Priority queues.
Disjoint set collections

DS 2018/2019

Content

- Priority queues and “max-heap”.
- Disjoint set collections and “union-find”.

Priority queues - examples

- Plane passengers
  - Priorities:
    - Buisness-class
    - Persons travelling with children / with reduced mobility
    - Other passengers
- Planes approaching the airport
  - Priorities:
    - Emergences
    - Fuel level
    - Distance to the airport

Priority queues: abstract

- Objects: data structures where the elements are called *atoms*; any atom has *key-field* called *priority*.

- Elements are stores function of their priorities and not their position.

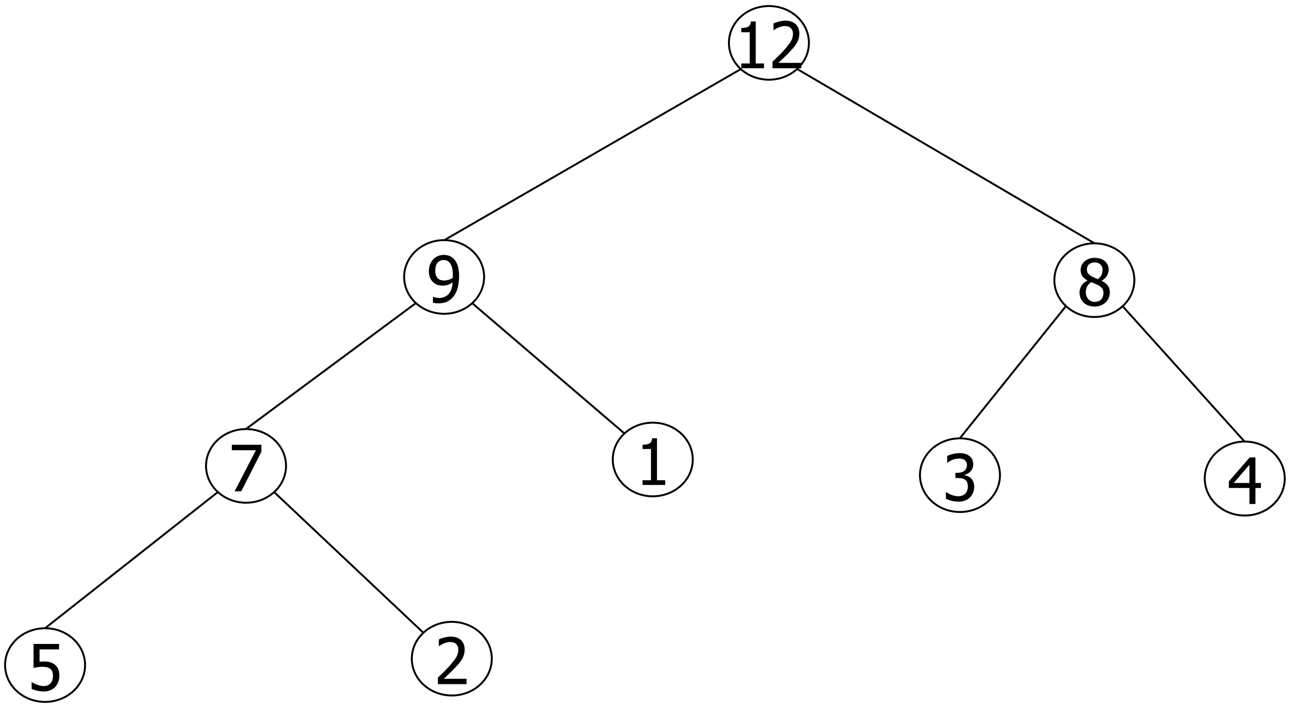
Priority queues: operations

- read
  - input: priority queue C
  - output: the atom from C with the highest priority
- delete
  - input: priority queue C
  - output: C from which the atom with the highest priority has been deleted
- insert
  - input: priority queue C and an atom **at**
  - output: C where the atom **at** has been added

maxHeap

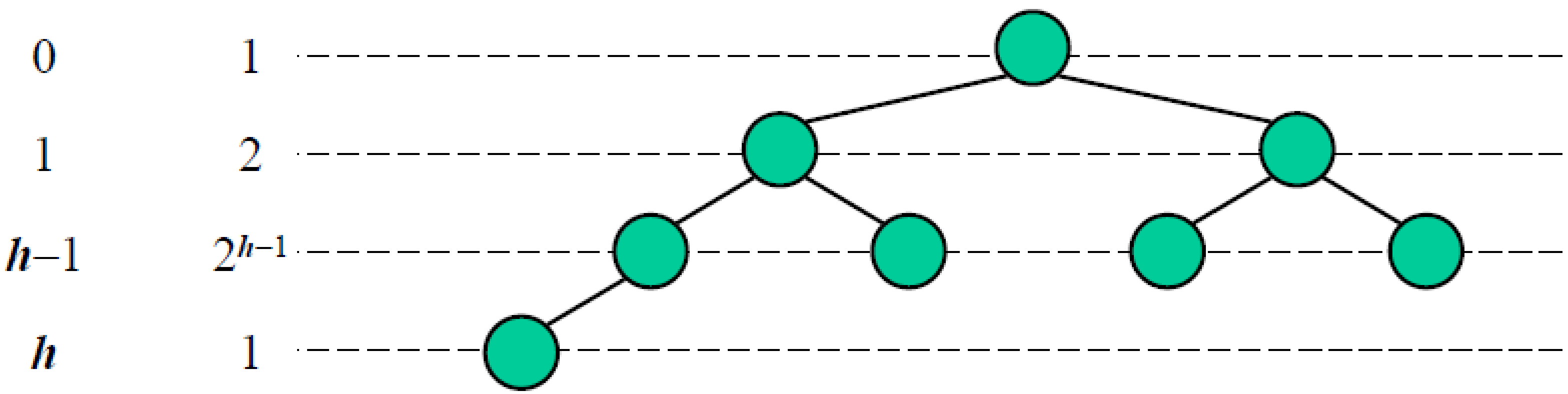
- Implements the priority queues.
- Binary tree with properties:
  - Nodes stores the fields “key”;
  - For any node, the node keys higher or equal than the child node keys;
  - The tree is complet. Let *h* be the tree height. Then,
    - For $i = 0, \ldots, h-1$, there are 2^i nodes of height *i*;
    - On level $h-1$, the internal nodes are on the left of external nodes;
  - The last node of a maxHeap is the rightmost node on the *h* level.

`maxHeap` - example



`maxHeap` height

- Theorem: A maxHeap with *n* keys has the height *O(log n)*.
- Proof:
  - The complete binary tree properties are used.
  - Let *h* a maxHeap height with *n* keys.
  - There are 2^i keys of depth *i*, for *i = 0, ..., h-1* and at least one key of depth *h*: $n \geq 1+2+4+\ldots+2^{h-1}+1 = 2^h$
  - It follows: $h \leq \log n$

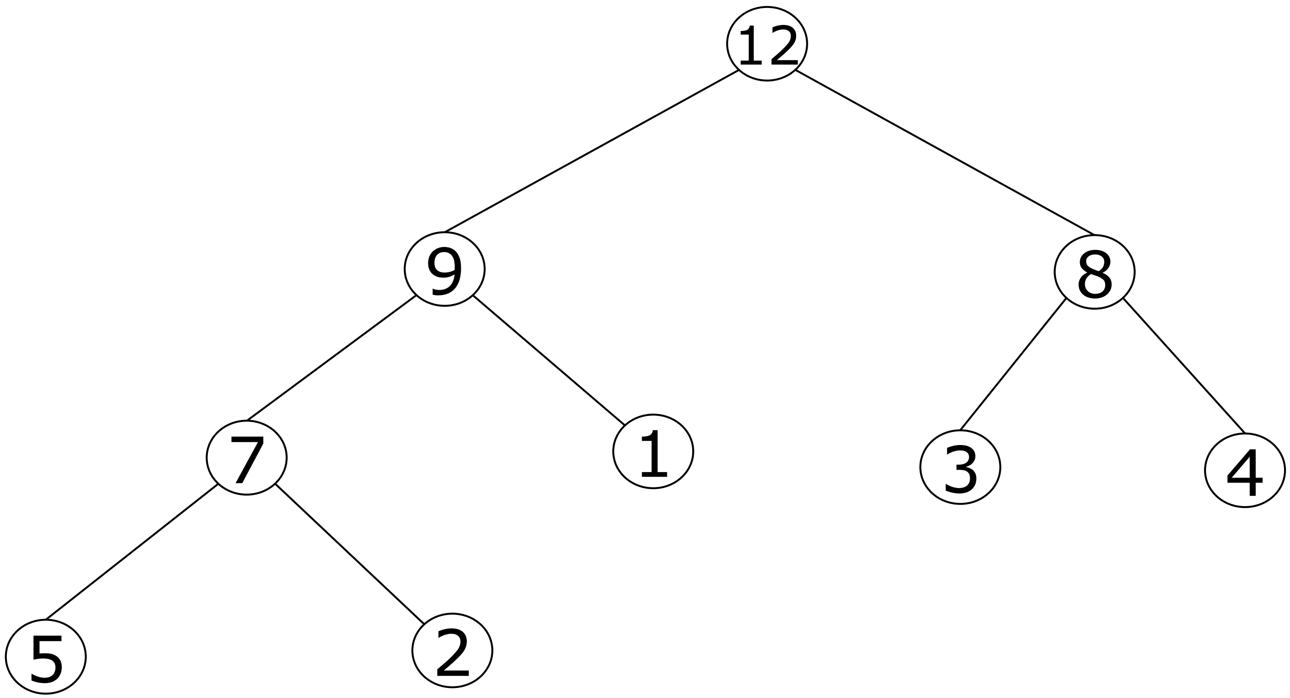


`maxHeap`: delete

- The heap root is deleted (it correspons to the atom with the highest probability).

- The algorithm has three stages:
  - The root key is replaced with the key of the last node;
  - The last node is deleted;
  - The maxHeap property is repaired.

`maxHeap`: delete

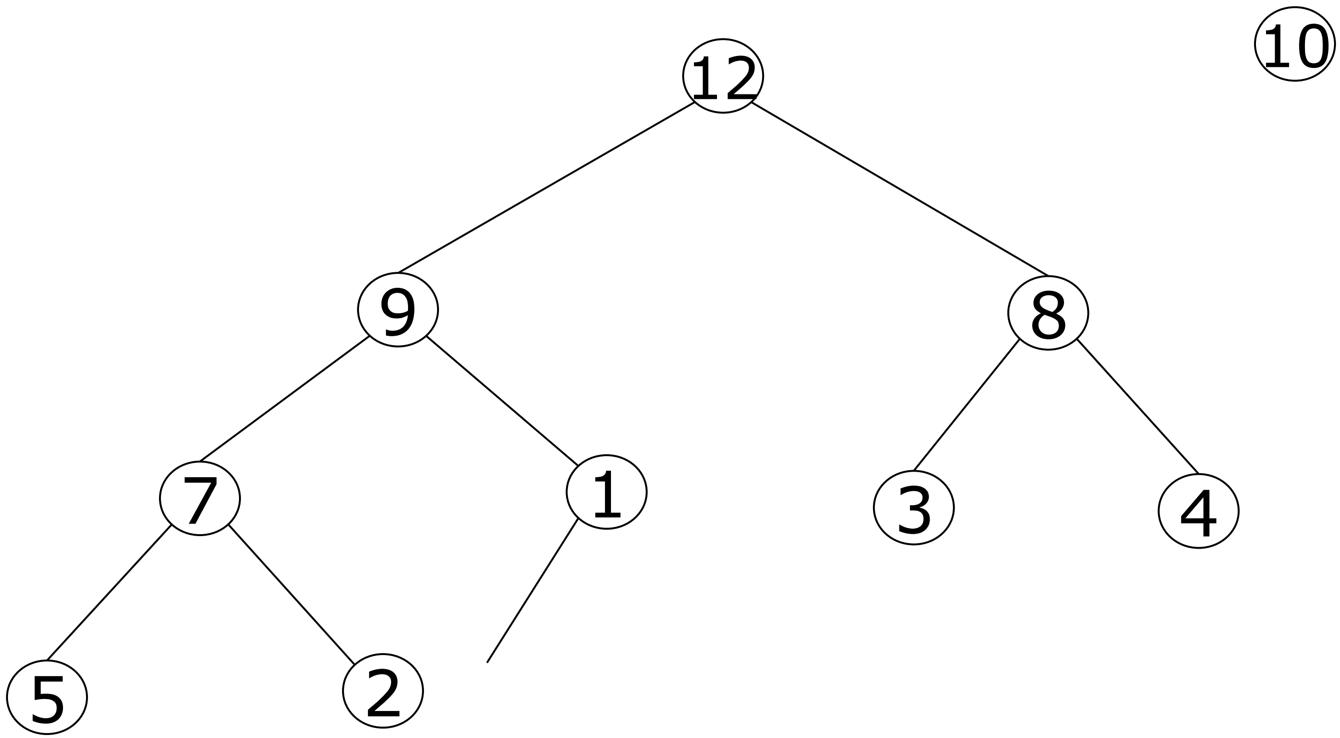


`maxHeap`: insert

- The new key is inserted in a new node.

- Algorithm has three stages:
  - A new node is added as the rightmost node on the last level;
  - The new key is inserted in this node;
  - The maxHeap property is repaired.

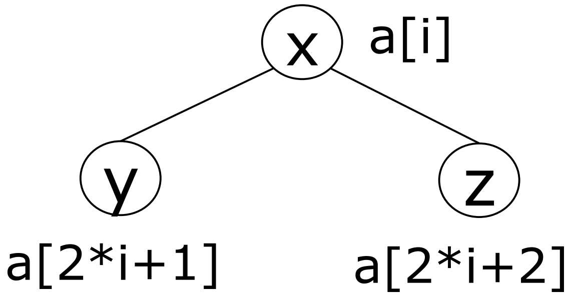
`maxHeap`: insert

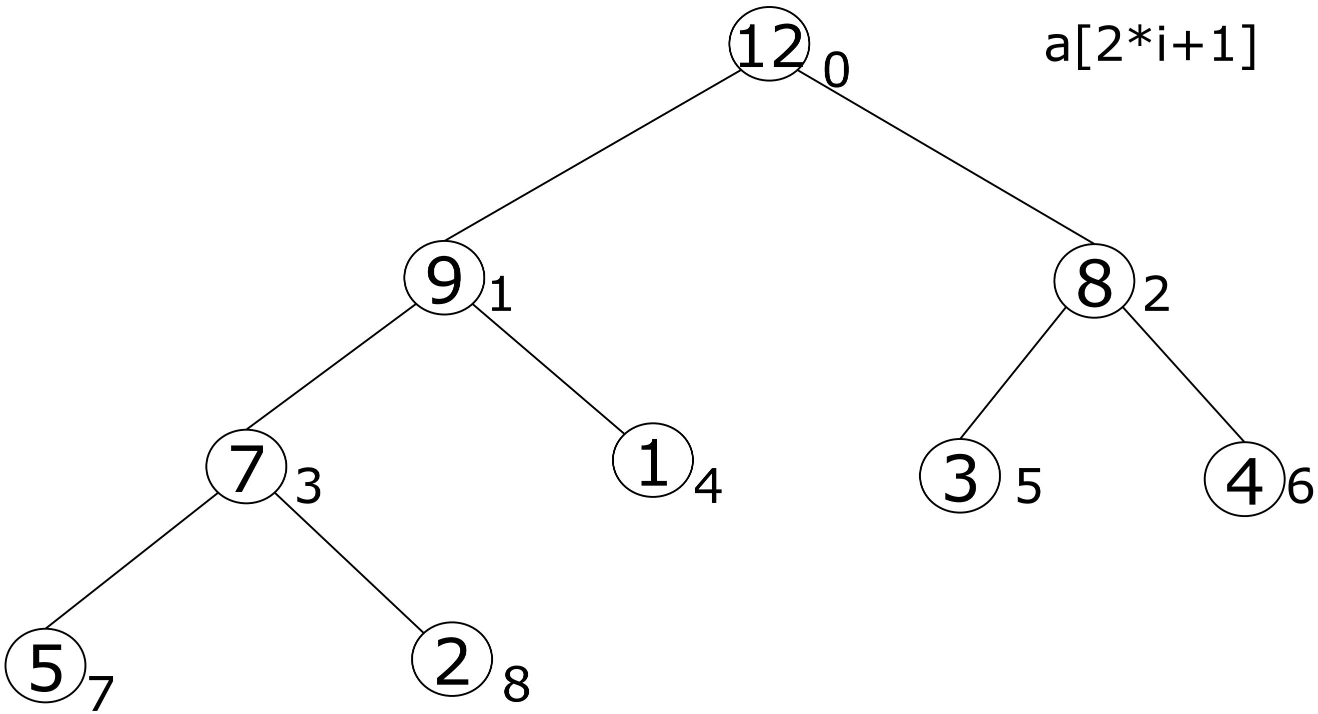


`maxHeap`: array implementation

$$(\forall k)\ 1 \leq k \leq n-1 \Rightarrow a[k] \leq a[(k-1)/2]$$

0	1	2	3	4	5	6	7	8
12	**9**	**8**	**7**	**1**	**3**	**4**	**5**	**2**





`maxHeap`: insert

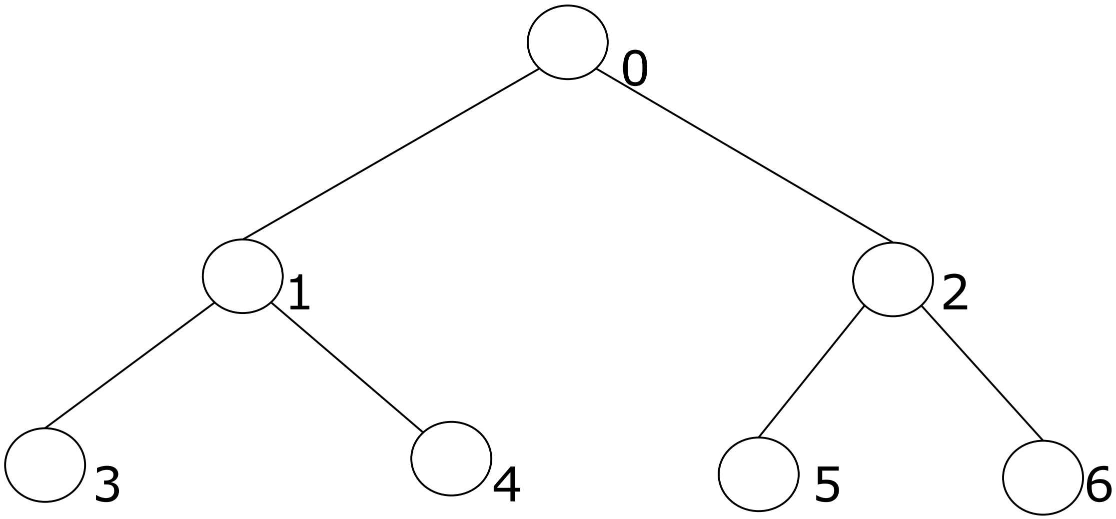
```
procedure insert(a, n, key)
begin
    n ← n+1
    a[n-1] ← key
   j ← n-1
   heap ← false
   while ((j > 0) and not heap) do
      k ← [(j-1)/2]
      if (a[j] > a[k])
      then swap(a[j], a[k])
           j ← k
      else heap ← true
end
```

`maxHeap` - delete

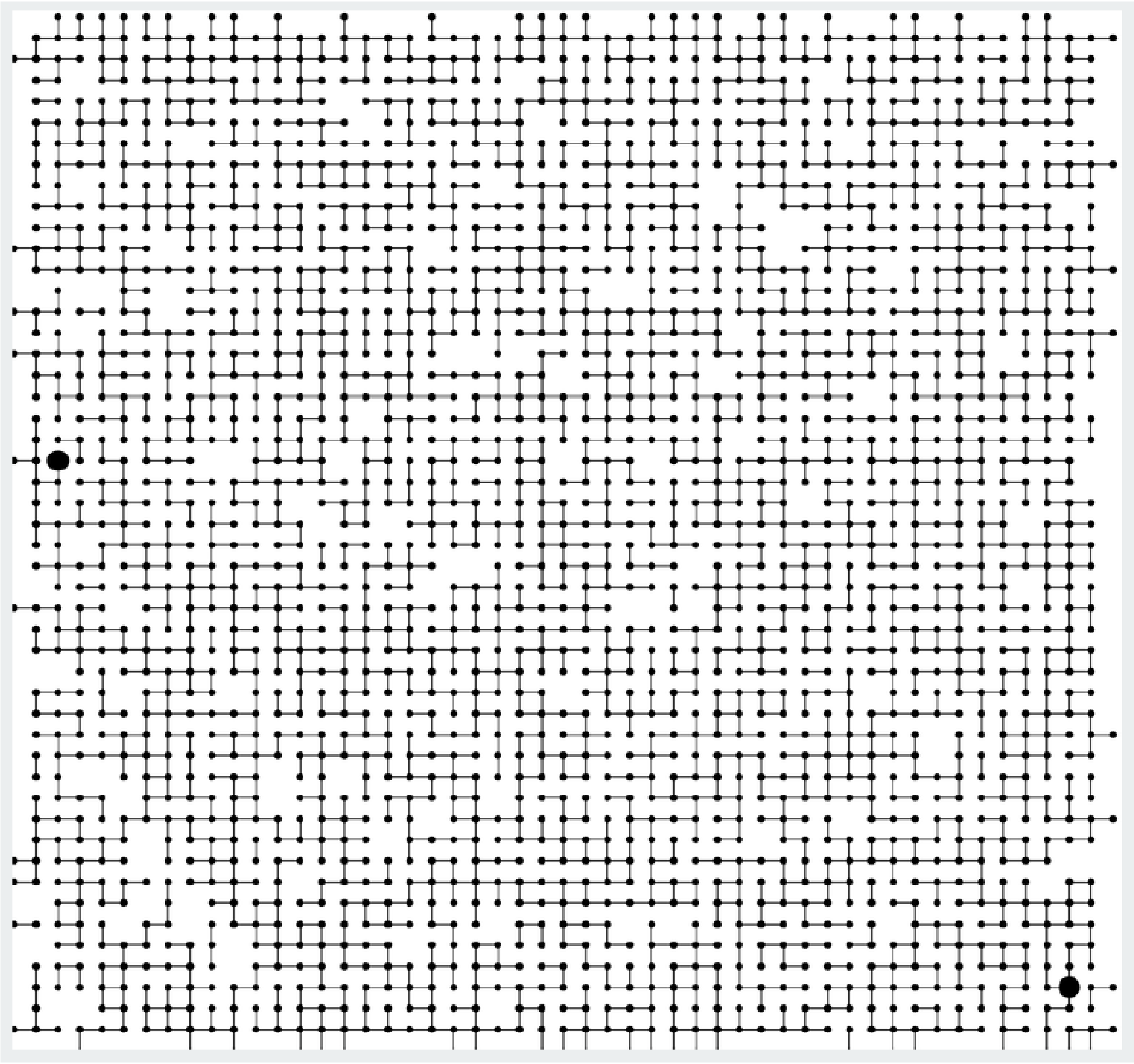
```
procedure delete(a, n)
begin
    a[0] ← a[n-1]
    n ← n-1
    j ← 0
    heap ← false
    while ((2*j+1 < n) and not heap) do
        k ← 2*j+1
        if ((k < n-1) and (a[k] < a[k+1]))
        then  k ← k+1
        if (a[j] < a[k])
        then  swap(a[j], a[k])
              j ← k
        else  heap ← true
end
```

`maxHeap`: execution time

- The insert/delete operations require the time $O(h) = O(\log n)$



Disjoint set collections



Applications:

- Computer networks
- Web pages(Internet)
- Pixels in a digital image

Disjoint sets collections: abstract data type

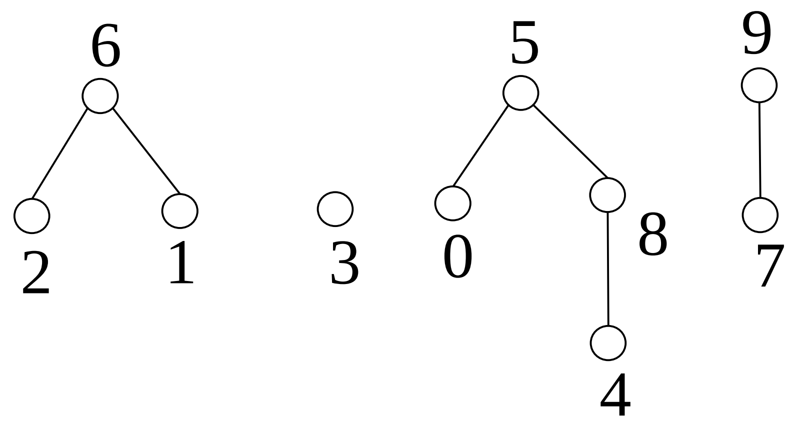
- objects: disjoint sets collections (partitions) of a universe set
- operations:
  - find()
    - input: a collection C, an element i from the universe set
    - output: the subset of C to which i belongs
  - union()
    - input: a collection C, two elements i and j from universe
    - output: C where the components of a i and j are joint
  - singleton()
    - input: a collection C, an element i from universe
    - output: C where the component of i has i as unique element.

Disjoint set collections: “`union-find`”

- The “union-find” structure
  - universe set = {0,1, ..., n-1}
  - subset = tree
  - collection = forest
  - Forest reprezentation by parent link

Disjoint set collections: “**`union-find`**”

- examples:
  - n=10, {1,2,6},{3}, {0,4,5,8}, {7,9}



	0	1	2	3	4	5	6	7	8	9
parent	5	6	6	-1	8	-1	-1	9	5	-1

Disjoint set collections: “**union-find**”

```
procedure singleton(C, i)
begin
  C.parent[i] ← -1
end
```

Disjoint sets collections: “**union-find**”

```
function find(C, i)
begin
  temp ← i
  while (C.parent[temp] >= 0) do
    temp ← C.parent[temp]
  return temp
end
```

Disjoint sets collections: "`union-find`"

```
procedure union(C, i, j)
begin
   ri ← find(i)
   rj ← find(j)
   if (ri != rj) then C.parent[rj]
     ← ri
end
```

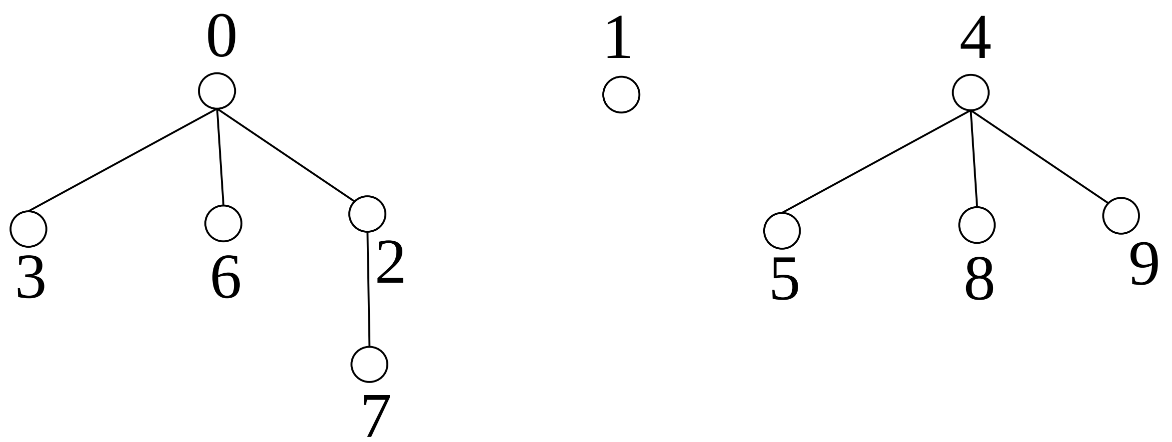
Balanced “union-find” structure

- Solution for the degenerated trees problem.

- Mechanism:
  - Store the number of node of the tree (with negative sign).
  - Tree flatting.

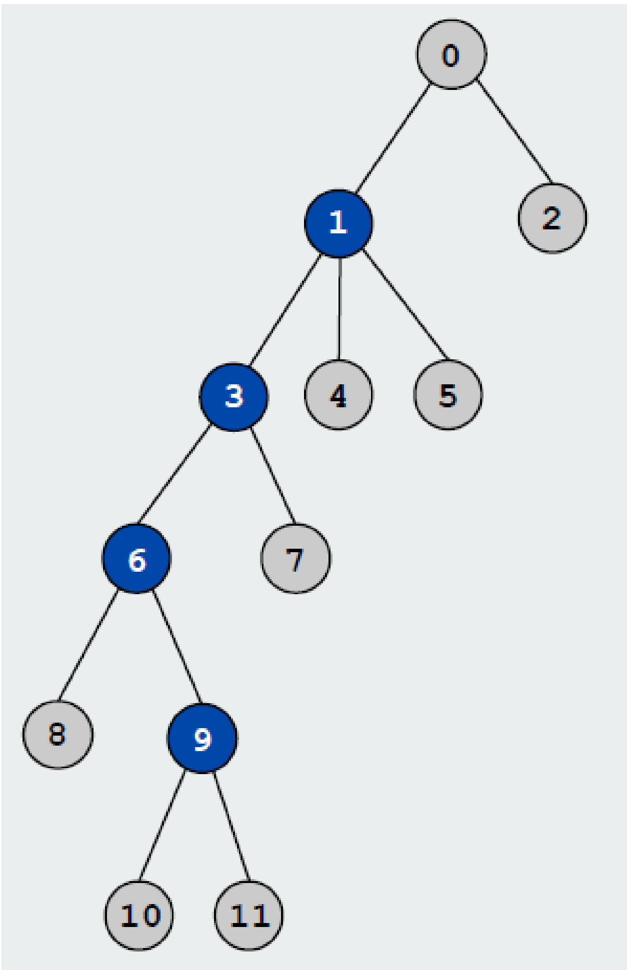
Balanced "union-find" structure

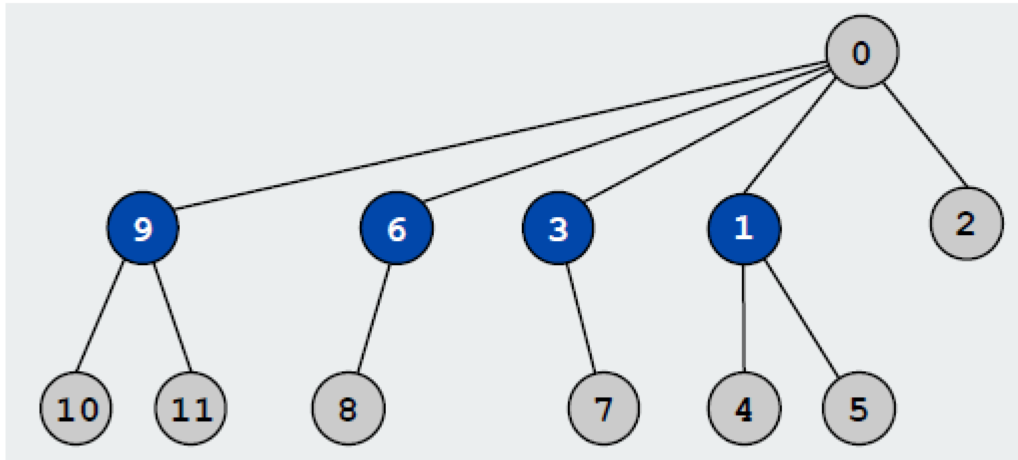
- example:



parent	-5	-1	0	0	-4	4	0	2	4	4
	0	1	2	3	4	5	6	7	8	9

Tree flating





find(9)

Balanced "union-find" structure

```
procedure union(C, i, j)
begin
   ri ← find(i); rj ← find(j)
   while (C.parent[i] >= 0) do
      temp ← i; i ← C.parent[i]; C.parent[temp]← ri
   while (C.parent[j] >= 0) do
      temp ← j; j ← C.parent[j]; C.parent[temp]← rj

   if (C.parent[ri] > C.parent[rj])
     then C.parent[rj] ← C.parent[ri]+C.parent[rj]
          C.parent[ri] ← rj
     else C.parent[ri] ← C.parent[ri]+C.parent[rj]
          C.parent[rj] ← ri
end
```

Balanced "union-find" structure

- *Theorem: Starting from an empty collection, any sequence of M operations "union" and "find" over N elements has complexity $O(N+M\ lg^*N)$.*
  - lg^*N = number of logarithms until 1 is obtained.