Priority queues

Mădălina Răschip, Cristian Gațu

Faculty of Computer Science
"Alexandru Ioan Cuza" University of Iași, Romania

DS 2020/2021

Content

Priority queues and "max-heap"

Disjoint set collections and "union-find"

Priority queues - examples

- Air passengers

  Priorities:

  – business-class;

  – persons traveling with children / with reduced mobility;

  – other passengers.

Priority queues - examples

- Air passengers

  Priorities:

  - business-class;
  - persons traveling with children / with reduced mobility;
  - other passengers.

- Planes approaching the airport

  Priorities:

  - emergencies;
  - fuel level;
  - distance to the airport.

Priority queues: abstract data type

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

- Elements are stores based on of their priorities and not their positions.

Priority queues – operations

- `read`
  - input: priority queue C
  - output: the atom from C with the highest priority.

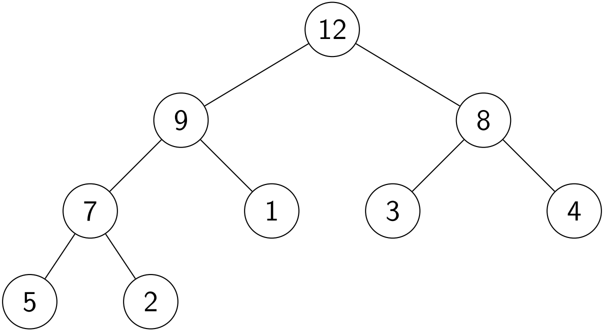
- `delete`
  - input: priority queue C
  - output: C from which the atom with the highest priority has been deleted.

- `insert`
  - input: priority queue C and some 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 complete. Let h be the tree height. Then,
    - for $i = 0, \cdots, h-1$, there are 2^i nodes of height i;
    - On level $h-1$ the internal nodes are on the left of the 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 $O(\log_2 n)$.*

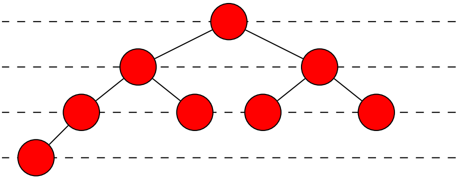
`maxHeap` height

Theorem

A `maxHeap` with n keys has the height $O(\log_2 n)$.

Demonstrație.

- The complete binary tree properties are used.
- Let h be the height of a `maxHeap` with n keys.
- There are 2^i keys of depth i, for $i = 0, \cdots, h-1$ and at least one key of depth h: $\Rightarrow n \geq 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 1 = 2^h$.
- Thus: $h \leq \log_2 n$



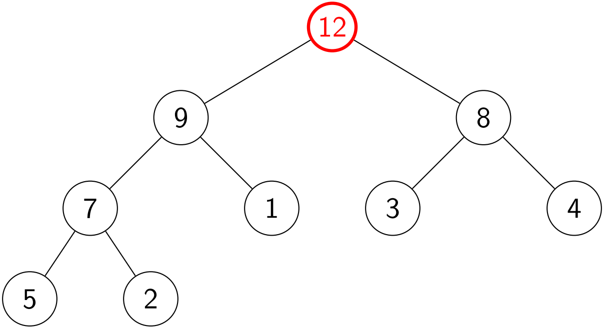
level	keys
0	2^0
1	2^1
$h-1$	2^{h-1}
h	1

□

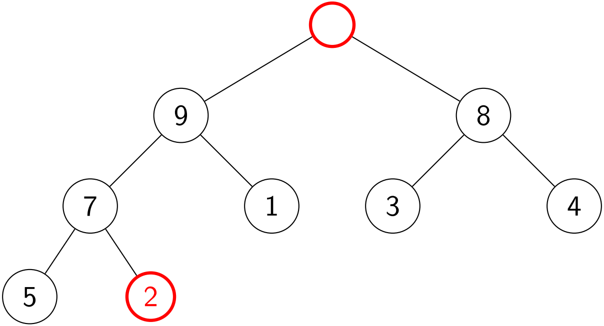
- The heap root is deleted
  (it corresponds to the atom with the highest priority).

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

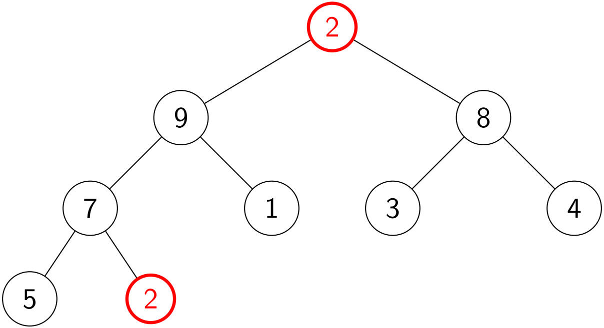
maxHeap: delete – example



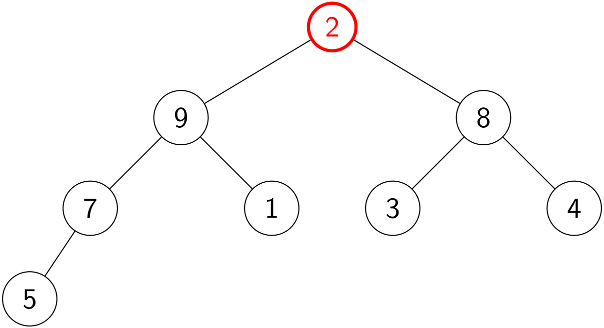
maxHeap: delete – example



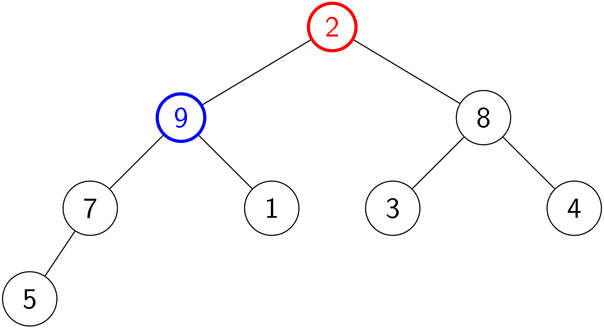
`maxHeap`: delete – example



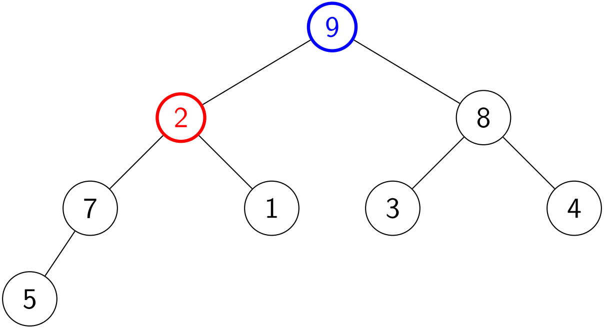
maxHeap: delete – example



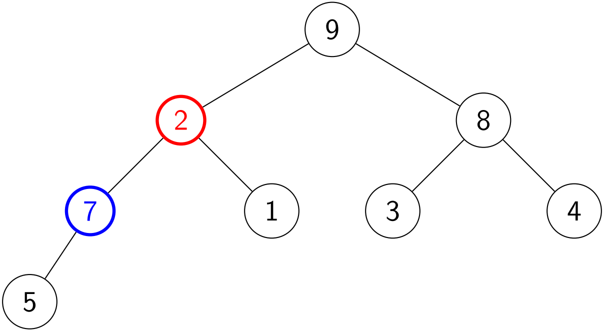
`maxHeap`: delete – example



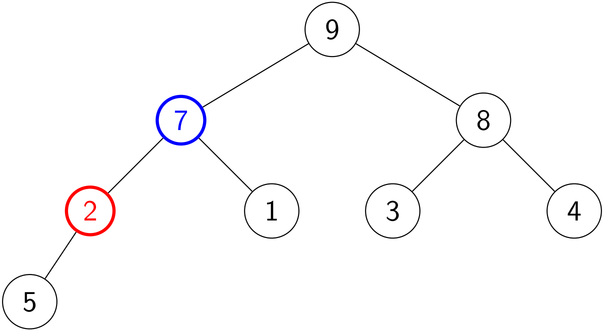
`maxHeap`: delete – example



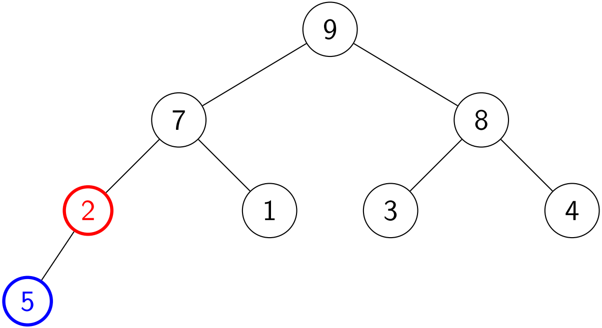
maxHeap: delete – example



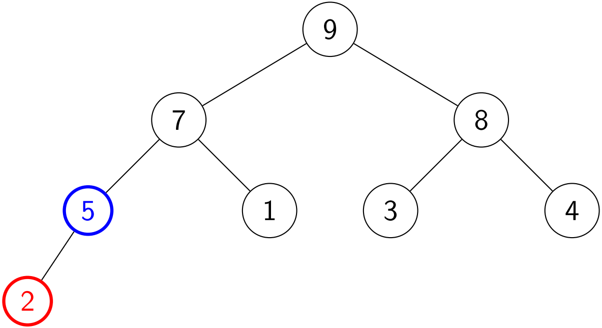
`maxHeap`: delete – example



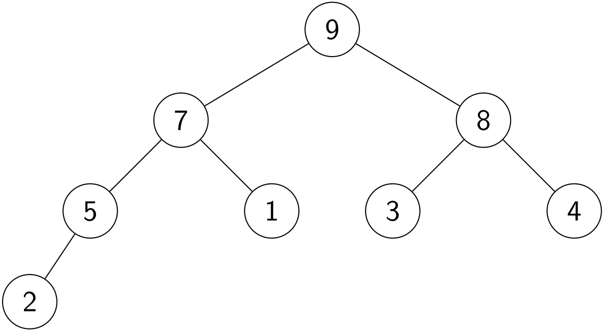
`maxHeap`: delete – example



`maxHeap`: delete – example



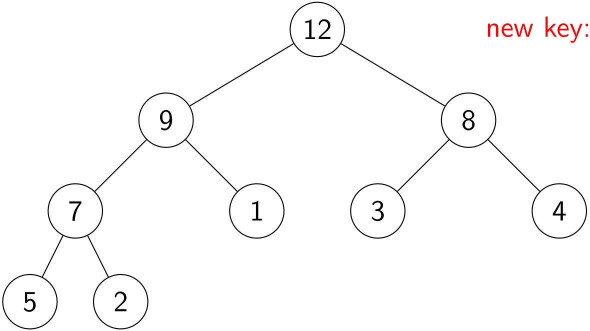
`maxHeap`: delete – example



maxHeap: insert

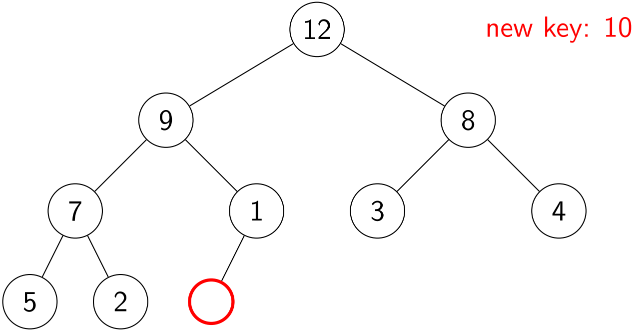
- The new key is inserted in a new node.
- The algorithm has three stages:
  - A new node is added as the rightmost node on the last level;
  - The new key is inserted in this node;
  - The maxHeap property is repaired.

maxHeap: insert – example

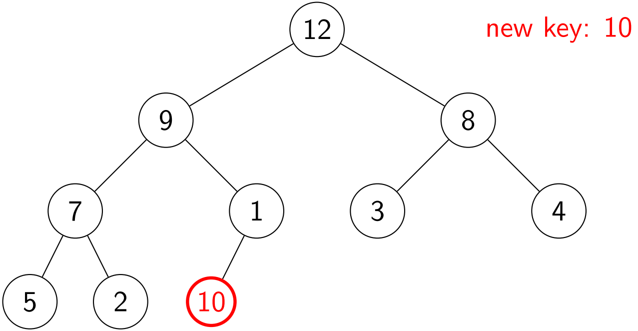


new key: 10

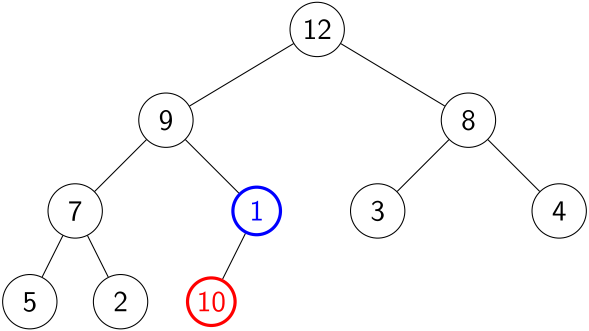
maxHeap: insert – example



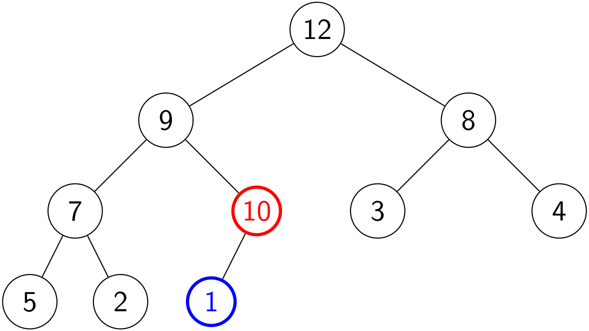
`maxHeap`: insert – example



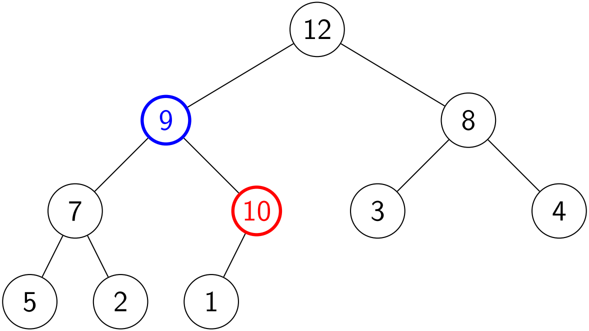
`maxHeap`: insert – example



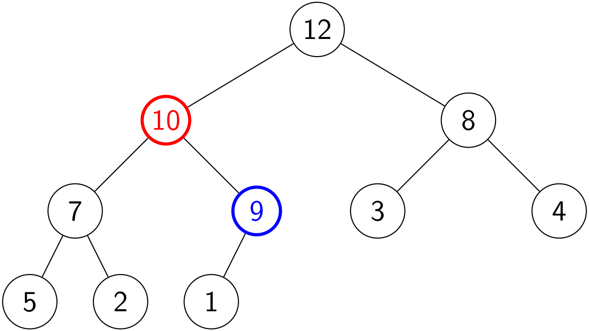
`maxHeap`: insert – example



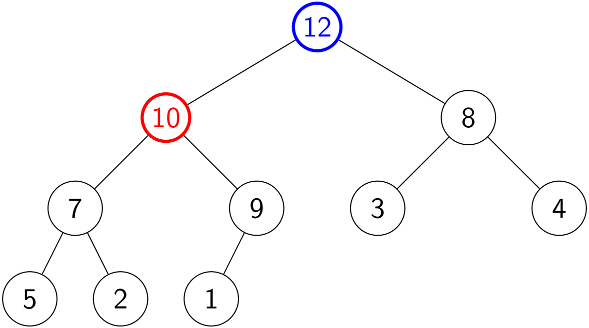
`maxHeap`: insert – example



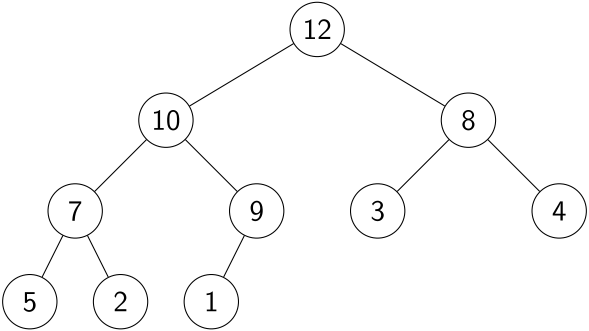
maxHeap: insert – example



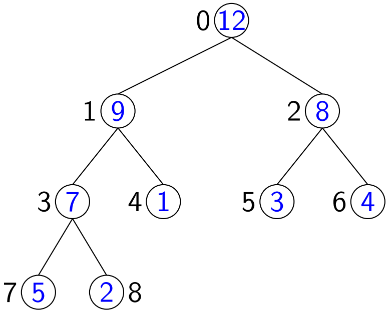
maxHeap: insert – example

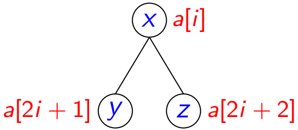


`maxHeap`: insert – example



maxHeap: array implementation





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

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

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
```

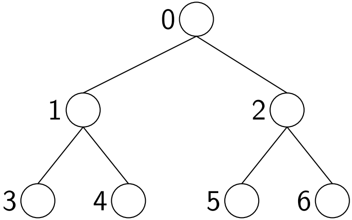
maxHeap: delete

```
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

$$O(h) = O(\log n).$$

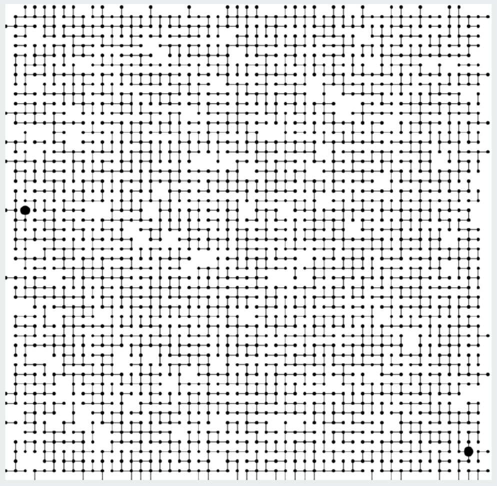


Content

Priority queues and "max-heap"

Disjoint set collections and "union-find"

Disjoint set collections



Applications:

- computer networks;
- web pages(Internet);
- pixels in a digital image.

Disjoint sets collection: abstract data type

- OBJECTS:
  disjoint sets collections (partitions) of some *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 the *universe* set;
    - output: C where the components (subsets) of i and j are joint.
  - `singleton`
    - input: a collection C, an element i from the *universe* set;
    - output: C where the component of i has i as unique element.

Disjoint set collections: `union-find`

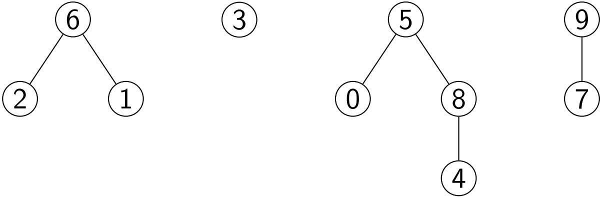
- The `union-find` structure:
  - universe set $\{0, 1, \cdots, n-1\}$;
  - a subset is given by a tree;
  - a collection (partition) is a tree collection ("forest");
  - a "forest" is represented through the "parent" link.

union-find: example

- $n = 10, \quad C = \{\{1,2,6\},\{3\},\{0,4,5,8\},\{7,9\}\}$

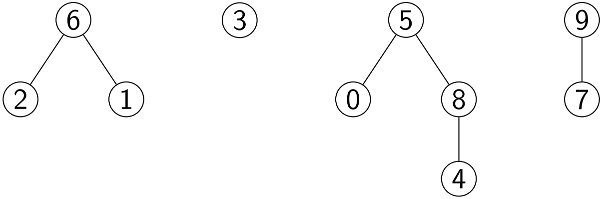
union-find: example

- $n = 10, \quad C = \{\{1,2,6\},\{3\},\{0,4,5,8\},\{7,9\}\}$



`union-find`: example

- $n = 10, \quad C = \{\{1, 2, 6\}, \{3\}, \{0, 4, 5, 8\}, \{7, 9\}\}$



parent

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

union-find: singleton

procedure *singleton(*C, i*)*
begin
    C.parent[i] $\leftarrow$ -1
end

union-find: find

procedure *find*(C, i)
begin

    temp ← i
    while (C.parent[temp] >= 0) **do**
        temp ← C.parent[temp]
    return temp

end

union-find: union

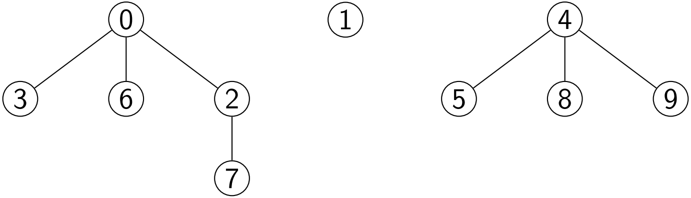
```
procedure union(C, i, j)
begin
    ri ← find(i)
    rj ← find(j)
    if ri != rj then
        C.parent[rj] ← ri
end
```

Balanced `union-find` structure

- Solution for the degenerated trees problem.
- Mechanism:
  - Store the number of nodes of the tree (with negative sign).
  - Tree flatting.

Balanced `union-find` structure: example

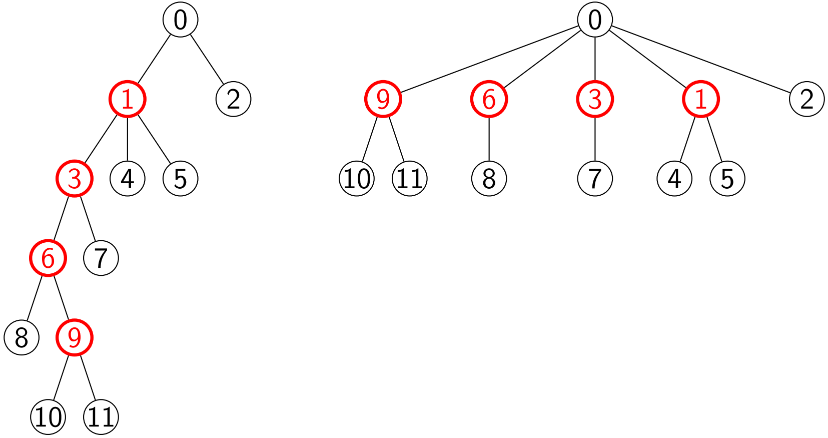
- $n = 10, \quad C = \{\{0, 2, 3, 6, 7\}, \{1\}, \{4, 5, 8, 9\}\}$



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

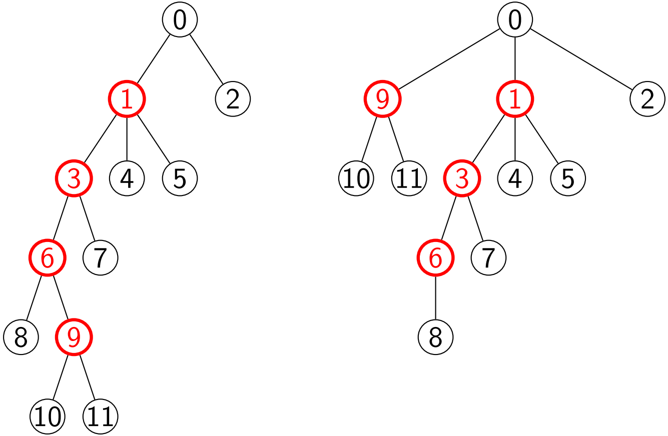
Tree flatting – example

- `find(9)`



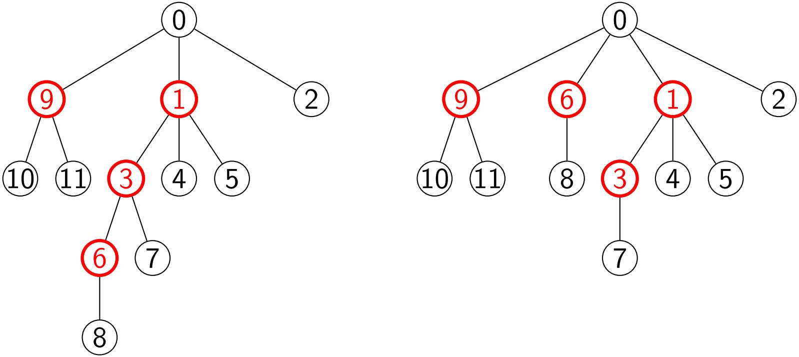
Tree flatting – example

- `find(9)`



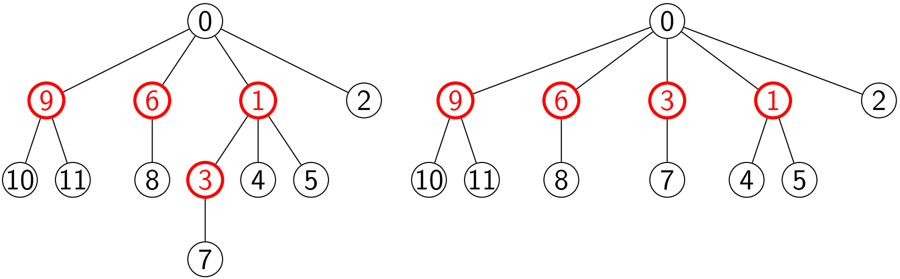
Tree flatting – example

- `find(9)`



Tree flatting – example

- `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 `union` and `find` operations over n elements has the time complexity $O(n + m \log^* n)$.

Note: $\log^* n$ is the number of logarithm applications until value 1 is obtained.