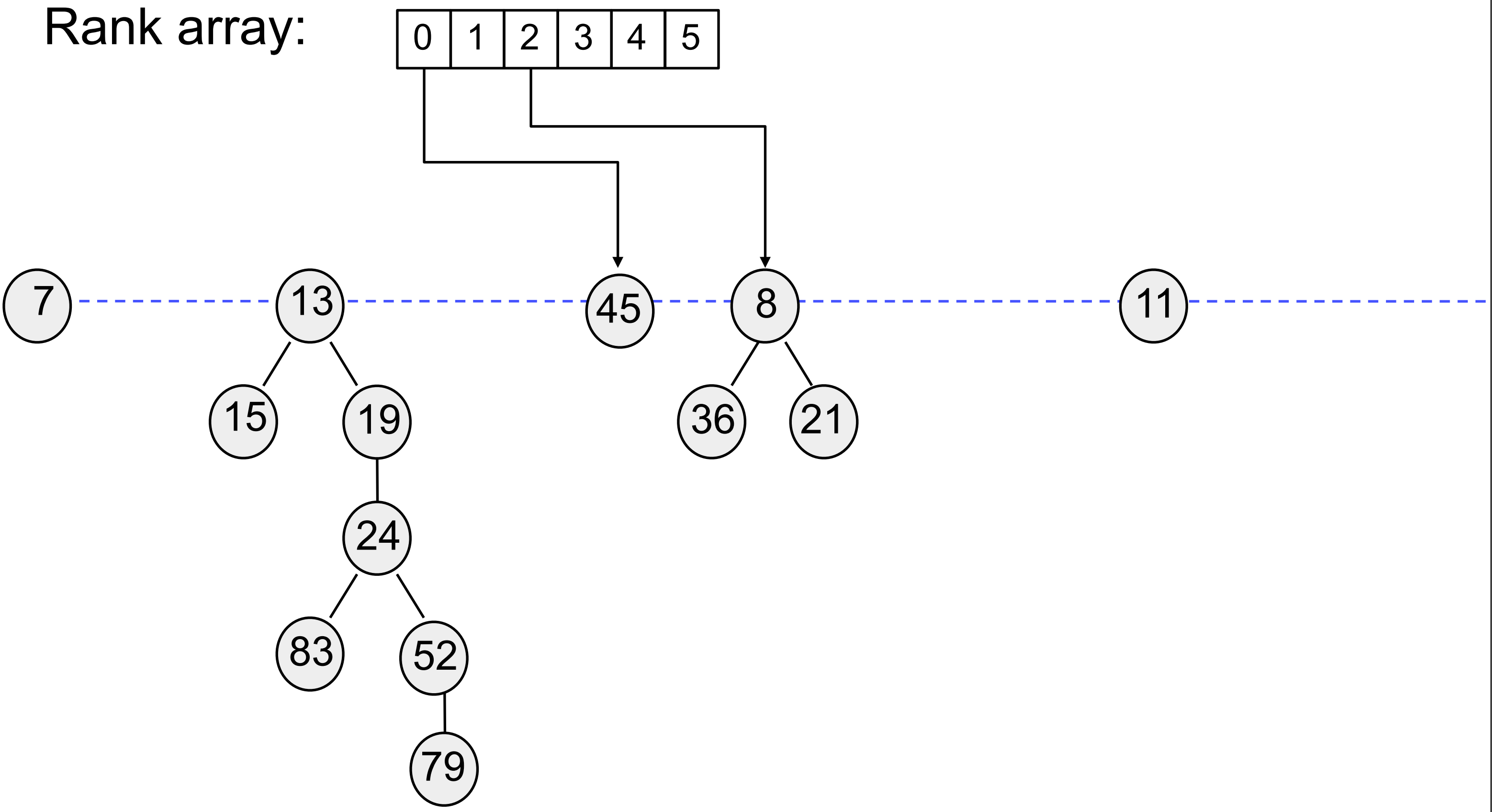
0	1	2	3	4	5



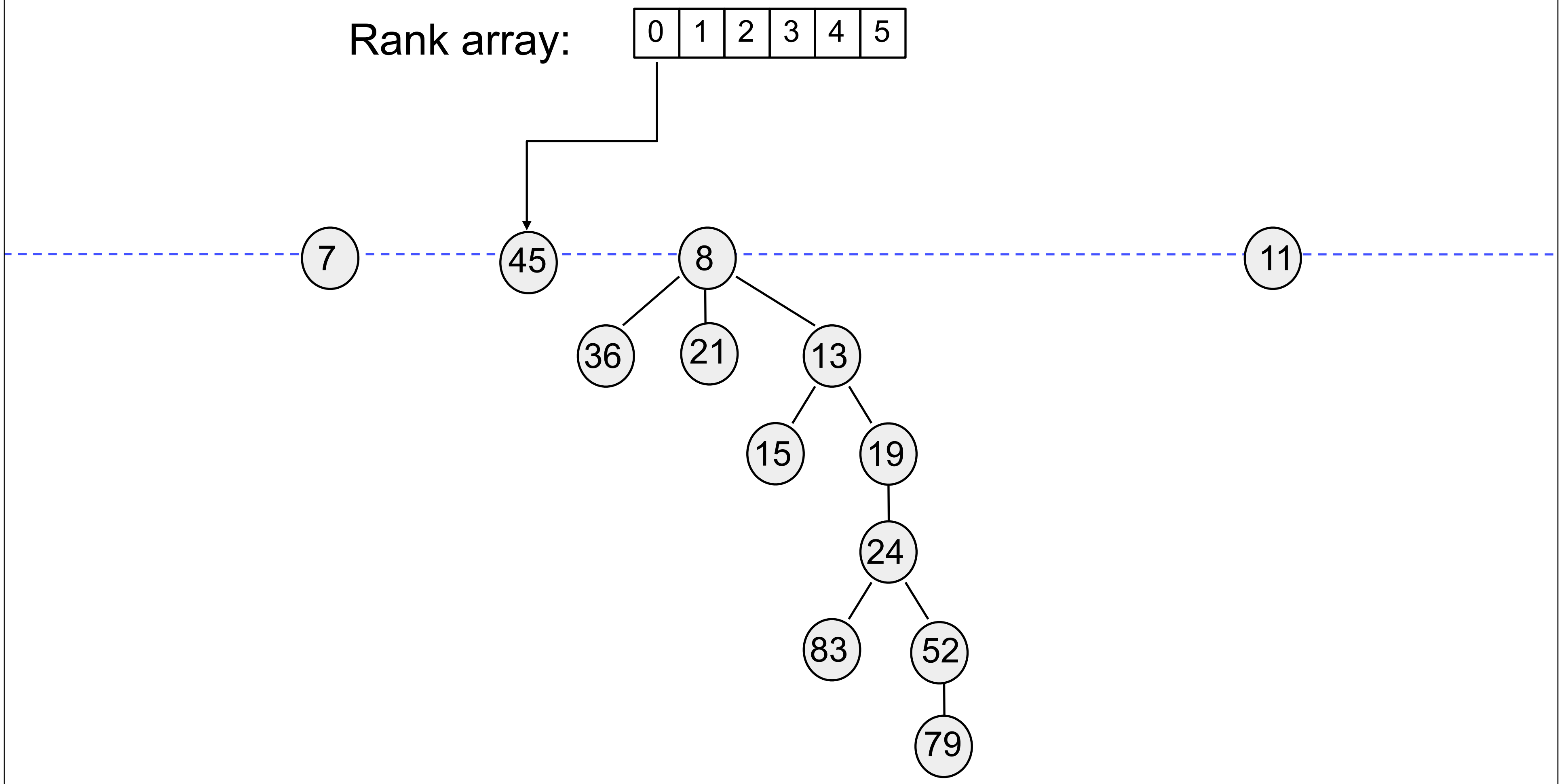
consolidate: **Example**



consolidate: **Example**

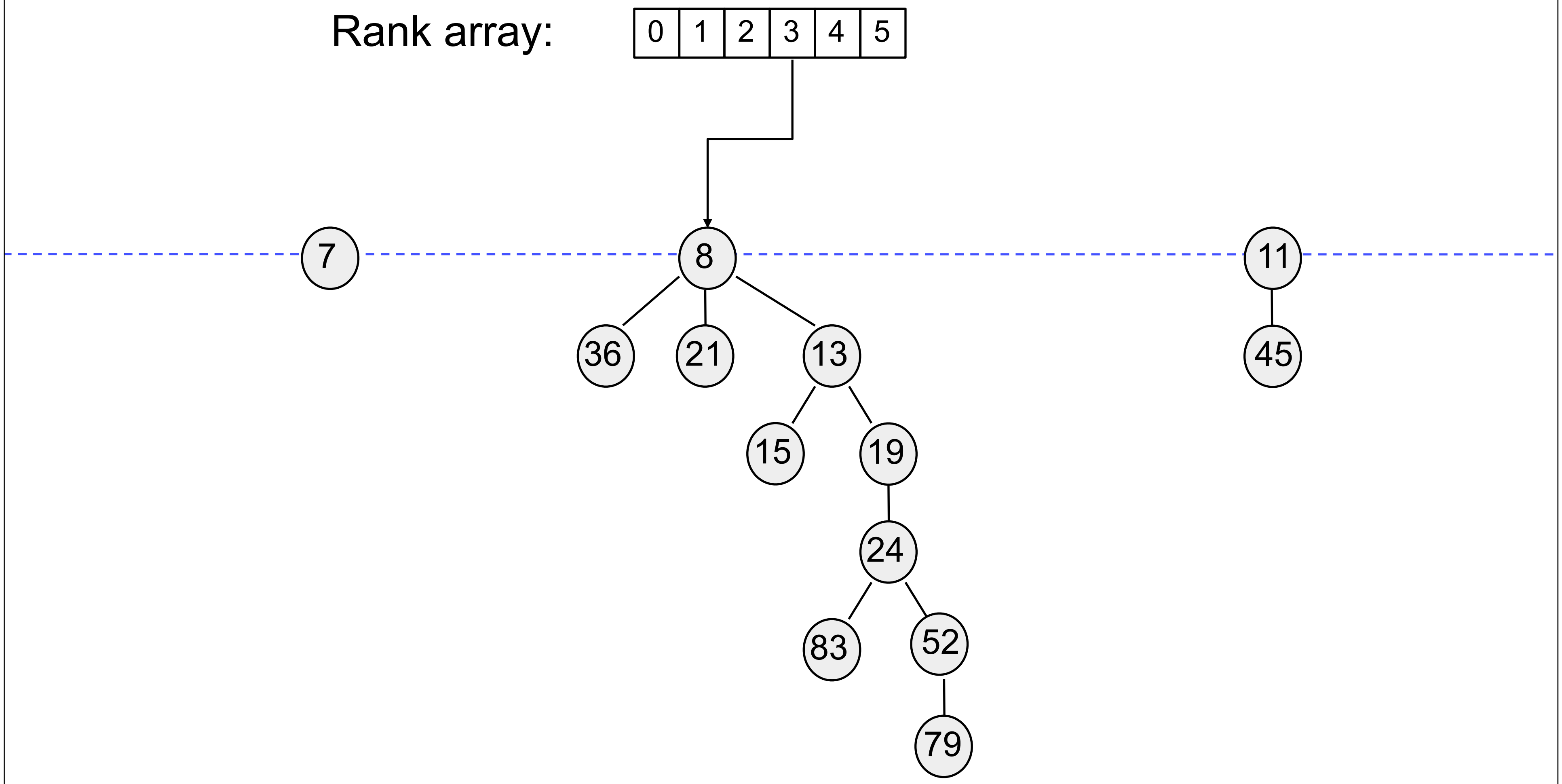


consolidate: **Example**

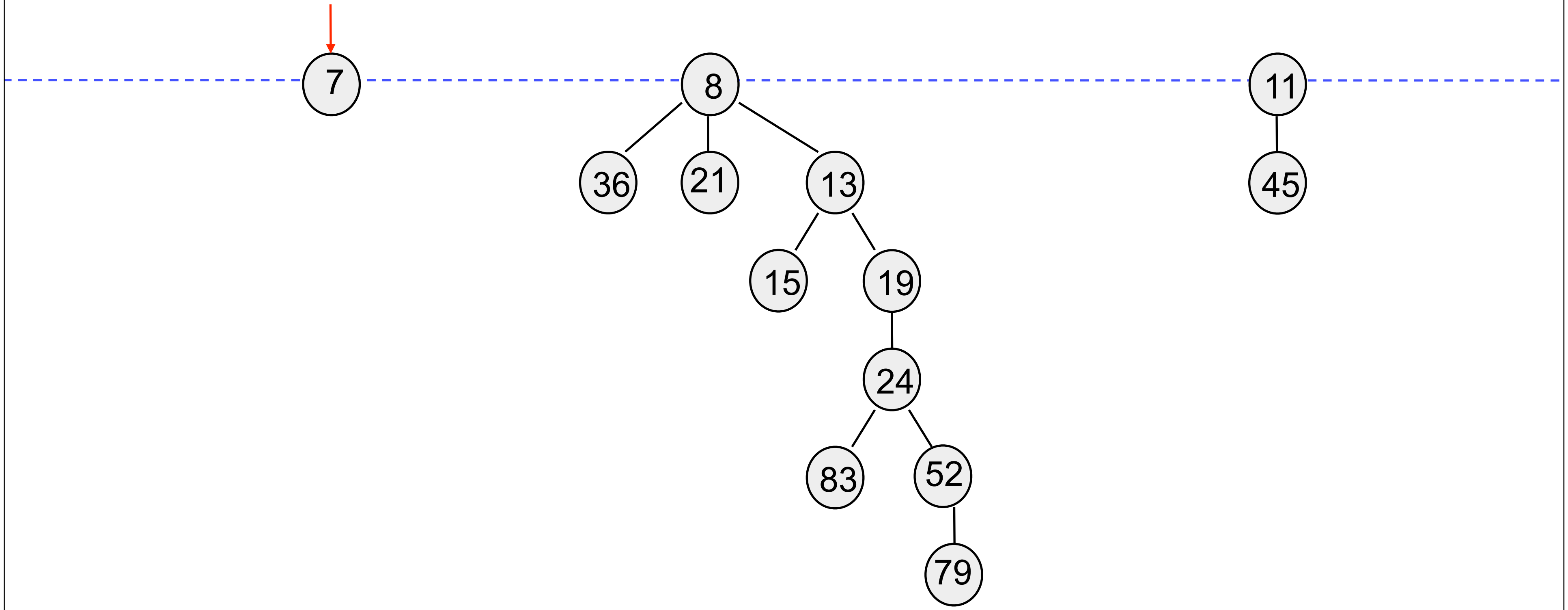


consolidate: Example

Rank array:



consolidate: Example



Analysis of *consolidate*

```
rankArray = new FibNode[maxRank(n)+1];  // Create array


for „each FibNode N in rootlist“ {
    while (rankArray[N.rank] != null) { // position occupied
        N = link(N, rankArray[N.rank]); // link trees
        rankArray[N.rank-1] = null;     // delete old position
    }
    rankArray[N.rank] = N;             // insert into the array
}
```

Analysis

```
for „each FibNode N in rootlist‟ {
    while (rankArray[N.rank] != null) {
        N = link(N, rankArray[N.rank]);
        rankArray[N.rank-1] = null;
    }
    rankArray[N.rank] = N;
}
```

Let k = #root node before the consolidation. These k nodes can be classified as

W = {Nodes which are in the root list in the end}

L = {Nodes which are appended to another node}

We have: *cost(for-loop)* = *cost*(W) + *cost*(L)

= |rankArray| + #links

Cost of *deletemin*

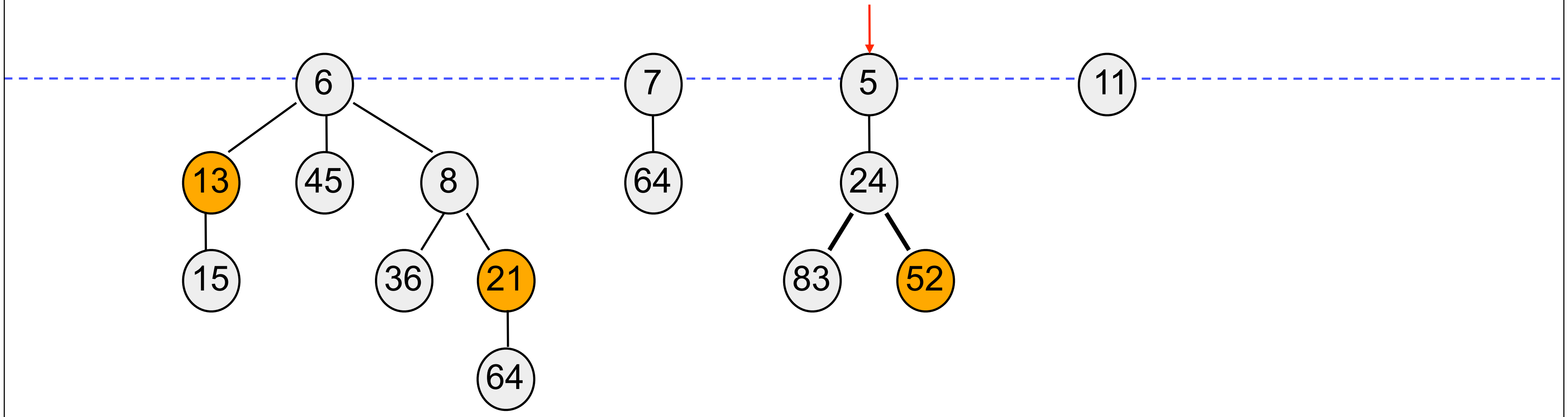
- **Worst case time: O(*maxRank*(*n*)) + O(#links)**
  - where maxRank(n) is the largest possible array entry, i.e. the largest possible root degree
- **Worst case: #links = n-2**
  - happens usually once at the beginning
- **Amortized Analysis**
  - sums up all the running times
  - and averages (worst case!) over all single Operations

Fibonacci-Heaps: *decreasekey*

`Q.decreasekey(FibNode N, int k):`

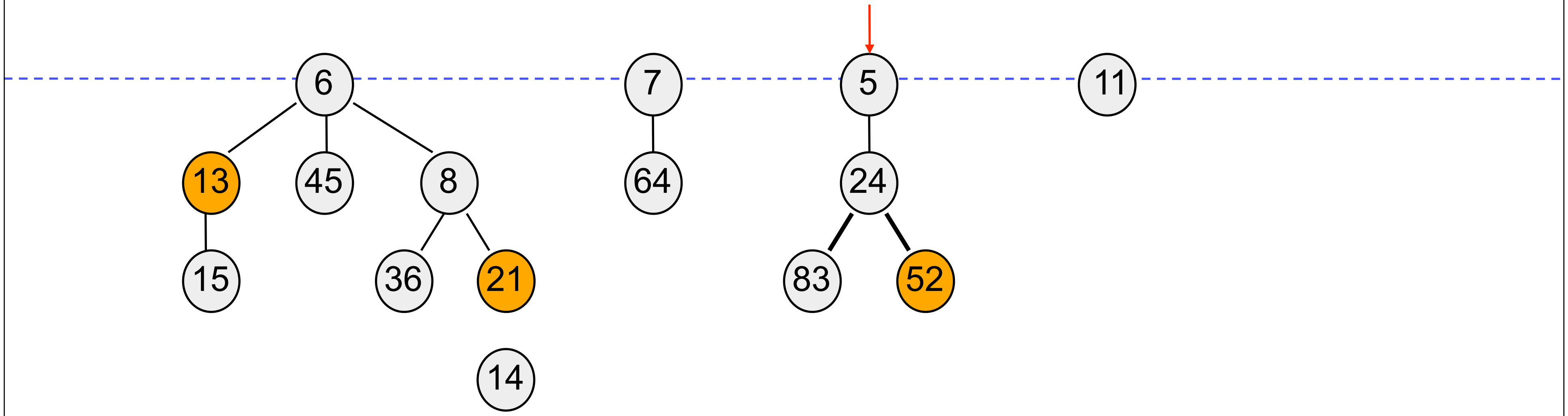
- **Decrease the key `N` to the value `k`**
- **If the heap condition does not hold (`k < N.parent.key`):**
  - Disconnect `N` from its father (using *cut*) and append it (to the right of the minimal node) into the rootlist and unmark it
  - If the father is marked(`N.parent.mark == true`), disconnect it from his father and unmark it; if his father is also marked, disconnect it, etc.(*„cascading cuts“*)
  - Mark the node whose son is disconnected at last (if it is node a root node).
  - Update the minimum pointer (if `k < min.key`).

Example for *decreasekey*

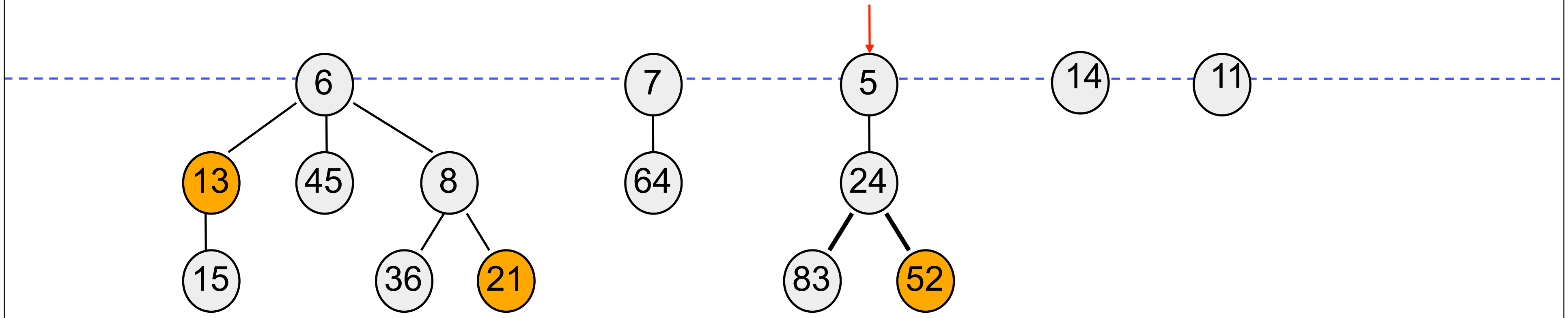


Decrease key 64 to 14

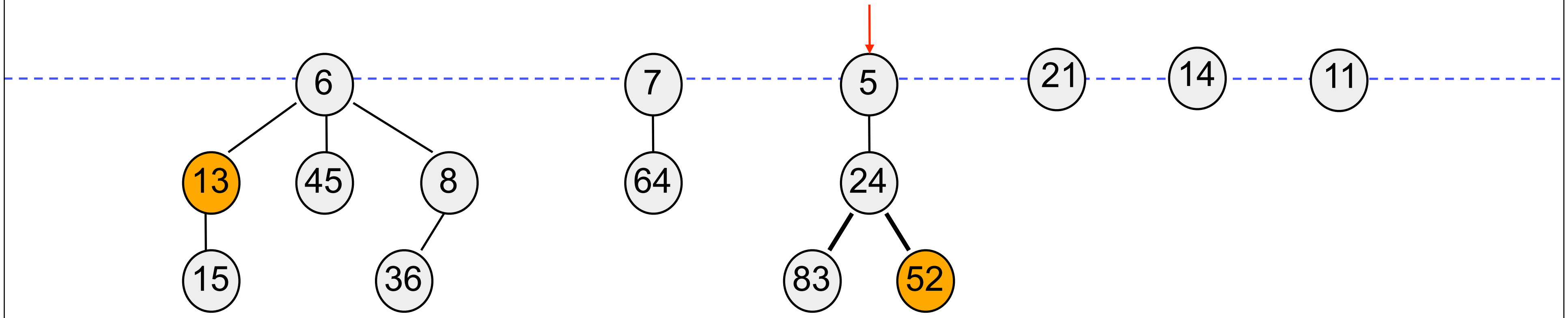
Example for *decreasekey*



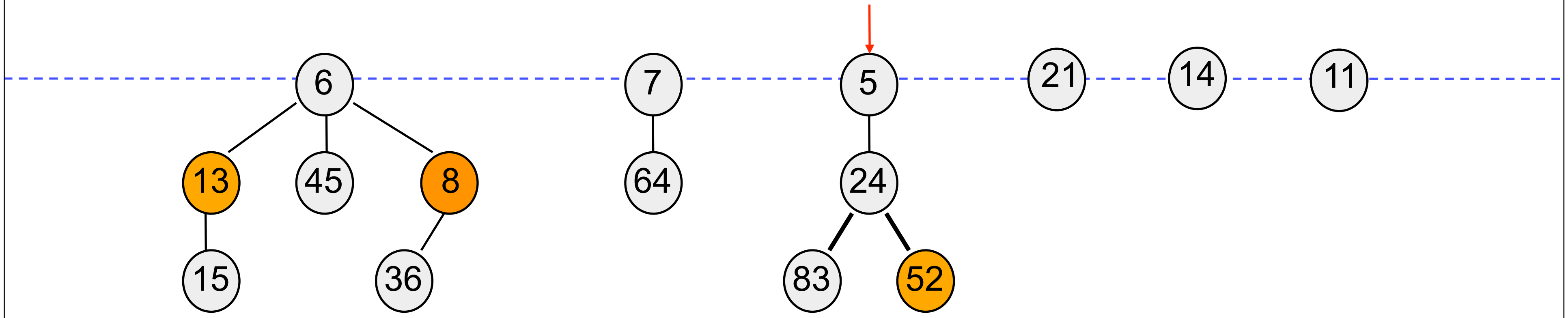
Example for *decreasekey*



Example for *decreasekey*



Example for *decreasekey*



Cost of *decreasekey*

- **Placement of key and comparison with father: O(1)**
- **Disconnect from father node and insertion into root list: O(1)**
- **Cascading cuts: #cuts**
- **Mark of the last node: O(1)**

→ Costs depends on the number of „cascading cuts“.

Worst case: #cuts = n-1

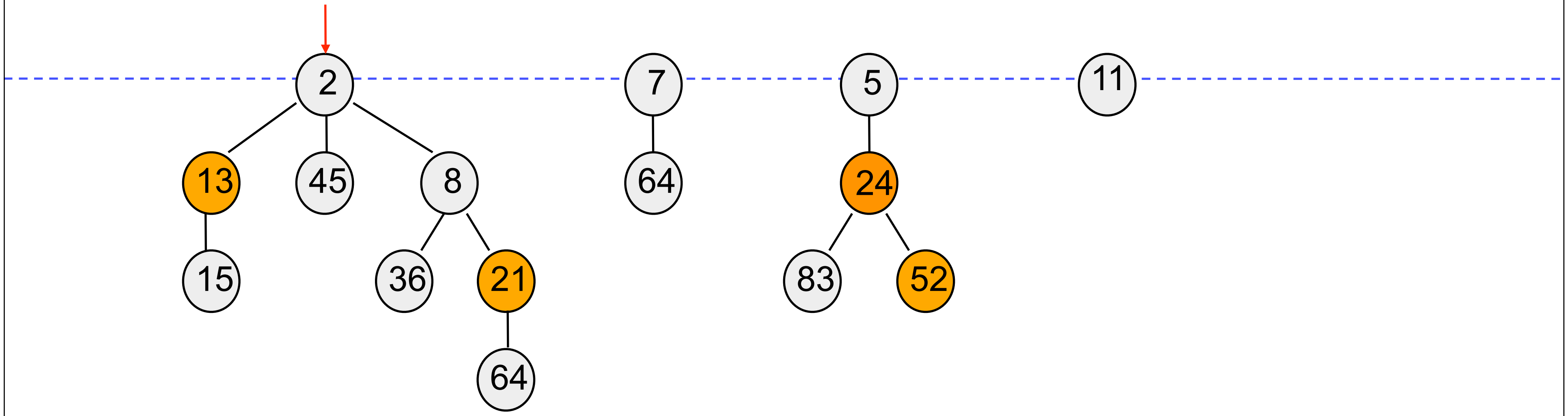
Amortized Analysis gives smaller value

Amortized Analysis – Potential Method

- **Assign every state *i* of the data structure a value Φ_i (Potential)**
  - **Φ_i =** potential after the i-th operation
- **The amortized costs a_i of the i-th operation is defined as**
  - $a_i = c_i + (\Phi_i - \Phi_{i-1})$,
  - the actual cost plus the change of the potential by the i-th operation
- **Define $\Phi_i = w_i + 2\, m_i$**
  - where w_i = number of root nodes
    and m_i = number of marked nodes (no roots)
- **Example: insert**
  - real costs: $c_i = O(1)$
    Potential increases by 1, i.e. $\Phi_i - \Phi_{i-1} = 1$

$$a_i = c_i + 1$$

Potential: Example



$w_i =$

$m_i =$

$\Phi_i =$

Potential Method: Cost of *insert*

- **Real coast: $c_i = O(1)$**
- **Change of potential:**
  $\Phi_i - \Phi_{i-1} = 1$
- **Amortized costs: $a_i = c_i + 1$**

Potential Method: Cost of *decreasekey*

- Real cost: $c_i = O(1) + \#\text{cascading cuts}$
- Change of potential:

  $w_i \leq w_{i-1} + 1 + \#\text{cascading cuts}$

  $m_i \leq m_{i-1} + 1 - \#\text{cascading cuts}$

  $$\begin{aligned}\Phi_i - \Phi_{i-1} &= w_i + 2 \cdot m_i - (w_{i-1} + 2 \cdot m_{i-1}) \\ &= w_i - w_{i-1} + 2 \cdot (m_i - m_{i-1}) \\ &\leq 1 + \#\text{cascading cuts} + 2 \cdot (1 - \#\text{cascading cuts}) \\ &= 3 - \#\text{cascading cuts}\end{aligned}$$

- Amortized costs: $a_i = c_i + (\Phi_i - \Phi_{i-1})$

  $\leq O(1) + \#\text{cascading cuts} + 3 - \#\text{cascading cuts} = O(1)$

Potential Cost
Cost of *deletemin*

- Real cost: $c_i \;=\; O(maxRank(n)) + \#\text{links}$
- Change of potential:

  $w_i = \; w_{i\text{-}1} - 1 + \text{rank}(\text{min}) - \#\text{links} \;\leq\; w_{i\text{-}1} + maxRank(n) - \#\text{links}$

  $m_i \leq \; m_{i\text{-}1}$

  $$\begin{aligned}\Phi_i - \Phi_{i\text{-}1} &= \; w_i + 2m_i - (w_{i\text{-}1} + 2m_{i\text{-}1}) \\ &= \; w_i - w_{i\text{-}1} + 2(m_i - m_{i\text{-}1}) \\ &\leq \; \text{maxRank}(\text{n}) - \#\text{links}\end{aligned}$$

- Amortized costs: $a_i \;=\; c_i + (\Phi_i - \Phi_{i\text{-}1})$

  $\leq\; O(maxRank(n)) + \#\text{links} + maxRank(n) - \#\text{links}$

  $= O(maxRank(n))$

maxRank

- ***maxRank(n)*** **is the maximal possible number of sons which a a node can have in a Fibonacci-Heap with *n* elements**
  - We want to show that this number is at most $O(\log n)$
  - Every node has a minimum number of successors which is exponential in the number of sons

maxRank(*n*)

Lemma 1:

Let N be a node in a Fibonacci-Heap and let $k = N$.rank. Consider the sons $C_1, \ldots, C_k$ of N in the order how they have been inserted (via *link*) to N. We have:

(1) C_1**.rank** ≥ 0

(2) C_i**.rank** $\geq i - 2$ **für** $i = 2, \ldots, k$

Proof: (1) straight forward

(2) When C_i became a son of N nodes $C_1, \ldots, C_{i-1}$ were already sons of N, i.e. N.rank $\geq i$-1. Since *link* always connects nodes with the same rank we had at the insertion also C_i.rank $\geq i$-1.

Since then C_i might have lost only one son (because of *cascading cuts*).

Hence we have: C_i.rank $\geq i - 2$

maxRank(*n*)

Lemma 2:

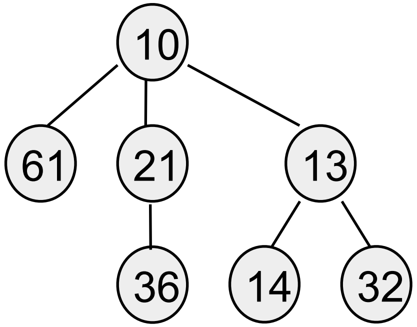
Let N be a node in a Fibonacci Heap and let $k = N$.rank.

Let $size(N)$ = be the number of nodes in a sub-tree with root N.

Then: $size(N) \geq F_{k+2} \geq 1.618^k$

i.e. a node with k children has at least F_{k+2} successors (including itself).

maxRank(*n*)



Proof: Let S_k = min {*size*(*N*) | *N* with *N*.rank = *k*},

i.e. the smallest possible size of tree with root rank *k*.

(clearly $S_0 = 1$ and $S_1 = 2$)

Let C_1, ..., C_k be the children *N* in the order how they were inserted into *N*

We have

$$
\begin{aligned}
size(N) \geq S_k &= \sum_{i=1}^{k} S_{C_i.\mathrm{rank}} \\
&= 1 + 1 + \sum_{i=2}^{k} S_{i-2} \\
&= 2 + \sum_{i=2}^{k} S_{i-2}
\end{aligned}
$$

maxRank(*n*)

Remember: Fibonacci-Numbers

$F_0 = 0$

$F_1 = 1$

$F_{k+2} = F_{k+1} + F_k$ für $k \geq 0$

Fibonacci numbers grow exponentially where $F_{k+2} \geq 1.618^k$

Furthermore: $$F_{k+2} = 2 + \sum_{i=2}^{k} F_i$$

(Follows by straight-forward induction over *k*.)

Summary

$$F_{k+2} = 2 + \sum_{i=2}^{k} F_i \qquad\qquad S_k = 2 + \sum_{i=2}^{k} S_{i-2}$$

- Furthermore $S_0 = 1 = F_2$

  and $S_1 = 2 = F_3$
- So, it follows: $S_k = F_{k+2}$ (Proof by induction)
- For a node N with rank k we observe

$$\text{size}(N) \geq S_k = F_{k+2} \geq 1.618^k$$

maxRank(*n*)

Theorem:

The maximal value of *maxRank*(*n*) of any node in a Fibonacci-Heap with *n* nodes is bound by $O(\log n)$.

Proof: Let *N* be an arbitrary node of a Fibonacci-Heaps with *n* nodes and let $k = N.\mathrm{rank}$.

Lemma 2 implies $n \geq size(N) \geq 1.618^k$

Therefore $k \leq \log_{1.618}(n) = O(\log n)$

Summary

	List	Heap	Binomial Queue	Fibonacci Heap
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)*
decrease key	O(1)	O(log n)	O(log n)	O(1)*
delete	O(n)	O(log n)	O(log n)	O(log n)*

* = amortized cost

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

ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Algorithm Theory

16 Fibonacci Heaps

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08

CoNe
Freiburg

IIF
INSTITUT FÜR
INFORMATIK
FREIBURG