Advanced algorithms

binary heap, *d*-ary heap, binomial heap,
amortized analysis, Fibonacci heap

Jiří Vyskočil, Radek Mařík

2013

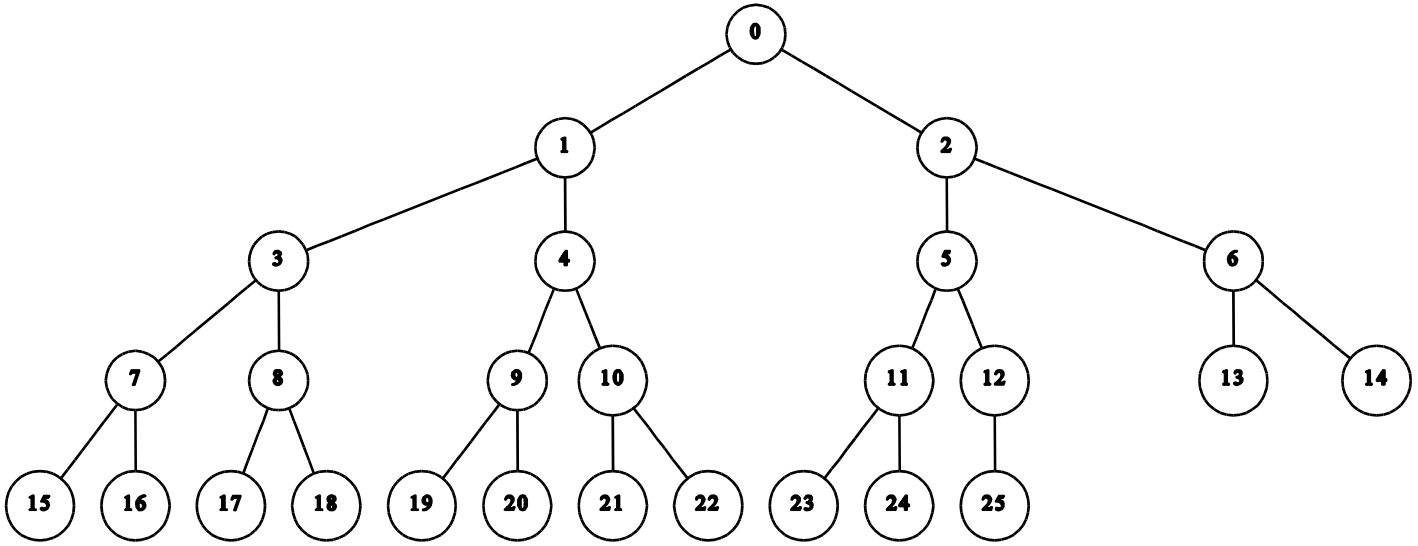
Heaps [haldy]

- heap
  - a *heap* is a specialized data structure (usually tree-based) that satisfies the **heap property**:

    If B is a child node of A, then key(B) ≥ key(A).
  - The heap is one the most efficient implementation of an abstract data type called a priority queue.
  - The operations commonly performed with a heap are:
    - **Insert** (*x*)
      - adds a new key *x* to the heap.
    - **AccessMin**
      - finds and returns the minimum item of the heap.
    - **DeleteMin**
      - removes the minimum node of the heap (usually, the minimum node is the root of a heap).
    - **DecreaseKey** (*x*,*d*)
      - decreases *x* key within the heap by *d*.
    - **Merge** (H_1, H_2)
      - joins two heaps H_1 and H_2 to form a valid new heap containing all the elements of both.
    - **Delete** (*x*)
      - removes a key *x* of a heap.

Binary Heap [binární halda]

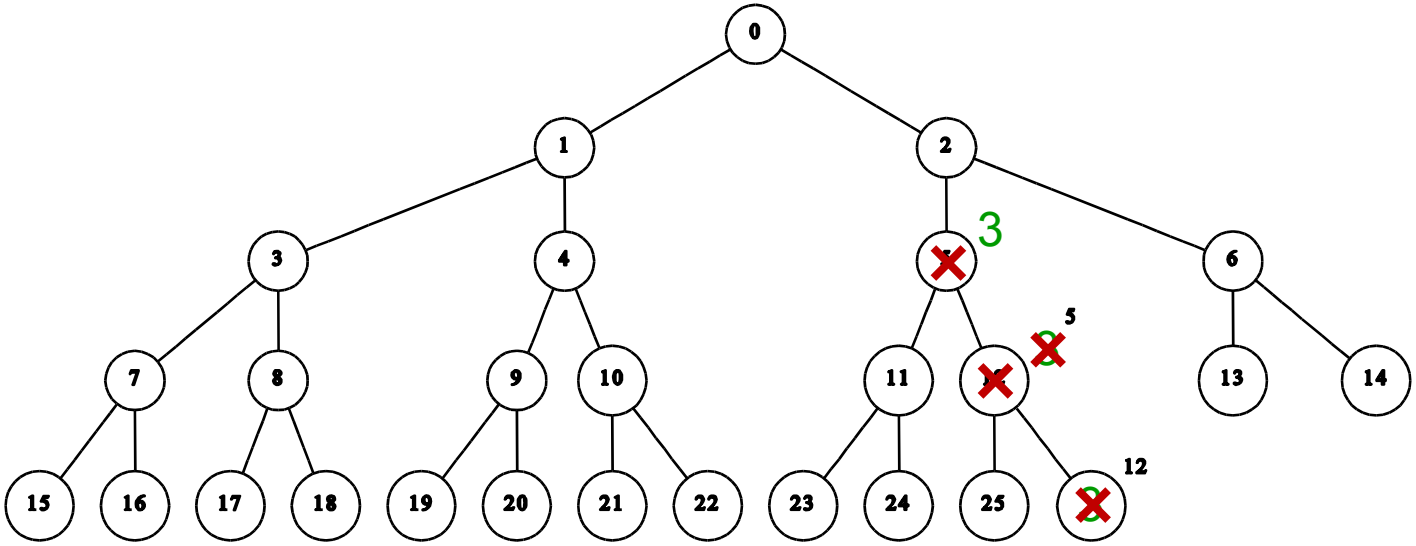
- **binary heap**
  - A *binary heap* is a binary tree with two additional constraints:
    1) It is a complete binary tree except the last level; that is, all levels of the tree, except possibly the last one (deepest) are fully filled. If the last level of the tree is not complete, the nodes of that level are filled from left to right.
    2) Each node is less than or equal to each of its children according to a comparison predicate ≤ over keys.



Binary Heap - Insert

- **Insert** (x)
  1. Add a node x at the end of the heap;
  2. **while** (key(parent(x)†) > key(x)) **{**
  3. Swap a location of the node x with the node parent(x);
  4. **}**

†parent(x) returns the parent of a node x. It returns x in the case where x has no parent.



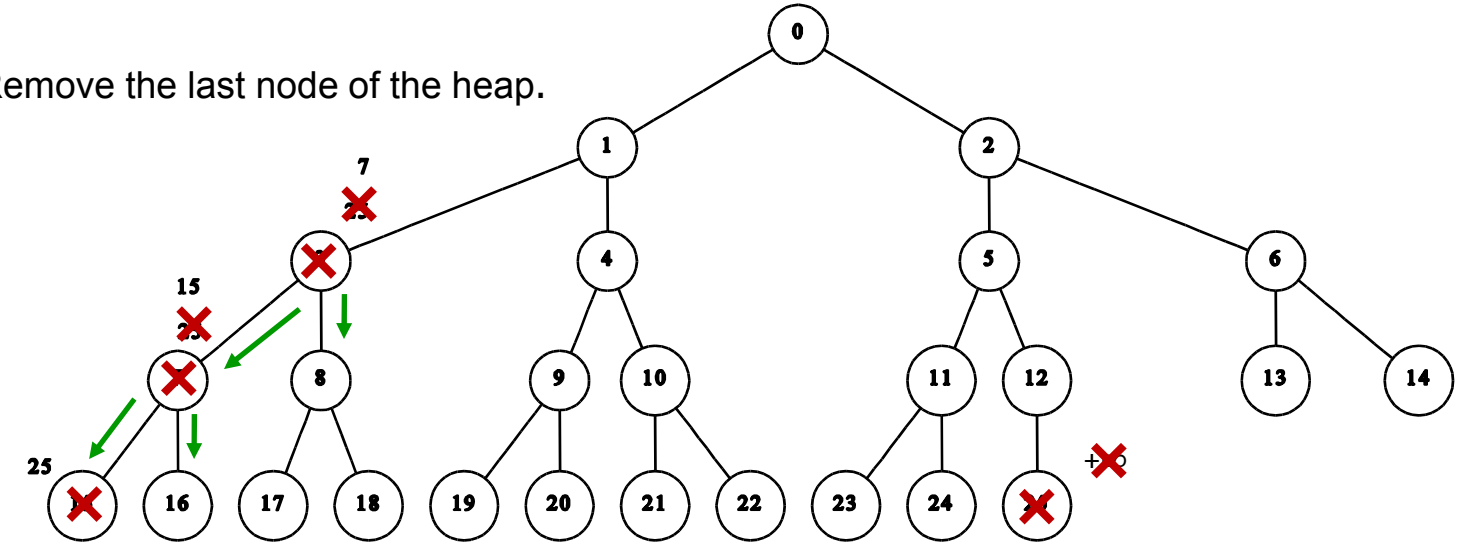
Binary Heap

- **AccessMin**
  - Returns the root of the heap's binary tree.
- **DeleteMin**
  1. $\&x$ = a location of the root of the heap;
  2. $\text{key}(x) = +\infty$;
  3. $\&y$ = a location of the last node of the heap;
  4. **do {**
  5. Swap a location of the node x with a location of the node y;
  6. $\&x = \&y$;
  7. **for each** $z \in \text{descendants}(x)$ **do**
  8. **if** ($\text{key}(y) > \text{key}(z)$) **then** $\&y = \&z$;
  9. **} while** ($\&x \neq \&y$);
  10. Remove the last node of the heap.
- **DecreaseKey** (x, d)
  - First, decrease the key of x by d and then apply the similar algorithm as in *Insert* case.

Binary Heap - Delete

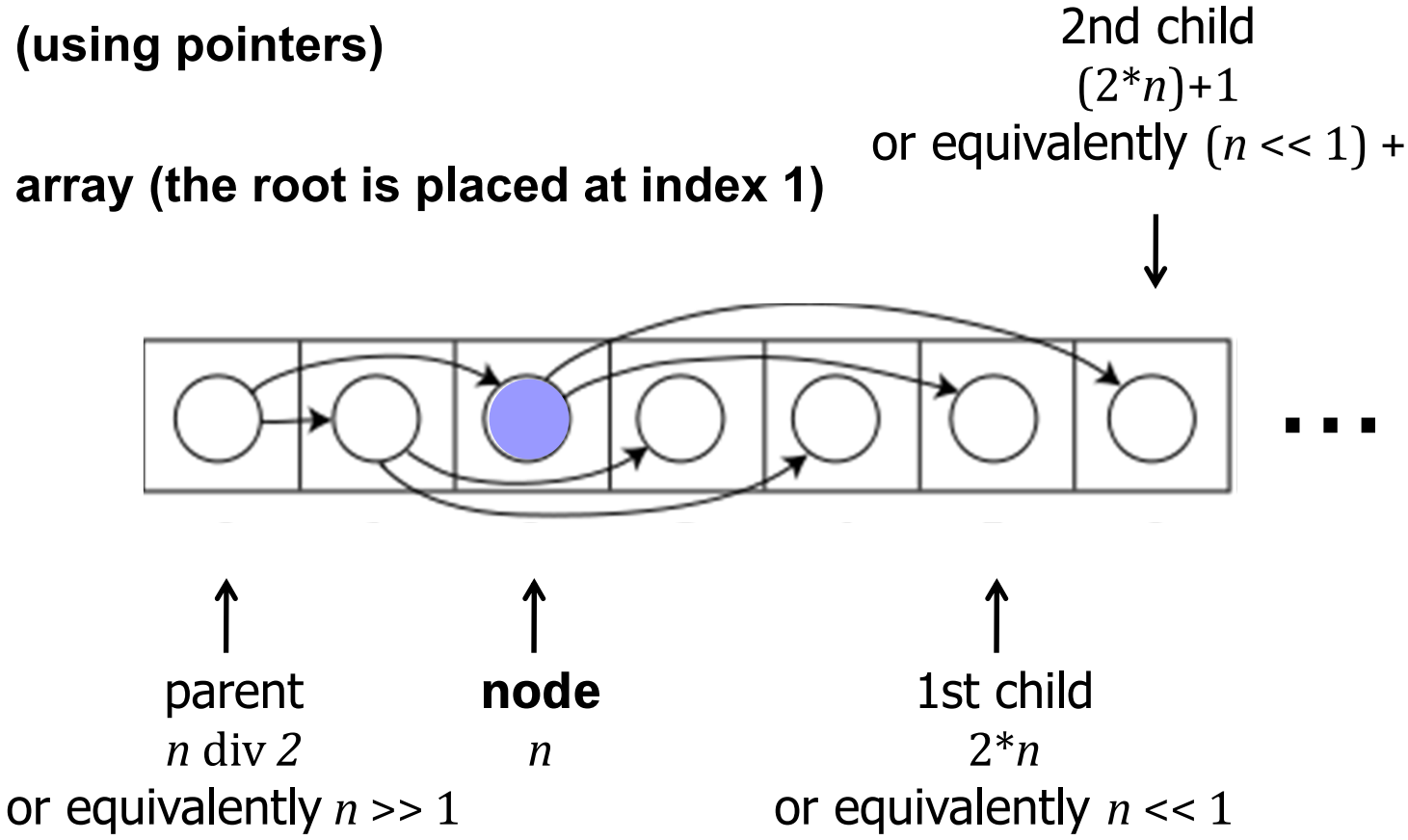
- **Delete** (&x)

1. key(x) = +∞; &y = a location of the last node of the heap;
2. **do {**
3. Swap a location of the node x with a location of the node y;
4. &x = &y;
5. **for each** z ∈ descendants(x) **do**
6. **if** (key(y) > key(z)) **then** &y = &z;
7. **} while** (&x ≠ &y);
8. **while** (key(parent(x)) > key(x)) **{**
9. Swap a location of the node x with the node parent(x);
10. **}**
11. Remove the last node of the heap.



Binary Heap - Representation

- **data representation**
  - **bidirectional tree structure (using pointers)**
  - **array (the root is placed at index 1)**



Binary Heap - BuildHeap

- **BuildHeap** (array A)

1. **for** $i = \left\lfloor \frac{\text{length}(A)}{2} \right\rfloor$ **downto** 1 **do {**
2. **Heapify**(A,i);
3. **}**

- **Heapify** (array A, index i)

1. $min = i$;
2. **do {**
3. $left = 2 \cdot i$;
4. $right = 2 \cdot i + 1$;
5. **if** ($left \leq \text{length}(A)$) **and** ($A[left] < A[min]$) **then** $min = left$;
6. **if** ($right \leq \text{length}(A)$) **and** ($A[right] < A[min]$) **then** $min = right$;
7. **if** $min = i$ **then break**;
8. **swap** $A[i] \leftrightarrow A[min]$;
9. $i = min$;
10. **} while true**;

Binary Heap – Time Complexity

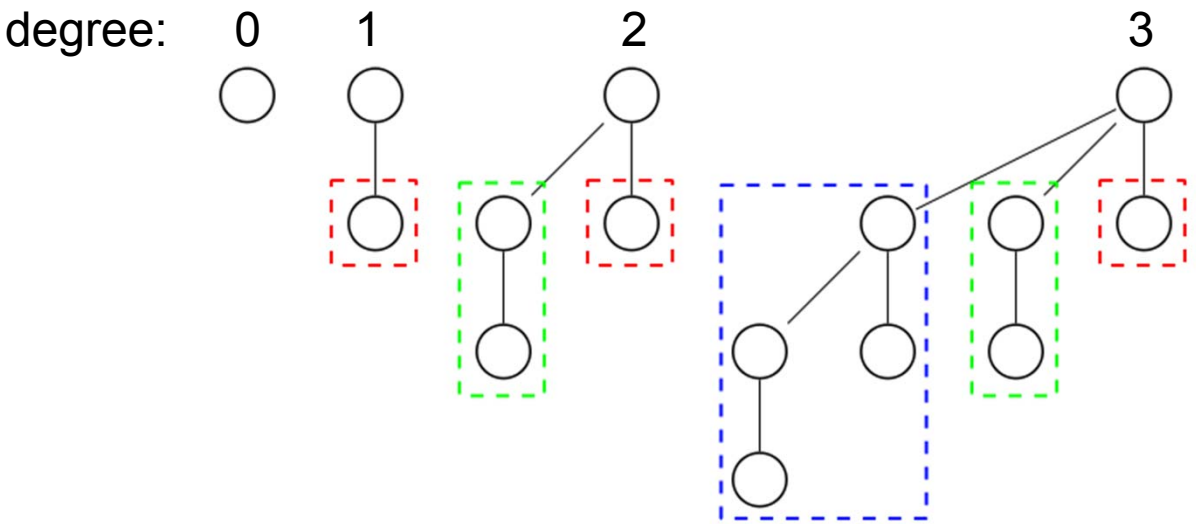
- **Insert**
  - O(log(n))
- **Delete**
  - O(log(n))
- **AccessMin**
  - O(1)
- **DeleteMin**
  - O(log(n))
- **DecreaseKey**
  - O(log(n))
- **BuildHeap**
  - $\sum_{h=0}^{\lceil \log(n) \rceil} (\text{number of nodes at heigh } h) \cdot O(h) \leq \sum_{h=0}^{\lceil \log(n) \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) \leq O(n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h}) =$ $O(n)$
- **Merge**
  - O(n) by building a new heap.

d-ary heap [d-regulární halda]

- A *d*-ary heap is a generalization of the binary heap in which the nodes have *d* children instead of 2.
- Operations for *d*-ary heap are analogical to the operations for binary heap.
- Asymptotic time complexity of *d*-ary heap operations is the same as binary heap operations.
- Exact complexity differs because of a different logarithm base (the base is *d*). For Delete operation it is needed to check *d* instead of 2 descendants in every loop.
- For an efficient implementation it is convenient to choose *d* as powers of 2. In this case, bit shifts can be used for traversing the array representation.
- A *d*-ary heap typically runs much faster than a binary heap for heap sizes that exceed the size of the computer's cache memory.

Binomial Heap [binomiální halda]

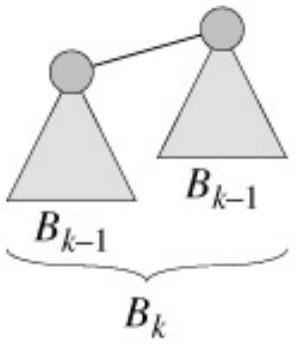
- **binomial heap**
  - A *binomial heap* is a collection of binomial trees of degrees: i=0,…,[log(n)]. There can only be either *one* or *zero* binomial trees for each degree, including zero degree. Each binomial tree in a heap obeys the heap property: the key of a node is less than or equal to the key of its child.
  - A **binomial tree** is defined recursively:
    - A binomial tree of order 0 is a single node
    - A binomial tree of degree k has a root node whose children are roots of binomial trees of degrees k−1, k−2, ..., 2, 1, 0 (in this order).



Binomial Heap – Binomial Tree

- **for a binomial tree B_k (of degree *k*) it holds:**
  - It satisfies the heap property,
  - the height of the tree is *k*,
  - its root has *k* children,
  - there are 2^k nodes,
  - there are exactly $\binom{k}{i}$ nodes at depth *i* for *i* = 0, 1, ..., *k*.

- **an alternative definition of a binomial tree:**
  - A binomial tree B_k (of degree *k*) consists of two binomial trees B_{k-1} (of degree *k*-1) that are ***linked*** together: the root of one, which is greater than the other, is the leftmost child of the root of the other.



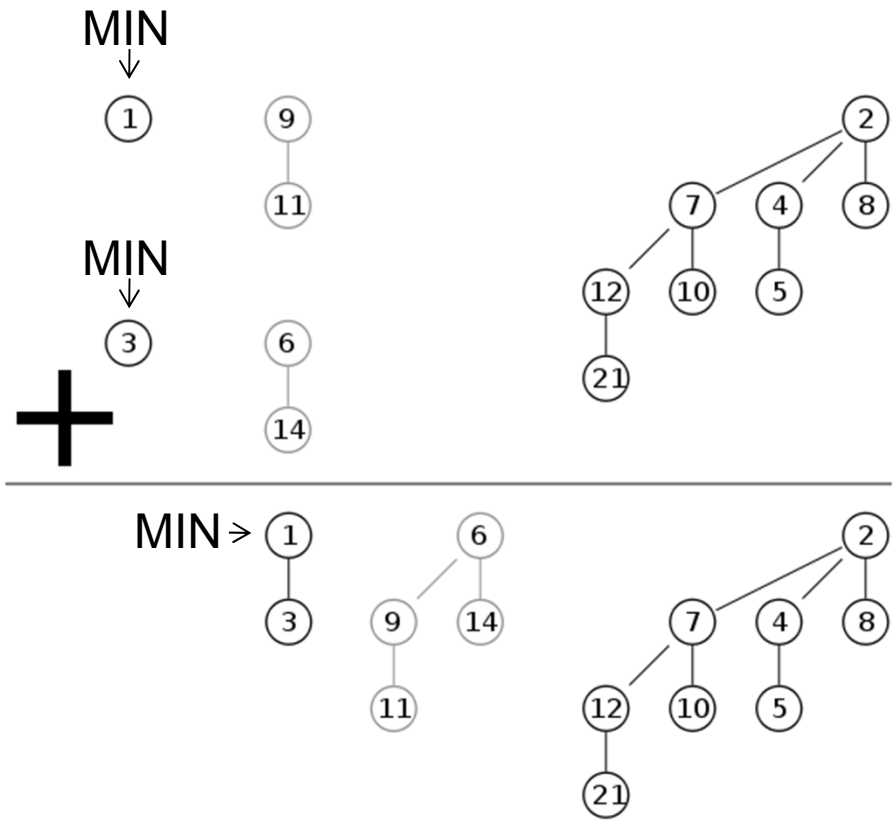
Binomial Heap – representation

- Because no operation requires random access to the root nodes of the binomial trees, the roots of the binomial trees can be stored in a linked list, ordered by increasing degree of the tree. But of course, binomial trees can be stored in array as well.
- The whole binomial heap is formed by binomial trees and an additional pointer to a binomial tree with a the minimum node of the whole heap (*MIN pointer*). *MIN* is always root by the heap property. *MIN* must be updated when performing any operation other than AccessMin. This can be done in O(log *n*) without raising the running time of any operation.

Binomial Heap – Insert, AccessMin, Merge

- **Insert** (x)
  1. Create a new heap containing only this element (there is only one tree of degree 0).
  2. Merge it with the original heap.
- **AccessMin**
  - It returns the root of a binomial tree from *MIN* pointer.
- **Merge** (H_1, H_2)
  - Because each binomial tree in a binomial heap corresponds to a bit in the binary representation of its size, there is an analogy between the merging of two heaps and the binary addition of the sizes of the two heaps, from right-to-left. Whenever a carry occurs during addition, this corresponds to a merging of two binomial trees during the merge. Due to the structure of binomial trees, they can be merged trivially. As their root node is the smallest element within the tree, by comparing the two keys, the smaller of them is the minimum key, and becomes the new root node. Then the other tree becomes a subtree of the combined tree. In the end, we update *MIN* pointer.

Binomial Heap - Merge



Binomial Heap - DeleteMin

1. **procedure** DeleteMin(binomial_heap H)
2. tree_with_minimum = H.*MIN*;
3. **for each** tree ∈ tree_with_minimum.subTrees **do {**
4. tmp.addTree(tree);
5. **}**
6. H.removeTree(tree_with_minimum)†;
7. H = **Merge**(H, tmp);

† Technically, this operation removes only the root of tree_with_minimum. All children subtrees of the root are used in tmp heap which is merged at line 7.

Binomial Heap – DecreaseKey, Delete

- **DecreaseKey**
  - It is analogical to binary heap DecreaseKey.
  - After decreasing the key of an element, it may become smaller than the key of its parent, violating the heap property. If this is the case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the heap property is no longer violated. Each binomial tree has height at most log n, so this takes O(log n) time.
- **Delete** (x)
  1. decrease x key to -∞ (that is, some value lower than any element in the heap) by DecreaseKey.
  2. delete the minimum in the heap by DeleteMin.

Binomial Heap – Time Complexity

- **Merge**
  - O(log(n))
- **Insert**
  - O(log(n))
  - The amortized complexity is O(1). It is analogical to a binary counter increment.
- **AccessMin**
  - O(1)
- **DeleteMin**
  - O(log(n))
- **DecreaseKey**
  - O(log(n))
- **Delete**
  - O(log(n))

Amortized Complexity [amortizovaná složitost]

- In an ***amortized analysis***, the time required to perform a sequence of data-structure operations is averaged over all the operations performed.
- Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation within the sequence might be expensive.
- Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the ***average performance of each operation in the worst case***.

Amortized Complexity

- Example: **A Complexity of INSERT in a dynamic array**
  - A *dynamic array* is an array which resizes by a doubling in size in the case that it is full, and uses the reserved space for future expansions.
  - INSERT without resize requires O(1), for *N* elements without resize O(*N*).
  - If the array is full then the reallocation (resizing) is needed. In the worst case, this operation takes O(*N*).
  - For insertion of *N* elements including reallocation we need in the worst case O(*N/2*) + O(*N/4*) +...+ O(*N*/2$^{\lfloor \log N \rfloor}$) + O(*N*) = O(*N*) + O(*N*) = O(*N*).

$$\sum_{i=0}^{\lfloor \log N \rfloor} \left\lfloor \frac{N}{2^i} \right\rfloor < N \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2N$$

  - Then the amortized time complexity for one INSERT operation is O(*N*)/*N* = O(1).

Fibonacci Heap [Fibonacciho halda]

- A **Fibonacci heap**, in fact, is loosely based on binomial heap.
- Fibonacci heaps have a more relaxed structure than binomial heaps, however, allowing for improved asymptotic time bounds.
- Fibonacci heaps support the same operations but have the advantage that operations that do not involve deleting an element (**AccessMin**, **Merge**, and **DecreaseKey**) run in **O(1) amortized time**.
- Operations **Delete** and **DeleteMin** have **O(log(n))** amortized time complexity.
- The usage of Fibonacci heaps is not suitable for real-time systems, because some operations can have a linear time complexity in the worst case.
- From a practical point of view, however, the constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary binary (or d-ary) heaps for most applications.

Fibonacci Heap

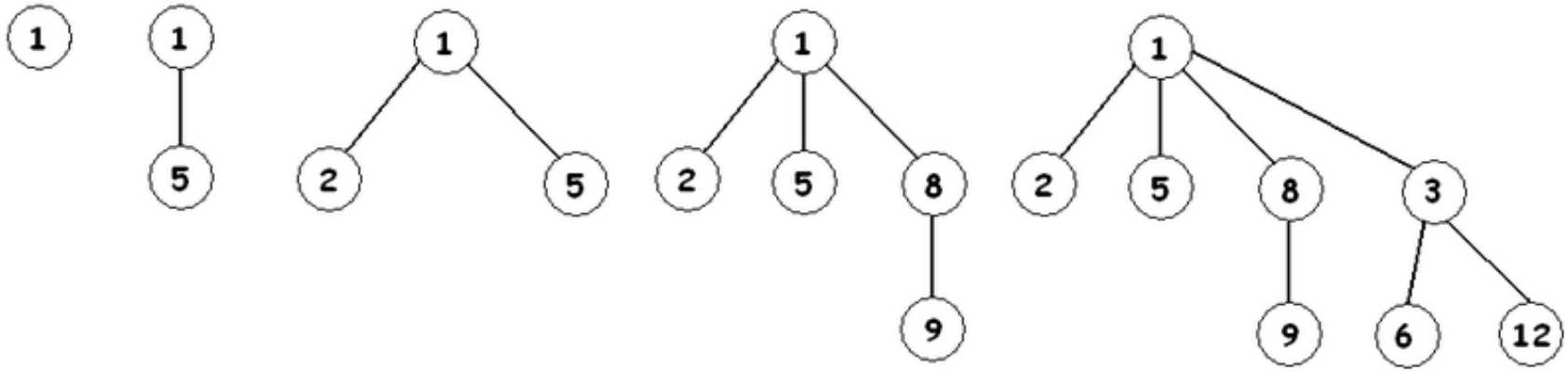
- Like a binomial heap, a ***Fibonacci heap*** is a collection of trees that satisfy the heap property.
- Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted but unordered.
- An ***unordered binomial tree*** is like a binomial tree, and it is also defined recursively. The unordered binomial tree U_0 consists of a single node, and an unordered binomial tree U_k consists of two unordered binomial trees U_{k-1} so that the root of one is made into *any* child of the root of the other.
- Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree.
- This flexibility allows some operations to be executed in a "lazy" manner, postponing the work for later operations. For example merging heaps is done simply by concatenating the two lists of trees, and sometimes operation *decrease key* cuts a node from its parent and forms a new tree.

Fibonacci Heap – Fibonacci Trees

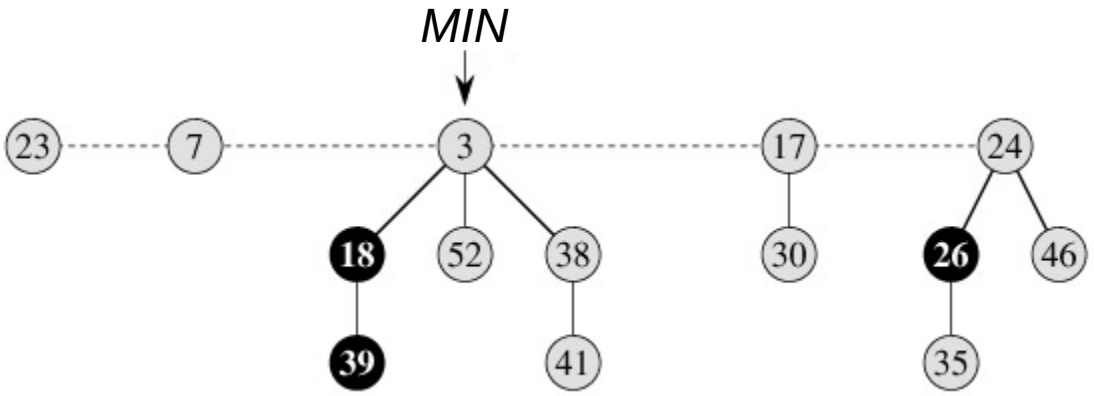
- Every node has degree (= the number of children) at most O(log n) and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the k-th Fibonacci number.

$$F_n = \begin{cases} 0, & \text{for } n = 0; \\ 1, & \text{for } n = 1; \\ F_{n-2} + F_{n-1} & \text{otherwise.} \end{cases} \quad \Leftrightarrow \quad F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}} \text{ where } \varphi = \frac{1+\sqrt{5}}{2} \approx 1.618;$$

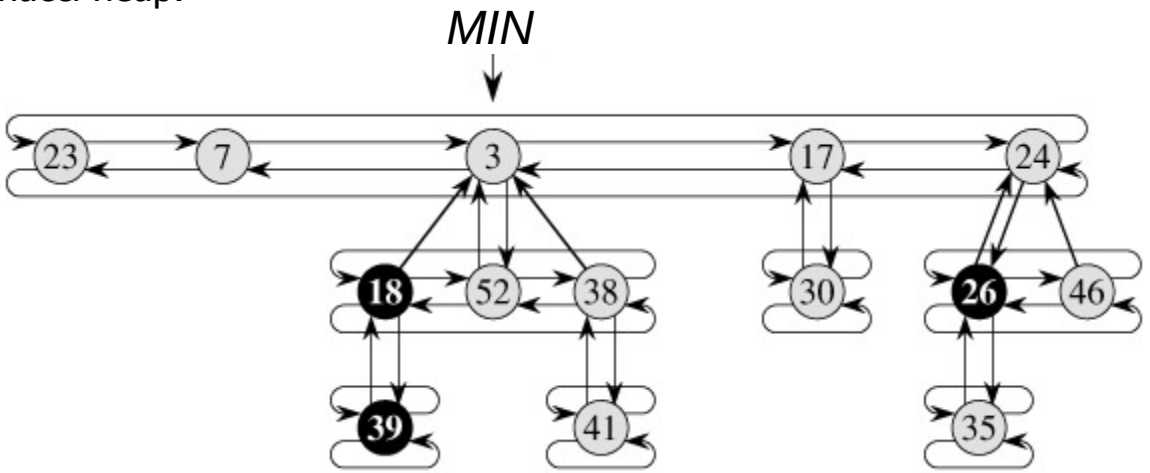
- This is achieved by the following two Fibonacci tree rules:
  1. **We can cut at most one child of each non-root node.**
  2. **When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree.**
- The number of trees is decreased in the operation DeleteMin, where trees are consolidated together.



Fibonacci Heap – Representation



- Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted but **unordered**. Each node x contains a pointer to its parent and a pointer to any one of its children. The children of x are linked together in a **circular, doubly linked list**.
- The roots of all the trees in a Fibonacci heap are linked together into a circular, doubly linked list called the ***root list*** of the Fibonacci heap.



Fibonacci Heap – Representation

- N is the actual number of elements in the heap.
- MIN is a pointer to the minimum element in the heap. It must be always a root from the root list of the heap.
- $\mathbf{key}(x)$ is a value of the key of the element x.
- $\mathbf{mark}(x)$ is a Boolean value which indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node.
- $\mathbf{descendants}(x)$ returns all children of x.
- $\mathbf{parent}(x)$ returns parent of a node x. It returns x in case where x has no parent.

Fibonacci Heap – Merge, Insert

- **Merge** (H_1, H_2)
  - Connect both doubly cyclic linked lists to one and then update pointer to *MIN*.
  - O(1)
- **AccessMin**
  - It returns the root of the Fibonacci tree from *MIN* pointer.
  - O(1)
- **Insert** (x)
  1. Create a new heap containing only x element (there is only one tree of degree 0).
  2. mark(x) = false;
  3. Merge it with the original heap.
  - O(1)

Fibonacci Heap – DeleteMin

- **DeleteMin**

```
1.   z = MIN;
2.   if z ≠ null then {
3.       for each x ∈ descendants(z) do
4.           add x to the root list of the heap;
5.       remove z from the root list of the heap;
6.       if N = 1 then
7.           MIN = null
8.       else {
9.           MIN = any pointer to a root from the root list of the heap;
10.          Consolidate;
11.      }
12.      N--;
13.  }
```

time complexity: **O(*N*)**
amortized: **O(log(*N*))**

Fibonacci Heap – Consolidate

Consolidate

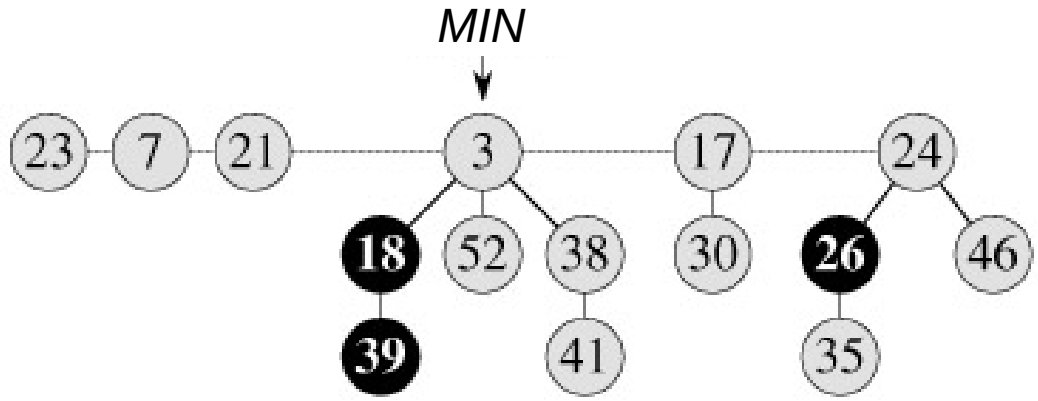
```
1.  for i = 0 to max. possible degree of a tree in Fibonacci heap of size N  do A[i] = null;
2.  for each w ∈ all trees in the root list of the heap do {
3.    x = w; d = a degree of the tree w;
4.    while A[d] ≠ null do {
5.      y = A[d];
6.      if key(x) > key(y) then swap x and y;
7.      remove y from the root list of the Heap;
8.      make y a child of x, incrementing the degree of x;
9.      mark(y) = false; A[d] = null; d++;
10.   }
11.   A[d] = x;
12. }
13. MIN = null;
14. for i = 0 to max. degree of a tree in the array A do
15.   if  A[i] ≠ null then {
16.     add A[i] to the root list of the heap;
17.     If  (MIN = null) or (key(A[i]) < key(MIN)) then MIN = A[i];
18.   }
```

time complexity: **O(*N*)**

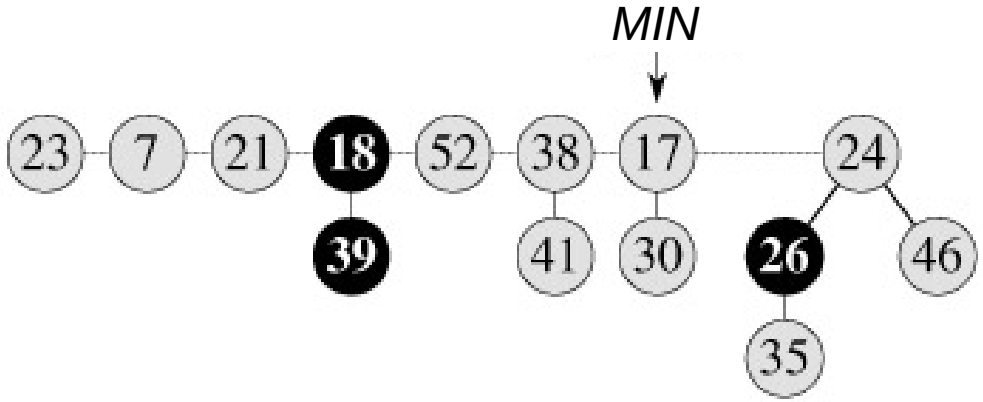
amortized: **O(log *N*)**

Fibonacci Heap – DeleteMin Example

1. Consider the following Fibonacci heap.

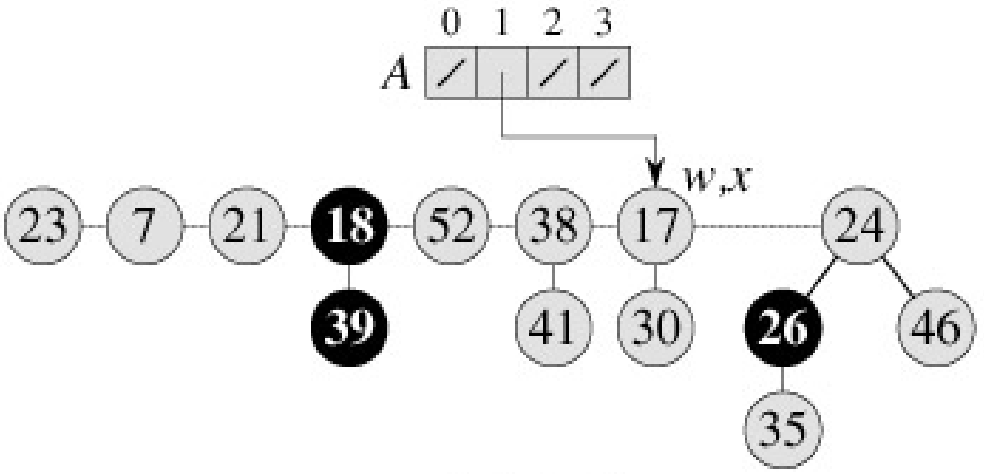


2. The situation after the minimum node *z* is removed from the root list and its children are added to the root list.

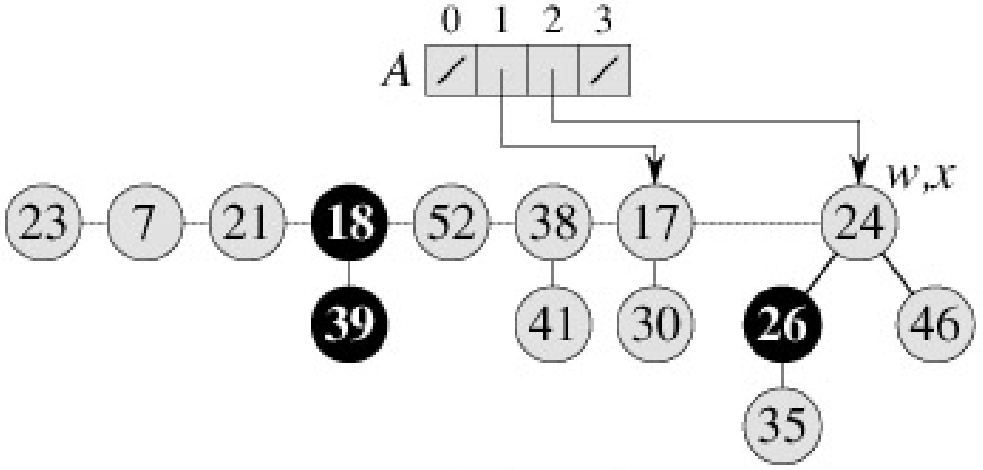


Fibonacci Heap – DeleteMin Example

3. The array *A* and the trees after each of the first three iterations of the *for each* loop of lines 2-12 of the procedure Consolidate. The root list is processed by starting at the node pointed to by *MIN* and following *right* pointers. Each part shows the values of *w* and *x* at the end of an iteration.

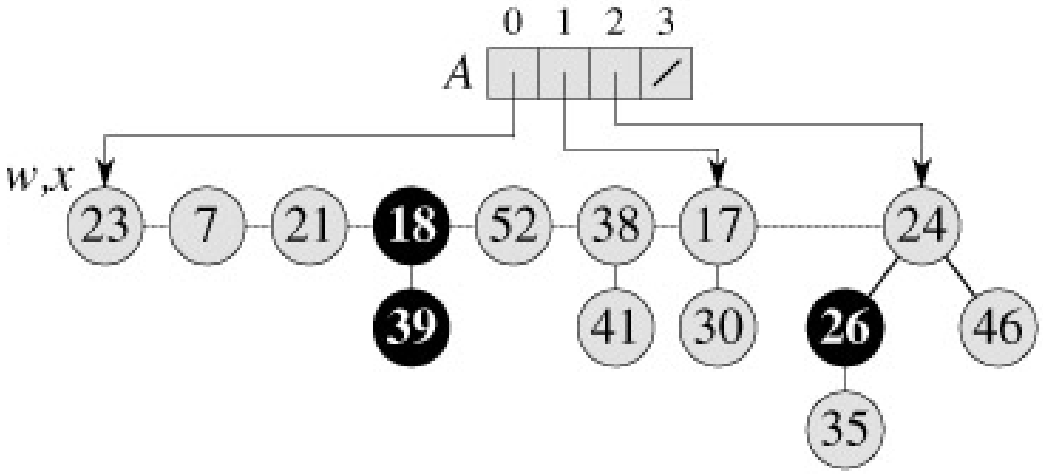


4.

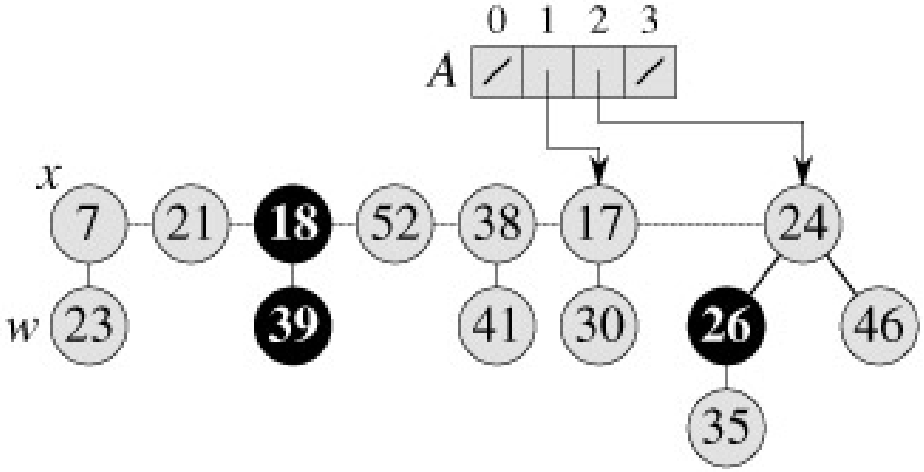


Fibonacci Heap – DeleteMin Example

5.

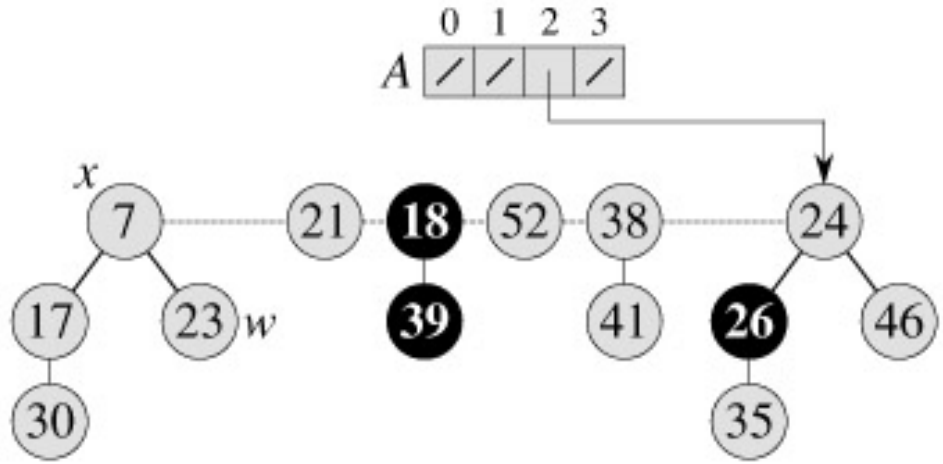


6. The Figure shows the situation after the first time through the *while* loop. The node with key 23 has been linked to the node with key 7, which is now pointed to by *x*.

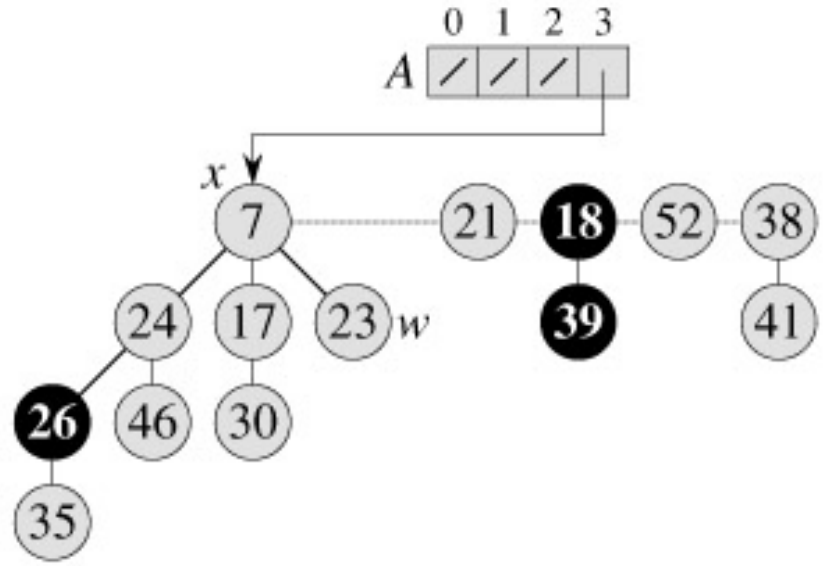


Fibonacci Heap – DeleteMin Example

7. The node with key 17 has been linked to the node with key 7, which is still pointed to by *x*.

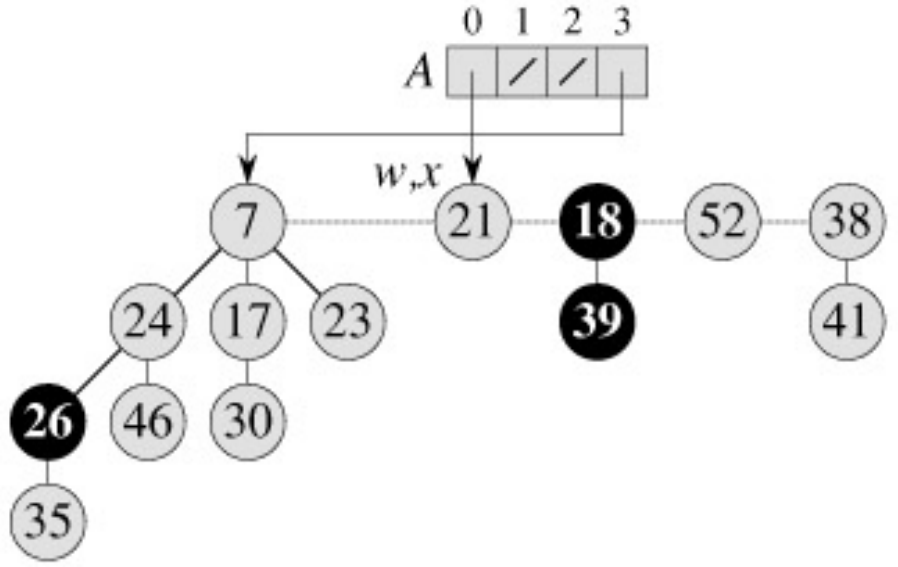


8. The node with key 24 has been linked to the node with key 7. Since no node was previously pointed to by *A*[3], at the end of the *for each* loop iteration, *A*[3] is set to point to the root of the resulting tree.

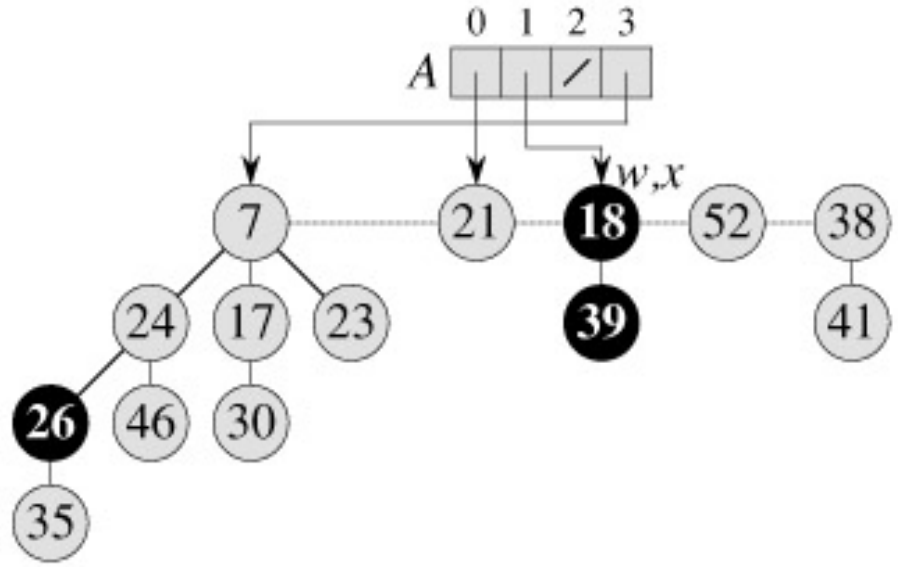


Fibonacci Heap – DeleteMin Example

9. The situations after each of the next four iterations of the *for each* loop.

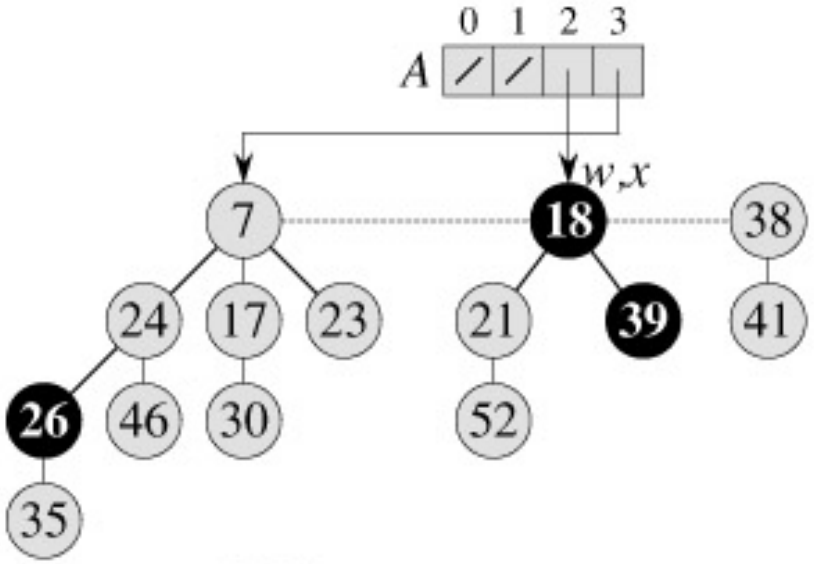


10.

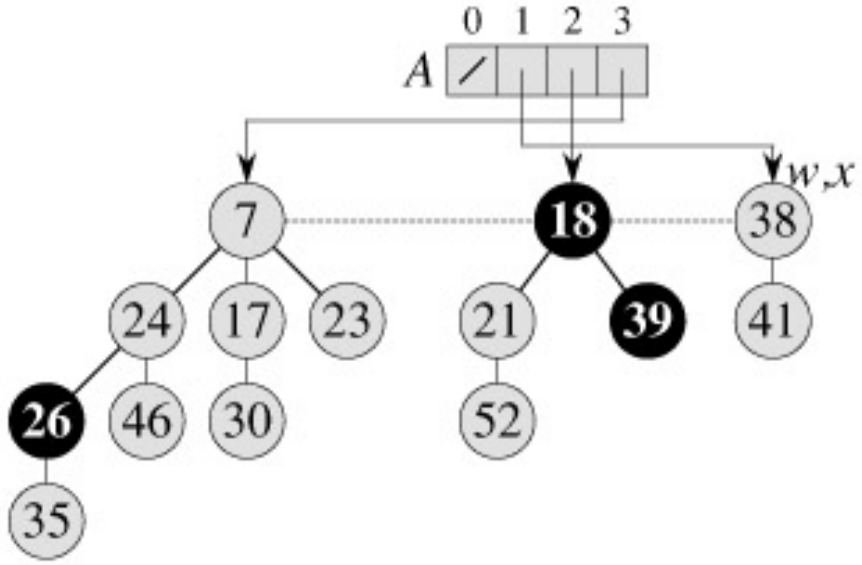


Fibonacci Heap – DeleteMin Example

11.

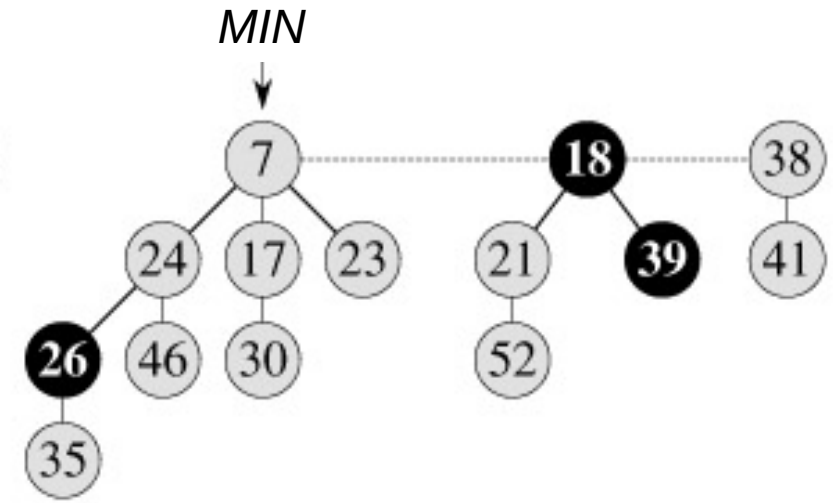


12.



Fibonacci Heap – DeleteMin Example

13. The Fibonacci heap after reconstruction of the root list from the array *A* and determination of the new *MIN* pointer.



Fibonacci Heap – DecreaseKey, Delete

- **DecreaseKey** (x,d)

```
1.  key(x) = key(x) – d;
2.  y = parent(x);
3.  if (x ≠ y) and (key(x) < key(y)) then {
4.      Cut(x,y);
5.      Cascading-Cut(y);
6.  }
7.  If key(x) < key(MIN) then MIN = x;
```

time complexity: **O(log *N*)**
amortized: **O(1)**

- **Cut**(x,y)

1. remove x from the child list of y, decrementing the degree of y;
2. add x to the root list of the heap;
3. mark(x) = **false**;

time complexity: **O(1)**

- **Delete**(x)

1. DecreaseKey(x,∞)
2. DeleteMin;

time complexity: **O(*N*)**
amortized: **O(log *N*)**

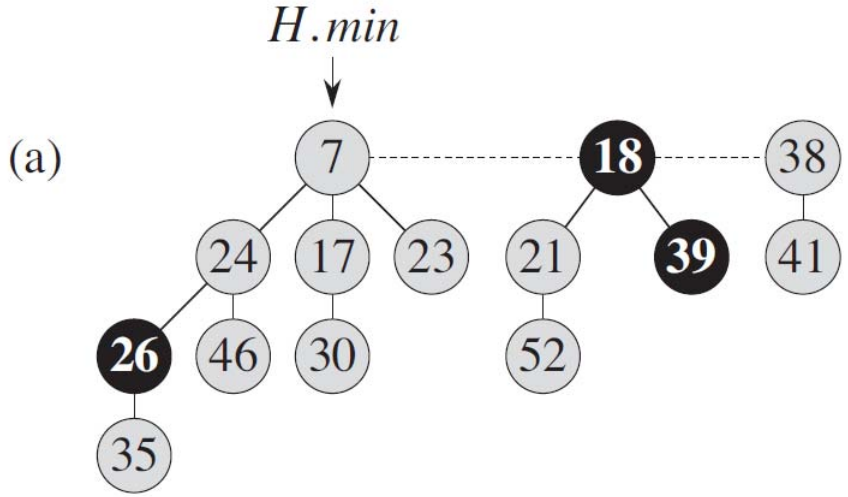
- **Cascading-Cut** (y)

```
1.  z = parent(y);
2.  if (y ≠ z) then
3.      if mark(y) = false then mark(y) = true
4.      else {
5.          Cut(y,z);
6.          Cascading-Cut(z);
7.      }
```

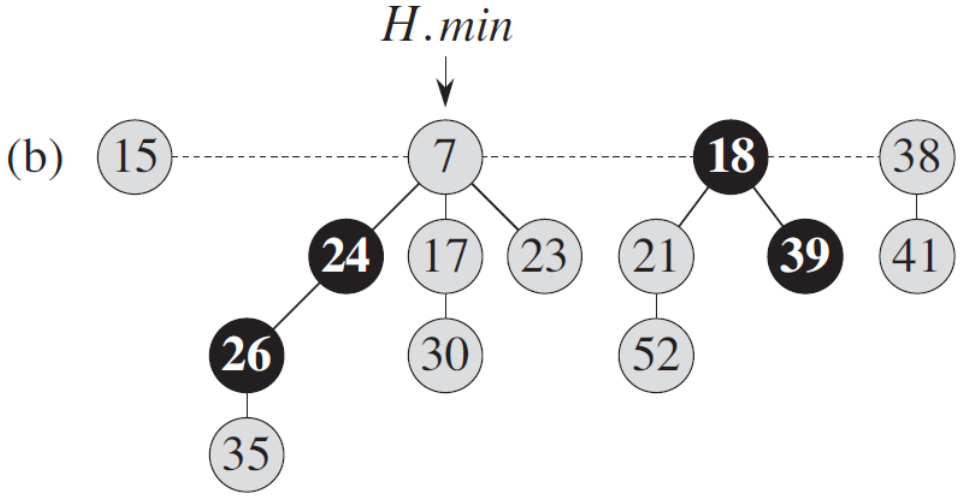
time complexity: **O(log *N*)**
amortized: **O(1)**

Fibonacci Heap – DecreaseKey Example

(a) The initial Fibonacci heap.

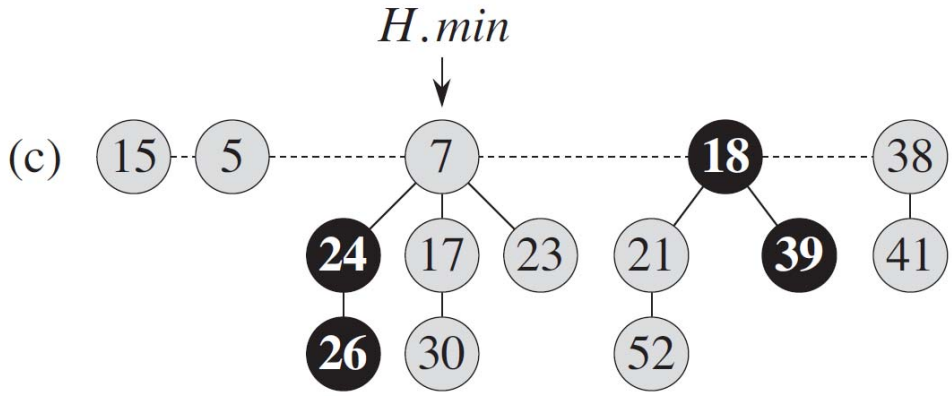


(b) The node with key 46 has its key decreased to 15. The node becomes a root and its parent (key 46, previously unmarked) becomes marked.

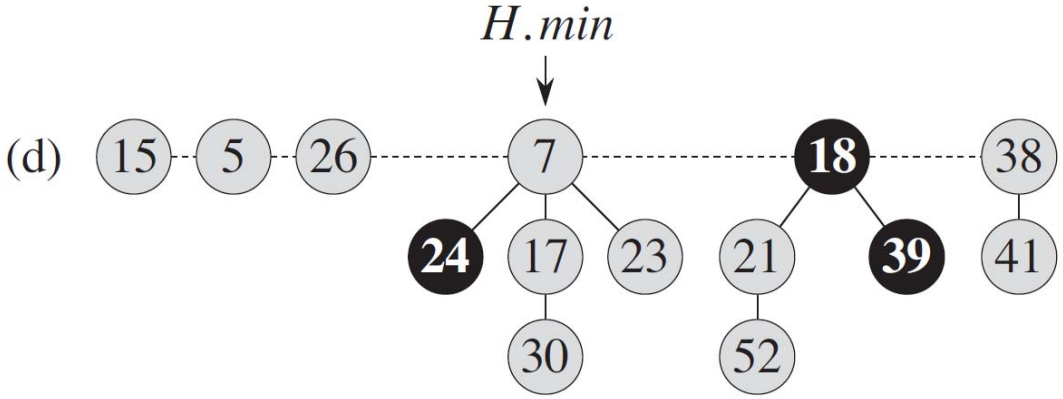


Fibonacci Heap – DecreaseKey Example

(c) The node with key 35 has its key decreased to 5. It now becomes a root. Its parent, with key 26, is marked, so a **cascading cut** occurs.

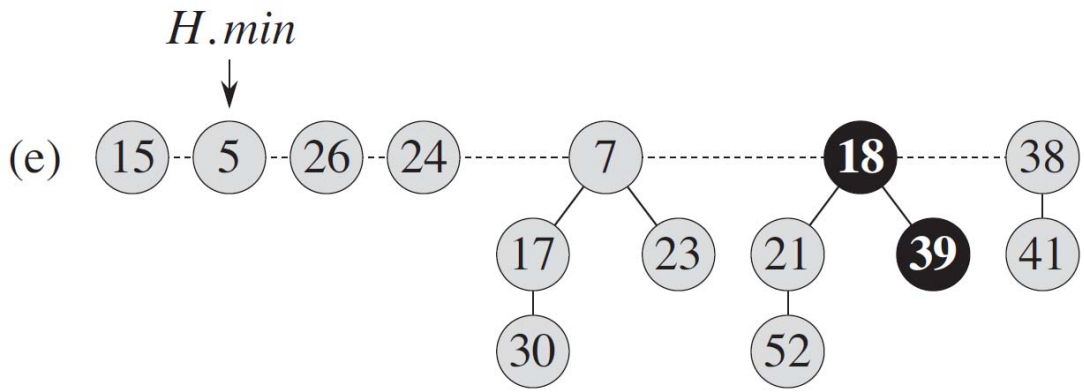


(d) The node with key 26 is cut from its parent and made an unmarked root. Another **cascading cut** occurs, since the node with key 24 is marked as well.



Fibonacci Heap – DecreaseKey Example

(e) The node with key 24 is also cut form its parent and made an unmarked root. The **cascading-cut** stops at this point since the node with key 7 is a root. The H.min pointer is updated.



Heaps – Comparison of Time Complexity

	binary heap	*d*-ary heap	binomial heap	Fibonacci heap
AccessMin	Θ(1)	Θ(1)	Θ(1)	Θ(1)
DeleteMin	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n) amortized: O(log(n))
Insert	Θ(log(n))	Θ(log(n))	O(log(n)) amortized: O(1)	Θ(1)
Delete	Θ(log(n))	Θ(log(n))	O(log(n))	O(n) amortized: O(log(n))
Merge	Θ(n)	Θ(n)	O(log(n))	Θ(1)
DecreaseKey	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n)) amortized: O(1)

References

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). *Introduction to Algorithms (2nd ed.)*. MIT Press and McGraw-Hill. ISBN 0-262-53196-8.
- Fredman, M. L. ; Tarjan, R. E. *Fibonacci heaps and their uses in improved network optimization algorithms.* J. ACM 34(1987), 596-615.