Data Structures and Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Waiting Lines in a System

- LIFO: stacks
- FIFO: queues
- **Priorities**: heaps
  - Implementation (Static/Dynamic)?
  - Time/Space complexity? Trade-offs?

Who's next?

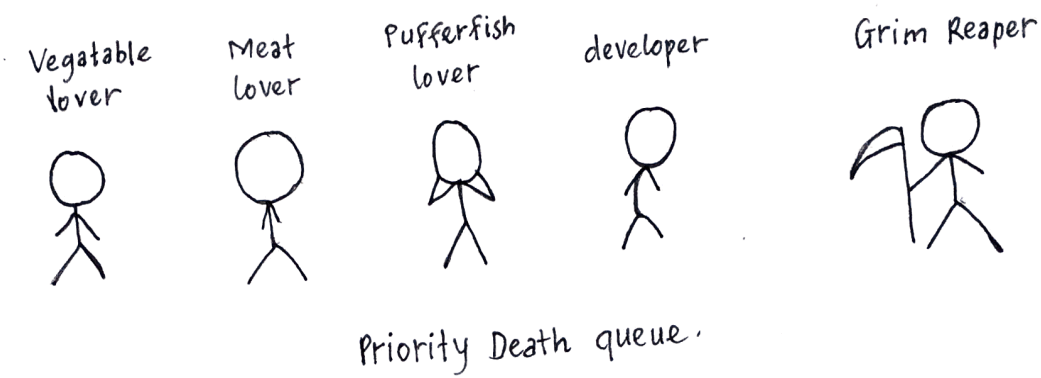
Choosing the next task: **Priorities**

- More general situation: Each element comes with its own (user/application defined) priority.

priority = number = int/long/double/...

- the next element = one with maximum (resp., minimum) priority

tie break using FIFO/LIFO/...



Data Structure: **Priority Queues**

- **bool** `empty()`: asserts whether the structure contains no element

- **void** `insert(T&,` **int**`)`: insertion of a new element (of type `T`) with some priority.

- `T& peek()`: returns the element with minimum priority (variant: returns the element with maximum priority).

- **void** `delete_min()`: removes from the structure the element with minimum priority (variant: `delete_max()`).

Data Structure: **Priority Queues**

Wlog we may only consider the priorities!

- **bool** `empty()`: asserts whether the structure contains no element

- **void** `insert(T&,` **int**`)`: insertion of a new element (of type `T`) with some priority. $\Longrightarrow$ **void** `insert(`**int**`)`

- `T& peek()`: returns the element with minimum priority (variant: returns the element with maximum priority). $\Longrightarrow$ **int** `peek()`

- **void** `delete_min()`: removes from the structure the element with minimum priority (variant: `delete_max()`).

Extended set of operations

- **int** `size()`.

Extended set of operations

- **int** `size()`.
- `delete(T&)`: removes any given element.

Remark: the location of each element in the structure can be stored in a Hash table

Extended set of operations

- **int** `size()`.
- `delete(T&)`: removes any given element.

Remark: the location of each element in the structure can be stored in a Hash table

- **void** `decrease_key(T&,int)`/**void** `increase_key(T&,int)`: modification of priority
  - Decrease/Increase key can be simulated by a deletion+insertion.
  - Deletion can be simulated by a decrease key + delete min

Extended set of operations

- **int** `size()`.
- `delete(T&)`: removes any given element.

Remark: the location of each element in the structure can be stored in a Hash table

- **void** `decrease_key(T&,int)`/**void** `increase_key(T&,int)`: modification of priority
  - Decrease/Increase key can be simulated by a deletion+insertion.
  - Deletion can be simulated by a decrease key + delete min
- **void** `meld(priority_queue& q)`: merge of two priority queues.

$\longrightarrow$ not all implementations of priority queues are "meldable"

FIFO and LIFO as particular cases

- Simulating a queue with a priority queue.

- Simulating a stack with a priority queue.

FIFO and LIFO as particular cases

- Simulating a queue with a priority queue.
  - add a counter: `int p = 0;`
  - adding a new element: with priority `p--;`
  - getting the first element: with maximum priority
- Simulating a stack with a priority queue.

FIFO and LIFO as particular cases

- Simulating a queue with a priority queue.
  - add a counter: **int** `p = 0;`
  - adding a new element: with priority `p--;`
  - getting the first element: with maximum priority
- Simulating a stack with a priority queue.
  - add a counter: **int** `p = 0;`
  - adding a new element: with priority `p++;`
  - getting the last element: with maximum priority

FIFO and LIFO as particular cases

- Simulating a queue with a priority queue.
  - add a counter: **int** `p = 0;`
  - adding a new element: with priority `p--;`
  - getting the first element: with maximum priority
- Simulating a stack with a priority queue.
  - add a counter: **int** `p = 0;`
  - adding a new element: with priority `p++;`
  - getting the last element: with maximum priority

$\Longrightarrow$ Induces complexity overhead

Applications

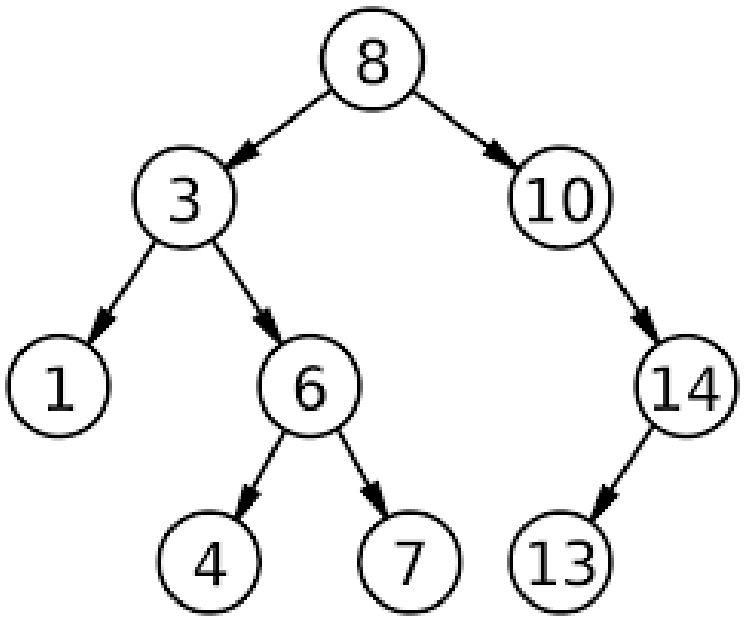
- Can be used to model various scenarii:
  - lambda/premium users
  - length/time required for a task, etc.
- Algorithmic application: **Sorting**.

```
void sort(vector<int>& v) {
    priority_queue q;
    for(int i = 0; i < v.size(); i++) { insert(q,v[i]); }
    for(int i = 0; !empty(q); i++) {
        v[i] = peek(q); delete_min(q);
    }
}
```

Complexity?

Implementation: Self-balanced binary research trees

Example: all elements stored in an AVL.



→ `empty()` and `size()` in $\mathcal{O}(1)$

→ insertion/deletion/peek in $\mathcal{O}(\log n)$

Discussion: Pros/Cons

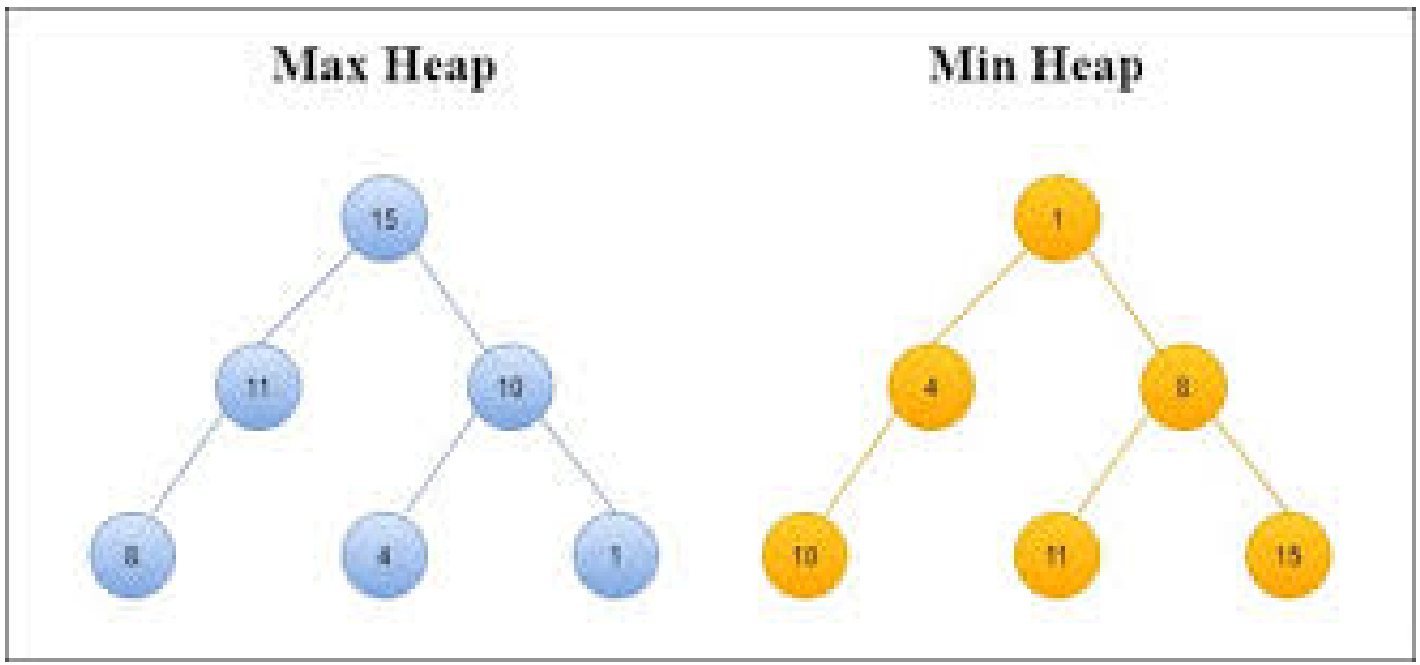
- Pros:
  - Modularity (novel use of a DS with already many applications)
  - Performances: $\mathcal{O}(\log n)$ worst-case per operation
- Cons
  - Complex implementation (rotations)
  - Non-discrimination between the operations: can we support more operations in $\mathcal{O}(1)$?
  - Not meldable

Heaps

Definition

Disjoint union of trees (=forest) that all satisfy the "Min-Heap Property":

- The key (value) store at any node is smaller than the keys stored at any of its children.



Remark: this is a property over the keys. The underlying tree is arbitrary.

Operations on a Heap

- `empty()`/`size()` are straightforward... $\mathcal{O}(1)$
- `peek()`?

Operations on a Heap

- `empty()`/`size()` are straightforward... $\mathcal{O}(1)$
- `peek()`?
  - If only one tree: Return the root (Direct consequence of the Heap property...) $\mathcal{O}(1)$

    – Ex: Binary Heap

Operations on a Heap

- `empty()`/`size()` are straightforward... $\mathcal{O}(1)$
- `peek()`?
  - If only one tree: Return the root (Direct consequence of the Heap property...) $\mathcal{O}(1)$
    – Ex: Binary Heap
  - If forest of k trees: Iterate over all roots $\mathcal{O}(k)$
    Efficient if $k = \mathcal{O}(\log n)$ – Ex: Binomial Heap

Operations on a Heap

- `empty()`/`size()` are straightforward... $\mathcal{O}(1)$
- `peek()`?
  - If only one tree: Return the root (Direct consequence of the Heap property...) $\mathcal{O}(1)$
    – Ex: Binary Heap
  - If forest of k trees: Iterate over all roots $\mathcal{O}(k)$
    Efficient if $k = \mathcal{O}(\log n)$ – Ex: Binomial Heap
  - General case: Keep a pointer to the smallest root $\mathcal{O}(1)$
    – Ex: Fibonacci Heap

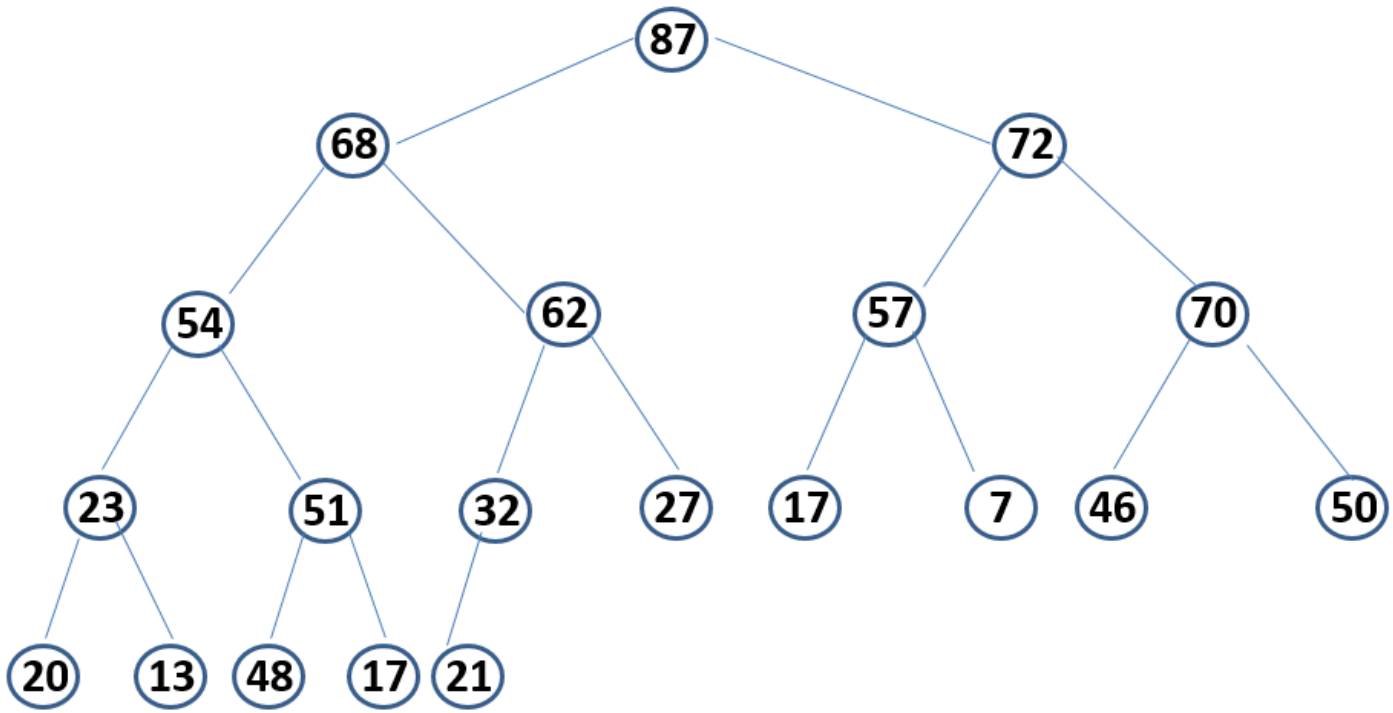
Operations on a Heap

- `empty()`/`size()` are straightforward... $\mathcal{O}(1)$
- `peek()`?
  - If only one tree: Return the root (Direct consequence of the Heap property...) $\mathcal{O}(1)$
    – Ex: Binary Heap
  - If forest of k trees: Iterate over all roots $\mathcal{O}(k)$
    Efficient if $k = \mathcal{O}(\log n)$ – Ex: Binomial Heap
  - General case: Keep a pointer to the smallest root $\mathcal{O}(1)$
    – Ex: Fibonacci Heap
- Insertion/Deletion depends on the underlying tree.
  Natural goal: keep the height to $\mathcal{O}(\log n)$

Binary Heap

almost complete tree: only the last level may not be complete

$+ \forall$ node, more leaves in the left subtree

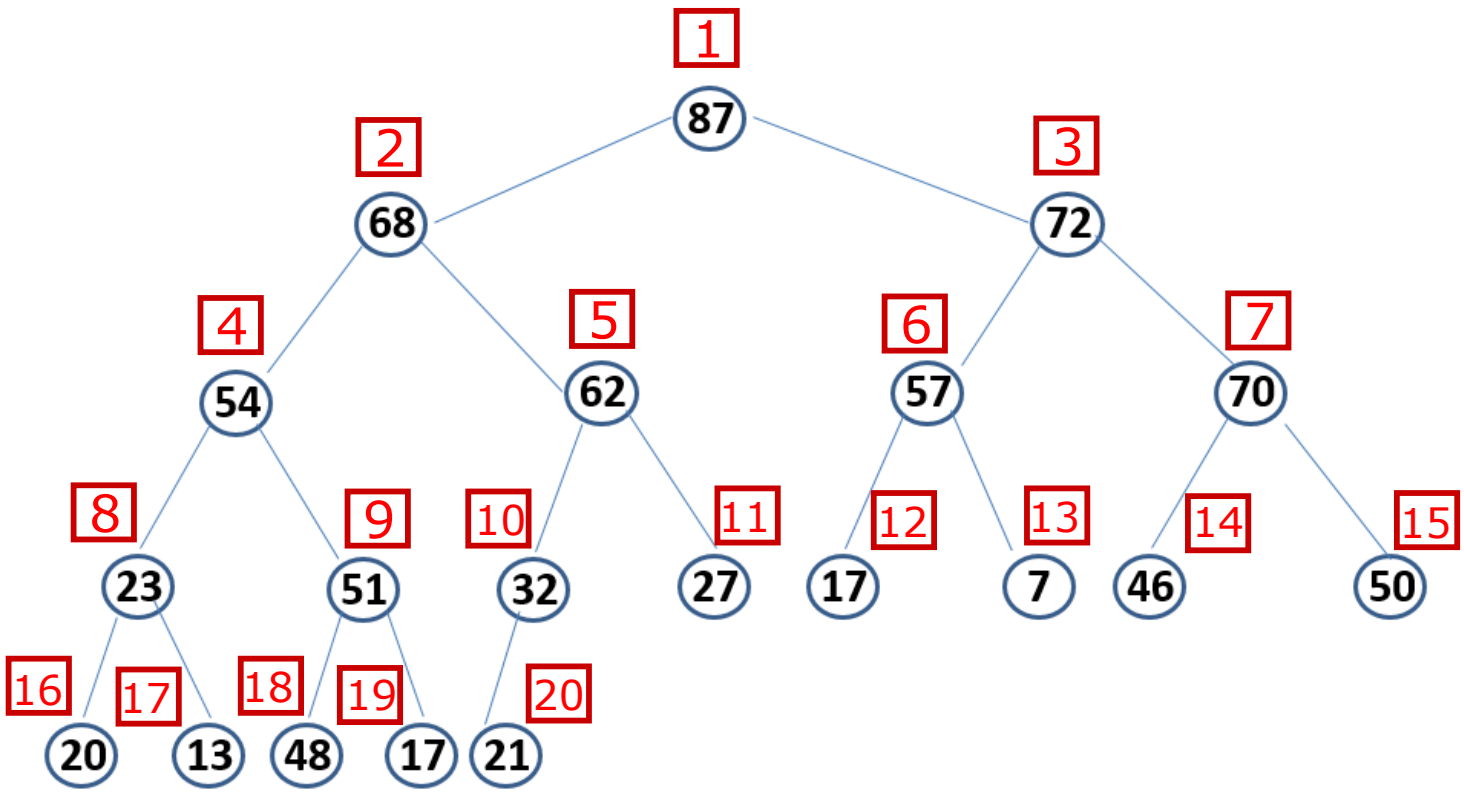


Observation # 1: Binary Tree but not a Binary *Research* Tree.

Observation # 2: for our examples, we consider a Binary Max Heap.

Encoding Binary Heaps (1/2)

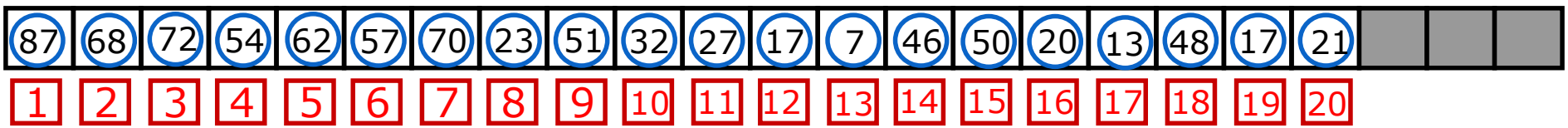
BFS from the root.
Children visited from left (largest subtree) to right (smallest subtree).



Observation: Node i has for children Nodes $2i, 2i + 1$.

Encoding Heaps (2/2)

Using an array!



$\texttt{left\_child}[i] = 2i$; $\texttt{right\_child}[i] = 2i + 1$.
$\texttt{father}[i] = \lfloor i/2 \rfloor$.

Array length = maximum capacity (if unknown, then use doubling arrays).

Keep track of the current number of nodes: counter c (here: $c = 20$).

Automatically handled by classic C++ implementation:

```
typedef vector<int> BinaryHeap;
```

Insertion

Insert a new ~~item v with~~ key p.

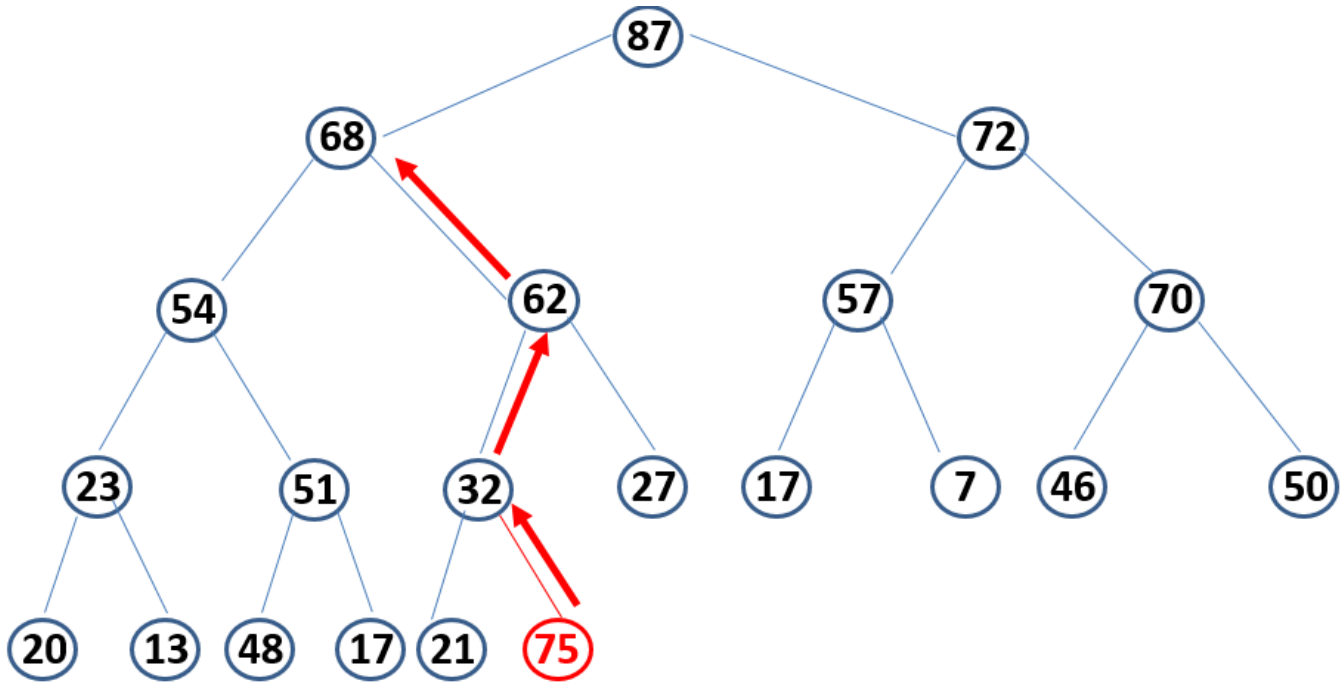
- Push back p in the vector.

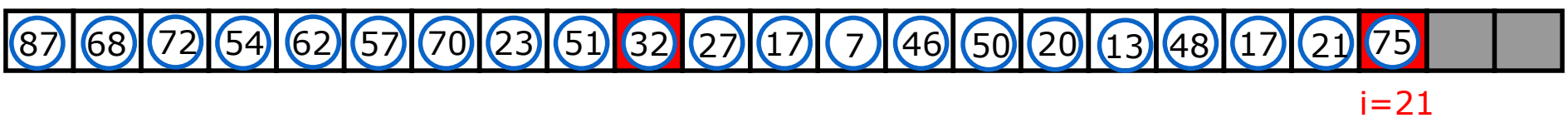
- Repeatedly swap p with its father until the Heap property is once more satisfied.

```
void insert(BinaryHeap& h, int p) {
   h.push_back(p); int i = h.size()-1;
   while(i > 0 && h[father(i)] < p) {
      swap(h,i,father(i)); i = father(i);
   }
}
```

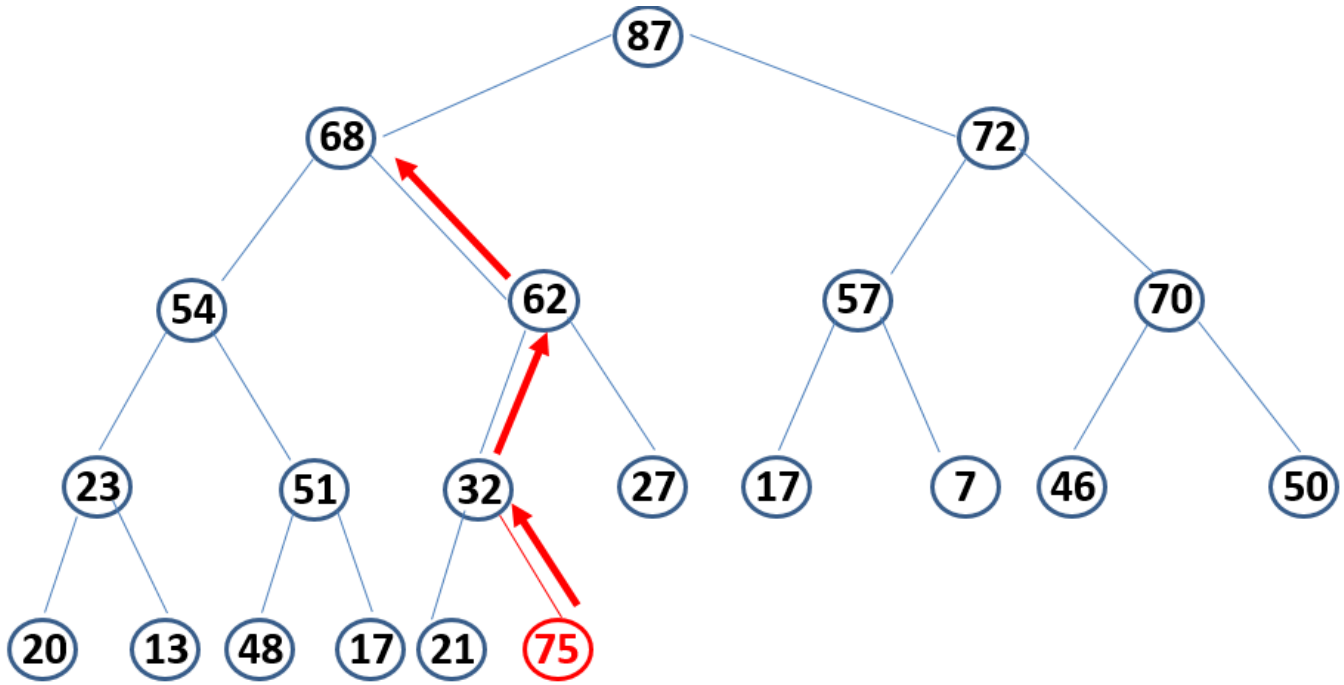
Complexity: $\mathcal{O}(\log n)$

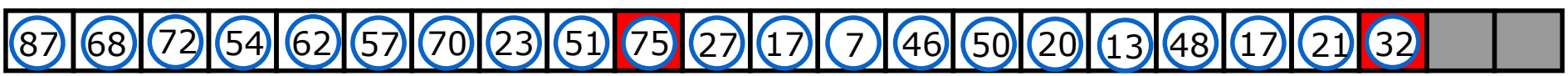
Insertion: Example





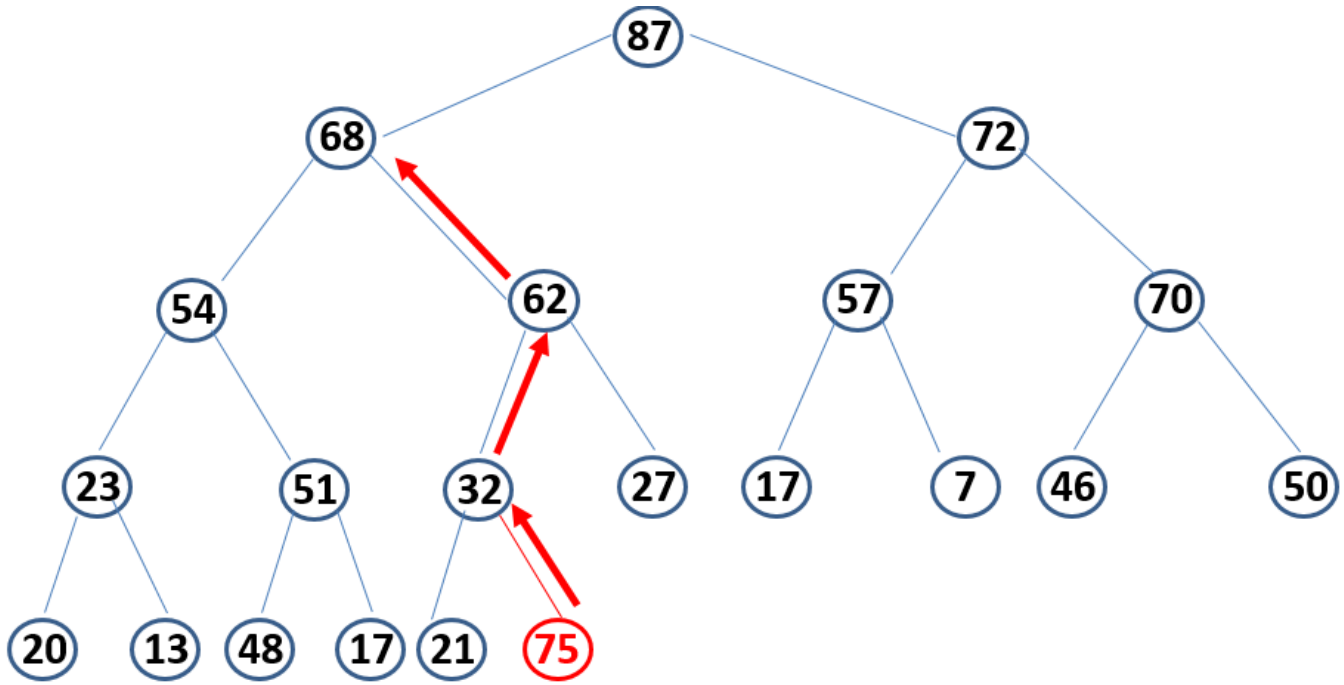
Insertion: Example

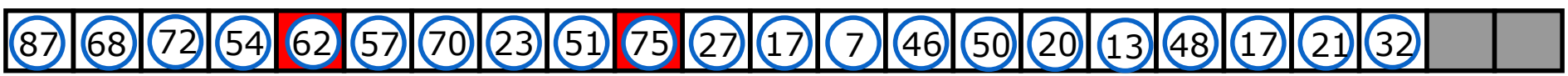




i=10

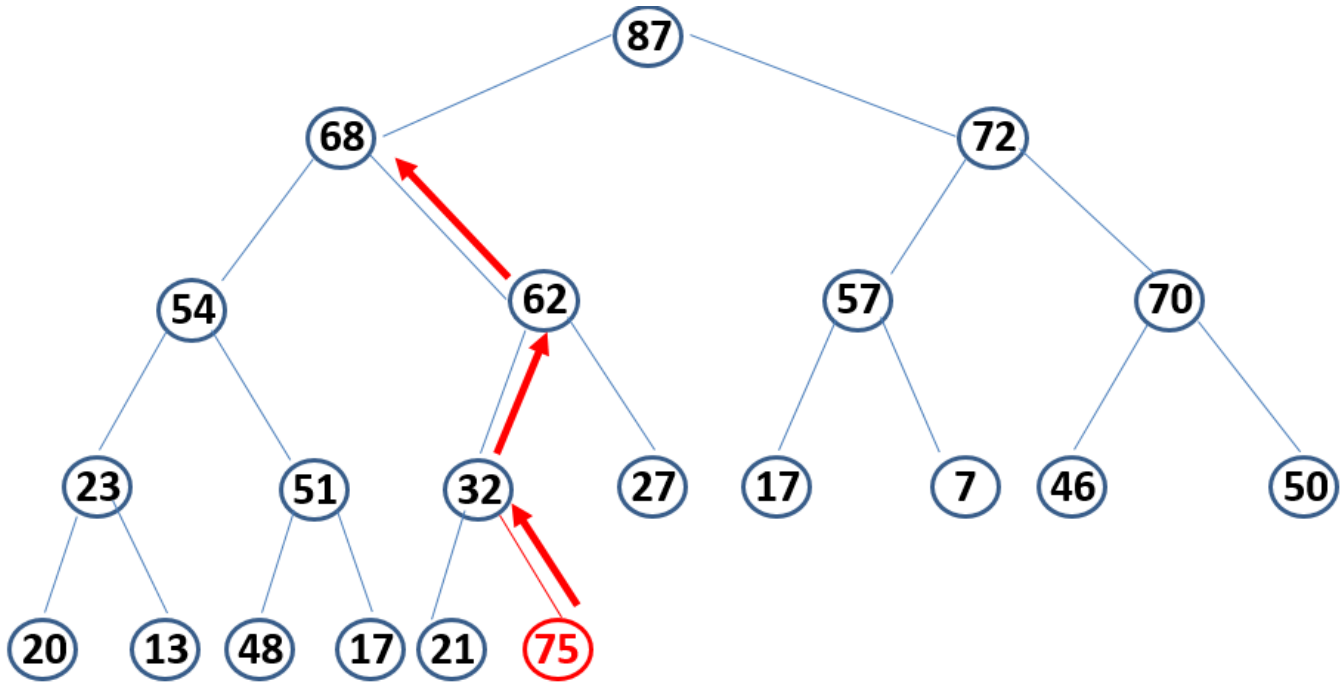
Insertion: Example

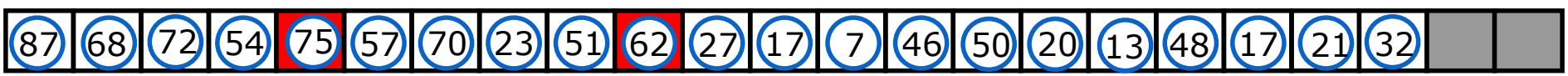




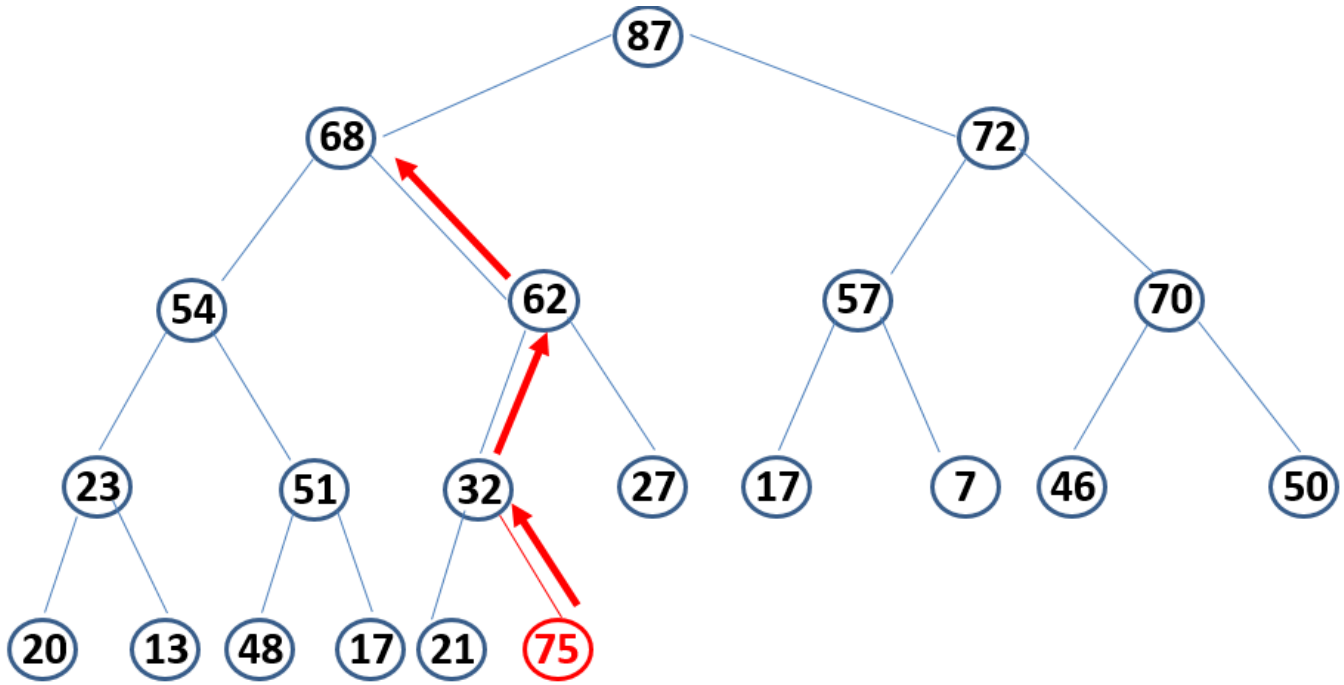
i=10

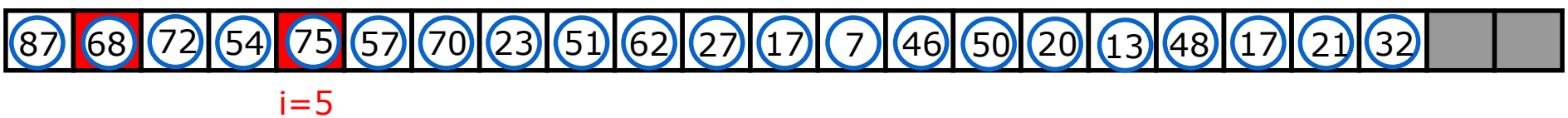
Insertion: Example



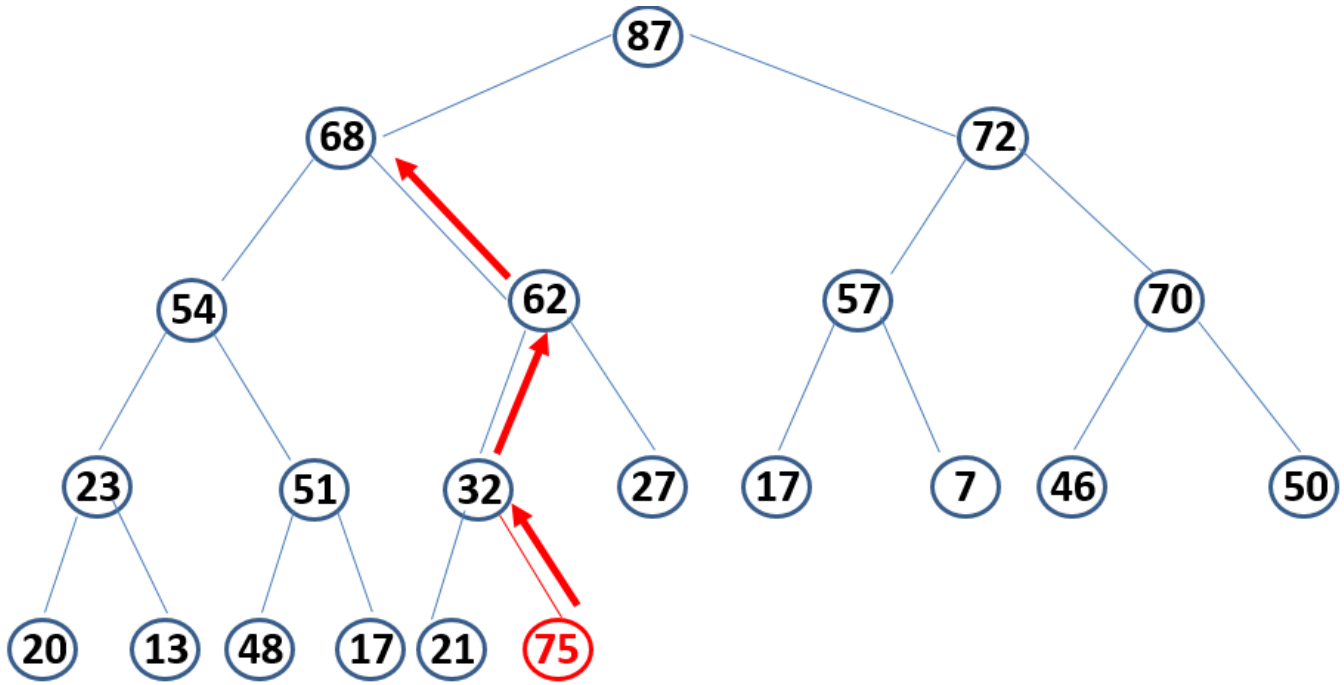


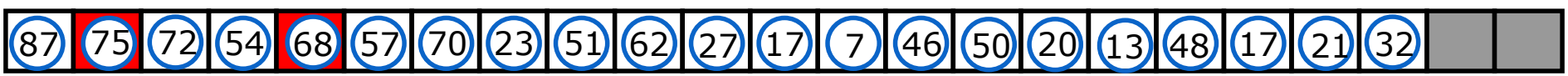
Insertion: Example





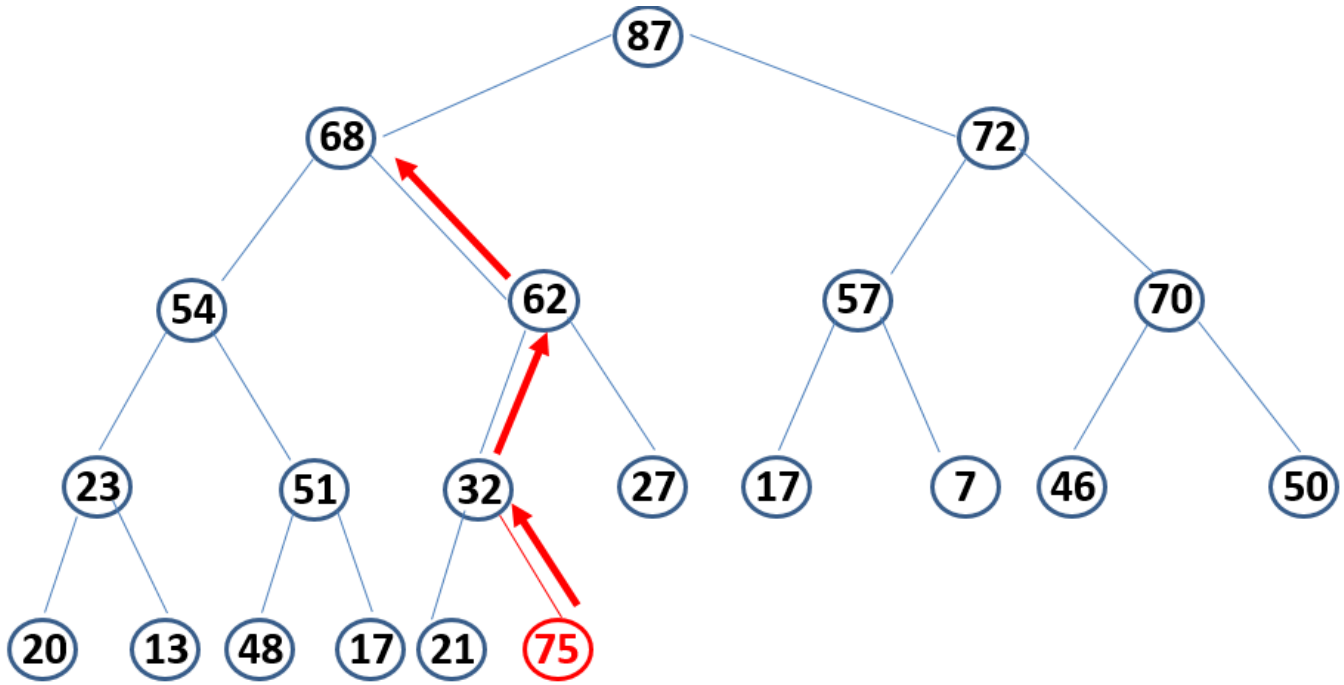
Insertion: Example

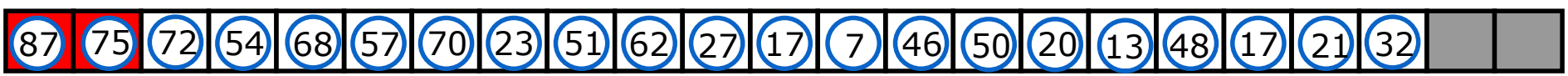




i=2

Insertion: Example





i=2

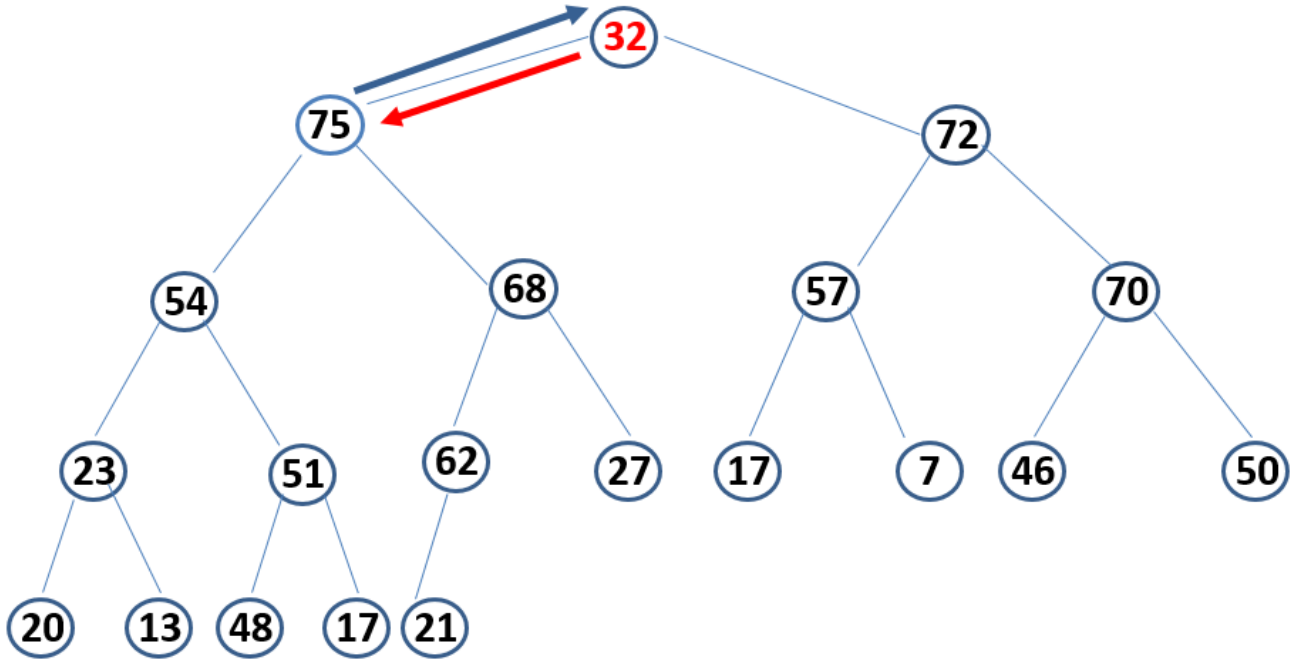
Deletion of the root

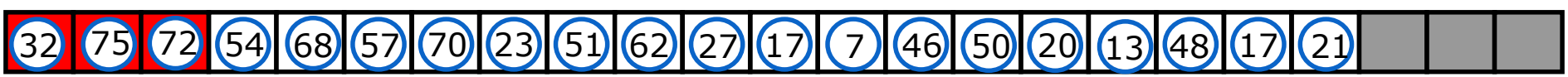
Swap the root with the rightmost leaf on the last level.

- Repeatedly swap with a largest child until the Heap property is once more satisfied.

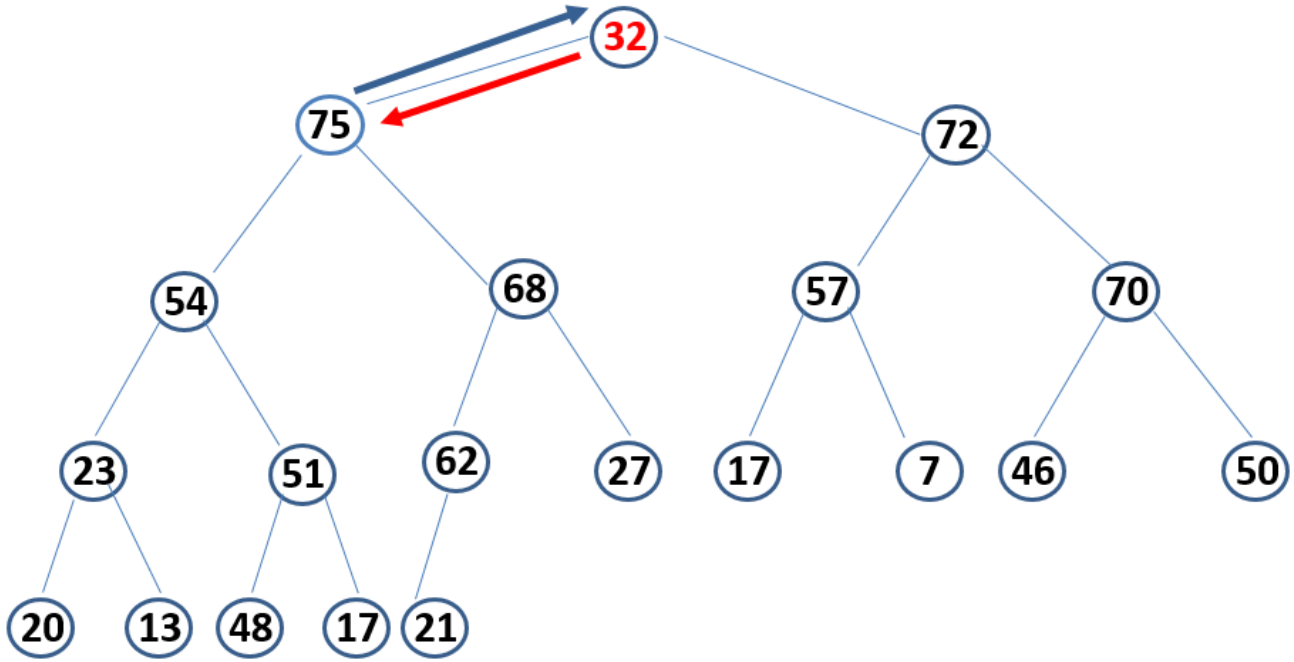
```
void delete_max(BinaryHeap& H) {
    swap(H,0,H.size()-1); H.pop_back(); int i = 0;
    while(left[i] < H.size()) {
        int j = left[i]; //largest child
        if(right[i] < H.size() && H[right[i]] >= H[left[i]])
            j = right[i];
        if(H[i] < H[j]) { swap(H,i,j); i = j; }
        else break;
    }
}
```

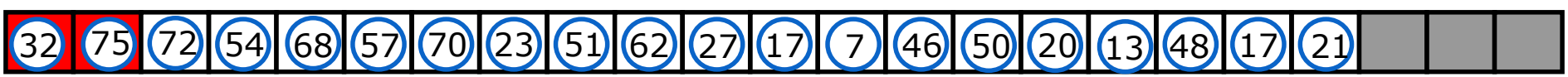
Deletion of the root: Example



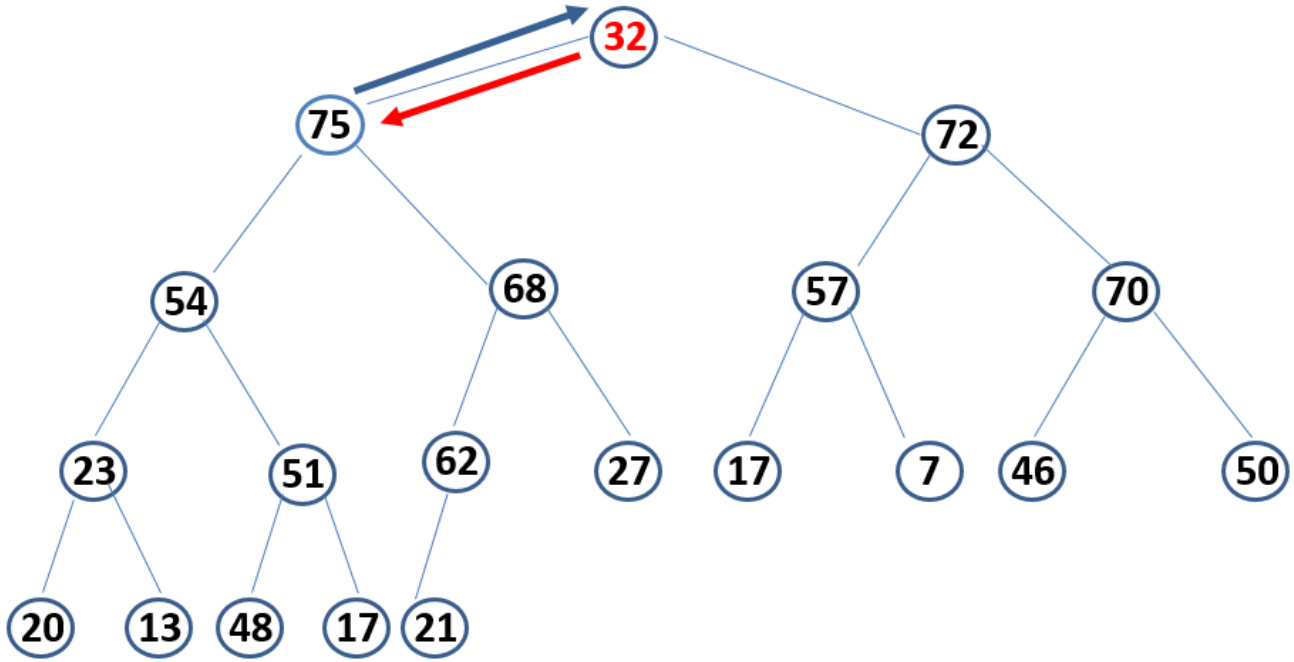


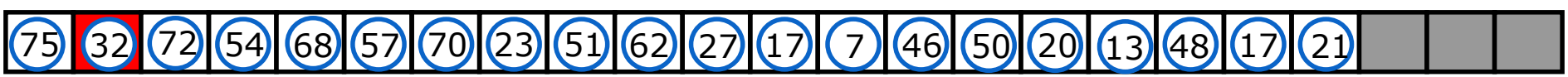
Deletion of the root: Example



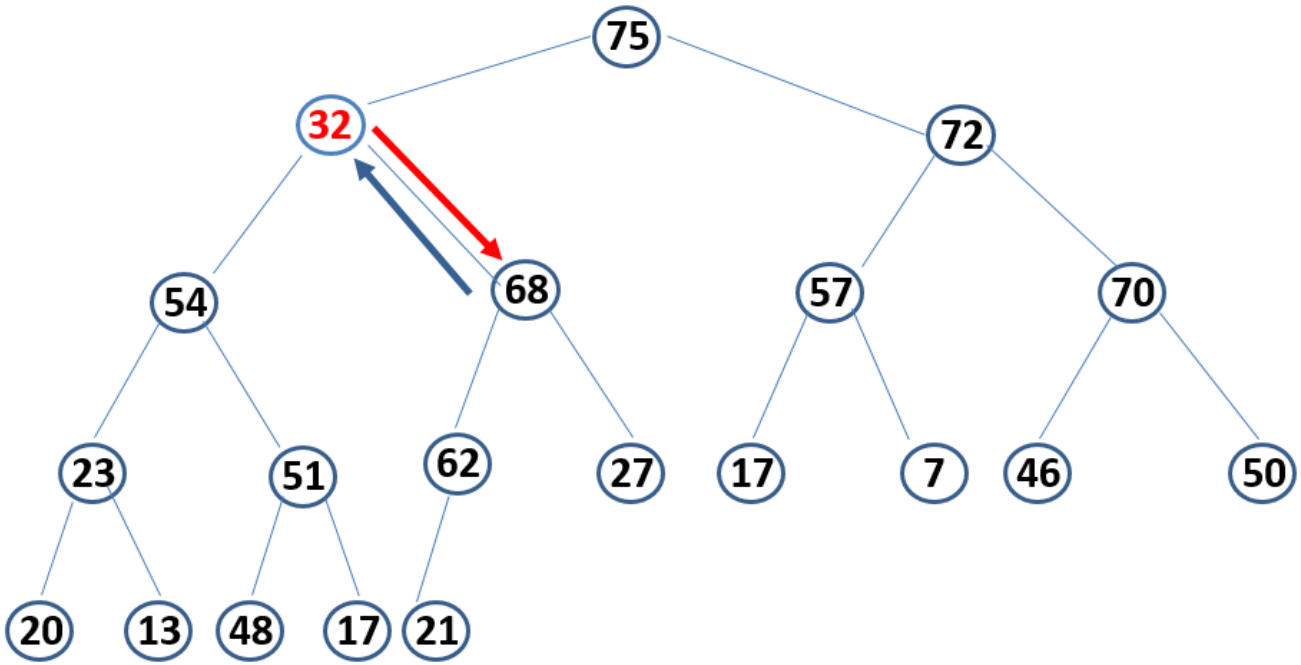


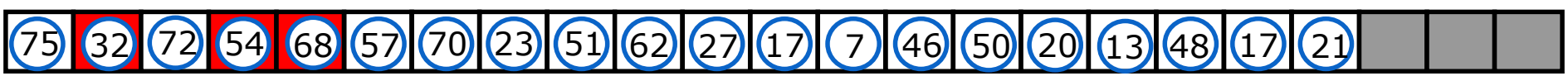
Deletion of the root: Example



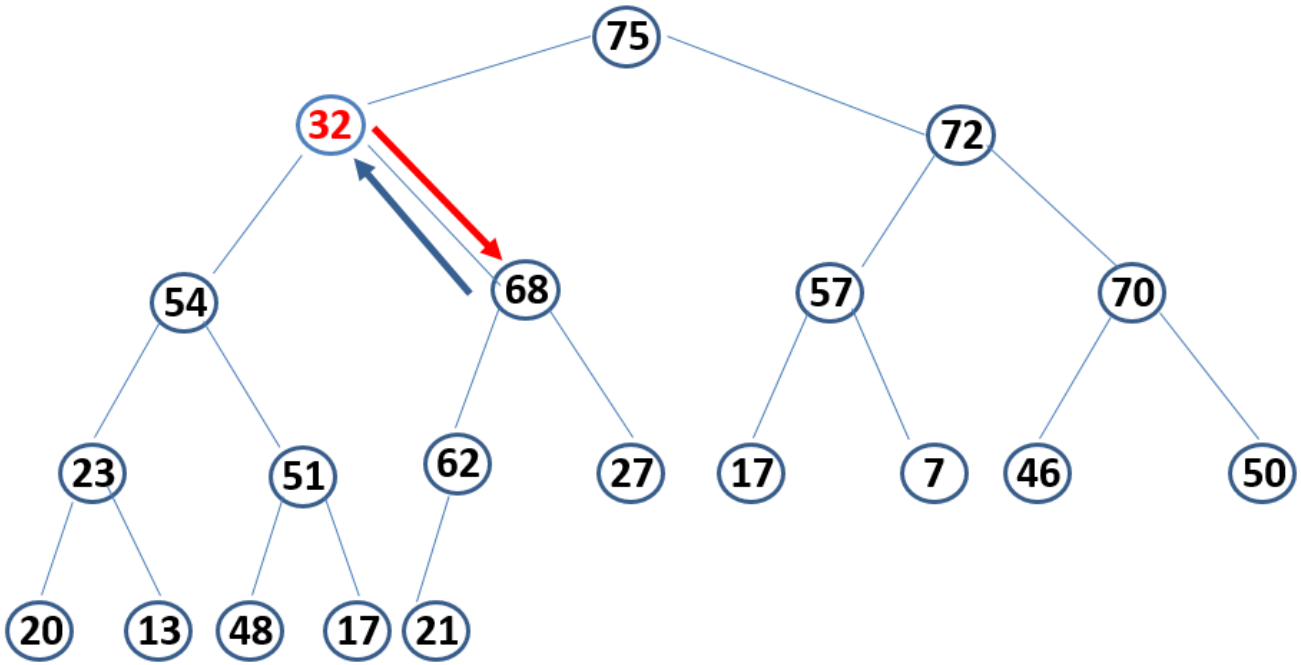


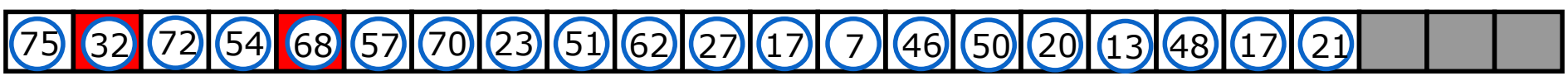
Deletion of the root: Example



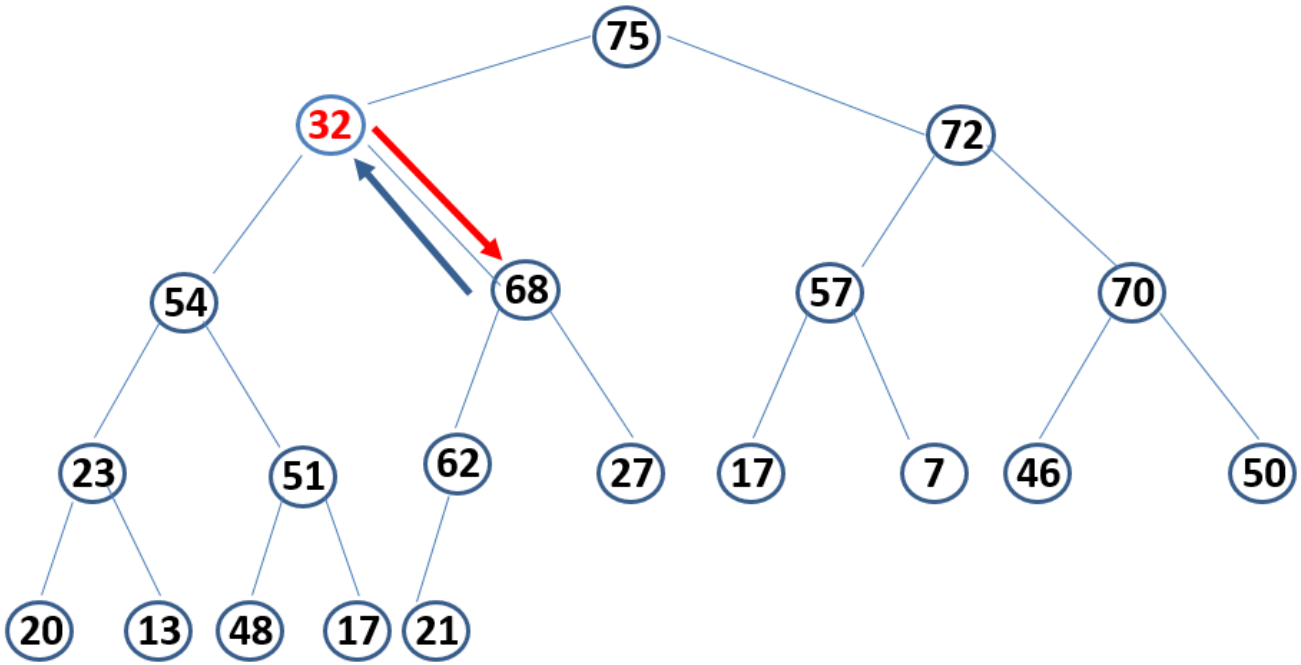


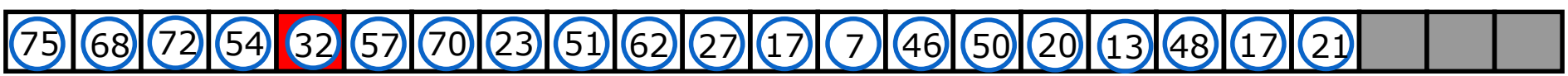
Deletion of the root: Example



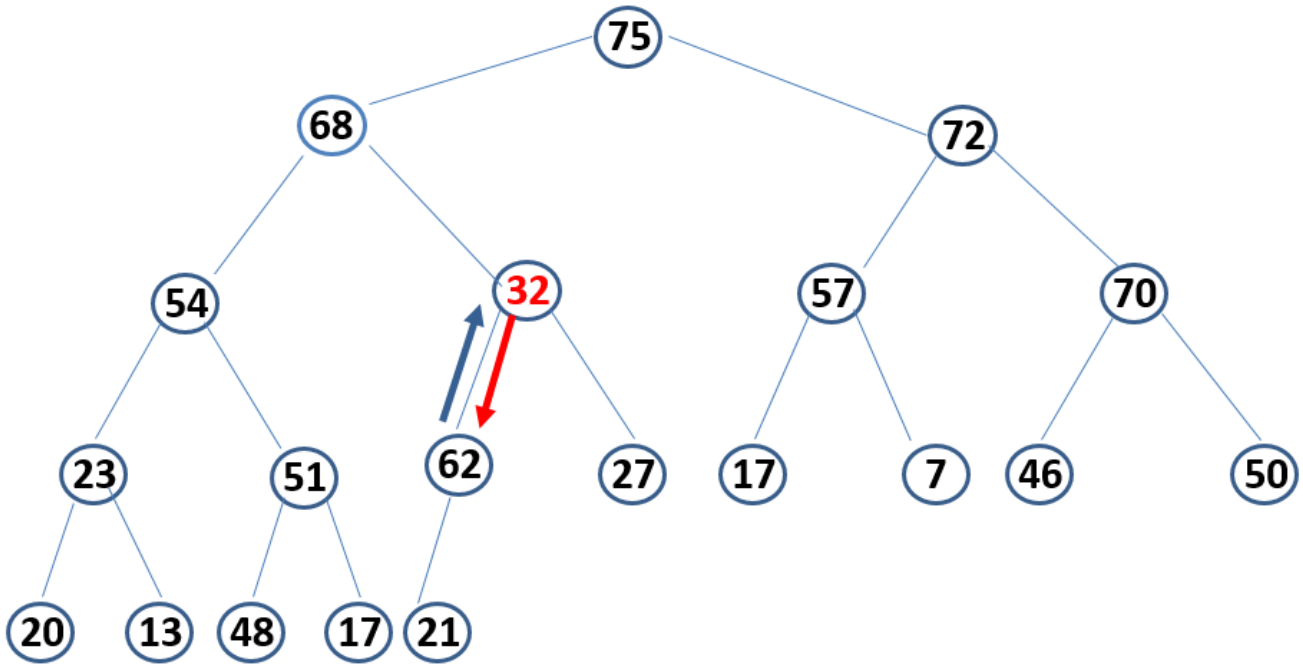


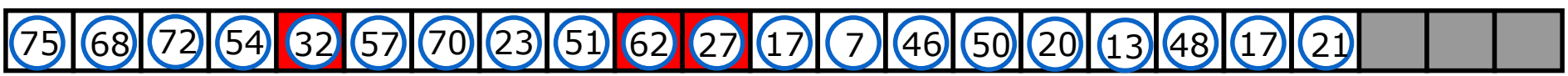
Deletion of the root: Example



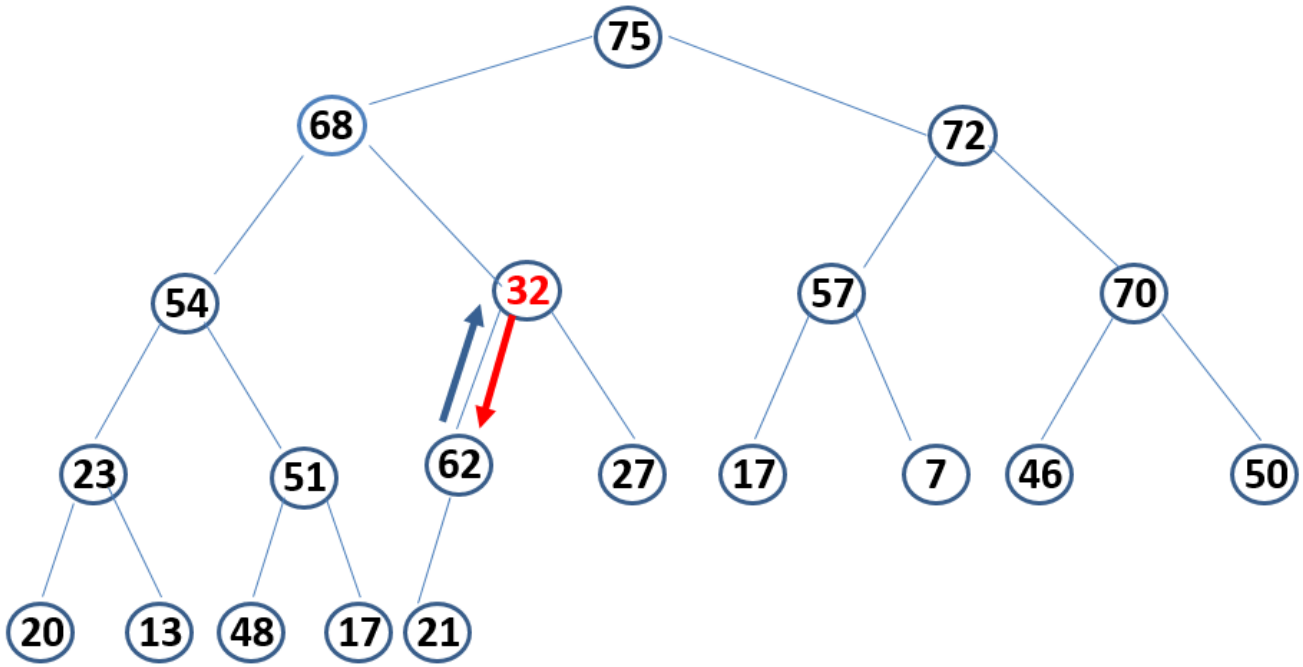


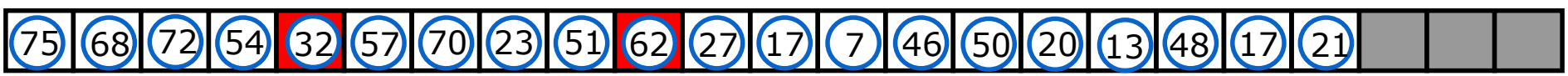
Deletion of the root: Example



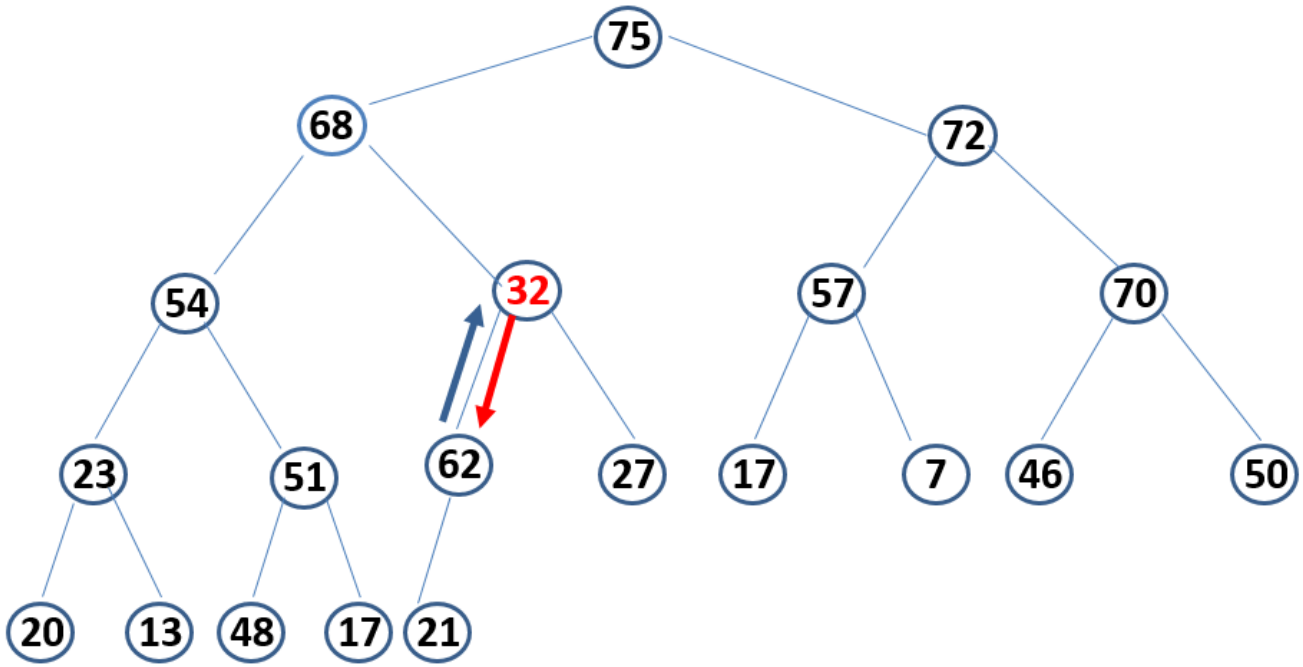


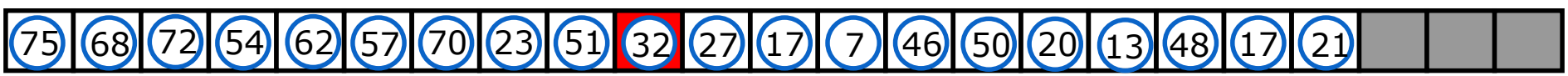
Deletion of the root: Example



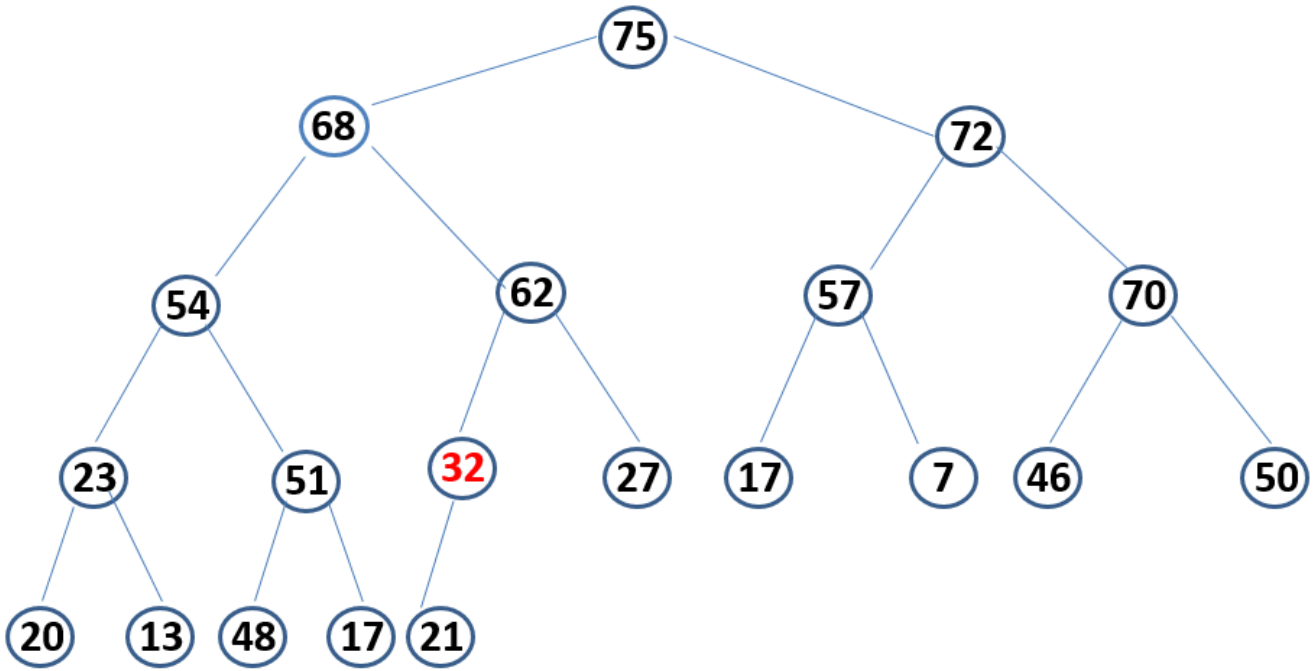


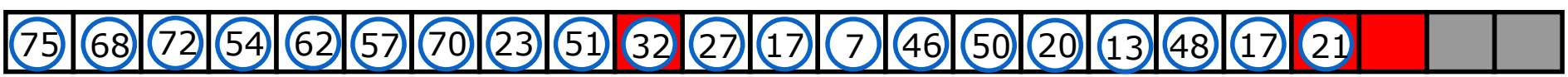
Deletion of the root: Example





Deletion of the root: Example





Deletion of an arbitrary node

Simplified input: index i of the node in the array (reminder: can be retrieved, *e.g.*, using a Hash table)

- First strategy: repeatedly swap the element with its father until it becomes the root.

```
void delete(BinaryHeap& H, int i) {
   if(i > 0) { swap(H,i,father(i)); delete(H,father(i)); }
   else delete_max(H);
}
```

- Second strategy: swap with the last node in the array and repeatedly go up/down.

Heapify

All elements in a vector can be inserted in (and then removed from) a Binary Heap with no extra space needed

(use the vector to implement the Binary Heap!)

```
void heapify(vector<int>& v) {
   for(int i = 1; i < v.size(); i++) {
      int j = i;
      while(j > 0 && v[father(j)] < v[j]) {
         swap(v,j,father(j)); j = father(j);
      }
   }
}
```

Complexity: $\mathcal{O}(n \log n)$.

Application: William's Heap Sort

Sorting a vector:

```
void sort(vector<int>& v) {
   heapify(v);
   for(int i = 0; i < v.size()-1; i++) {
      swap(v,0,v.size()-1-i); int j = 0;
      while(left(j) < v.size()-1-i) {
         int k = left(j);
         if(right(j) < v.size()-1-i && v[right(j)] >= v[left(j)])
            k = right(j);
         if(v[k] > v[j]) { swap(v,k,j); j = k; }
         else break;
      }
   }
}
```

Complexity: $\mathcal{O}(n \log n)$ + In-place

Binary Heap: Pros and Cons

Pros:

- Competitive with BST
- Simpler implementation
- William's Heap Sort

Cons:

- Insertion/Deletion in both $\mathcal{O}(\log n)$
- not meldable

Binomial Heaps

Main differences with binary heaps:

- The structure may store > 1 trees (*i.e.*, it is a forest)
- The trees may not be binary: they are in fact so-called "binomial trees".

$\rightarrow$ **meldable**!

Roadmap:

1) Presentation of binomial trees + their main properties

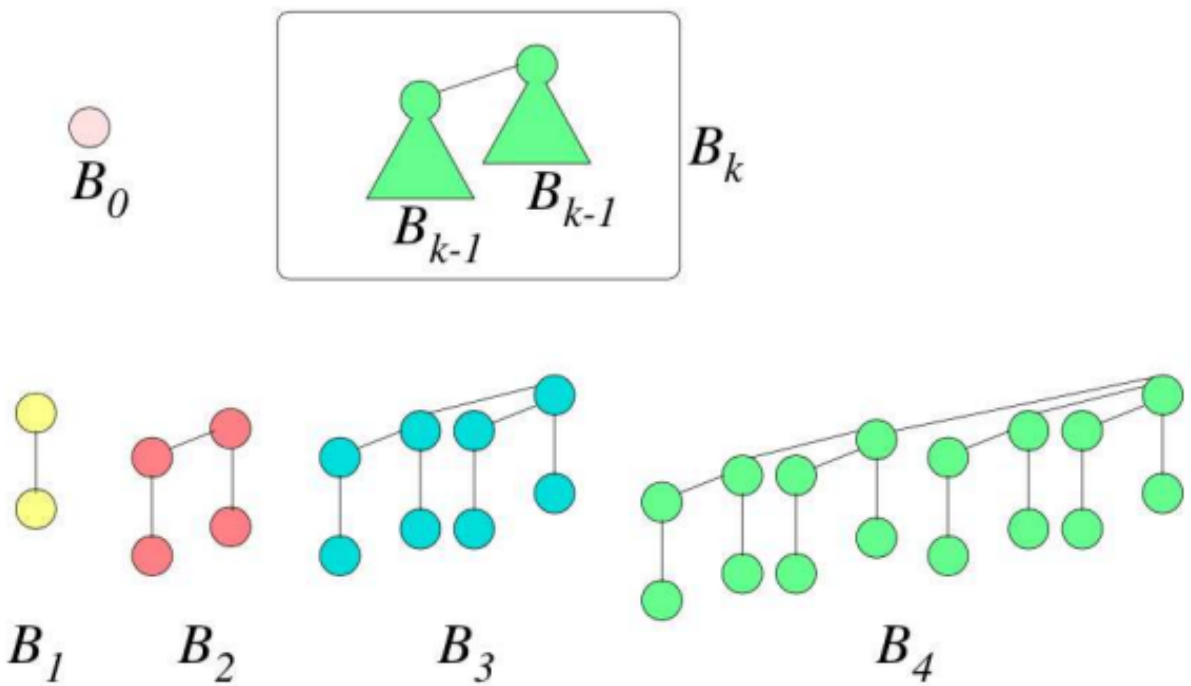
2) Presentation of binomial heaps + their encoding

3) Operations + implementation

Binomial Trees

Definition (Binomial Trees)

- The one-node tree B_0 is binomial;
- If B_{k-1} is binomial, then let the rooted tree B_k be obtained from two isomorphic B_{k-1}'s by making the root of one copy the leftmost child of the root of the other copy. We also have that B_k is binomial.



Properties

Easy to prove by induction on k:

- B_k has order (number of nodes) 2^k.
- B_k has height k. $\Longrightarrow$ Balanced tree
- There are $\binom{k}{i}$ nodes at level i.
- The degree of B_k equals k (logarithmic in the order). Furthermore, the root is the only node with k children.
- The subtrees rooted at children of the root are (from left to right) $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$.

Encoding

For every $k \geq 0$, there is only one binomial tree B_k whose root has height/degree k.

$\Longrightarrow$ We needn't any special structure. **Any tree implementation storing the degree/height at each node is sufficient!**

```
struct node {
   int value;
   int degree; //==k iff it is B_k
   node *father, *child, *previous, *next;
};

typedef node *BinomialTree;
```

Merge of two B_{k-1}'s

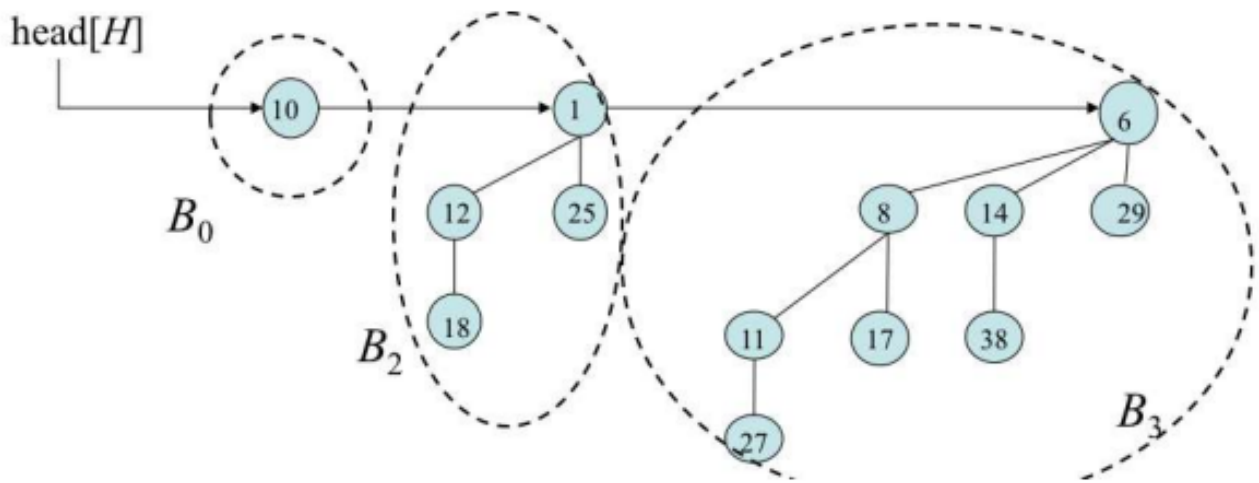
```
//Complexity: O(1)
void addChild(BinomialTree& T1, BinomialTree& T2){
   T1->degree++;
   if(T1->child != nullptr)
      T1->child->previous = T2;
   T2->next = T1->child;
   T1->child = T2; T2->father = T1;
}

//Complexity: O(1)
void merge(BinomialTree& T1, BinomialTree& T2) {
   if(T1->value <= T2->value)
      addChild(T1,T2);
   else {
      addChild(T2,T1); T1 = T2;
   }
}
```

Binomial Heap: specifications

- A binomial heap is an ordered collection of binomial trees (list where the binomial trees are ordered w.r.t. their degree/height).

- For every $k \geq 0$ there is <u>at most one</u> copy of B_k in the list.

- Each tree respects the (Min/Max) Heap property.




```
typedef list<BinomialTree> BinomialHeap;
```

Fundamental Property: Number of Trees

- If there are k binomial trees in the heap then the number of elements must be at least:

$$n(B_0) + n(B_1) + \ldots + n(B_{k-1}) = \sum_{i=0}^{k-1} 2^i = 2^k - 1$$

A Binomial Heap with n elements contains at most $\mathcal{O}(\log n)$ trees.

Remark: this is no longer true if we can have multiple copies of each binomial tree

–Simulation of a list with B_0's

- Finer analysis: one-to-one mapping with the binary representation of n:

$$B_k \text{ in the Heap} \iff \text{the } k^{th} \text{ bit of } n \text{ is set to 1}$$

Binomial Heaps are meldable!

In order to make the union of two binomial Heaps H1,H2, we scan both lists in order:

*) If both lists contain a copy of B_k, then we merge both copies into a copy of B_{k+1}. Note that there may exist copies of B_{k+1} in H1 and/or H2.

$\rightarrow$ if both H1 and H2 contain a copy of B_{k+1}, then both copies shall be merged in some copy of B_{k+2} at a later stage of the scan. Therefore, the third copy of B_{k+1} shall be preserved.

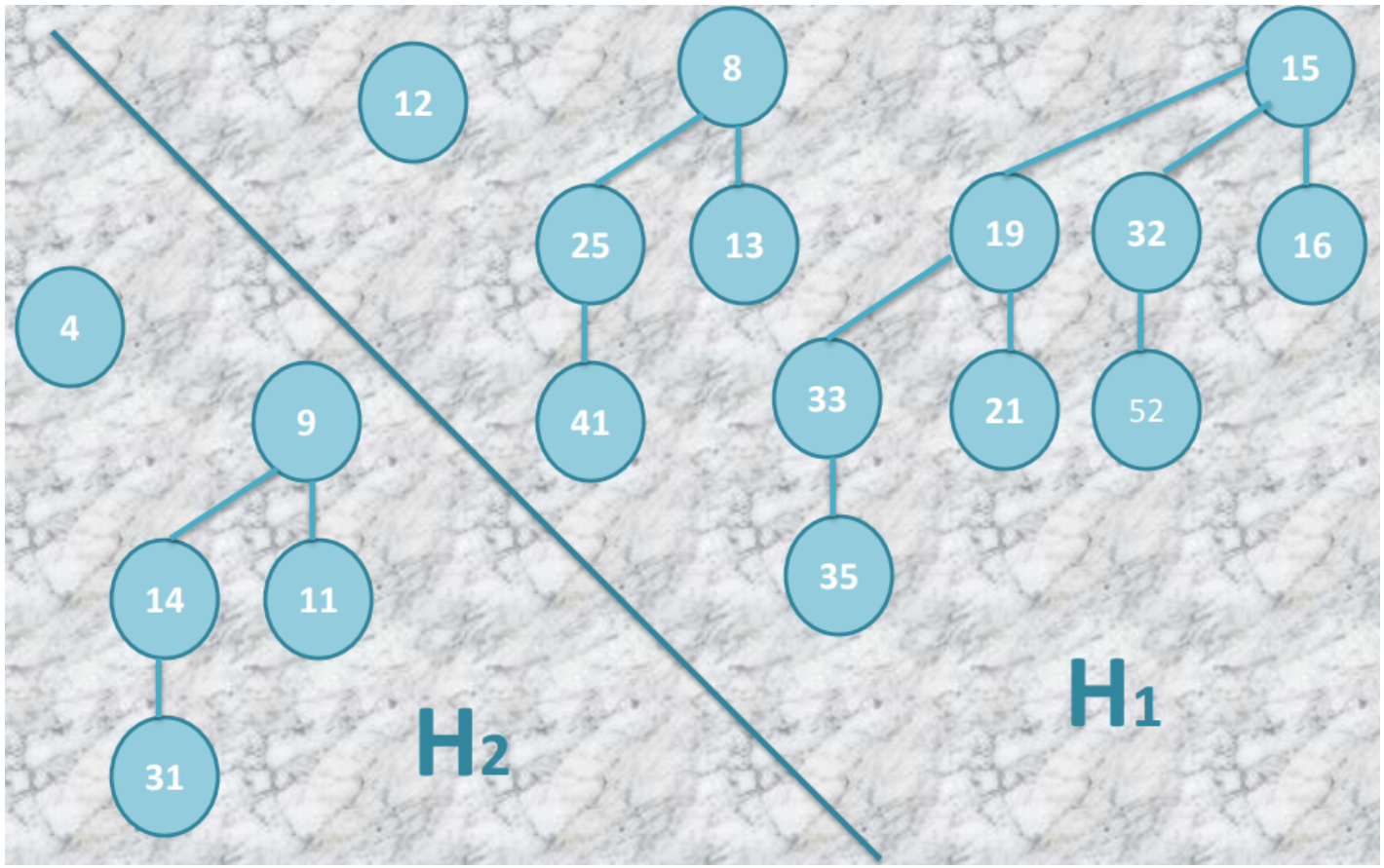
$\rightarrow$ otherwise, we insert B_{k+1} at the head of any of H1,H2 that does not already contain a copy of B_{k+1}. We continue the scan.

Complexity: $\mathcal{O}(\texttt{H1.size}() + \texttt{H2.size}()) = \mathcal{O}(\log n)$

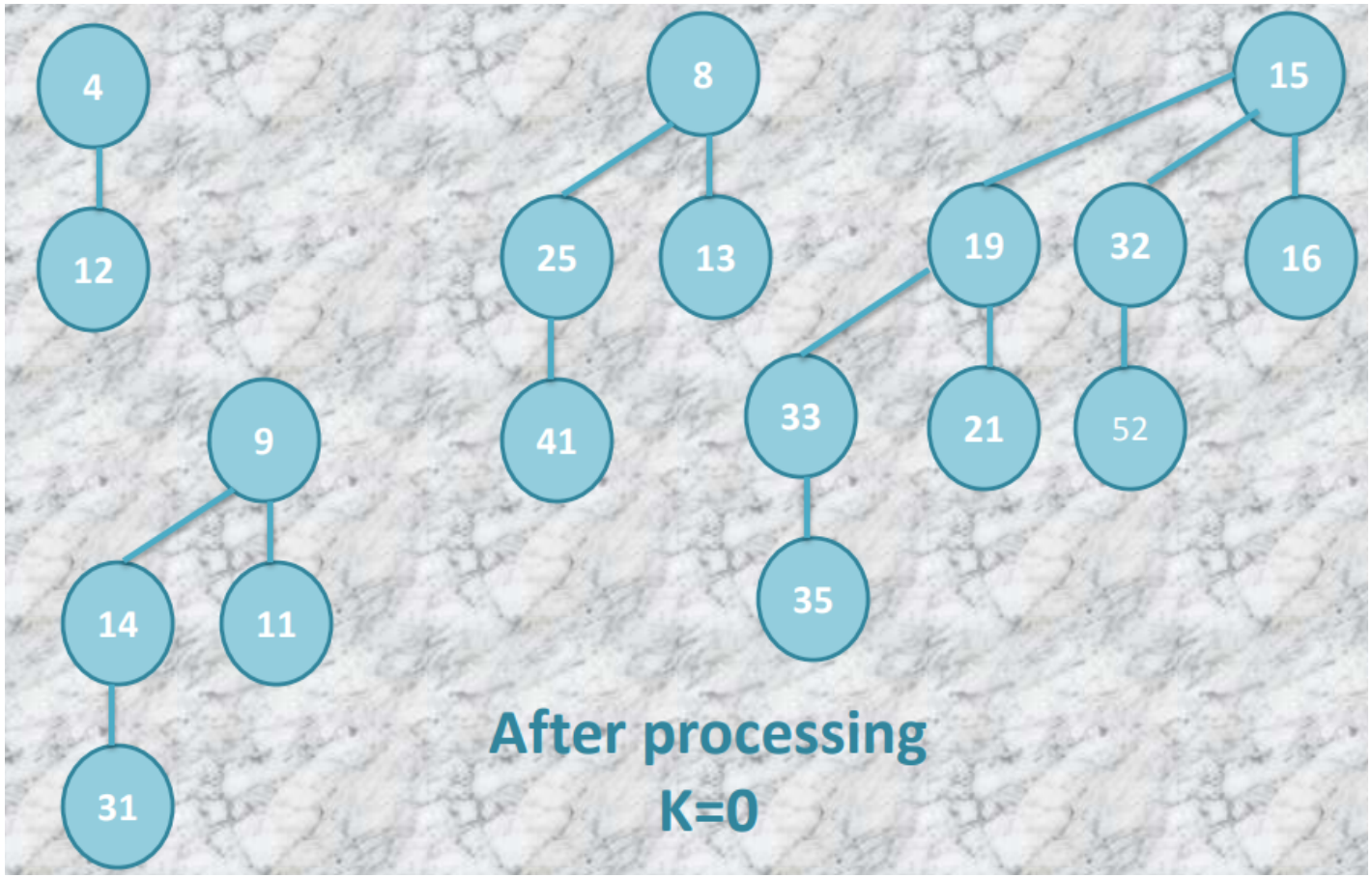
Meld: Implementation

```
//the union of both heaps is stored in H1
void meld(BinomialHeap& H1, BinomialHeap& H2) {
    if(H1.empty()) H1 = H2;
    else if(!H2.empty()) {
        BinomialTree T1 = H1.front(), T2 = H2.front();
        if(T1->degree < T2->degree) {
            H1.pop_front(); meld(H1,H2); H1.push_front(T1);
        } else if(T2->degree < T1->degree) {
            H2.pop_front(); meld(H1,H2); H1.push_front(T2);
        } else { //T1-¿degree == T2-¿degree == k
            H1.pop_front(); H2.pop_front();
            merge(T1,T2); //T1 now stores B_{k+1}
            if(H1.empty() || (H1.front())->degree > T1->degree) {
                //B_{k+1} not in H1
                H1.push_front(T1); meld(H1,H2);
             }else if(H2.empty() || (H2.front())->degree > T1->degree) {
                //B_{k+1} not in H2
                H2.push_front(T1); meld(H1,H2);
            } else {
                meld(H1,H2); H1.push_front(T1);
}}}}
```

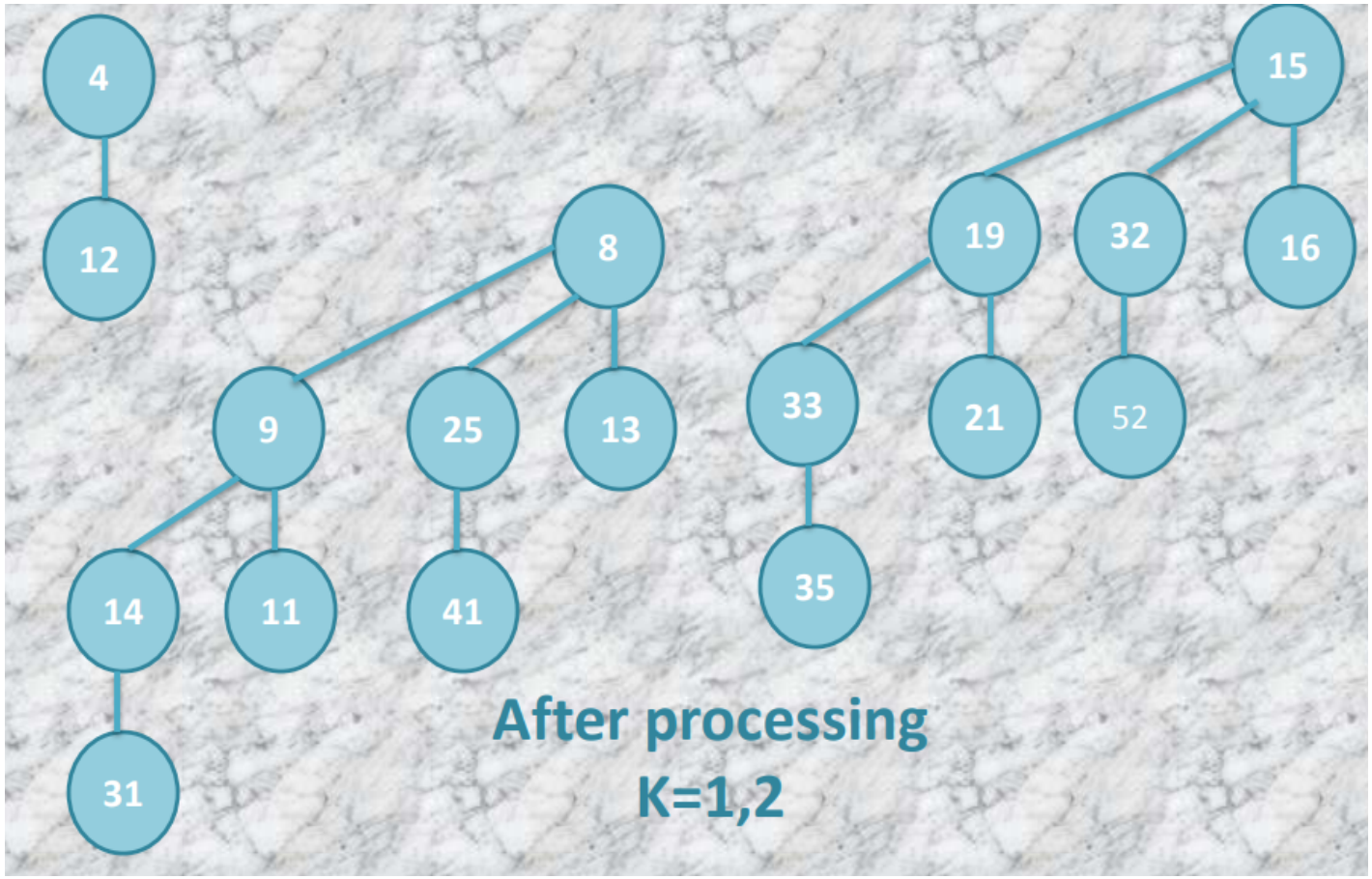
Meld: Examples



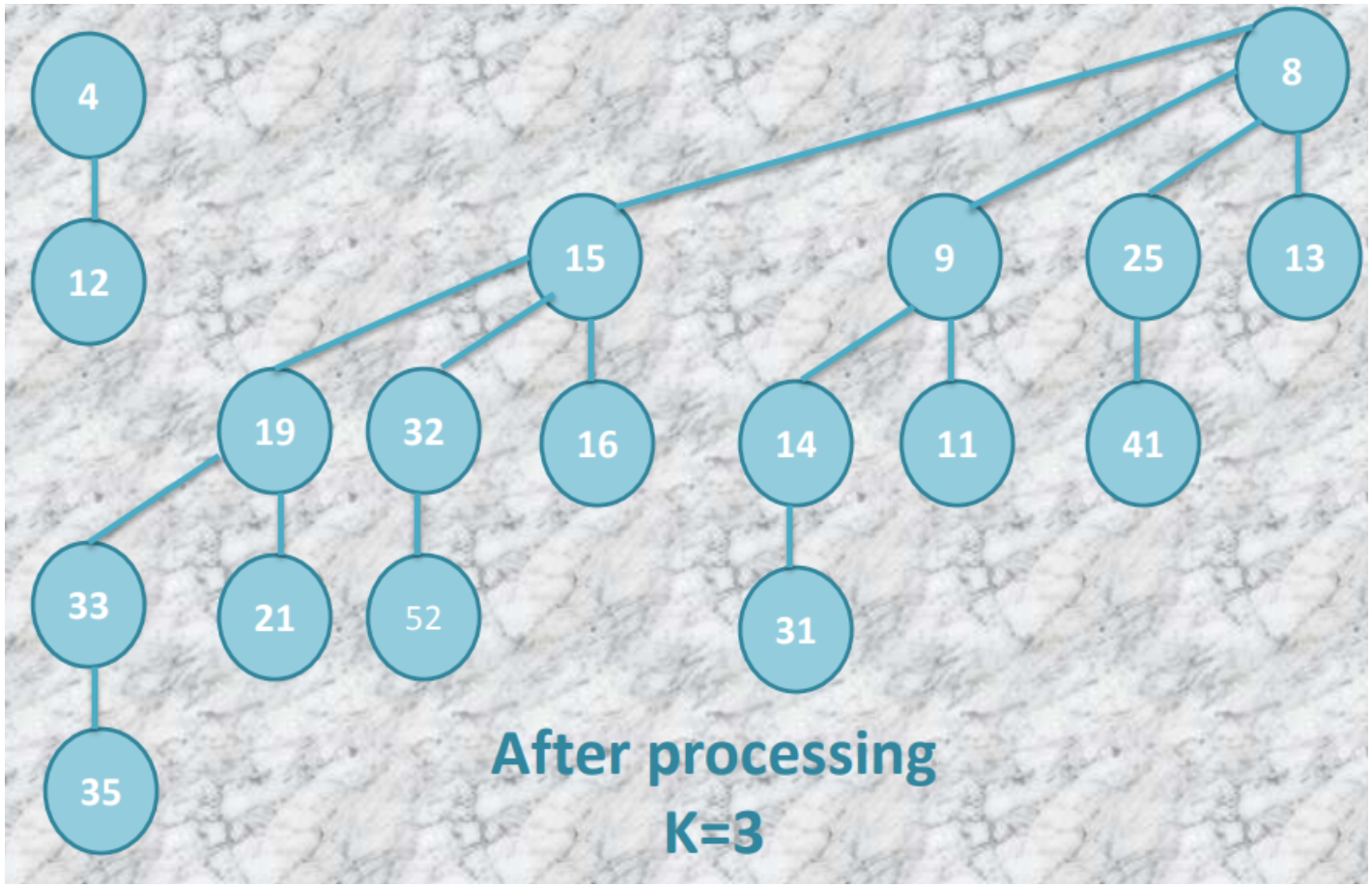
Meld: Examples



Meld: Examples



Meld: Examples



Insertion

- Special case of union of two heaps!

  – One heap is reduced to a B_0 containing the new element

```
void insert(BinomialHeap& H, int p) {
   BinomialTree T = new node; //B0
   T->value = p; T->degree=0;
   T->father=T->child=T->next=T->previous =nullptr;
   BinomialHeap Htmp; Htmp.push_back(T);
   meld(H,Htmp);
}
```

Complexity: $\mathcal{O}(\log n)$

Deletion of the minimum element

Reminder: the min. element is the root of some binomial tree in the heap (Min-heap property).

*) Removing the root of a binomial tree B_k leaves binomial trees $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$.

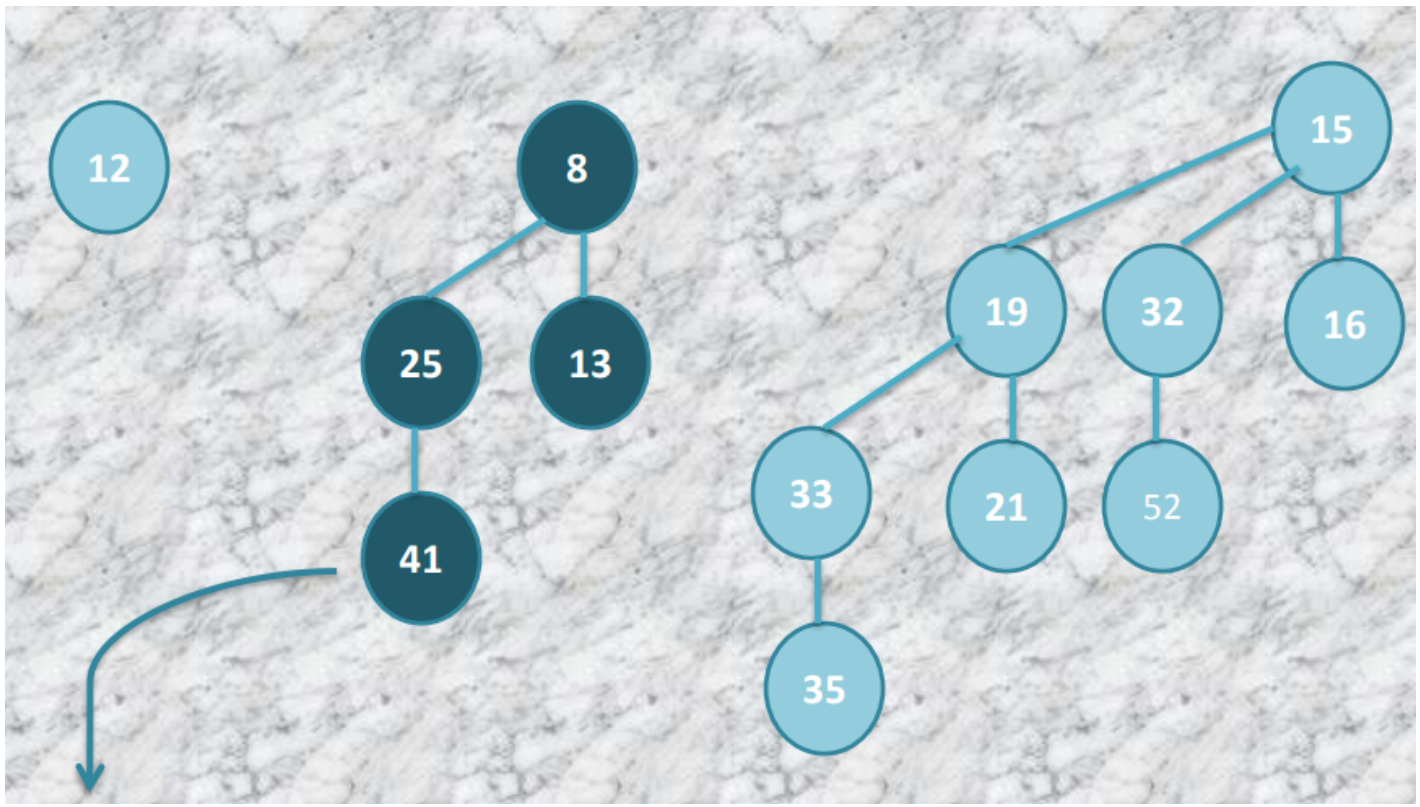
$\rightarrow$ **Union between** $H \setminus B_k$ **and** $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$

Complexity: $\mathcal{O}(\log n)$

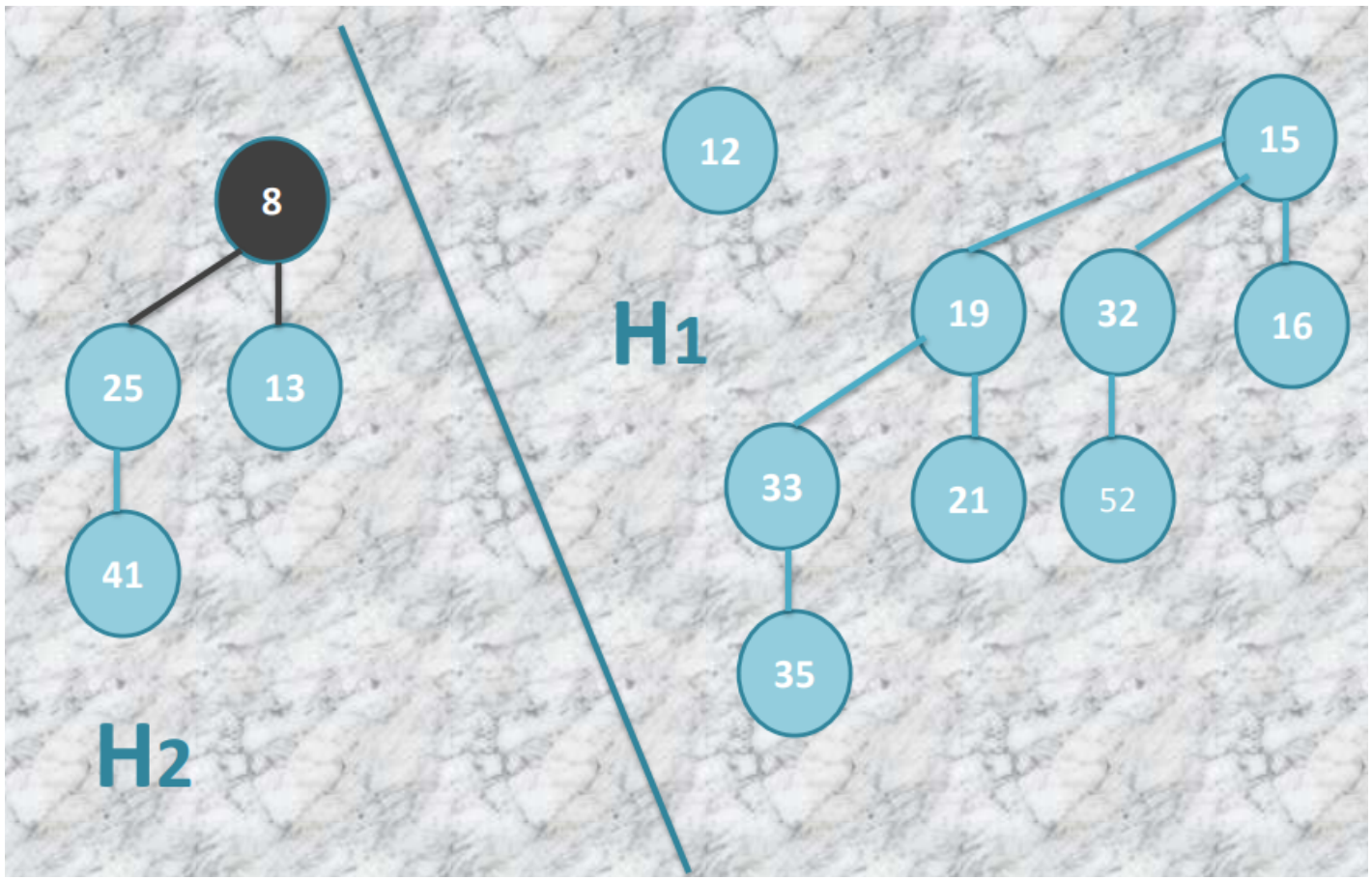
Min. Deletion: Implementation

```
void delete_min(BinomialHeap& H) {
   BinomialHeap H1,H2;

   for(BinomialTree T : H) {
      if(H2.empty() || (H2.front())->value > T->value) {
         for(BinomialTree T2 :  H2) { H1.push_back(T2); }
         H2.clear();
      }
      H2.push_back(T);
   }

   BinomialTree Tmin = H2.front(); H2.pop_front();

   for(BinomialTree T : H2) { H1.push_back(T); }
   H2.clear();
   for(BinomialTree T = Tmin->child; T != nullptr; T = T->next) {
      H2.push_front(T); T->father = nullptr;
   }

   meld(H1,H2); H = H1;
   delete Tmin;
}
```

Min. Deletion: Example



Min. Deletion: Example



Min. Deletion: Example

Decrease key

- Same as for binary heaps.

Simplified input: pointer to the node

Reminder: $\mathcal{O}(1)$-time access using Hash table

```
void decrease_key(BinomialHeap& H, node *n, int d){
   n->value -= d;
   while(n->father != nullptr && n->father->value > n->value){
      swap(n->value,n->father->value);
      n = n->father;
   }
}
```

Complexity: $\mathcal{O}(\log n)$

$\Longrightarrow$ Deletion of an arbitrary node also in $\mathcal{O}(\log n)$

Binomial Heaps: Pros and Cons

Pros:

- Competitive with BST and Binary Heaps
- Meldable

Cons:

- More complex implementation.
- Deletion and Insertion are both in $\mathcal{O}(\log n)$.

Fibonacci heaps

Ensures more flexibility than Binomial Heaps by allowing:

- possibly Non-Binomial Trees
- possibly isomorphic trees (copies of the same tree, but storing different values).
- a large number of trees stored (this number is opportunately reduced to at most $\mathcal{O}(\log n)$ after some operations such as deletions).

Balancedness is ensured by local mechanisms (degree of the root + markings on the nodes).

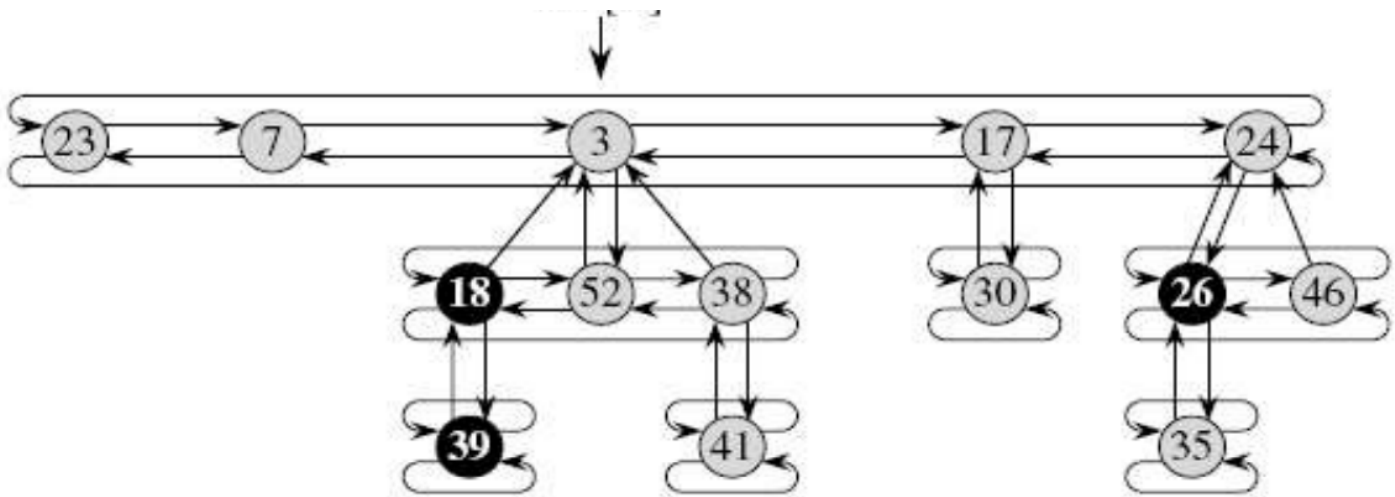
A price to pay: worst-case query time can be as big as $\Omega(n)$ for some queries. However, their **amortized** complexity shall remain logarithmic, or even constant.

Encoding

It suffices to augment any standard tree implementation with:

- one boolean field at each node to indicate whether it is marked (balancedness mechanism)
- pointers between the first and last child of any node (*i.e.*, the children of a node are put in some doubly linked circular list).

The roots of the trees are also put in some doubly linked circular list. The first accessible tree is always the one whose root is the minimum element in the Heap.



Implementation

We can use the `previous`/`next` fields at the root nodes in order to simulate the circular list in which we put all the trees.

```
struct node {
   int value;

   int degree;
   bool mark;

   node *father, *child;
   node *previous, *next;
};

typedef node *FiboHeap;
```

Potential function

For a Fibonacci Heap H let:

- $tree(H)$ denote the number of trees stored in the structure;
- $marked(H)$ denote the number of marked nodes.

The potential of H is defined as:

$$\Phi(H) = tree(H) + 2 \times marked(H)$$

Interpretation:

- Some easy operations (*e.g.*, insert), are sped up by allowing $tree(H)$ to grow. This will compensate **one** delete min. operation (but shall require reorganization of the structure at this point).
- Decrease key operations are sped up by splicing trees rather than using the classical swapping technique with parents. This may cause unbalance, and so sometimes we need to reorganize the structure. Some marks are added on nodes after splicing in order to compensate upcoming reorganizations.

Operations: meld

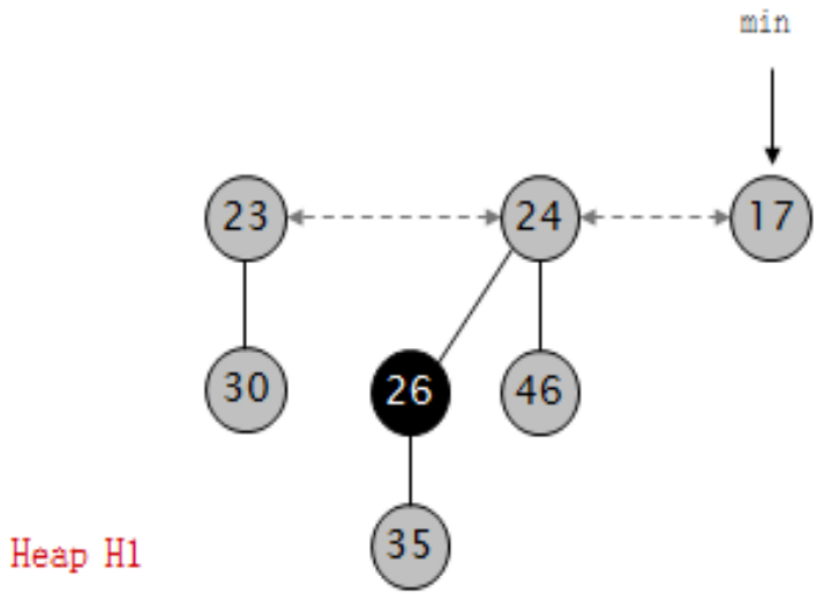
- Union of two heaps H_1, H_2 simply consists of a list concatenation:

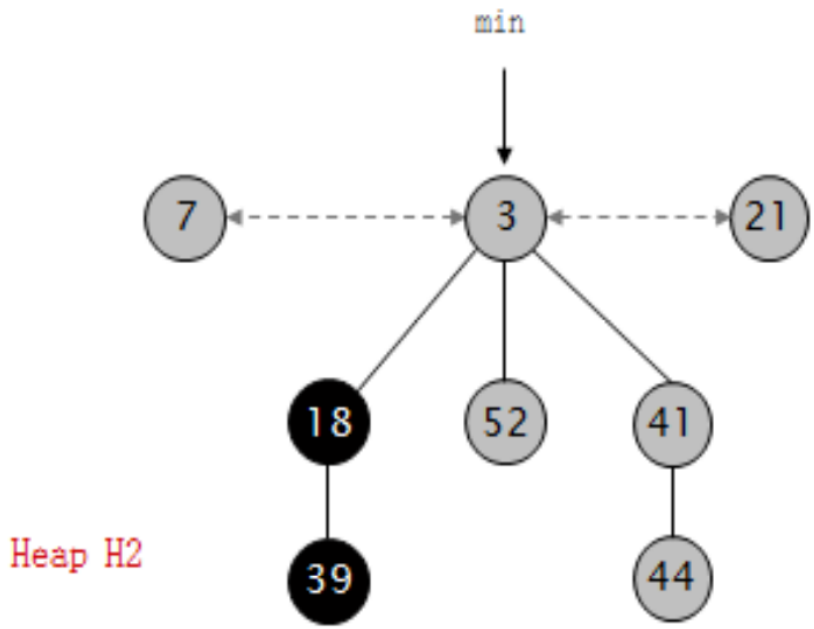
```
void meld(FiboHeap& H1, FiboHeap& H2) {
   node *tail = H1->previous;
   H1->previous = H2->previous; H1->previous->next = H1;
   tail -> next = H2; H2->previous = tail;
   if(H2->value < H1->value)
      H1 = H2; //pointer to min
}
```

Complexity: $\mathcal{O}(1)$.

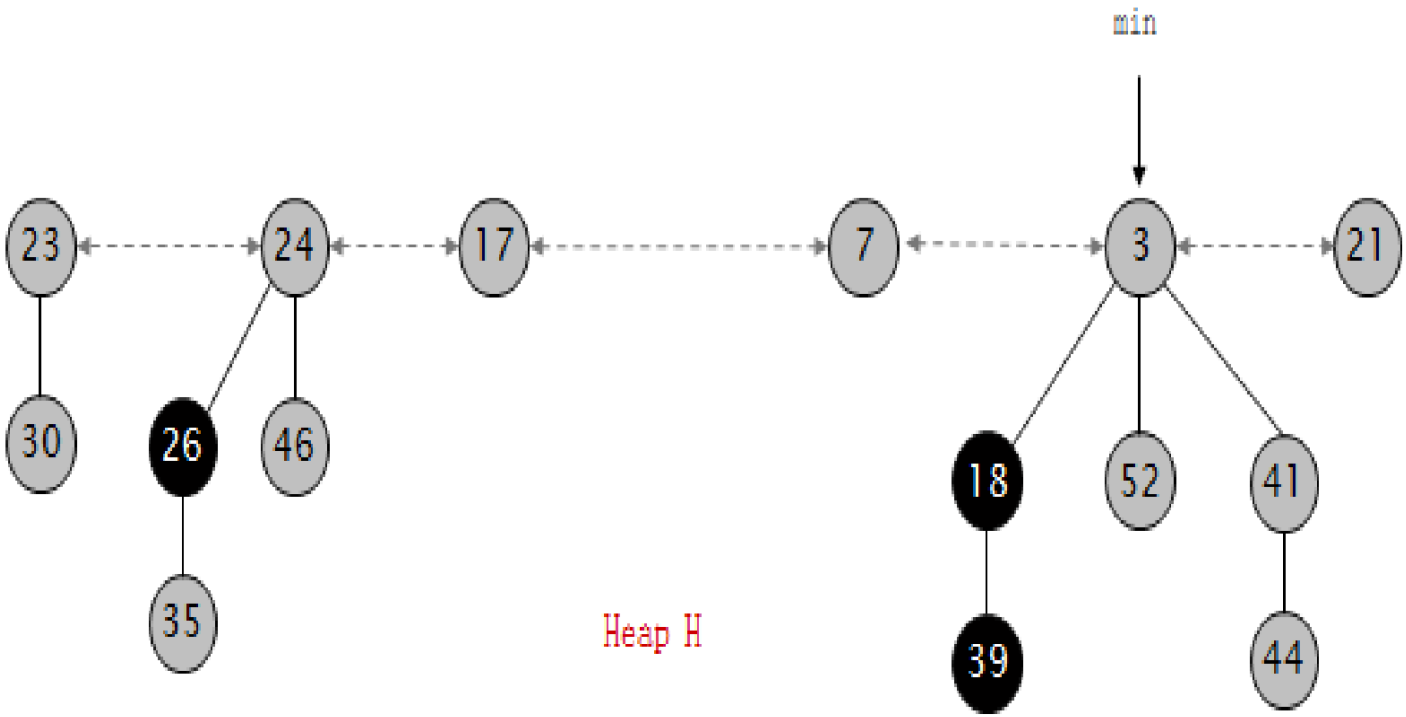
Potential: $\Phi(H_1 \cup H_2) = \Phi(H_1) + \Phi(H_2)$.

Example: meld





Example: meld



Operations: insert

- Special case of union (the same as for binomial heaps):

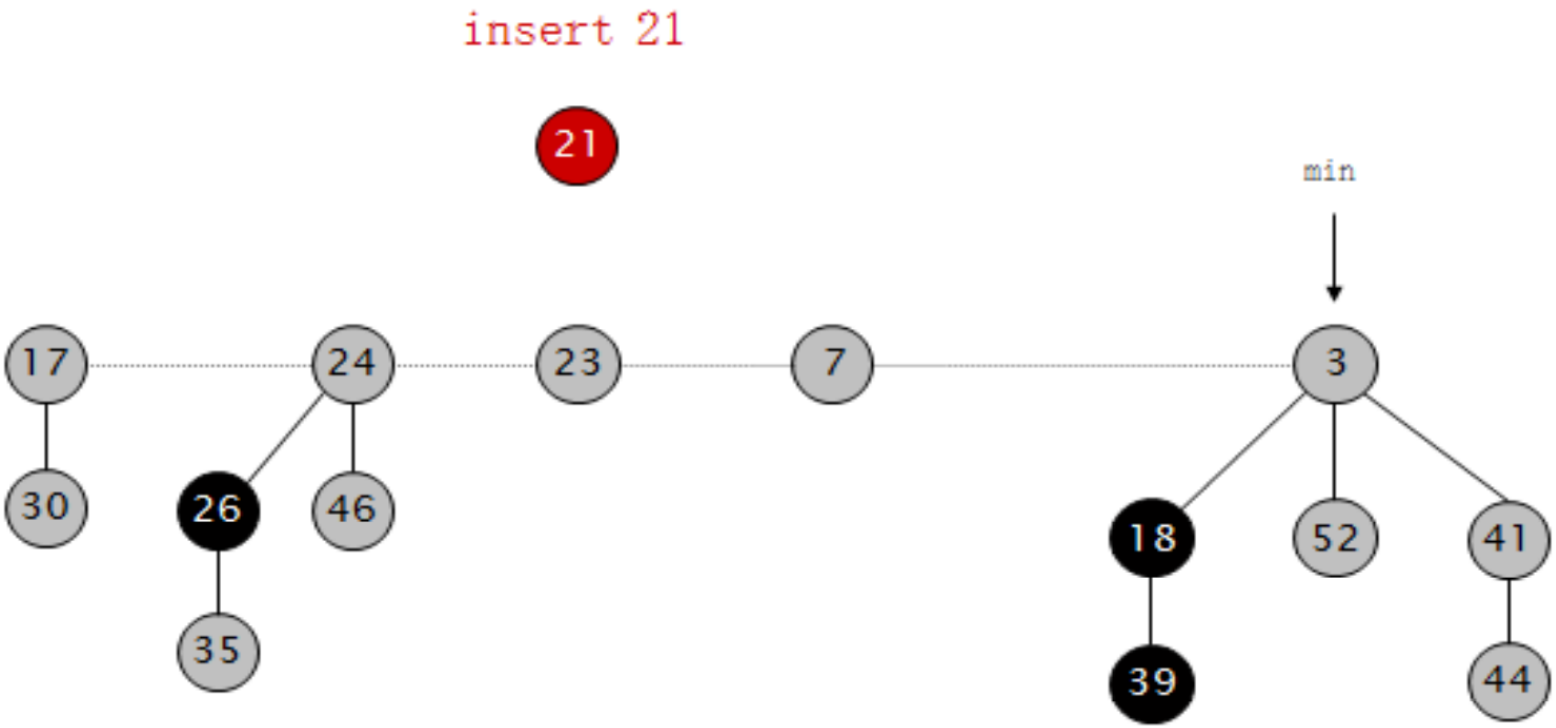
```
void insert(FiboHeap& H, int p) {
   FiboHeap N = new node;
   N->value = p;
   N->degree = 0; N->mark = 0;
   N->father = N->child = nullptr;
   N->next = N->previous = N;

   if(H != nullptr) meld(H,N);
   else H = N;
}
```

Complexity: $\mathcal{O}(1)$.

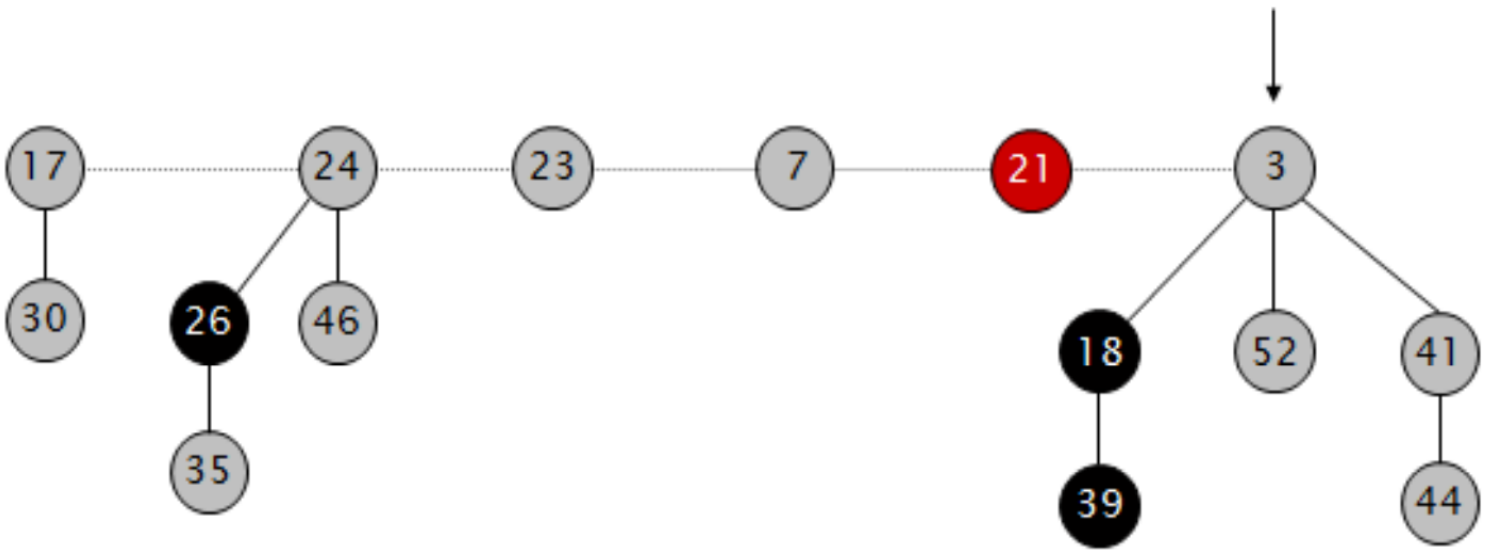
Potential: $+1$.

Amortized complexity: $\mathcal{O}(1)$

Example: insert



Example: insert



A naive version of Min. deletion

- Concatenation of the remaining trees with the children list of the node.

```
void naive_delete_min(FiboHeap& H) {
   node *root = H; //for future deletion
   if(H == H-> next) H = H->child; //case of a single tree
   else if(H->child == nullptr) {
      H->previous->next = H->next;
      H->next->previous = H->previous;
      H=H->next;
   } else {
      H->previous->next = H->child;
      H->next->previous = H->child->previous;
      H->child->previous = H->previous;
      H->next->previous->next = H->next;
      H = H->child;
   }
   locate_min(H); //see next slide
   delete root;
}
```

Location of the minimum element

```
void locate_min(FiboHeap& H) {
   if(H != nullptr) {
      node *MinNode = H;
      for(node *N = H->next; N != H; N = N->next){
         N->father = nullptr;
         if(N->value < MinNode->value) { MinNode = N; }
      }
      H = MinNode;
   }
}
```

Complexity: Linear in tree(H) + root.degree

Potential: + root.degree - 1

$\Longrightarrow$ No compensation!

The Fibonacci property

Definition

The structure must guaranteed that any subtree whose root has degree d must contain $\geq F(d)$ nodes.

$\rightarrow$ Trivially true after insertion because $F(0) = 1$.

Consequence: any node has degree $\mathcal{O}(\log n)$.

Next objective: reorganize the structure after min. deletion so that:

- The Fibonacci property still holds true;
- $tree(H)$ is decreased to $\mathcal{O}(\log n)$.

$\Longrightarrow$ Min. deletion in amortized $\mathcal{O}(\log n)$.

Consolidation

- We repeatedly merge any two trees with the same degree at the root, until all roots have pairwise different degrees.
  - Since $2F(d) \geq F(d+1)$, the Fibonacci property remains valid. In particular, the final number of trees must be in $\mathcal{O}(\log n)$.

Complexity: Linear in tree(H)

Potential: $+ \mathcal{O}(\log n)$ - tree(H)

Amortized complexity: $\mathcal{O}(\log n)$

Implementation

Auxiliary functions

```
int max_degree(FiboHeap& H) {
    int rank = H->degree;
    for(node *N = H->next; N != H; N = N->next)
        if(N->degree > rank) { rank = N->degree; }
    return rank;
}

void merge(FiboHeap& N1, FiboHeap& N2) {
    if(N1->value > N2->value) { merge(N2,N1); N1 = N2; }
    else{

        N1->degree++;

        N2->previous->next = N2->next;
        N2->next->previous = N2->previous;

        N2->father = N1;
        if(N1->child != nullptr) {
            N2->previous = N1->child->previous; N2->next = N1->child;
            N1->child->previous = N2;
            N2->previous->next = N2;
        } else { N2->previous = N2->next = N2; }
        N1->child = N2;
    }
```

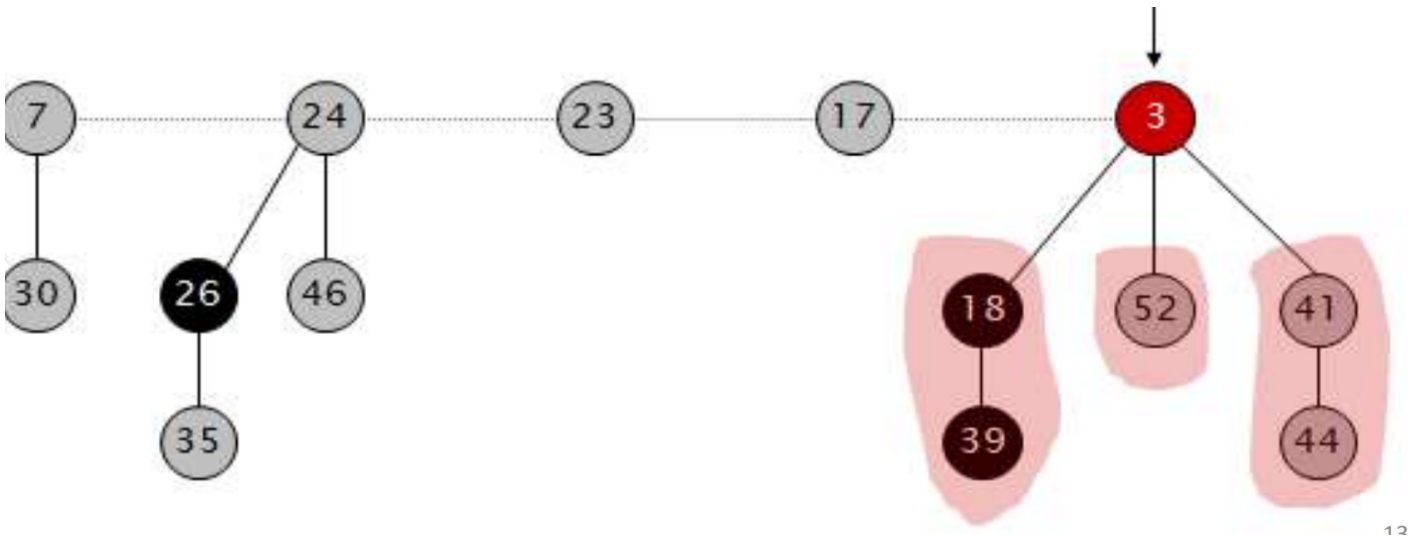
Implementation

Main loop

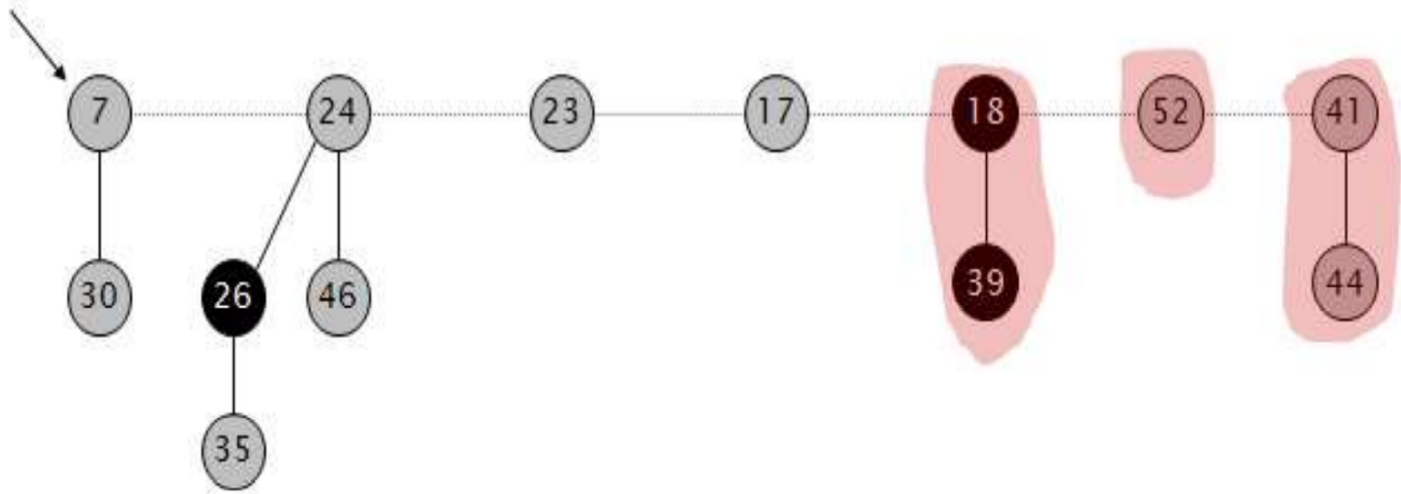
```
void consolidate(FiboHeap& H) {

    vector<int> rank(max_degree(H),nullptr);
    rank[H->degree] = H;

    for(node *N = H->next; N != H; N= N->next) {

        while(N->degree < rank.size() && rank[N->degree] != nullptr) {
            merge(N,rank[N->degree]); rank[N->degree -1] = nullptr;
        }

        if(N->degree == rank.size()) rank.push_back(N);
        else rank[N->degree] = N;
    }
}
```

Complexity: Linear in $tree(H)$ (since any while loop removes a tree)

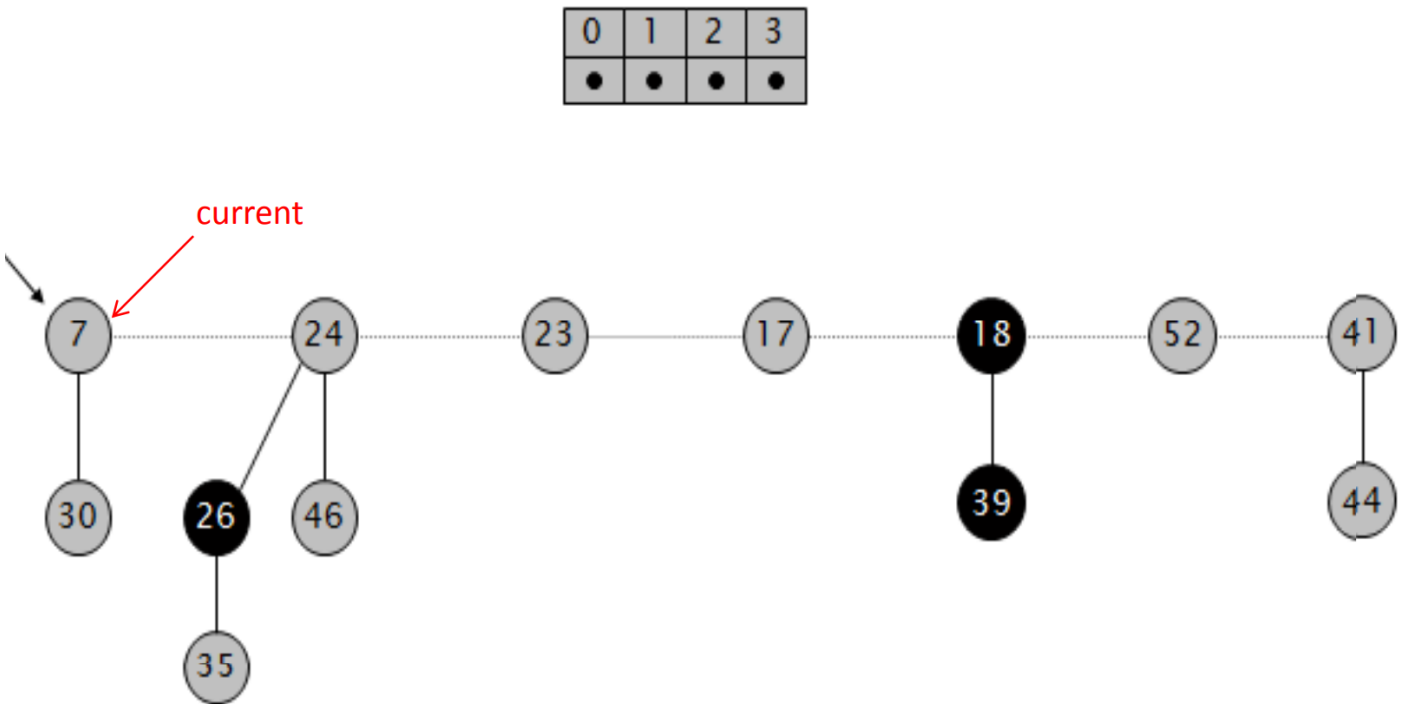
Example: Min. deletion



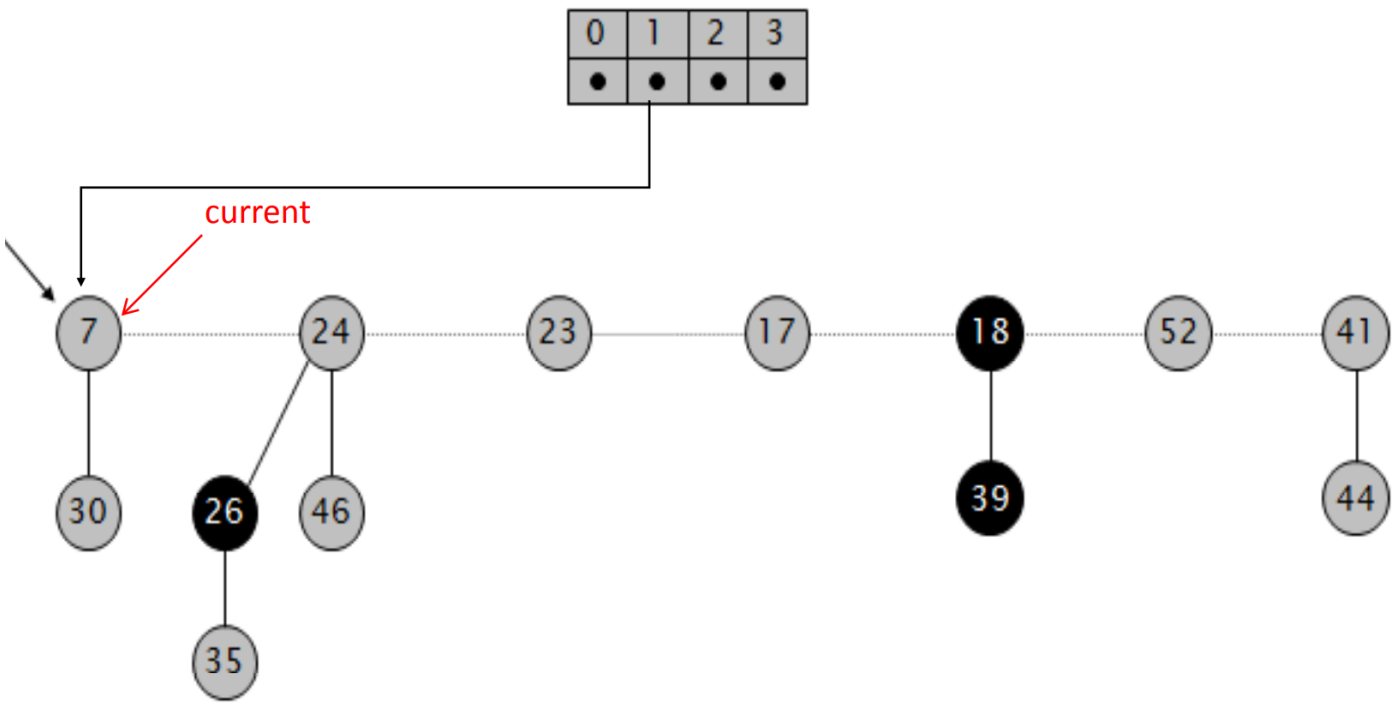
Example: Min. deletion



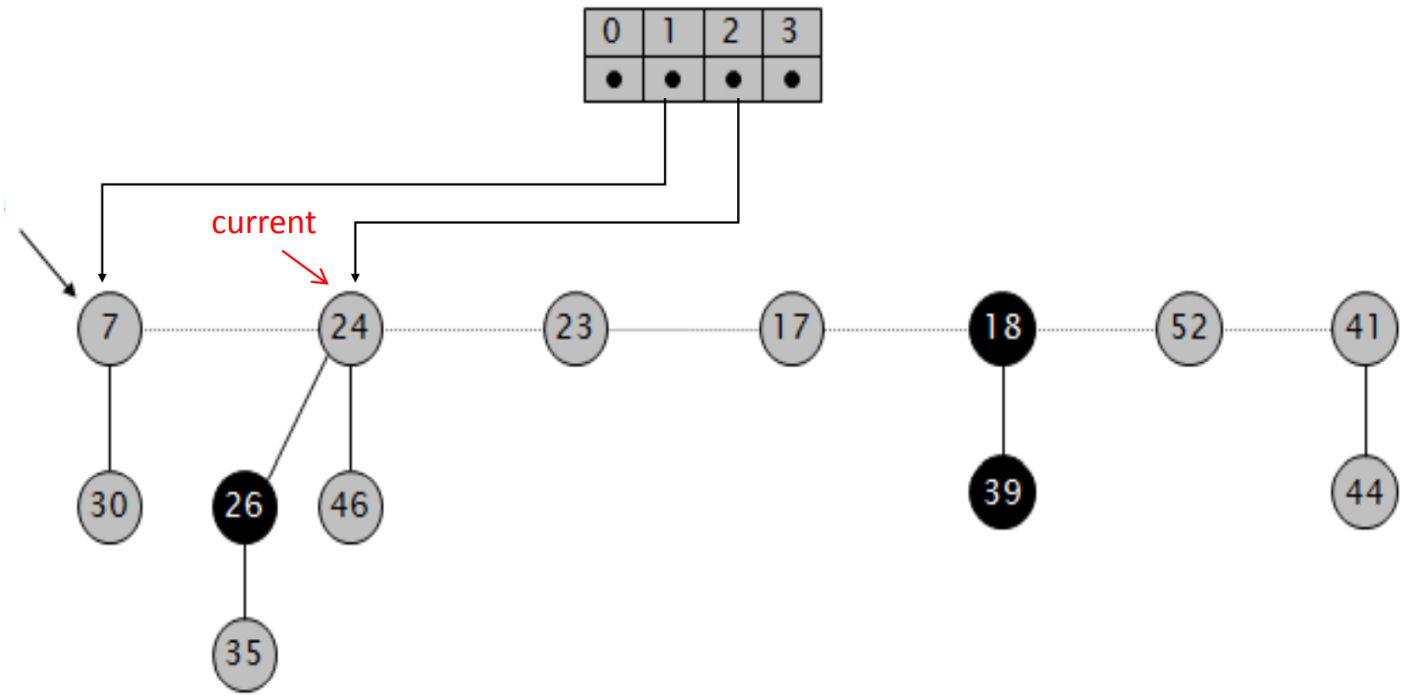
Example: Min. deletion



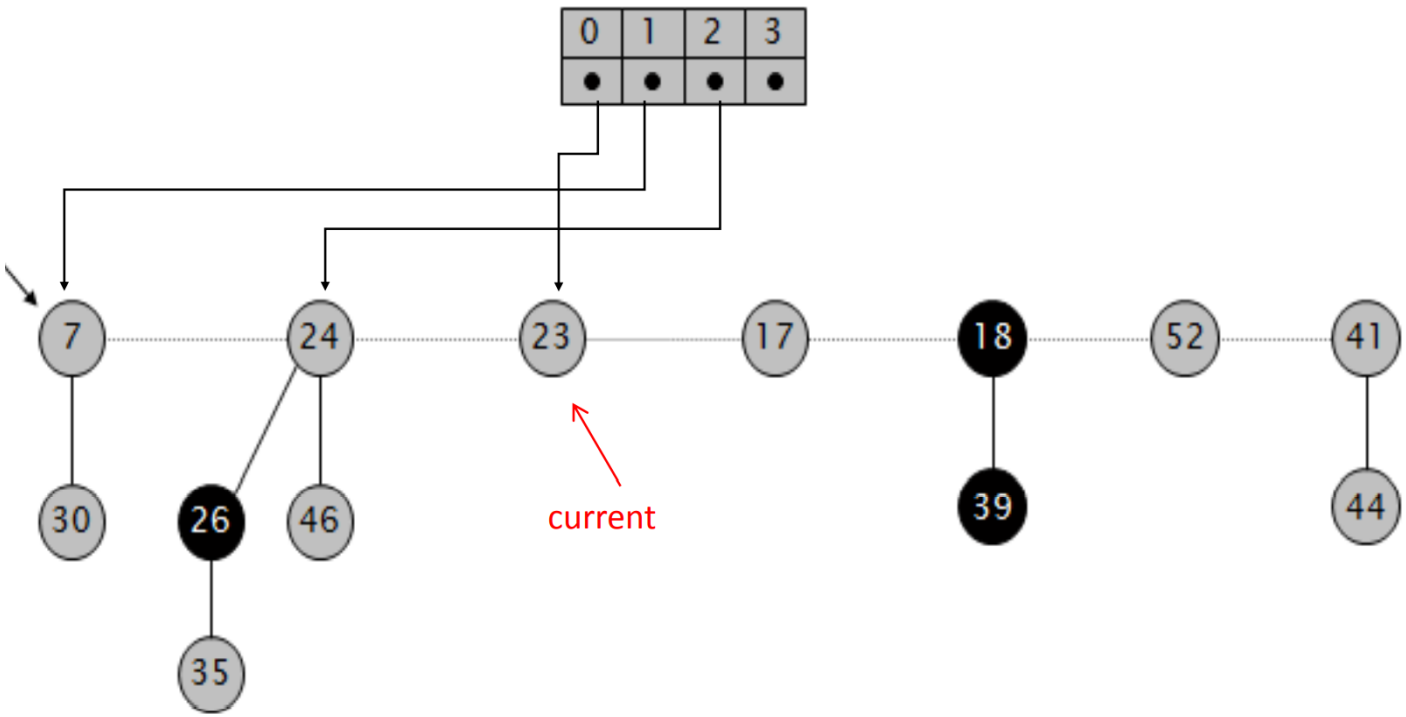
Example: Min. deletion



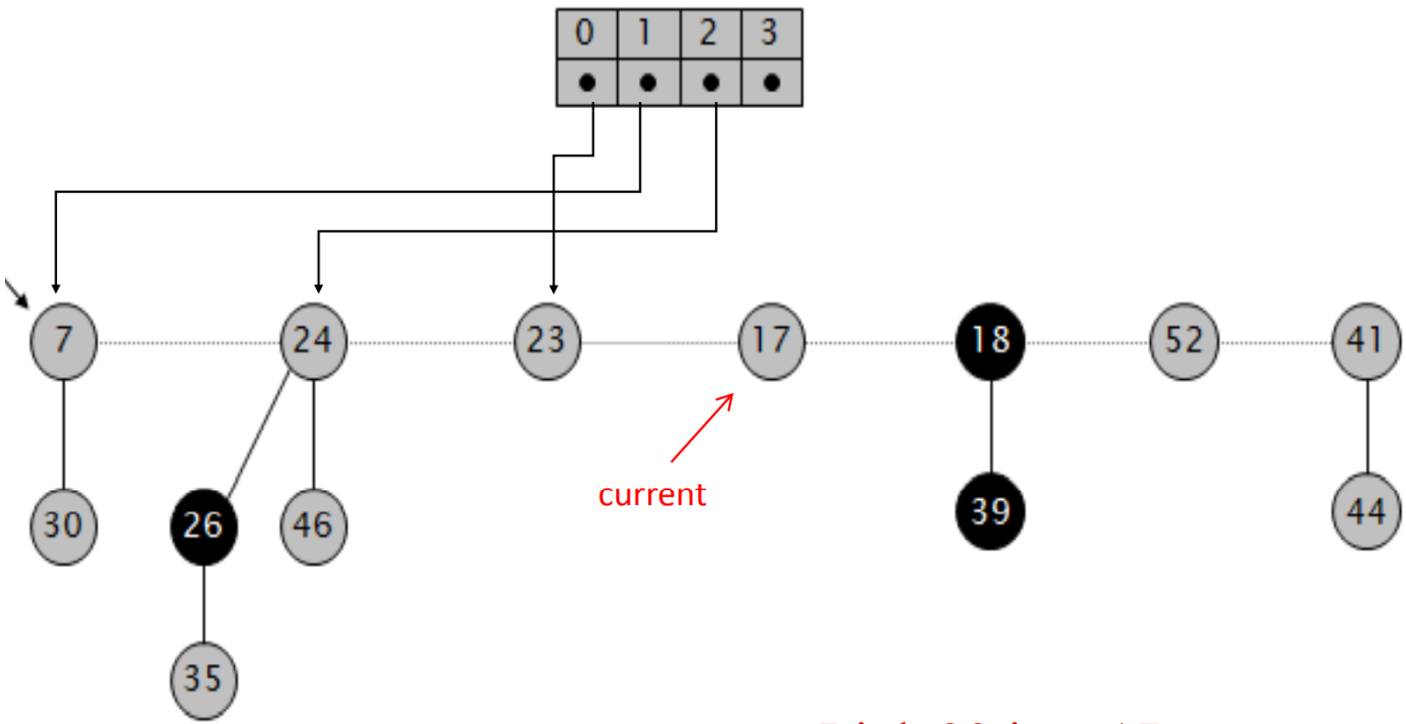
Example: Min. deletion



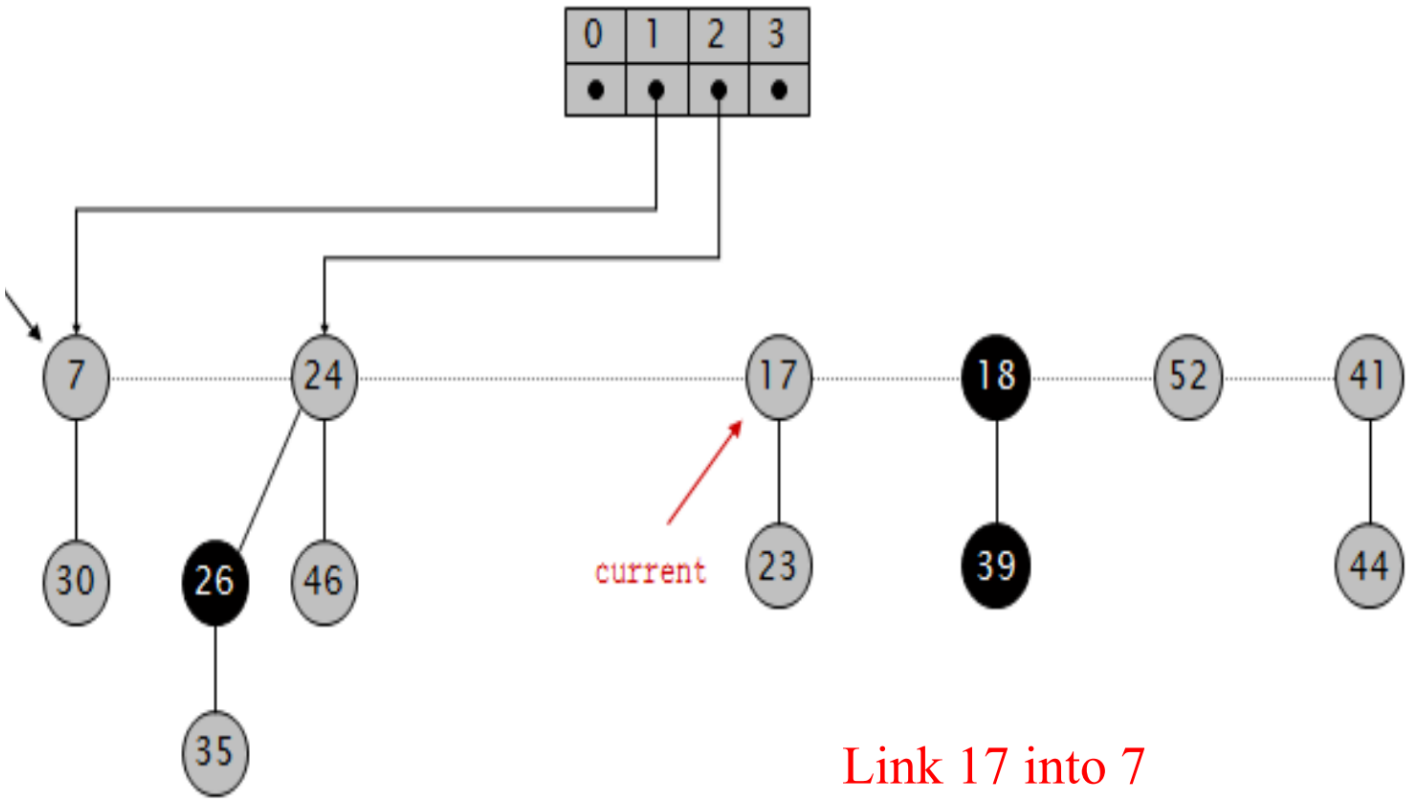
Example: Min. deletion



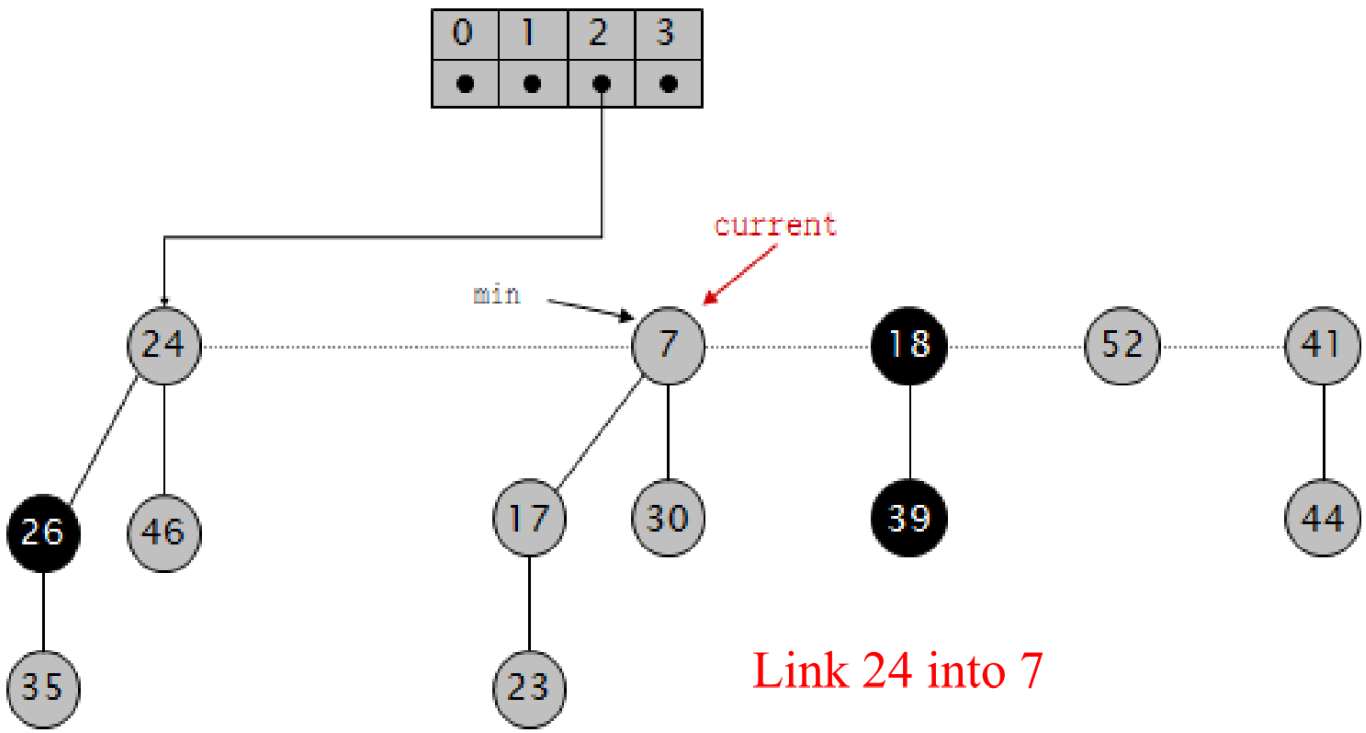
Example: Min. deletion



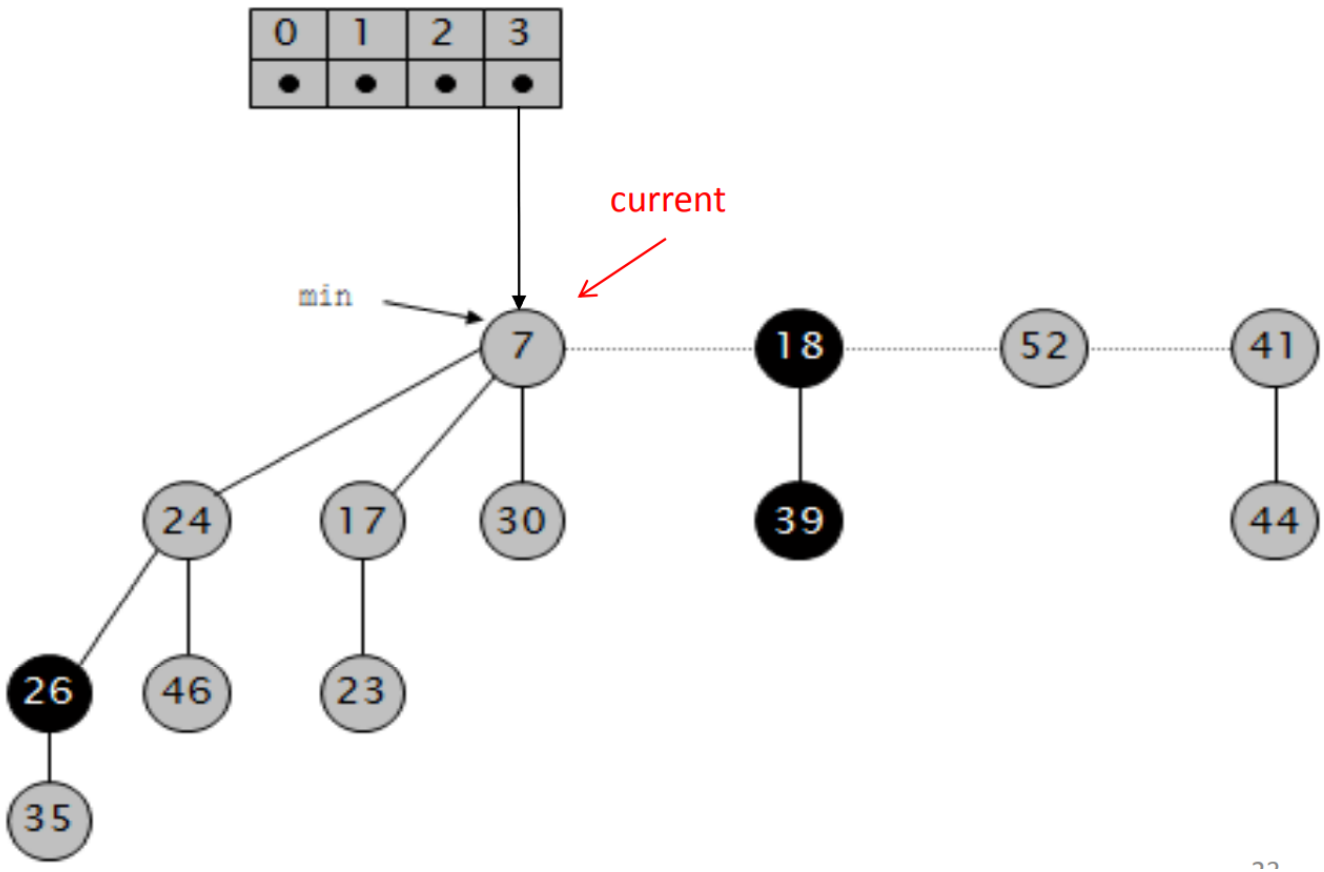
Example: Min. deletion



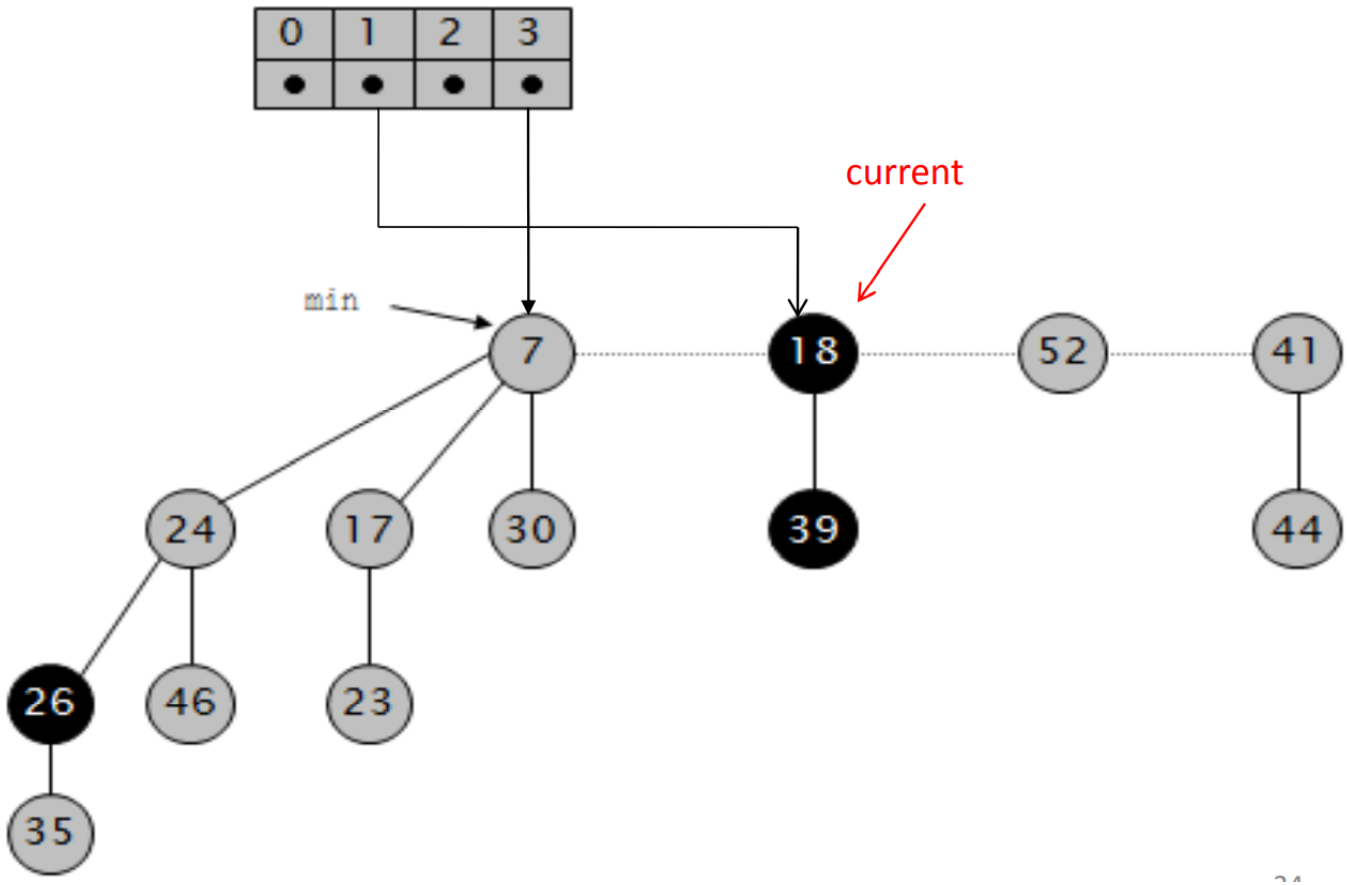
Example: Min. deletion



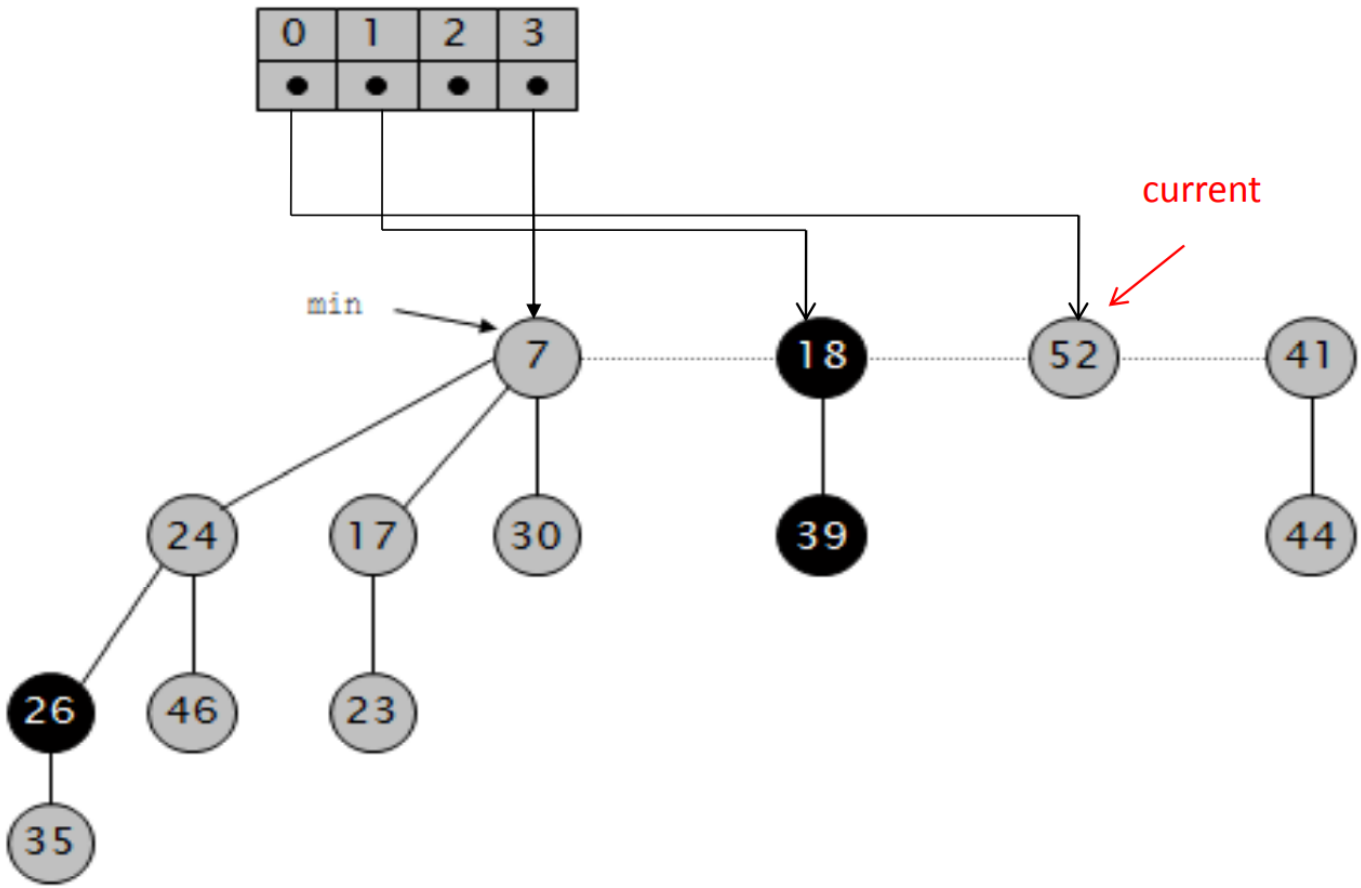
Example: Min. deletion



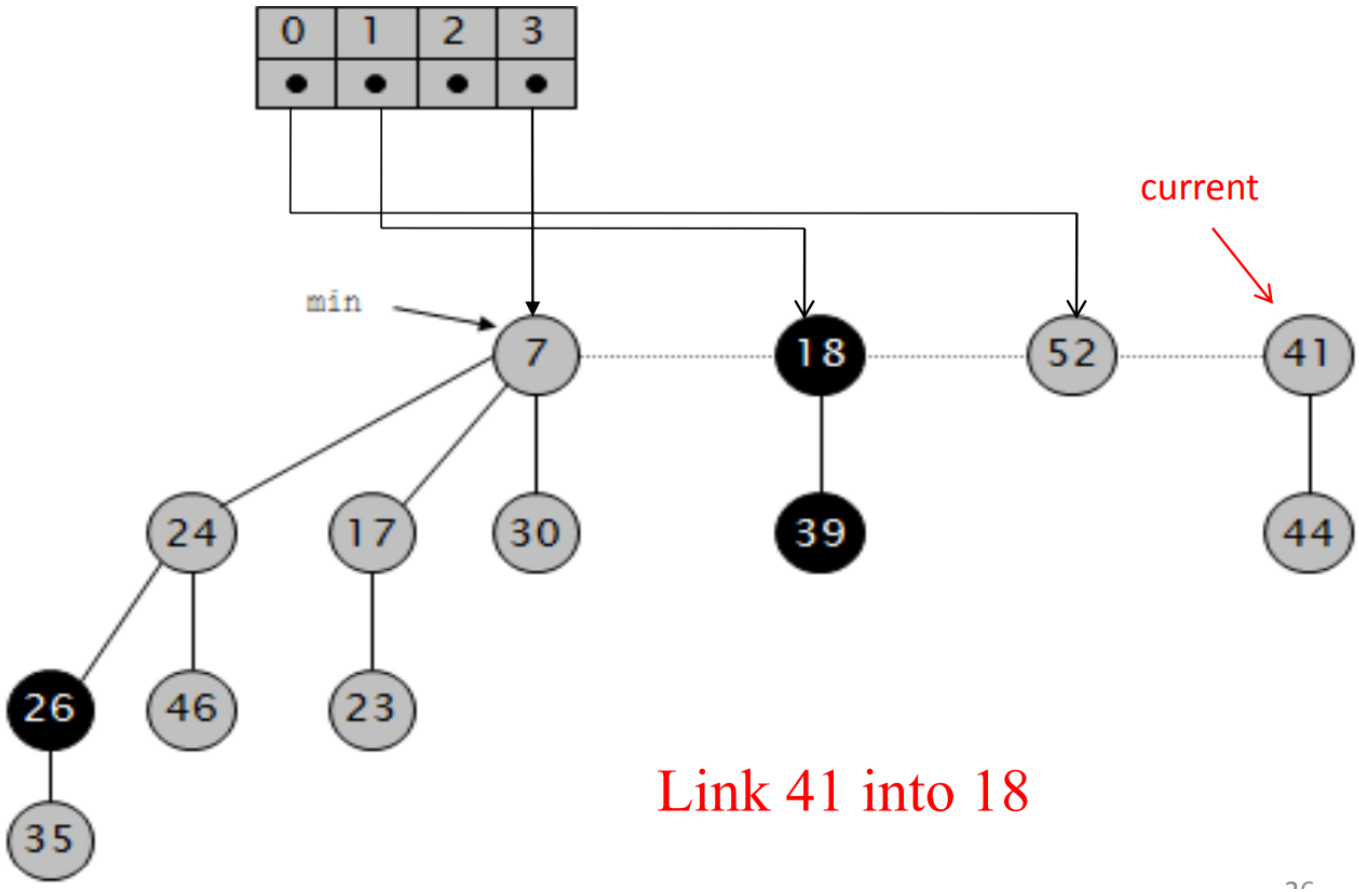
Example: Min. deletion



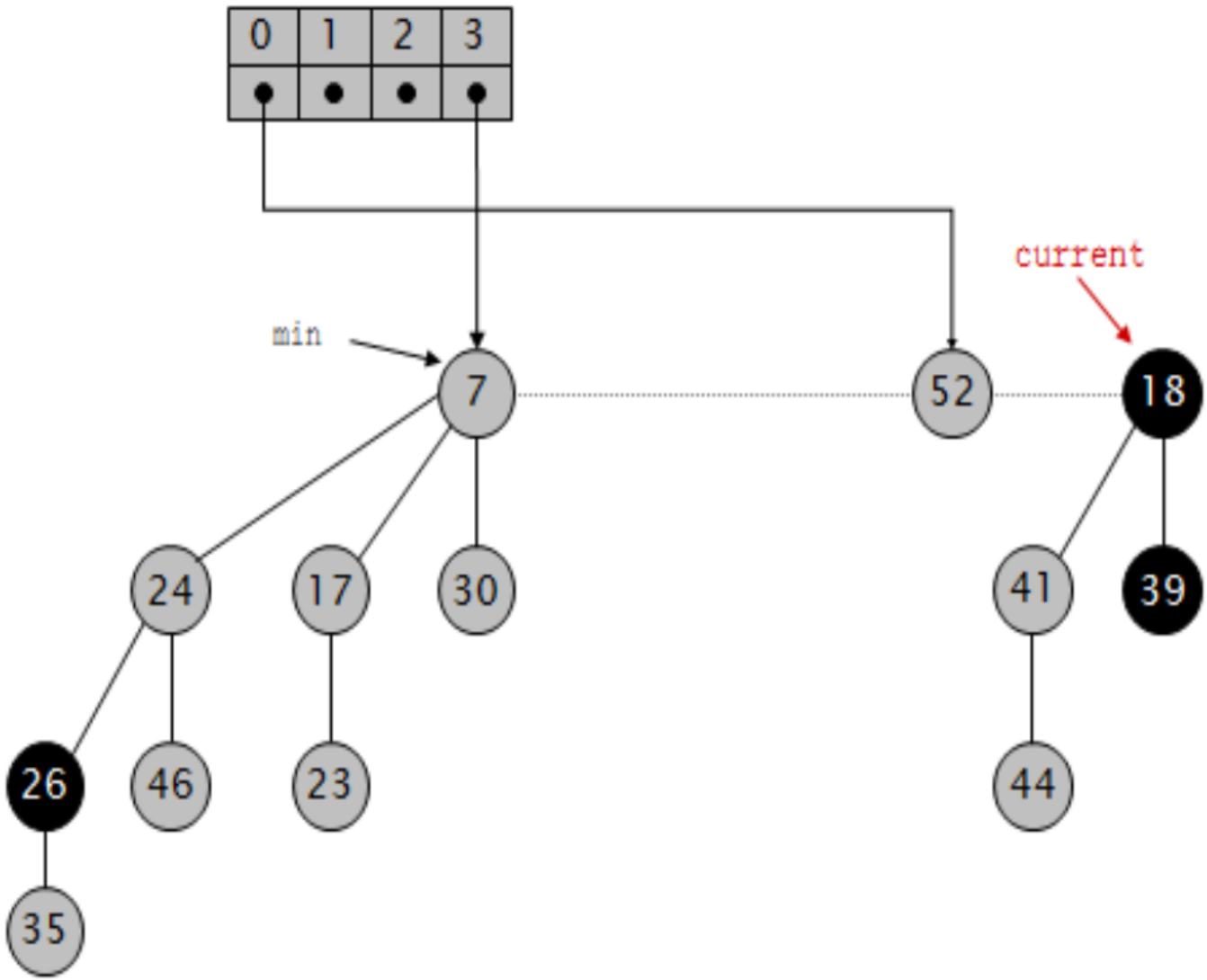
Example: Min. deletion



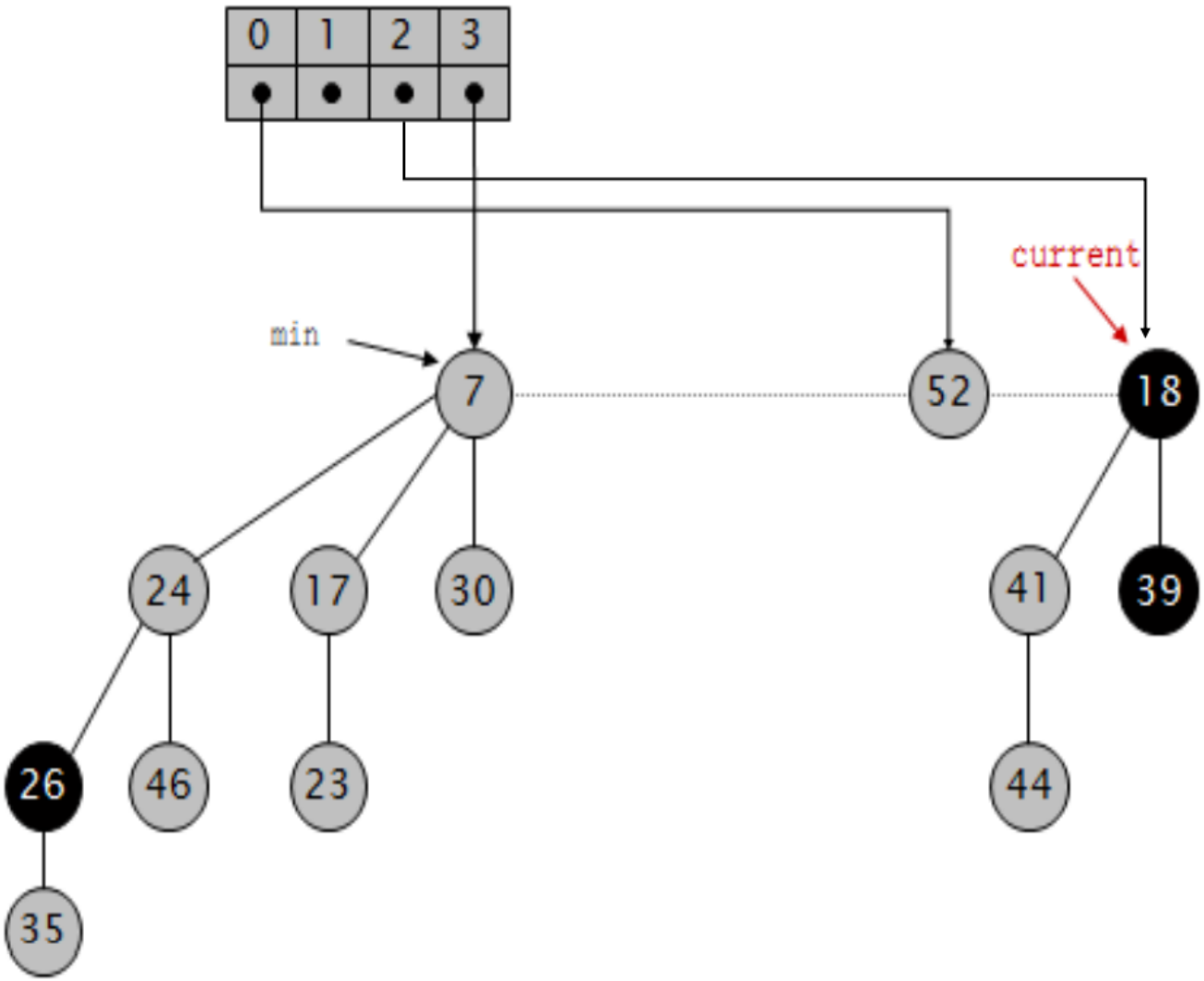
Example: Min. deletion



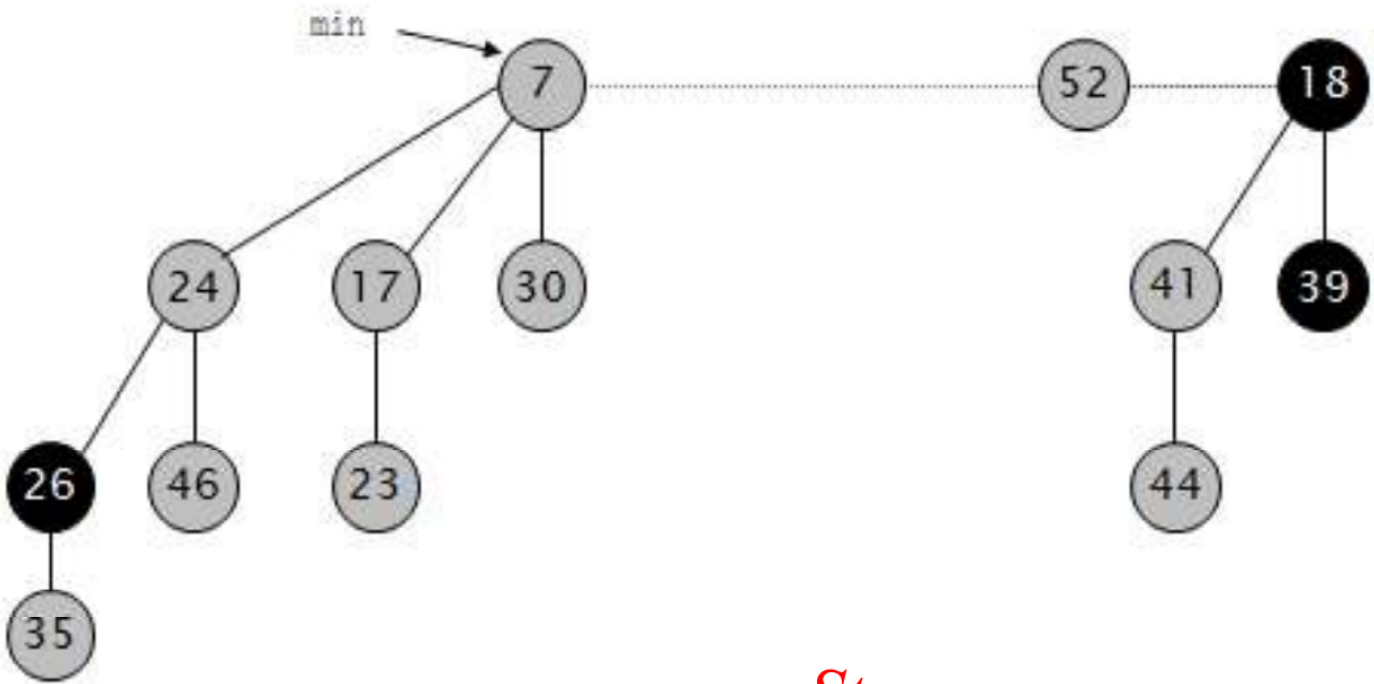
Example: Min. deletion



Example: Min. deletion



Example: Min. deletion



Stop

Decrease key

1) After decreasing the value of some node, we compare this node value with the one in its father.

2) If the Min-Heap property still holds, then there is nothing else to do.

3) Otherwise, we disconnect the node from its father and make it a root.

- If the father was unmarked, then we mark it (**unless** it is the root).
- Otherwise, we also disconnect the father to make it a root. If the grandfather was unmarked then... $\Longrightarrow$ **Disconnections in cascade!**

Complexity: Linear in t, the number of consecutive marked ancestors of the node.

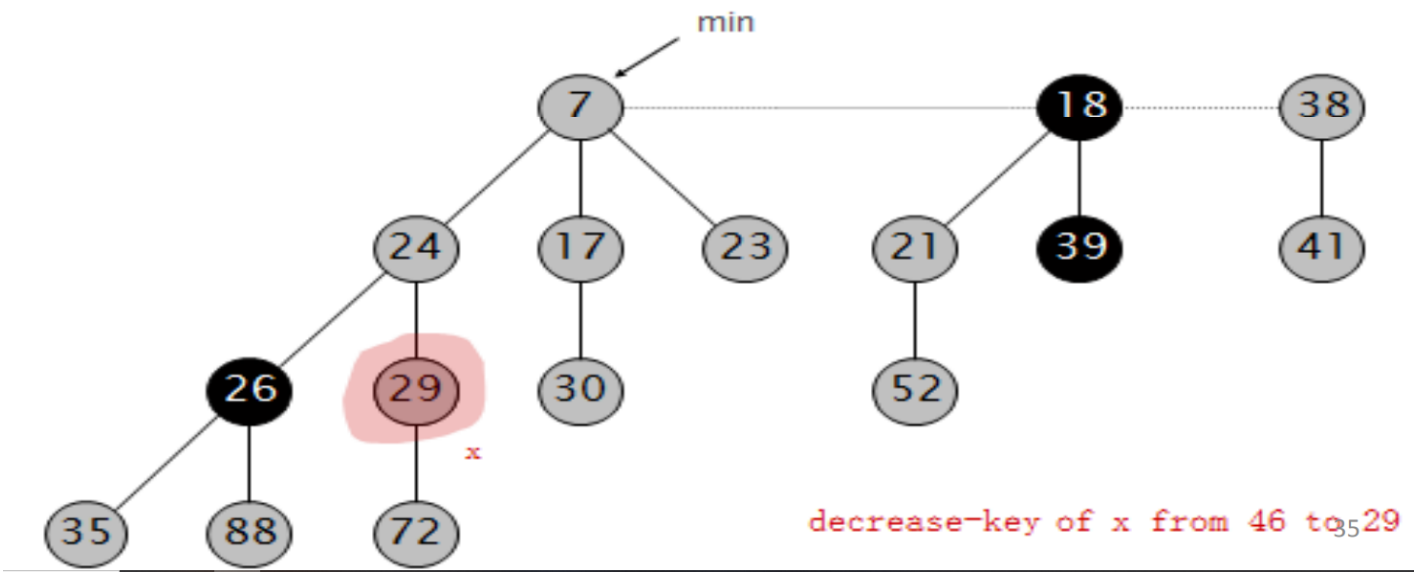
Potential: $-t + \mathcal{O}(1)$

Amortized complexity: $\mathcal{O}(1)$

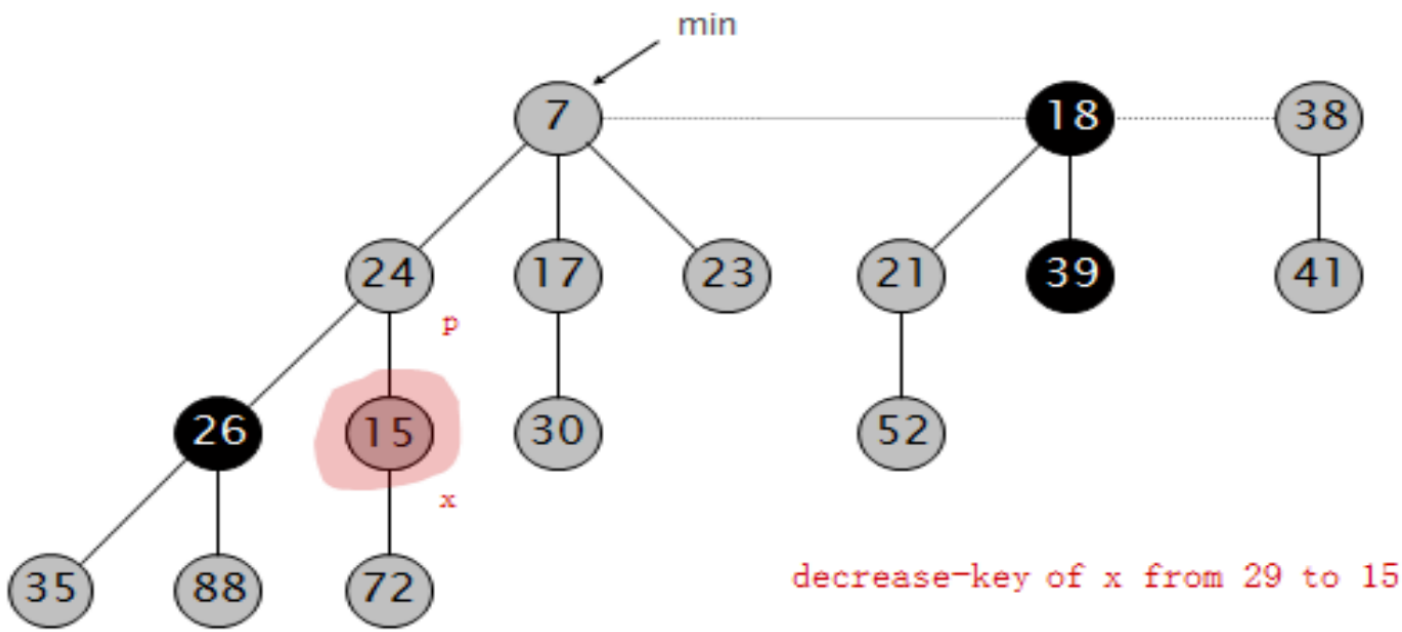
Implementation

```
void decrease_key(FiboHeap& H, node *n, int d) {
   n->value -= d;
   if(n->father != nullptr && n->father->value > n->value)
      cascade_cut(H,n);
}
void cascade_cut(FiboHeap& H, node *n) {
   node *f = n->father; n->father = nullptr;
   if(n->next == n) f->child = nullptr;
   else { //deletion from the children list
      n->previous->next = n->next;
      n->next->previous = n->previous;
      if(n == f->child) f->child = n->next;
   }
   n->next = n->previous = n;
   n->mark = 0; meld(H,n); //also updates the min. element
   if(f->father != nullptr){
      if(!f->marked) f->mark = 1;
      else cascade_cut(H,f);
}}
```

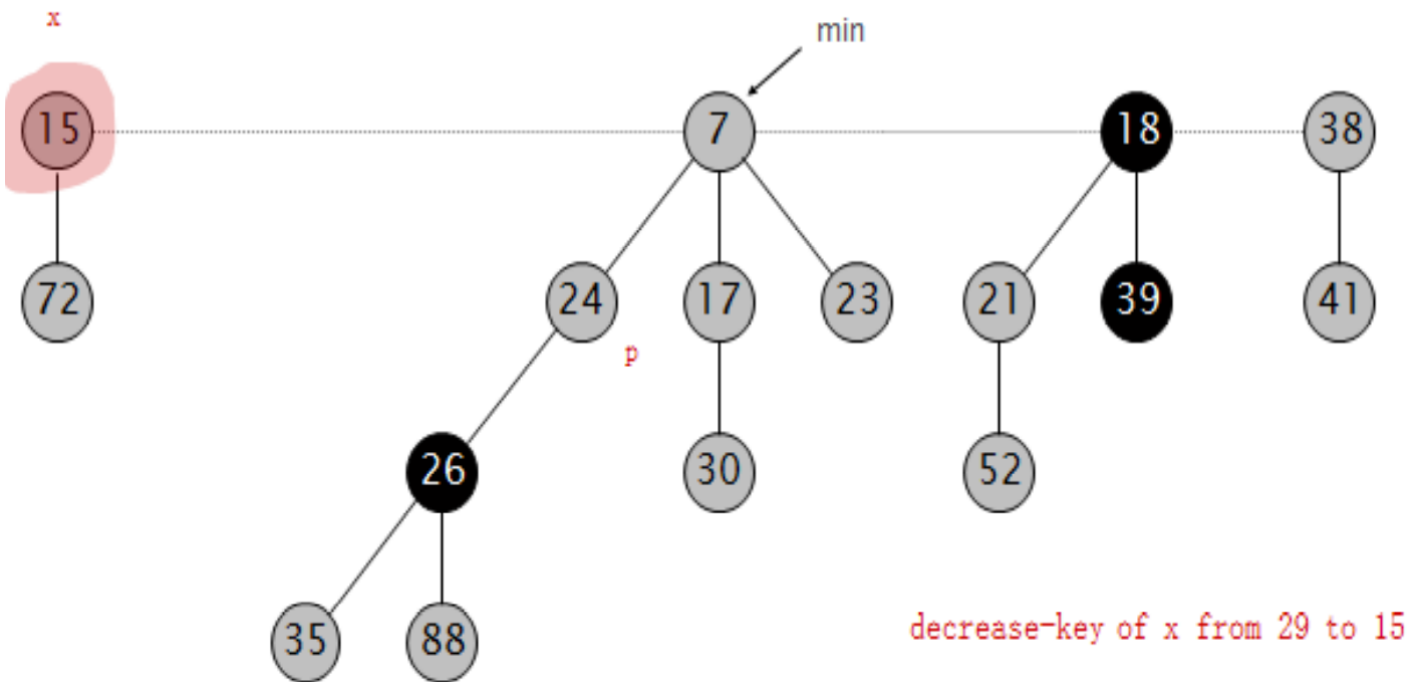
Example: decrease key



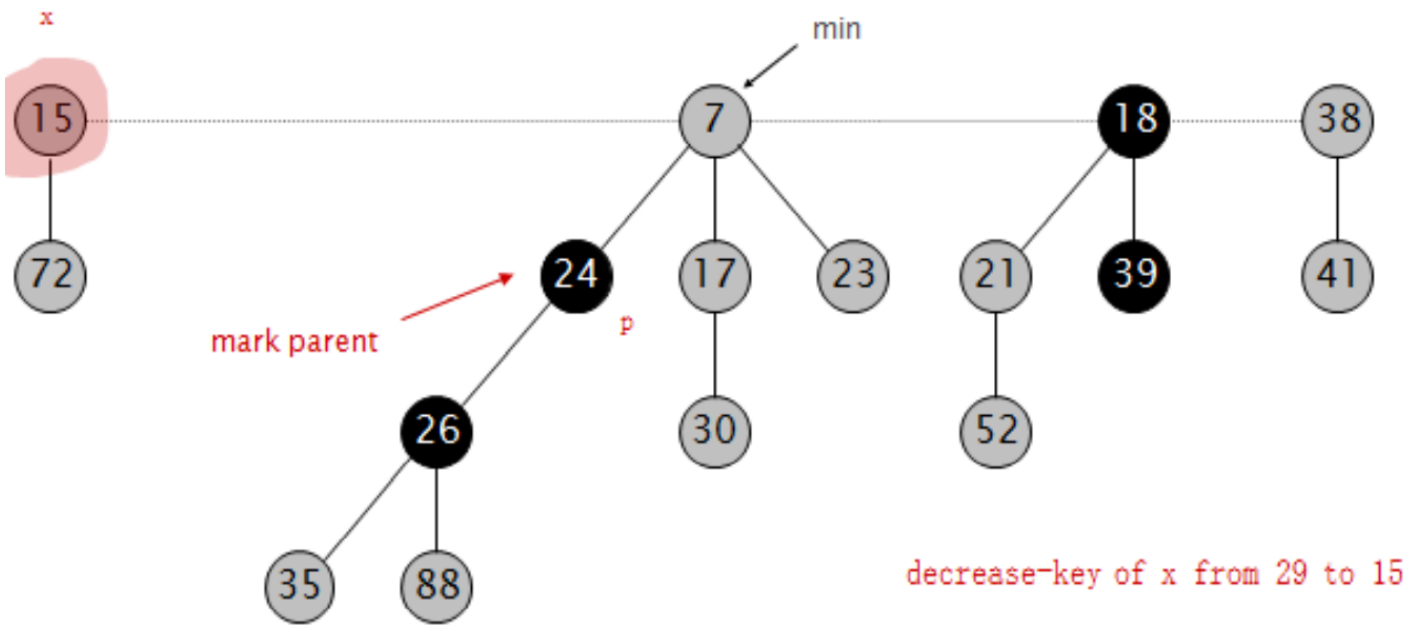
Example: decrease key



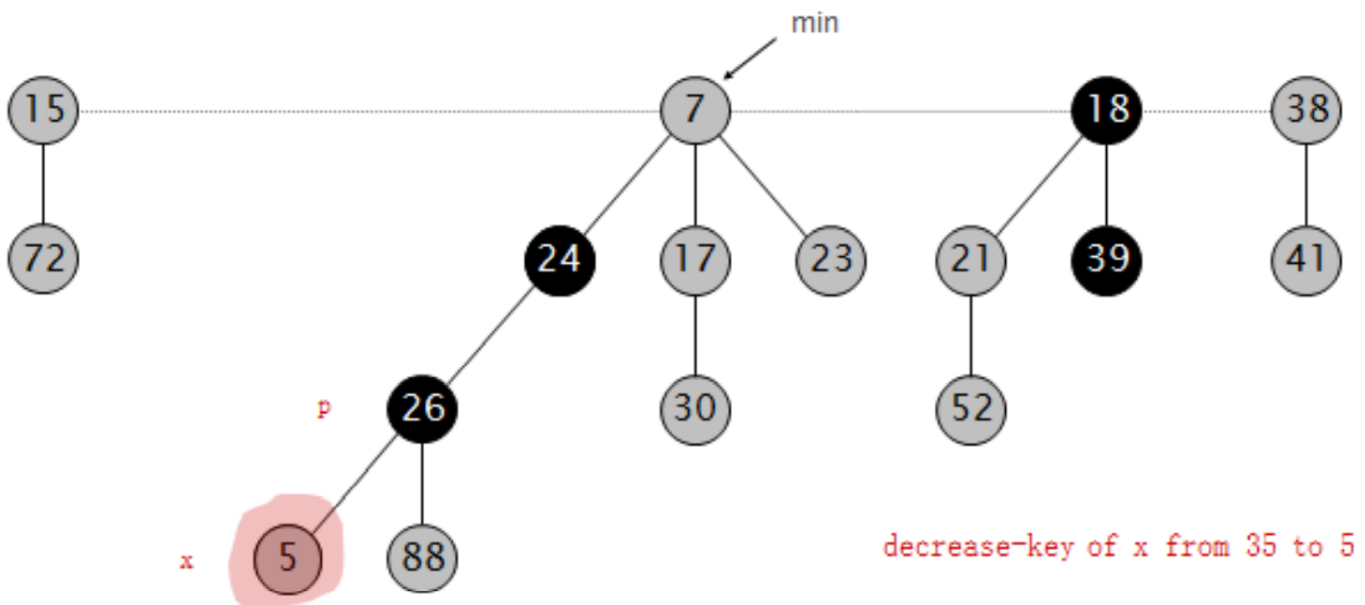
Example: decrease key



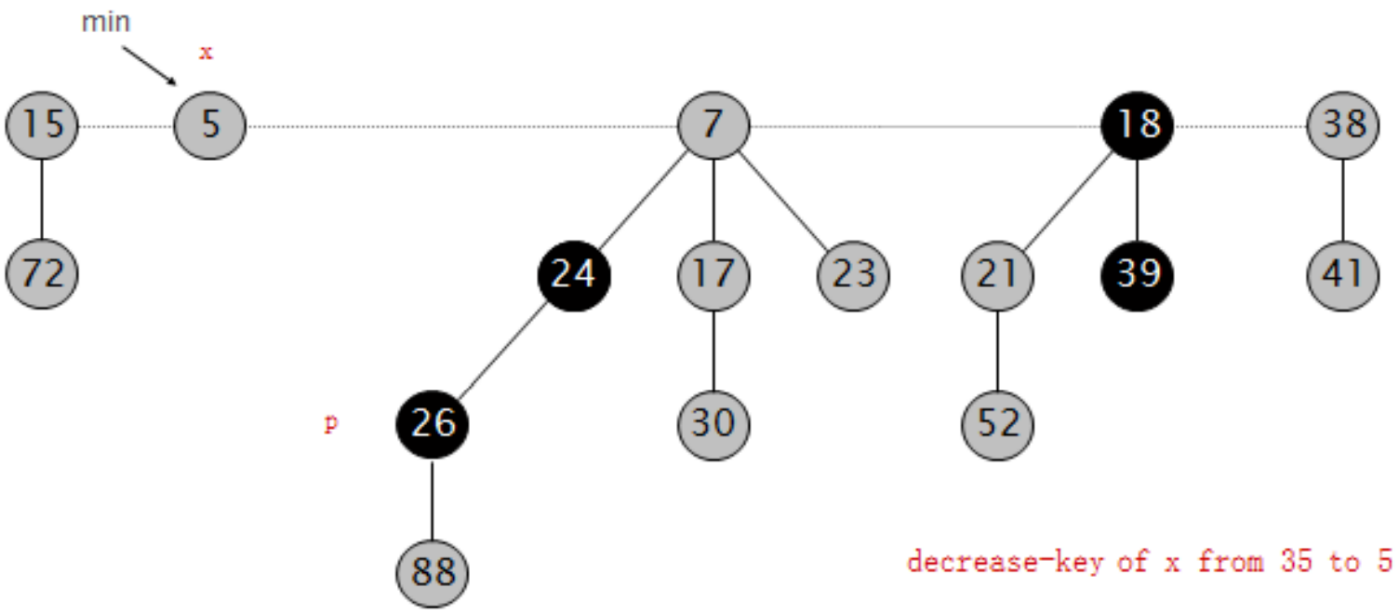
Example: decrease key



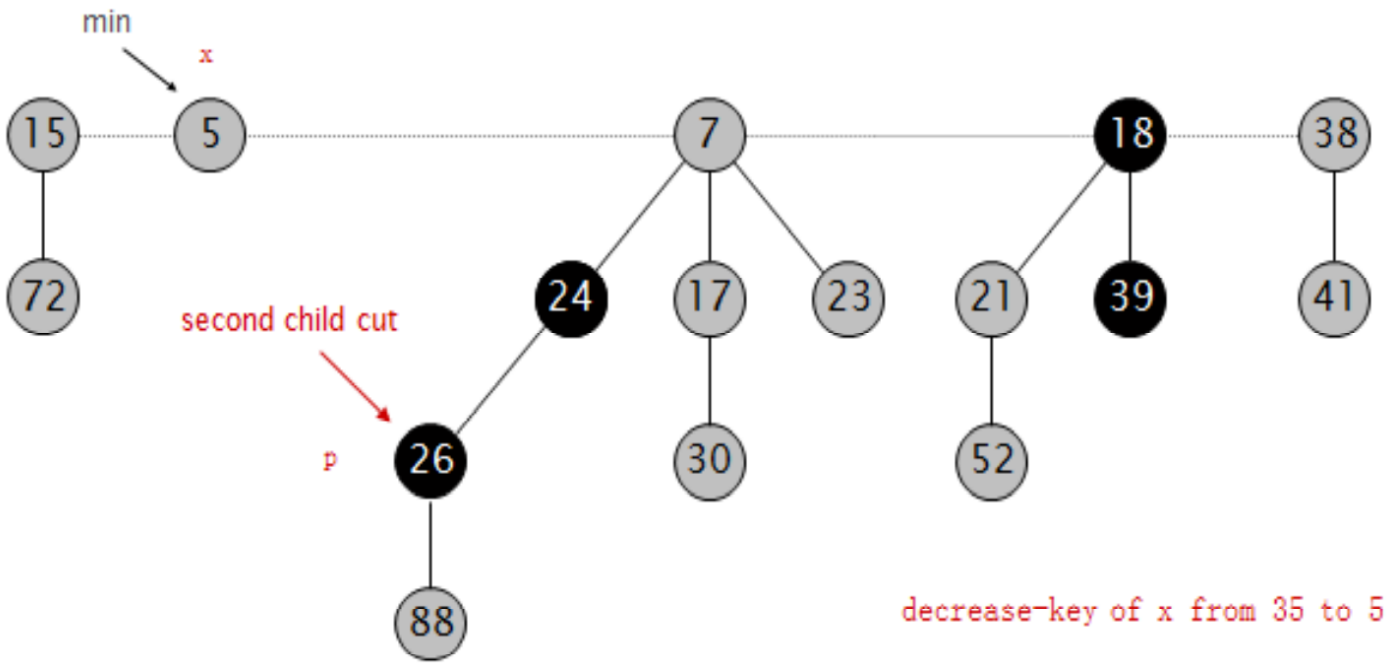
Example: decrease key



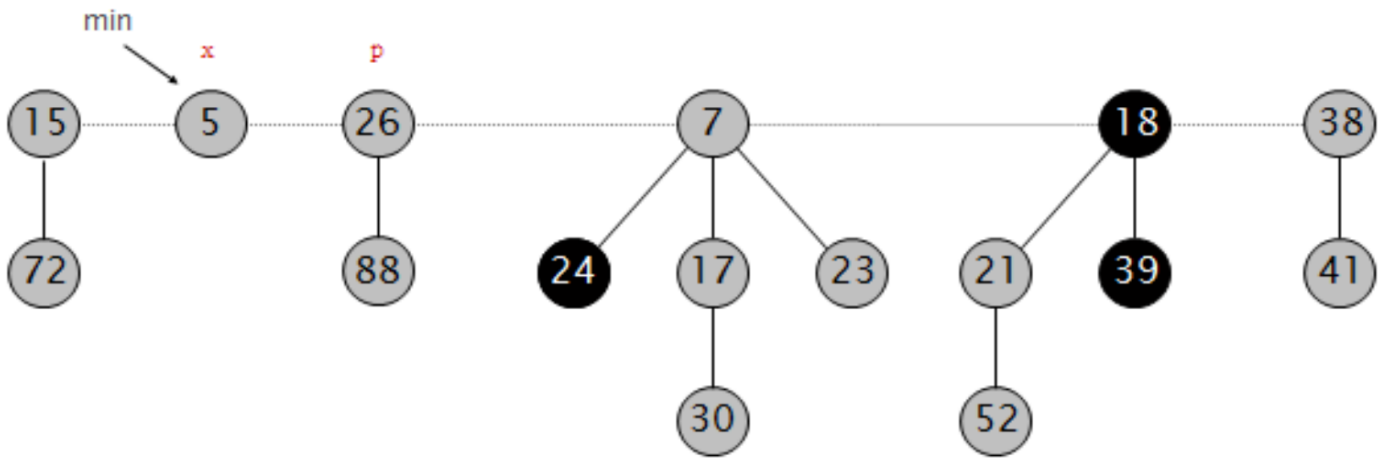
Example: decrease key



Example: decrease key

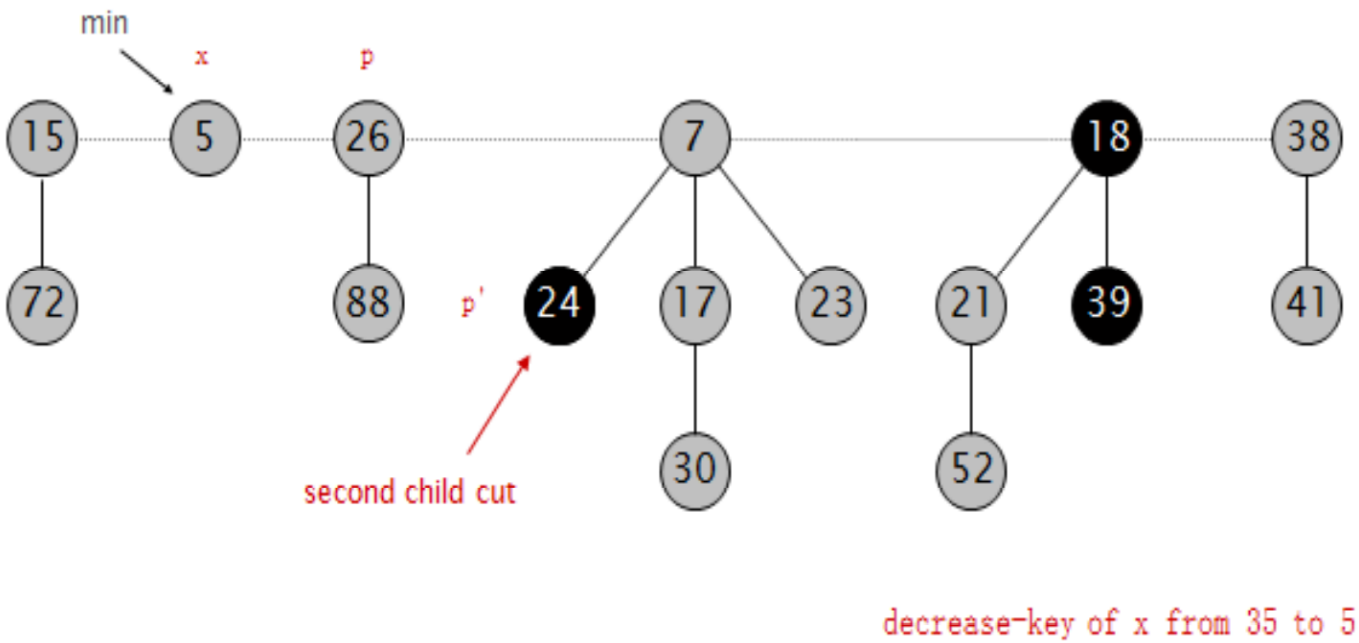


Example: decrease key

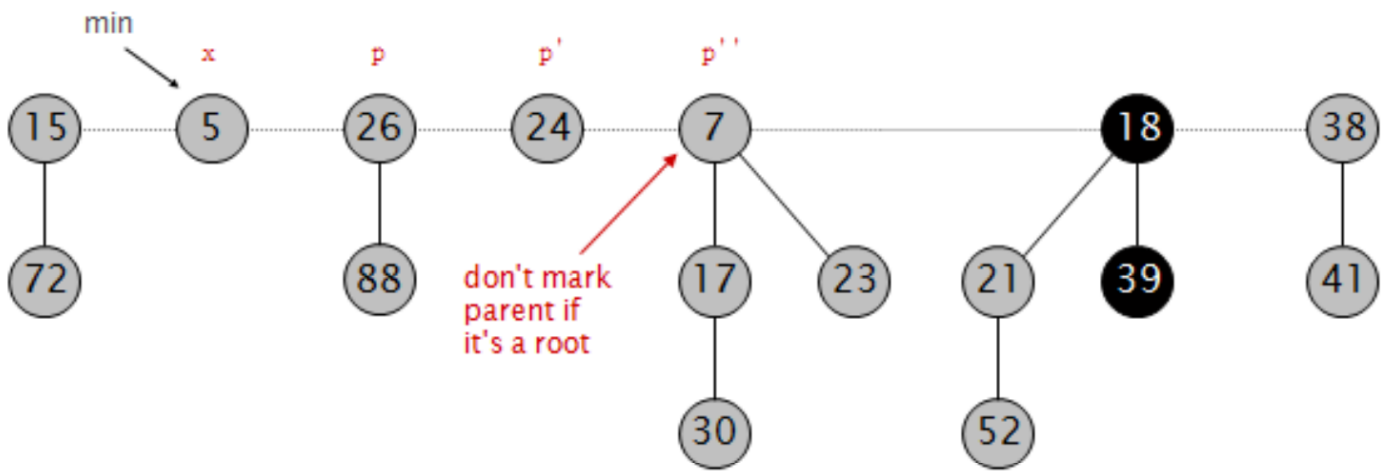


decrease-key of x from 35 to 5

Example: decrease key



Example: decrease key



decrease-key of x from 35 to 5

Discussion: Proving the Fibonacci property

- This is made complicated by the disconnection of marked nodes!
- Consider all children $y_1, y_2, \ldots, y_d$ of a node x, in the order they were merged with x.

  - At the time we merged y_i with x, both nodes had the same degree (see delete min. operation). In particular, the degree of y_i was at least $i-1$.
  - Since then, node y_i lost at most one child (otherwise, it should have been cut). In particular, node y_i has degree at least $i-2$.
  - By induction on the height y_i has at least $F(i-2)$ descendants. Therefore, x has at least:

$$1 + \sum_{i=1}^{d} F(i-2) = 1 + \sum_{i=0}^{d-2} F(i) = F(d)$$

    descendants!

Questions