Binary Trees, Heaps

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

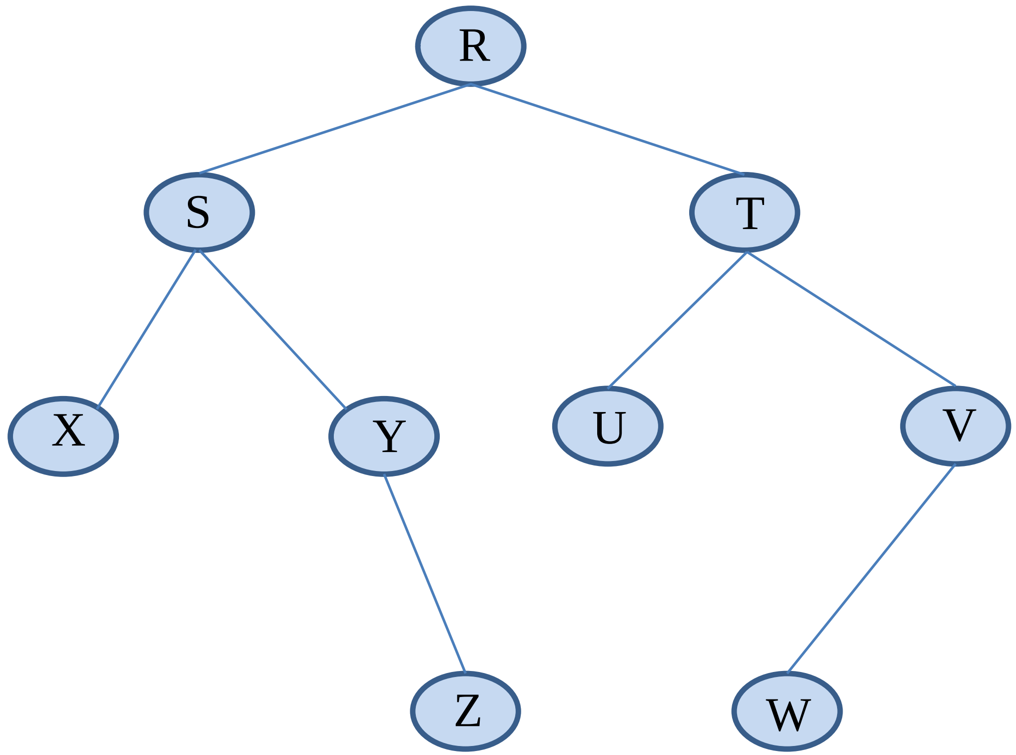
Κώστας Χατζηκοκολάκης

Binary trees

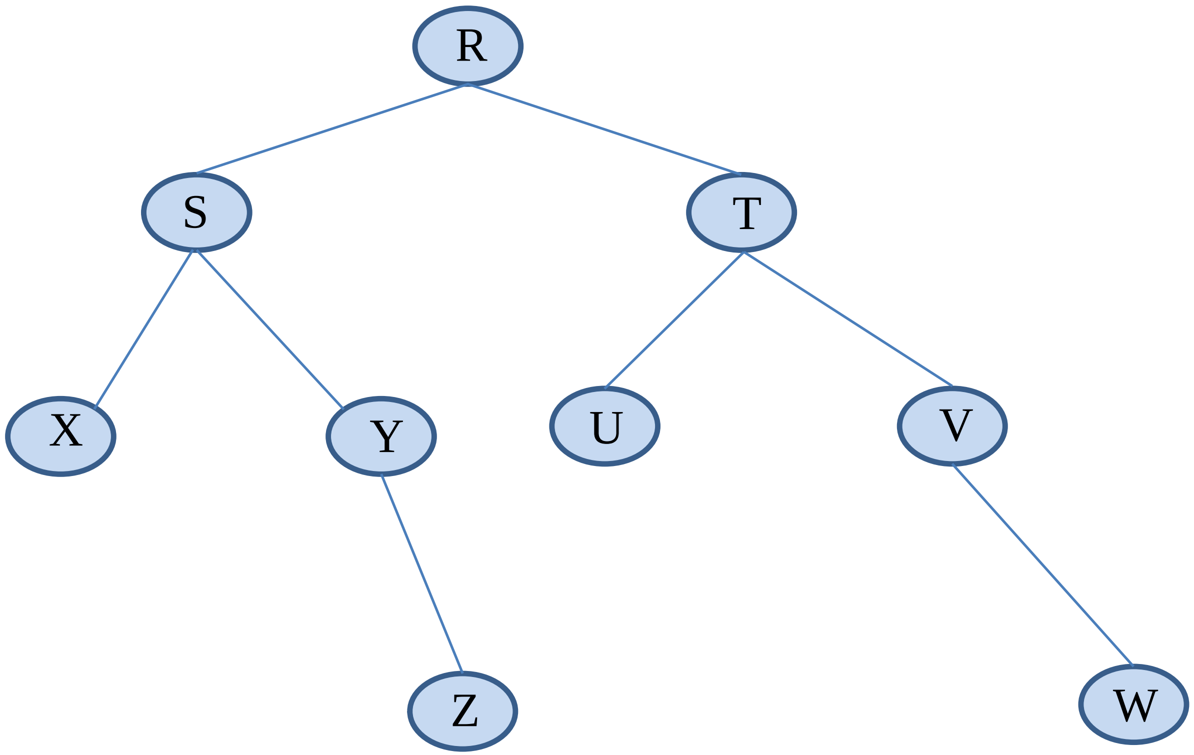
A **binary tree (δυαδικό δέντρο)** is a set of nodes such that:

- Exactly one node is called the **root**
- All nodes except the root have **exactly one parent**
- Each node has **at most two children**
    - and the are **ordered**: called **left** and **right**

Example: a binary tree



Example: a different binary tree



Whether a child is left or right matters.

Terminology

- **path**: sequence of nodes traversing from parent to child (or vice-versa)
- **length** of a path: number of nodes -1 (= number of “moves” it contains)
- **siblings**: children of the same parent
- **descendants**: nodes reached by travelling downwards along any path
- **ancestors**: nodes reached by travelling upwards towards the root
- **leaf / external node**: a node without children
- **internal node**: a node with children

Terminology

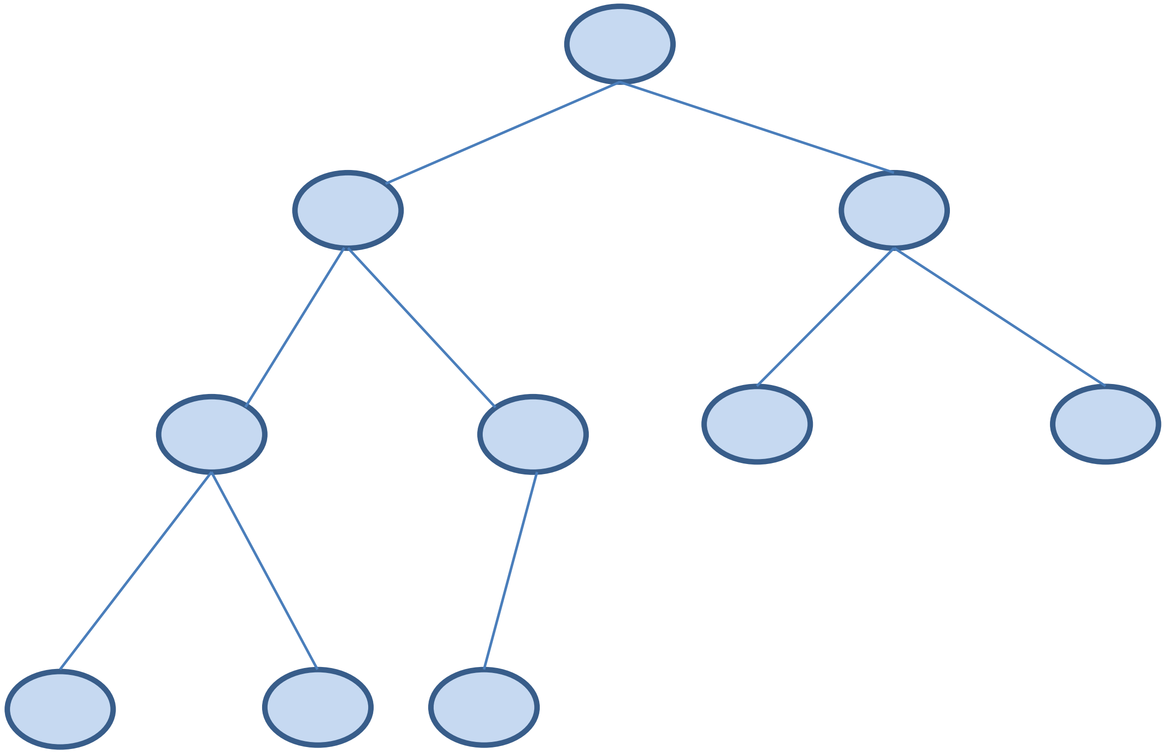
- Nodes tree can be arranged in **levels / depths**:
  - The root is at **level 0**
  - Its children are at **level 1**, their children are at **level 2**, etc.
- Note: node level = length of the (unique) path from the root to that node
- **height** of the tree: the largest depth of any node
- **subtree** rooted at a node: the tree consisting of that node and its descendants

Complete binary trees

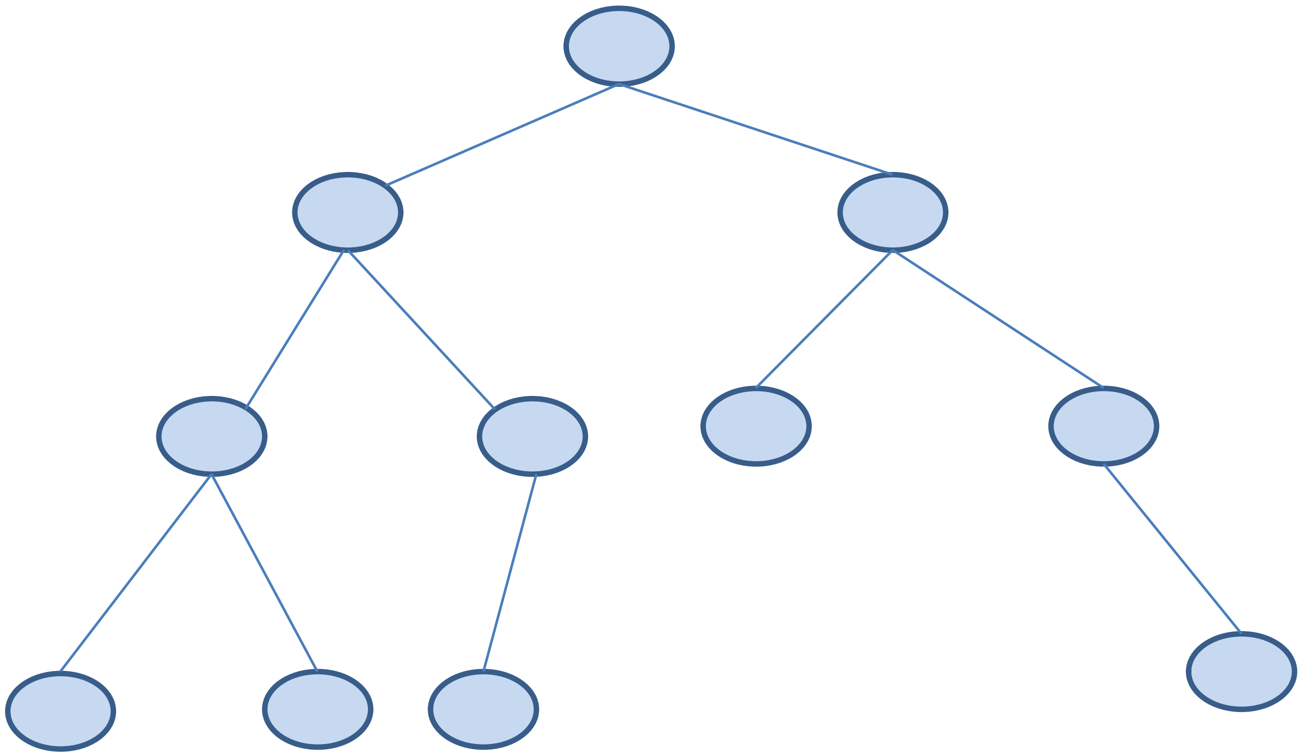
A binary tree is called **complete** (πλήρες) if

- All levels except the last are **“full”** (have the maximum number of nodes)
- The nodes at the last level fill the level “from left to right”

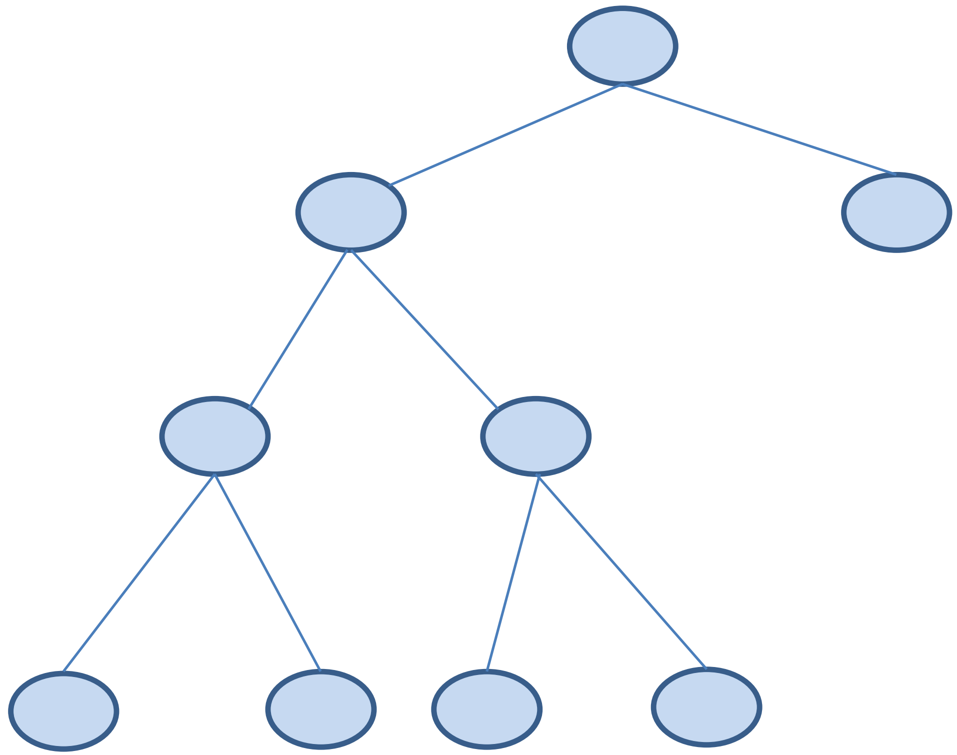
Example: complete binary tree



Example: not complete binary tree

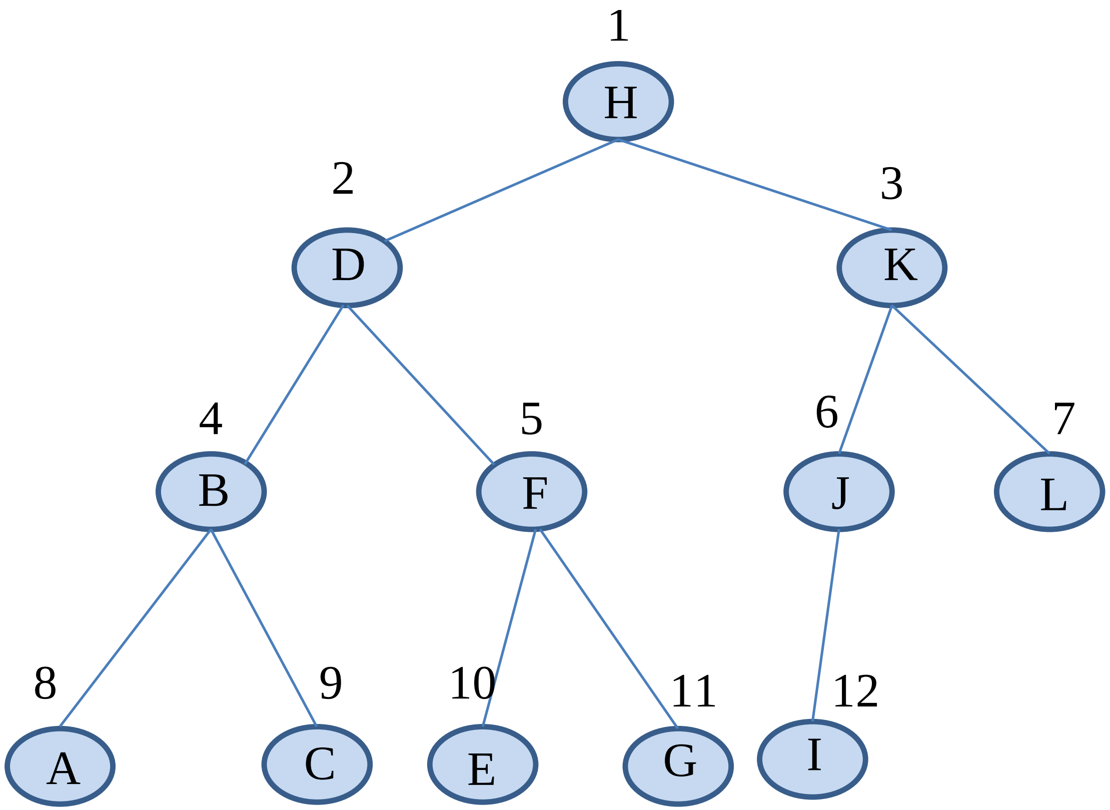


Example: not complete binary tree



Level order

Ordering the nodes of a tree **level-by-level** (and left-to-right in each level).



Nodes of a complete binary tree

- How many nodes does a complete binary tree have at each level?
- At most
  - 1 at level 0.
  - 2 at level 1.
  - 4 at level 2.
  - $\ldots$
  - 2^k at level k .

Properties of binary trees

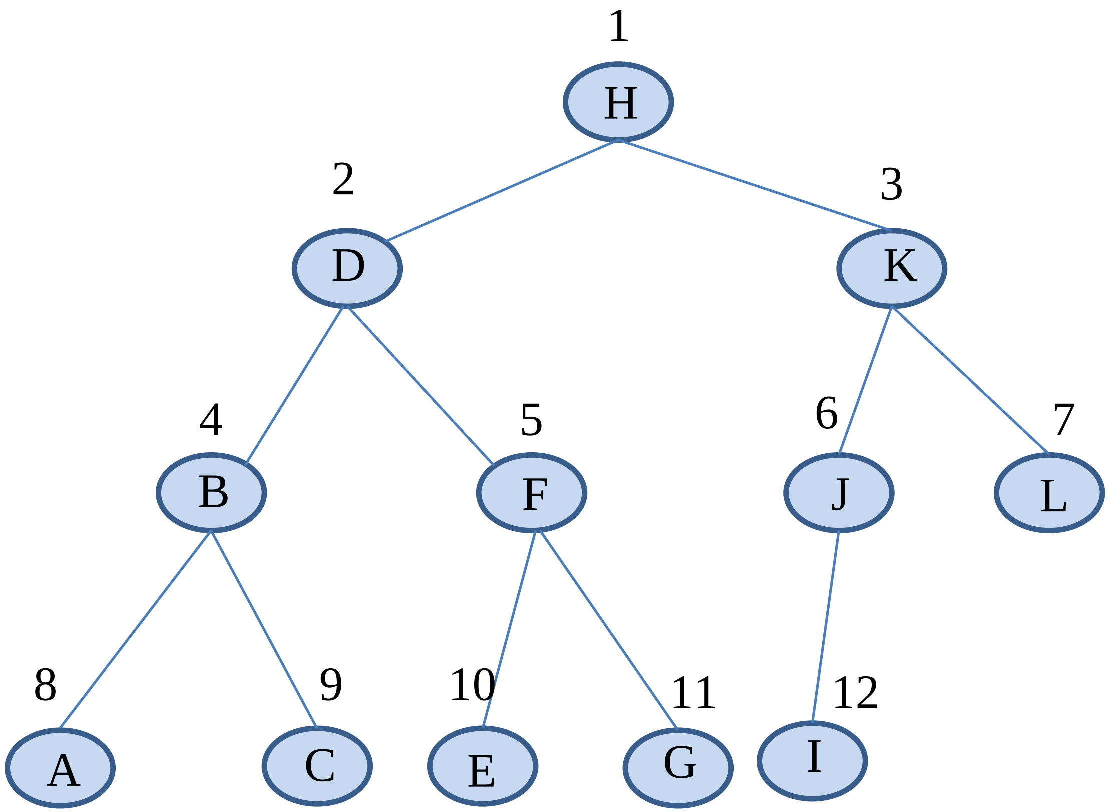
- The following hold:
  - $h + 1 \leq n \leq 2^{h+1} - 1$
  - $1 \leq n_E \leq 2^h$
  - $h \leq n_I \leq 2^h - 1$
  - $\log(n+1) - 1 \leq h \leq n - 1$
- Where
  - n: number of all nodes
  - n_I: number of internal nodes
  - n_E: number of external nodes (leaves)
  - h: height

Properties of complete binary trees

$h \le \log n$

- Very important property, the tree cannot be too "tall"!
- Why?
  - Any level $l < h$ contains exactly 2^l nodes
  - Level h contains at least one node
  - So $1 + 2 + \ldots + 2^{h-1} + 1 = 2^h \le n$
  - And take logarithms on both sides

How do we represent a binary tree?



Sequential representation

Store the entries in an **array** at **level order**.

A:		H	D	K	B	F	J	L	A	C	E	G	I
		1	2	3	4	5	6	7	8	9	10	11	12

- Common for **complete trees**
- A lot of **space** is wasted for non-complete trees
  - missing nodes will have empty slots in the array

How to find nodes

To Find:	Use	Provided
The left child of $A[i]$	$A[2i]$	$2i \leq n$
The right child of $A[i]$	$A[2i+1]$	$2i+1 \leq n$
The parent of $A[i]$	$A[i/2]$	$i > 1$
The root	$A[1]$	A is nonempty
Whether $A[i]$ is a leaf		$2i > n$

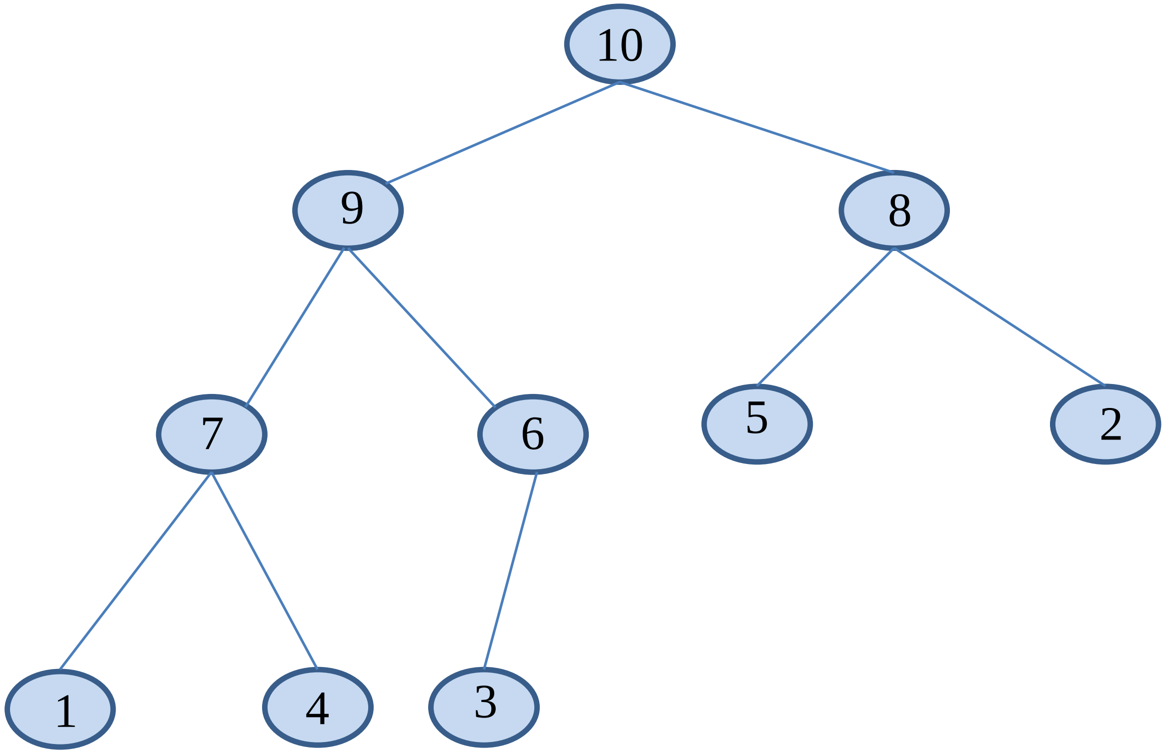
Heaps

A binary tree is called a **heap** (σωρός) if

- It is **complete**, and
- each node is **greater or equal than its children**

(Sometimes this is called a **max-heap**, we can similarly define a min-heap)

Example



Heaps and priority queues

- Heaps are a common data structure for implementing **Priority Queues**
- The following operations are needed
  - find max
  - insert
  - remove max
  - create with data
- We need to **preserve the heap property** in each operation!

Find max

- Trivial, the max is always **at the root**
  - remember: we always preserve the heap property
- Complexity?

Inserting a new element

- The new element can only be inserted at the **end**
  - because a heap must be a **complete** tree
- Now all nodes **except the last** satisfy the heap property
  - to restore it: apply the `bubble_up` algorithm on the last node

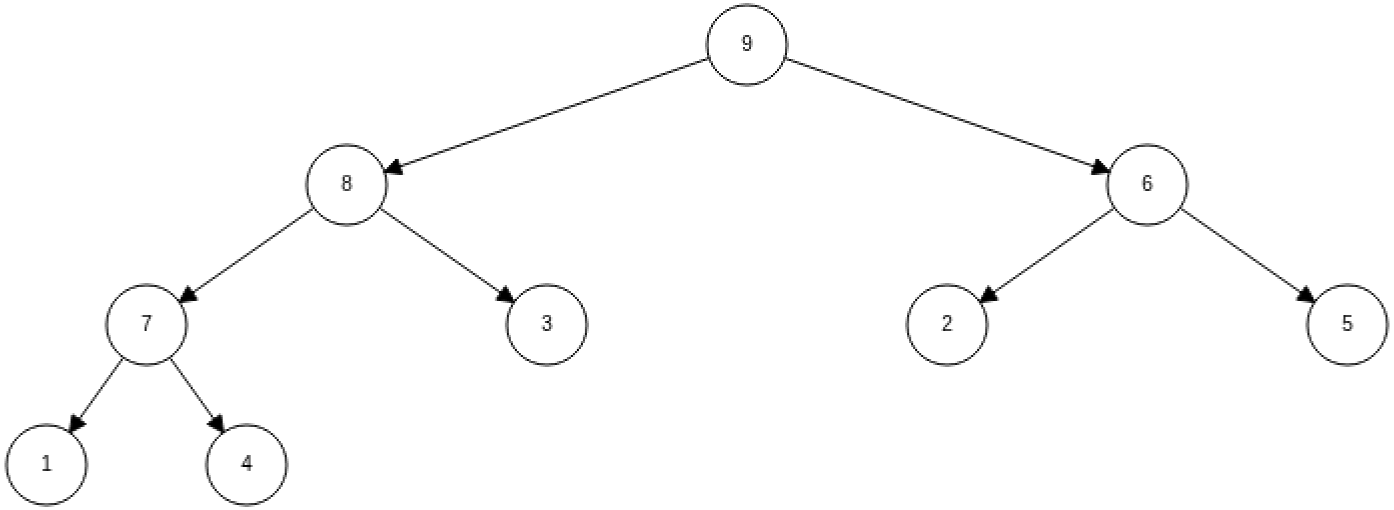
Inserting a new element

`bubble_up(node)`

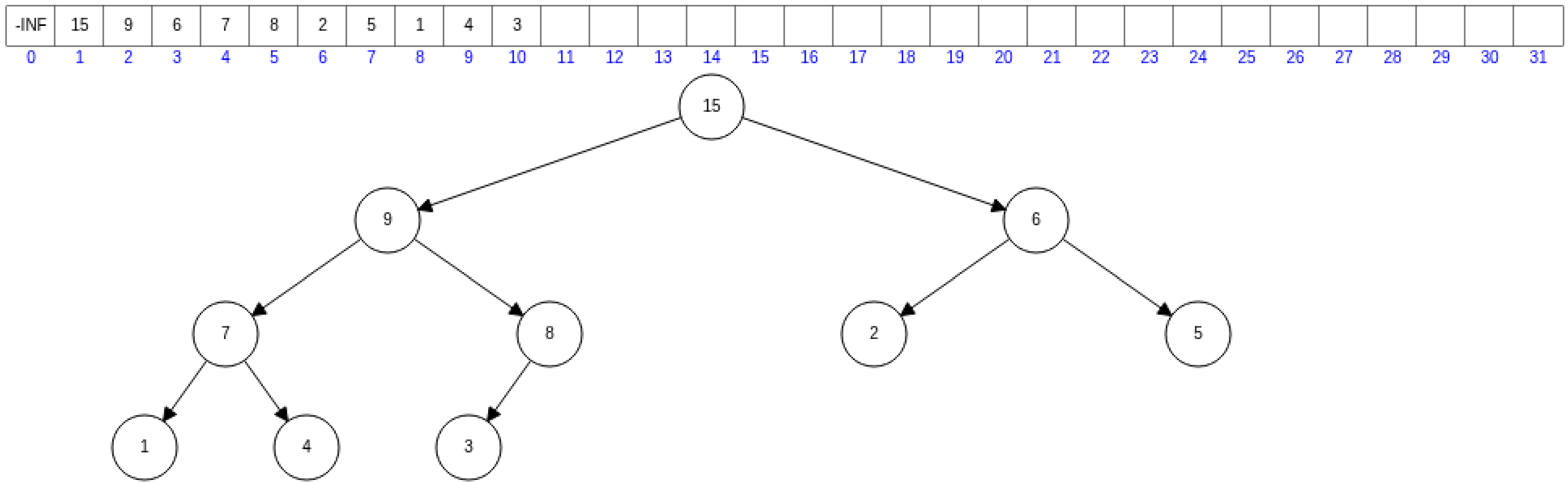
- **Before**
  - `node` might be **larger** than its parent
  - all other nodes satisfy the heap property
- **After**
  - all nodes satisfy the heap property
- **Algorithm**
  - if `node > parent`
    - **swap them** and call `bubble_up(parent)`

Example insertion

-INF	9	8	6	7	3	2	5	1	4																						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

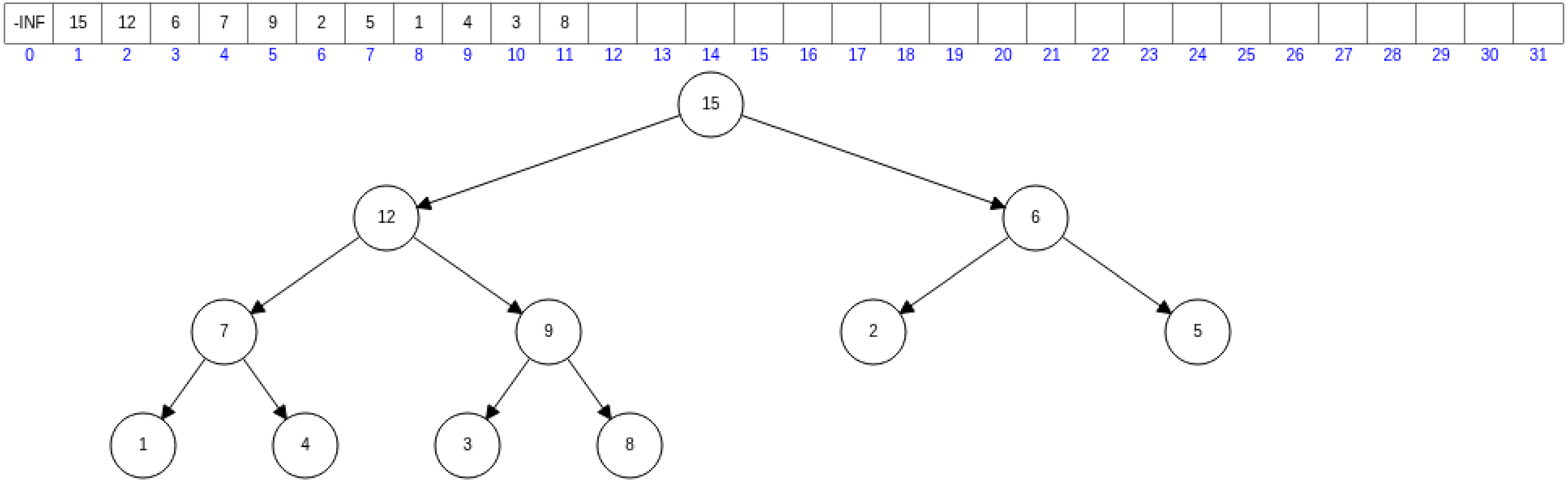


Example insertion



Inserting 15 and running bubble_up

Example insertion



Inserting 12 and running bubble_up

Complexity of insertion

- We travel the tree from the last node to the root
  - on each node: 1 step (constant time)
- So we need at most $O(h)$ steps
  - h is the height of the tree
  - but $h \leq \log n$ on a **complete tree**
- So $O(\log n)$
  - the “complete” property is crucial!

Removing the max element

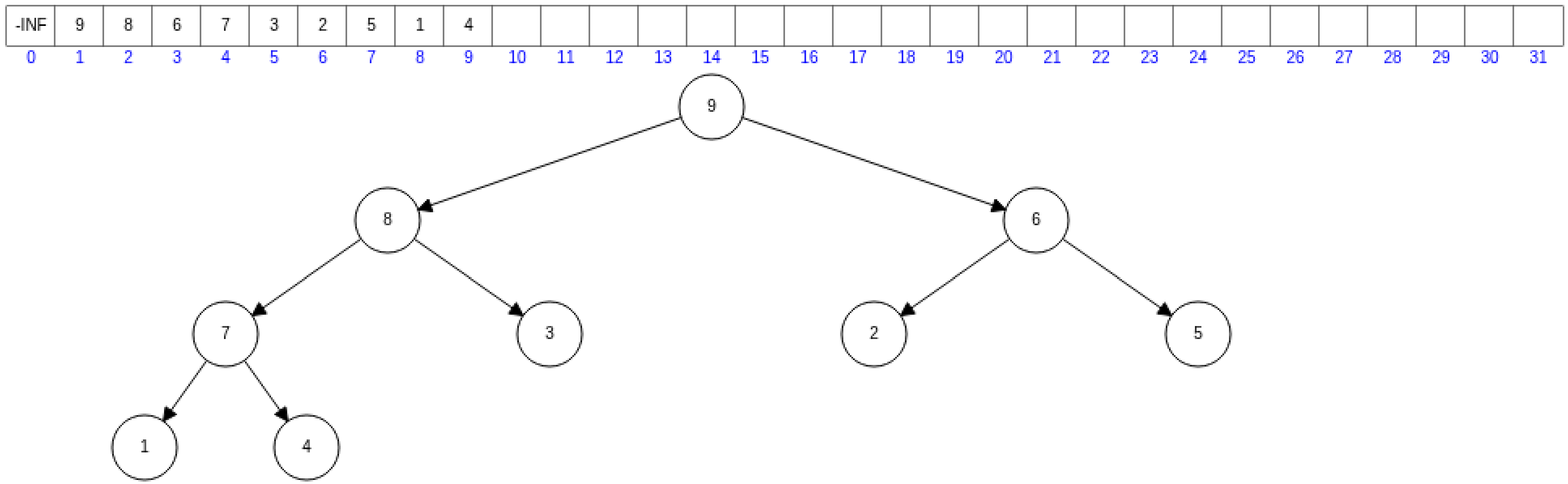
- We want to remove the root
  - but the heap must be a **complete** tree
- So **swap** the root with the **last** element
  - then remove the last element
- Now all nodes **except the root** satisfy the heap property
  - to restore it: apply the `bubble_down` algorithm on the root

Removing the max element

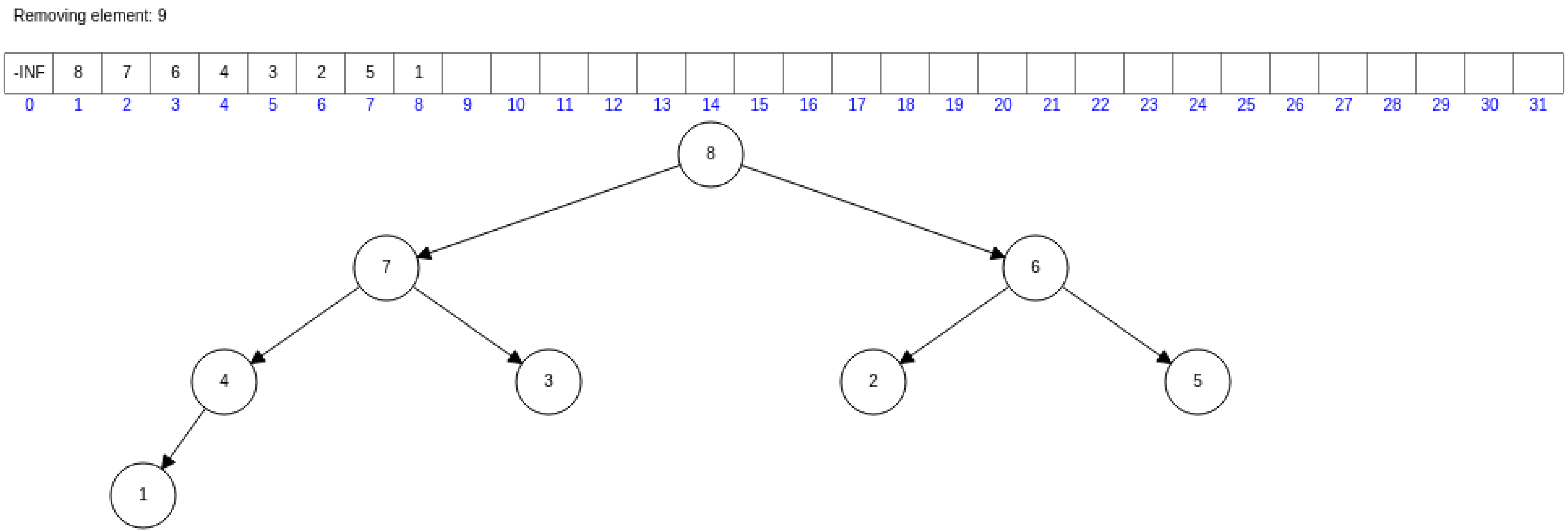
`bubble_down(node)`

- **Before**
  - `node` might be **smaller** than any of its children
  - all other nodes satisfy the heap property
- **After**
  - all nodes satisfy the heap property
- **Algorithm**
  - `max_child` = the **largest child** of `node`
  - If `node < max_child`
    - **swap them** and call `bubble_down(max_child)`

Example removal



Example removal



Removing 9 and restoring the heap property

Complexity of removal

- We travel a single path from the root to a leaf
- So we need at most $O(h)$ steps
  - h is the height of the tree
- Again $O(\log n)$
  - again, having a complete tree is crucial

Building a heap from initial data

- What if we want to create a heap that contains some **initial values**?
  - we call this operation **heapify**
- “Naive” implementation:
  - Create an empty heap and **insert elements one by one**
- What is the complexity of this implementation?
  - We do n inserts
  - Each insert is $O(\log n)$ (because of `bubble_up`)
  - So $O(n \log n)$ total
- Worst-case example?
  - sorted elements: each value with have to fully `bubble_up` to the root

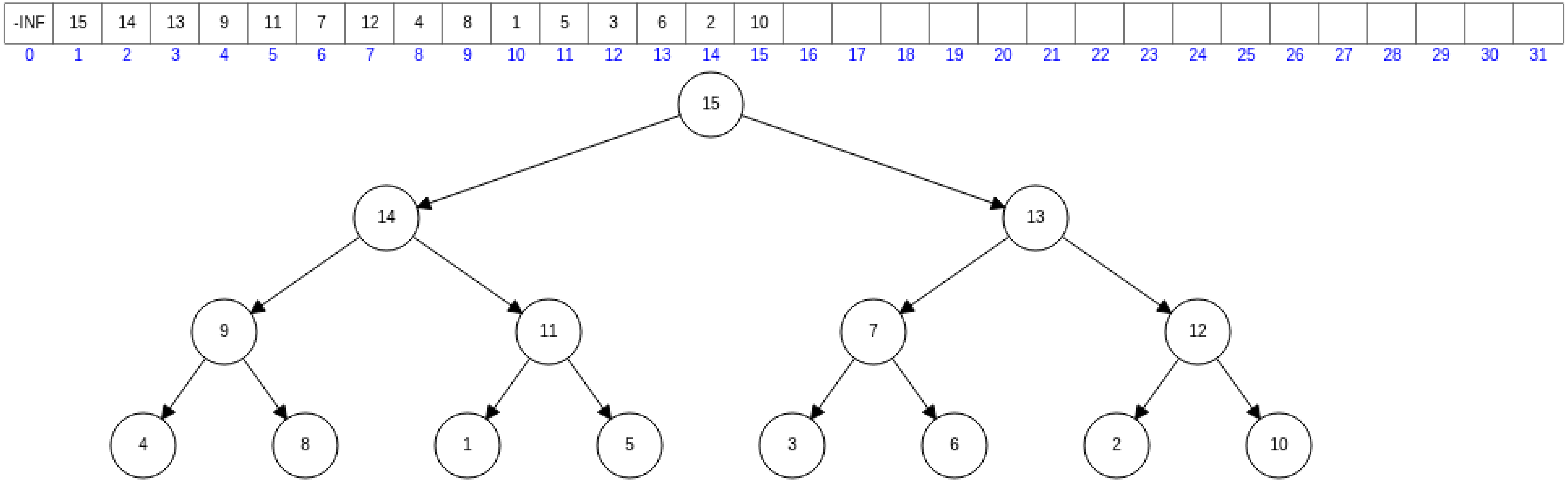
Efficient heapify

- Better algorithm:
  - Visit all **internal nodes** in **reverse level order**
    - last internal node: $\frac{n}{2}$ (parent of the last leaf n)
    - first internal node: 1 (root)
  - Call `bubble_down` on each visited `node`
- Why does this work?
  - when we visit `node`, its **subtree is already a heap**
    - except from `node` **itself** (the precondition of `bubble_down`)
  - So `bubble_down` restores the heap property **in the subtree**
  - After processing the root, the whole tree is a heap

Heapify example

-INF																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Heapify example



Visit internal nodes in inverse level order, call bubble_down.

Complexity of heapify

- We call `bubble_down` $\frac{n}{2}$ times
  - So $O(n \log n)$?
- But this is only an **upper-bound**
  - `bubble_down` is faster **closer to the leaves**
  - and **most nodes** live there!
  - we might be over-approximating the number of steps

Complexity of heapify

- More careful calculation of the number of steps:
  - If `node` is at level l, `bubble_down` takes at most $h - l$ steps
  - At most 2^l nodes at this level, so $(h-l)2^l$ steps for level l
  - For the whole tree: $\sum_{l=0}^{h-1}(h-l)2^l$
  - This can be shown to be less than $2n$ (exercise if you're curious)
- So we get worst-case $O(n)$ complexity

Efficient vs naive heapify

- For `naive_heapify` we found $O(n \log n)$
  - maybe we are also over-approximating?
- No: in the worst-case (sorted elements) we really need $n \log n$ steps
  - try to compute the exact number of steps
- The difference:
  - `bubble_up` is faster closer to the **root**, but **few** nodes live there
  - `bubble_down` is faster closer to the **leaves**, and **most** nodes live there
- Note: in the **average-case**, the naive version is also $O(n)$

Implementing ADTPriorityQueue

Types

```
// Ενα PriorityQueue είναι pointer σε αυτό το struct

struct priority_queue {
    Vector vector;              // Τα δεδομένα, σε Vector για μεταβλη
    CompareFunc compare;        // Η διάταξη
    DestroyFunc destroy_value;  // Συνάρτηση που καταστρέφει ένα στοι
};
```

ADTPriorityQueue implementation

Types.

```
// Ενα PriorityQueue είναι pointer σε αυτό το struct

struct priority_queue {
    Vector vector;              // Τα δεδομένα, σε Vector για μεταβλη
    CompareFunc compare;        // Η διάταξη
    DestroyFunc destroy_value;  // Συνάρτηση που καταστρέφει ένα στοι
};
```

ADTPriorityQueue implementation

Finding the max is trivial.

```
Pointer pqueue_max(PriorityQueue pqueue) {
    return node_value(pqueue, 1);       // root
}
```

ADTPriorityQueue implementation

For `pqueue_insert`, the non-trivial part is `bubble_up`.

```
// Αποκαθιστά την ιδιότητα του σωρού.
// Πριν: όλοι οι κόμβοι ικανοποιούν την ιδιότητα του σωρού, εκτός από
//       τον node που μπορεί να είναι _μεγαλύτερος_ από τον πατέρα το
// Μετά: όλοι οι κόμβοι ικανοποιούν την ιδιότητα του σωρού.

static void bubble_up(PriorityQueue pqueue, int node) {
    // Αν φτάσαμε στη ρίζα, σταματάμε
    if (node == 1)
        return;

    int parent = node / 2;      // Ο πατέρας του κόμβου. Τα node ids

    // Αν ο πατέρας έχει μικρότερη τιμή από τον κόμβο, swap και συνεχ
    if (pqueue->compare(node_value(pqueue, parent), node_value(pqueue
        node_swap(pqueue, parent, node);
        bubble_up(pqueue, parent);
    }
}
```

ADTPriorityQueue implementation

```
// Πριν: όλοι οι κόμβοι ικανοποιούν την ιδιότητα του σωρού, εκτός από
//       node που μπορεί να είναι _μικρότερος_ από κάποιο από τα παιδ
// Μετά: όλοι οι κόμβοι ικανοποιούν την ιδιότητα του σωρού.

static void bubble_down(PriorityQueue pqueue, int node) {
    // βρίσκουμε τα παιδιά του κόμβου (αν δεν υπάρχουν σταματάμε)
    int left_child = 2 * node;
    int right_child = left_child + 1;
    int size = pqueue_size(pqueue);
    if (left_child > size)
        return;

    // βρίσκουμε το μέγιστο από τα 2 παιδιά
    int max_child = left_child;
    if (right_child <= size && pqueue->compare(node_value(pqueue, lef
            max_child = right_child;

    // Αν ο κόμβος είναι μικρότερος από το μέγιστο παιδί, swap και συ
    if (pqueue->compare(node_value(pqueue, node), node_value(pqueue,
        node_swap(pqueue, node, max_child);
        bubble_down(pqueue, max_child);
    }
}
```

Other possible representations

Operation	Heap	Sorted List	Unsorted Vector
`pqueue_create` (with data)	$O(n)$	$O(n \log n)$	$O(1)$
`pqueue_remove`	$O(\log n)$	$O(1)$	$O(n)$
`pqueue_insert`	$O(\log n)$	$O(n)$	$O(1)$

All of them have **some** advantage

- Heaps provide a great compromise between insertions and removals

Using ADTPriorityQueue for sorting

- We can easily sort data using ADTPriorityQueue
  - create a priority queue with the data
  - remove elements in sorted order
- When ADTPriorityQueue is implemented by a **heap**
  - this algorithm is called **heapsort**
  - and runs in time $O(n \log n)$

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*. Chapter 9. Sections 9.1 to 9.6.

- R. Sedgewick. *Αλγόριθμοι σε C*. Κεφ. 5 και 9.

Proofs of given statements can be found in the following book:

- M. T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++*. 2nd edition. John Wiley and Sons, 2011.