

# Data Structures

## Fall 2018

# Heaps / Priority queues

Luis de Marcos Ortega

[luis.demarcos@uah.es](mailto:luis.demarcos@uah.es)

# Contents

- Heaps
- Implementations of heaps
- Heapsort
- Priority queues
- Implementations of priority queues

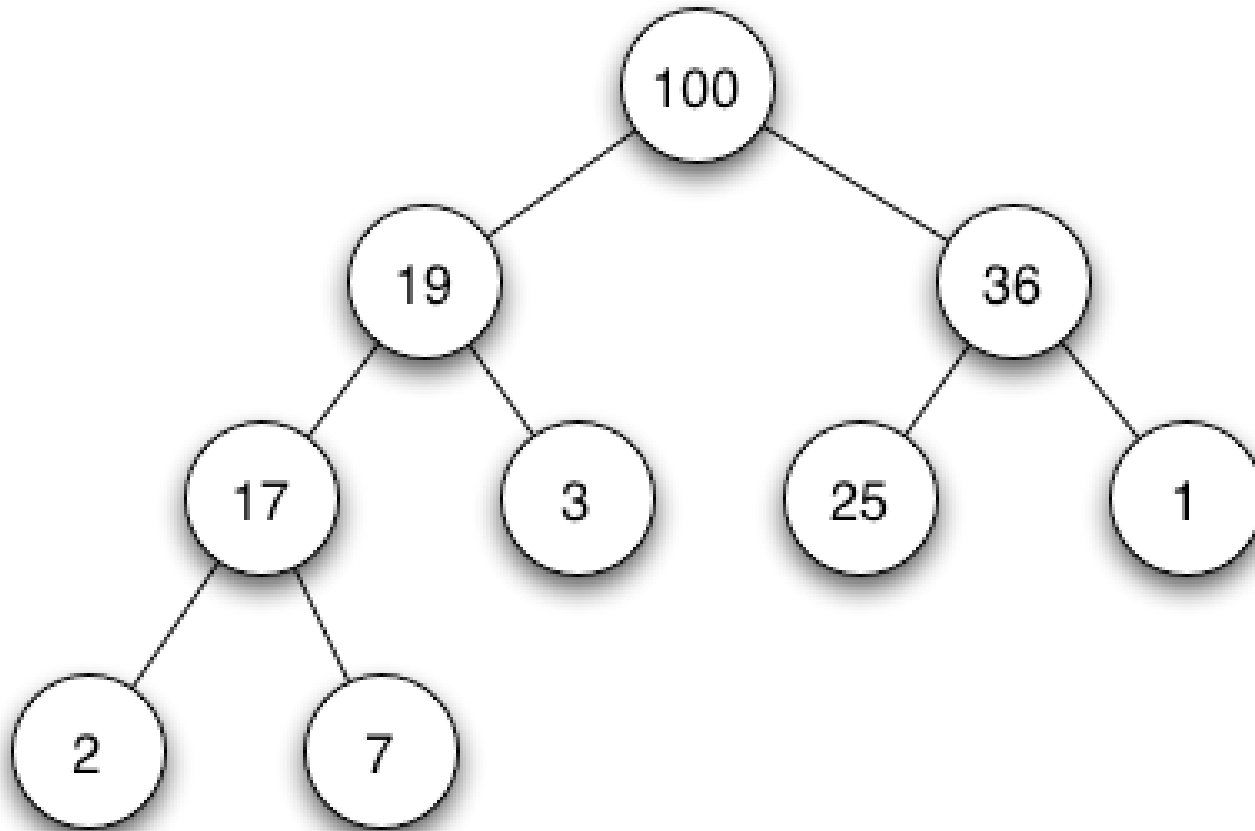
# Bibliography

- Chapter 4 of:
  - A.V. AHO., J.E. HOPCROFT., J.D. ULLMAN. 1987.  
“Data Structures and Algorithms.” Addison-Wesley.

# Heaps

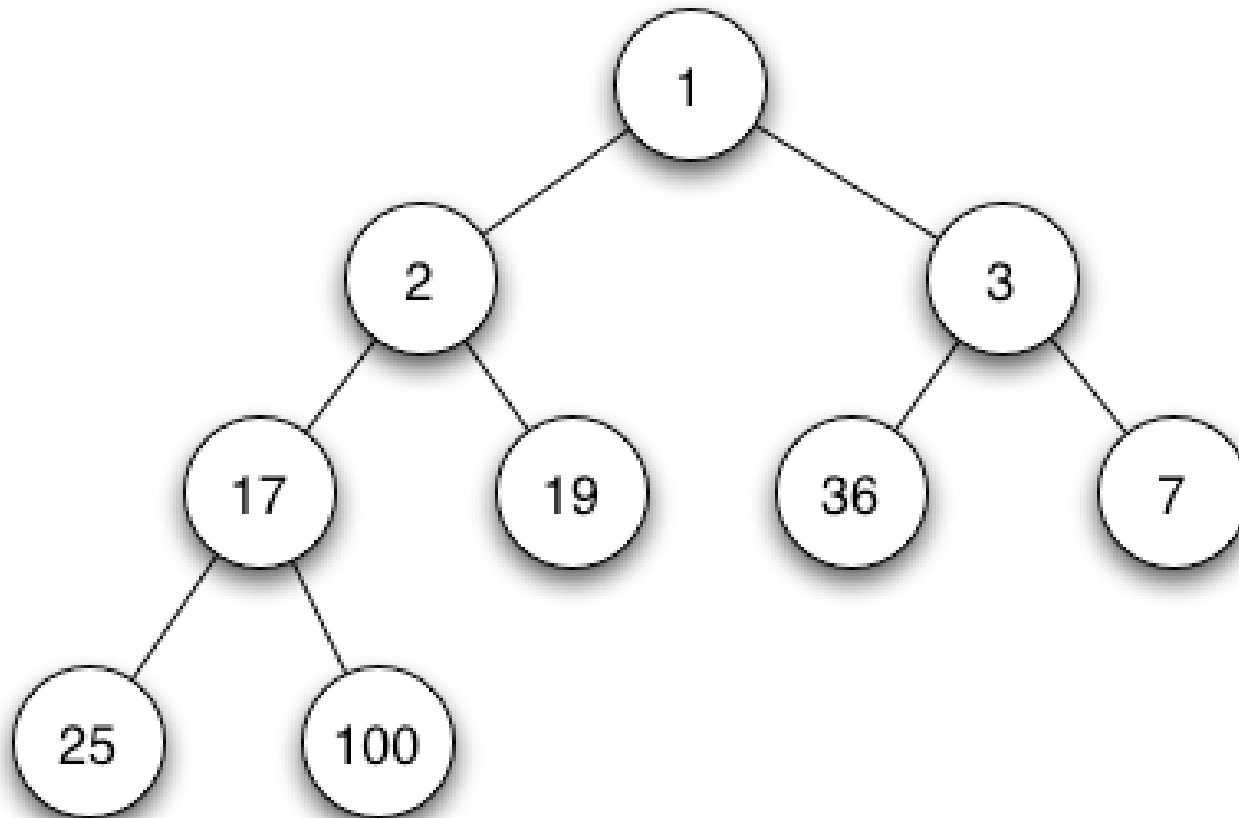
- A *heap* is a complete tree that satisfies the *heap property*
  - if  $B$  is a child node of  $A$ , then  $\text{key}(A) \geq \text{key}(B)$
- This implies that the element with the greatest key is always in the root node, and so such a heap is sometimes called a *max-heap*
  - Alternatively, if the comparison is reversed, the smallest element is always in the root node, which results in a *min-heap*

# Heaps



A binary max-heap

# Heaps



A binary min-heap

# Heaps

- There is no restriction as to how many children each node has in a heap, although in practice each node has at most two (*binary heap*).
- The heap is one maximally-efficient implementation of an abstract data type called a priority queue.
- Heaps are crucial in several efficient graph algorithms such as Dijkstra's algorithm, and in the sorting algorithm heapsort.

# The ADT heap

**spec** *HEAP*[*node*]

**genres** *heap*, *node*

**operations**

*find\_max*: *heap* -> *node*

[*find\_min*: *heap* -> *node*]

*delete\_max*: *heap* -> *heap*

[*delete\_min*: *heap* -> *heap*]

*insert*: *node* *heap* -> *heap*

*makenull*: *heap* -> *heap*

*create*: *node*[] -> *heap*

*empty*: *heap* -> *boolean*

**endspec**



# The ADT heap

- *find-max* or *find-min*: find the maximum item of a max-heap or a minimum item of a min-heap, respectively
- *delete-max* or *delete-min*: removing the root node of a max- or min-heap, respectively
- *create*: create a valid new heap containing all the elements of a given list of nodes.

# The ADT heap

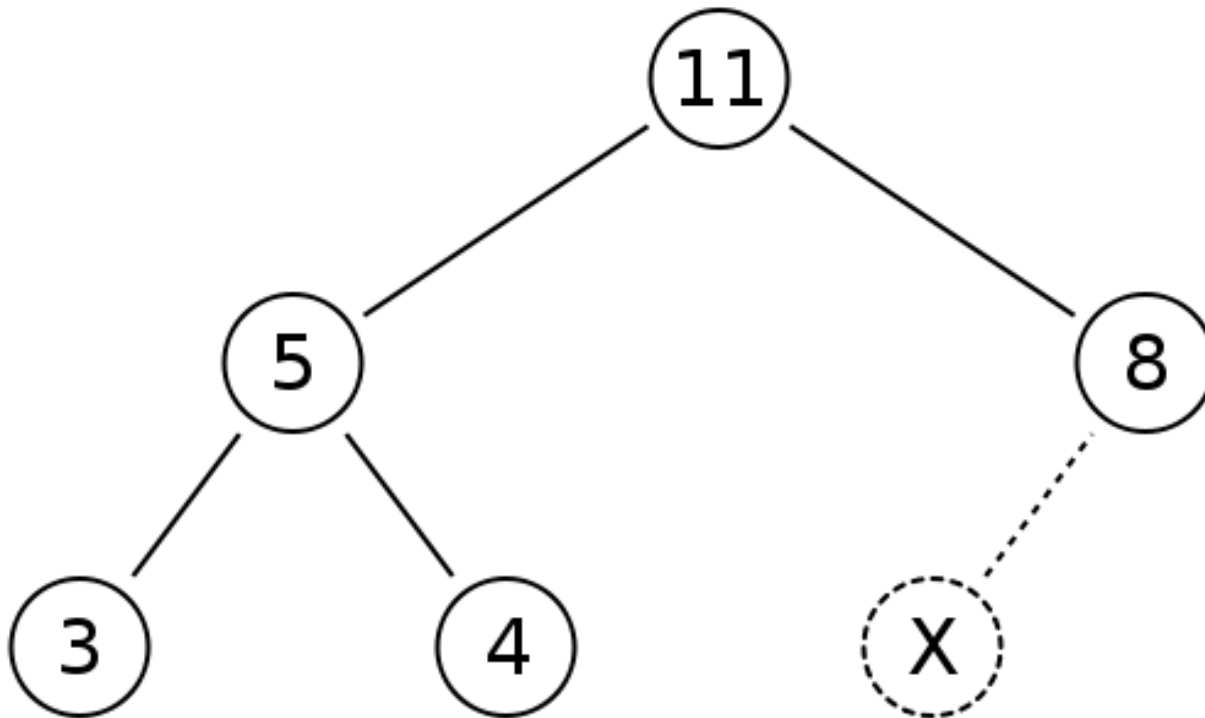
- To clarify things...
- A max-heap has the operations:
  - find-max, delete-max
- A min-heap has the operations:
  - find-min, delete-min

# Heap operations - Insert

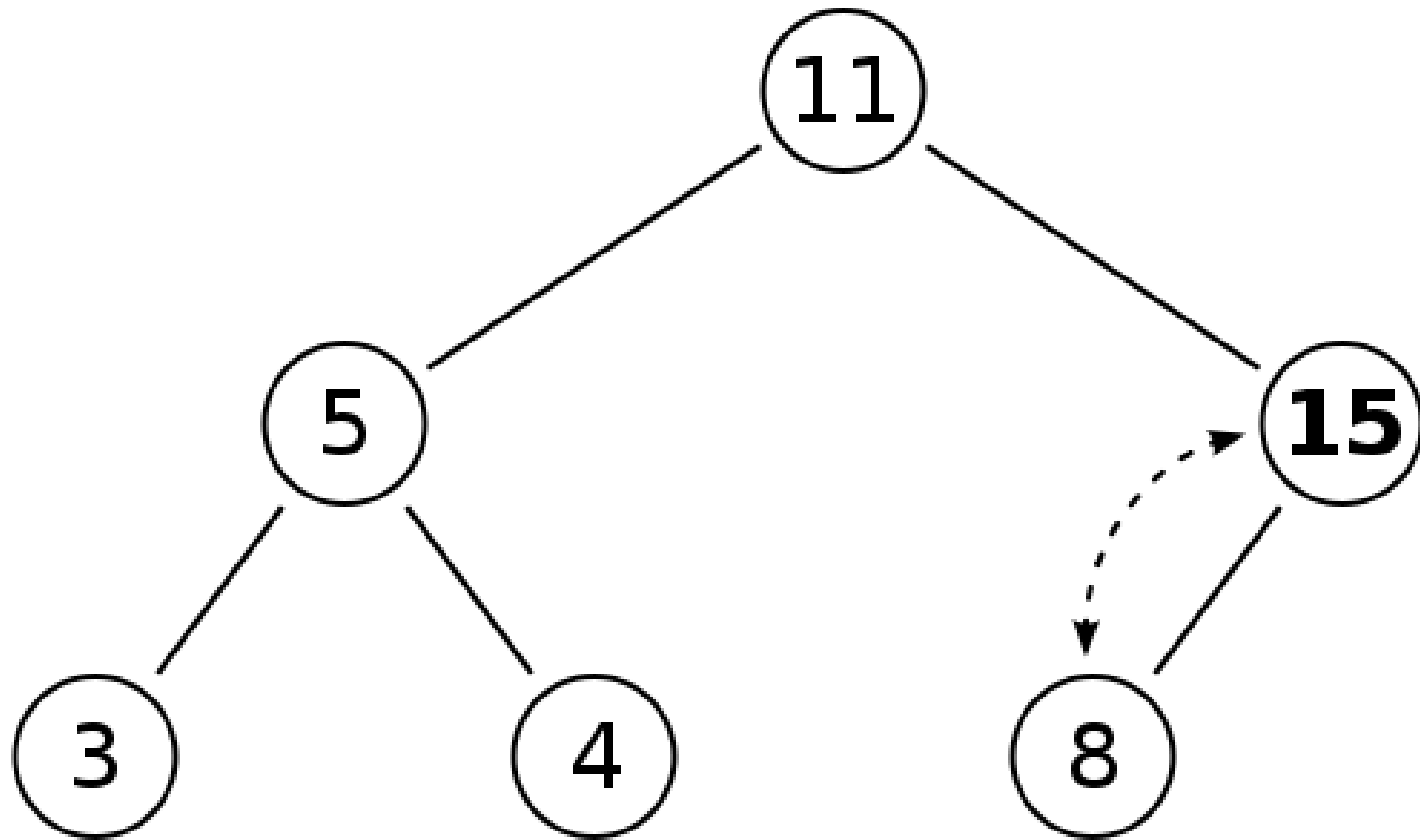
- To add an element to a heap we must perform an *up-heap* operation (also known as *bubble-up* or *heapify-up*) in order to restore the heap property. We can do this in  $O(\log n)$  time, using a binary heap, by following this algorithm:
  1. Add the element to the bottom level of the heap.
  2. Compare the added element with its parent; if they are in the correct order, stop.
  3. If not, swap the element with its parent and return to the previous step.

# Heap operations - Insert

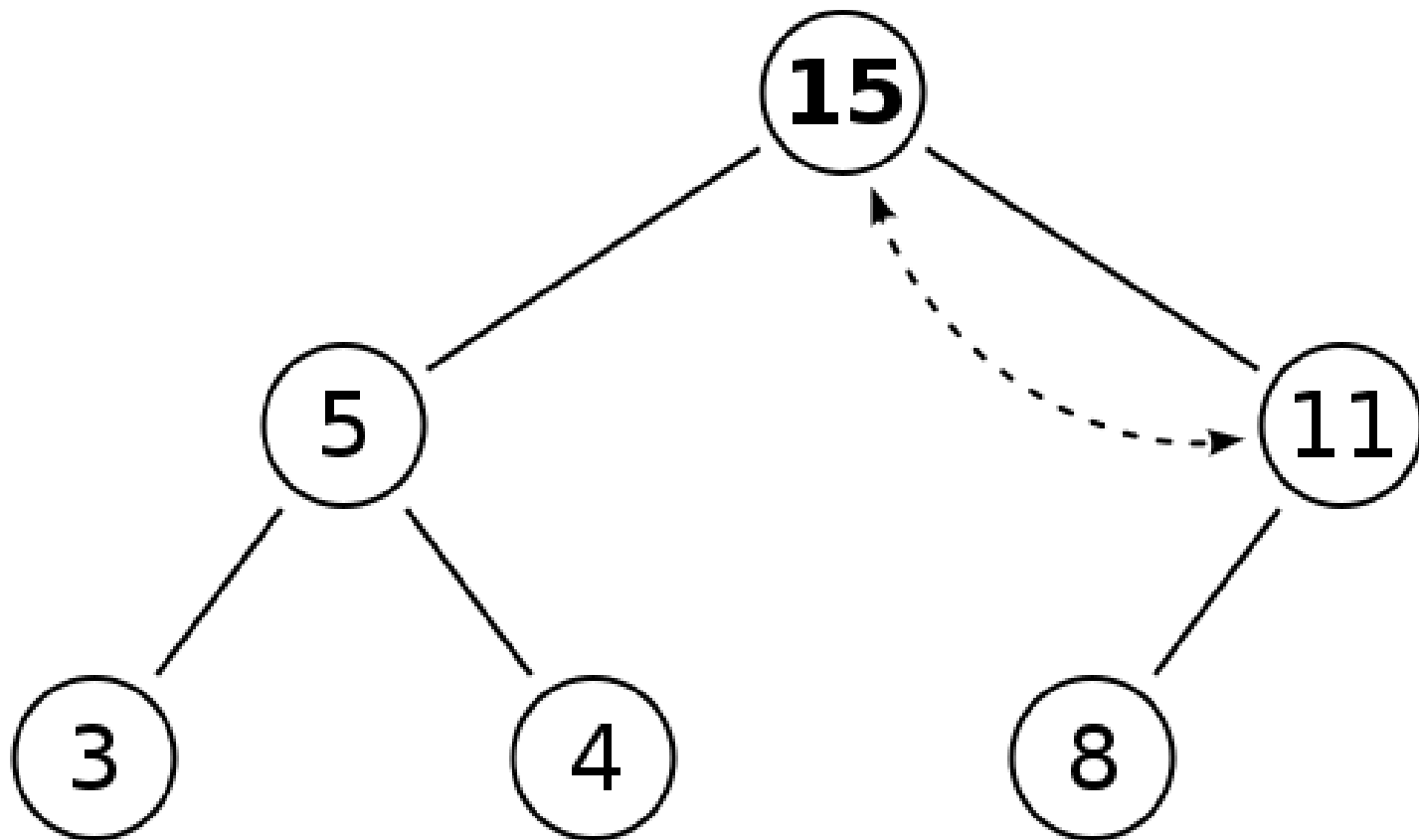
- Insert 15 on this max-heap



# Heap operations - Insert



# Heap operations - Insert



# Heap operations - Insert

- We do this at maximum once for each level in the tree—the height of the tree, which is  **$O(\log n)$** . However, since approximately 50% of the elements are leaves and 75% are in the bottom two levels, it is likely that the new element to be inserted will only move a few levels upwards to maintain the heap. Thus, binary heaps support insertion in *average* constant time  **$O(1)$** .

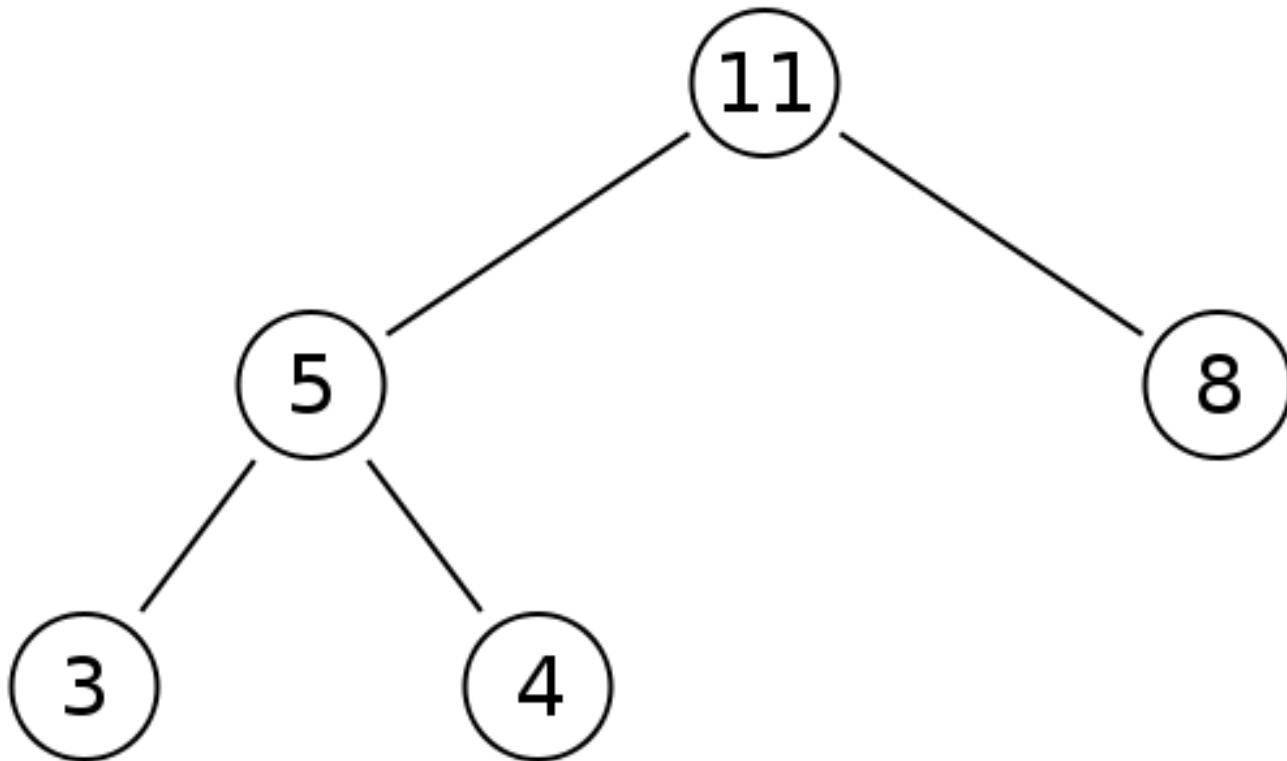
# Heap operations - Delete

- The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down* or *heapify-down*).
  1. Replace the root of the heap with the last element on the last level.
  2. Compare the new root with its children; if they are in the correct order, stop.
  3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

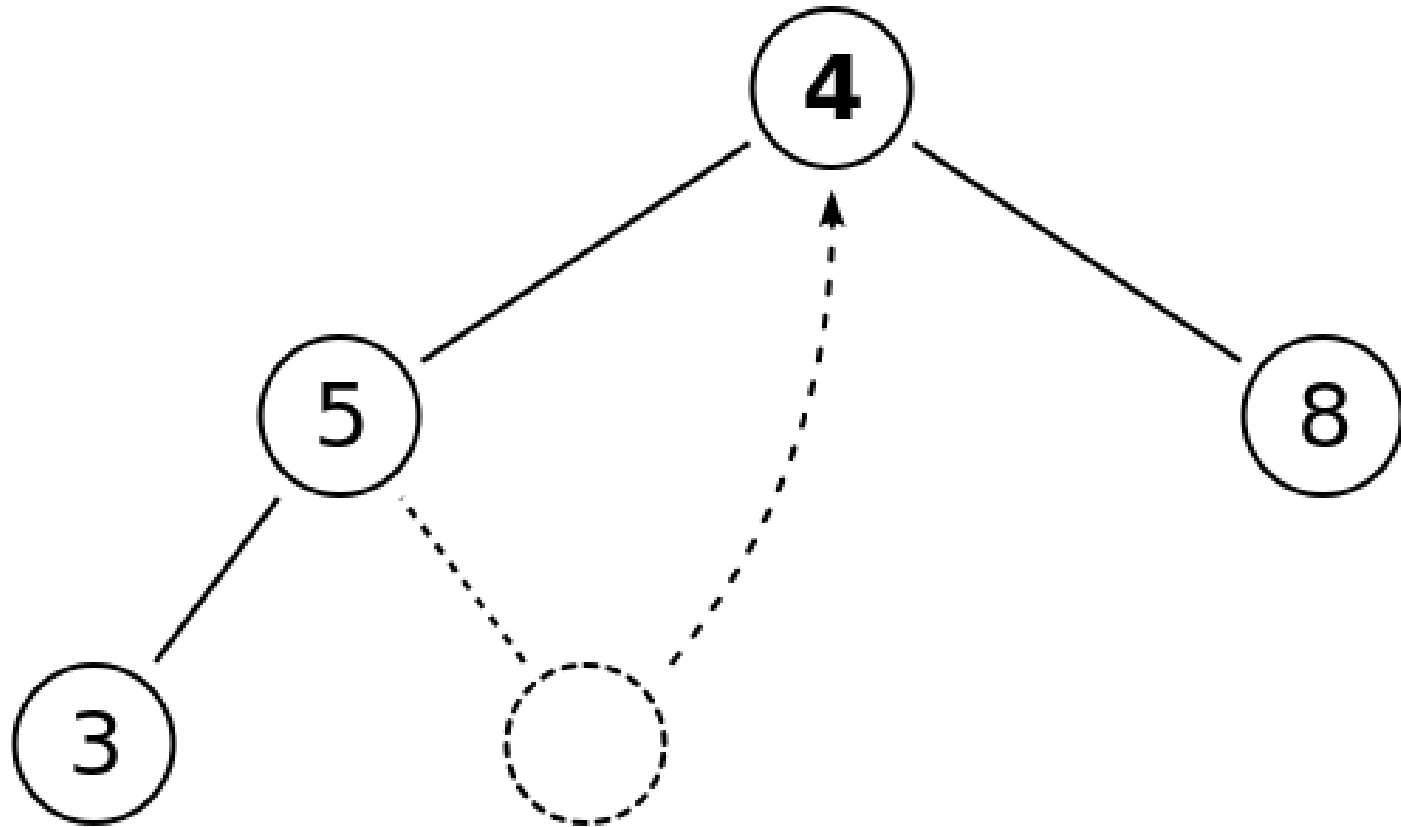


# Heap operations - Delete

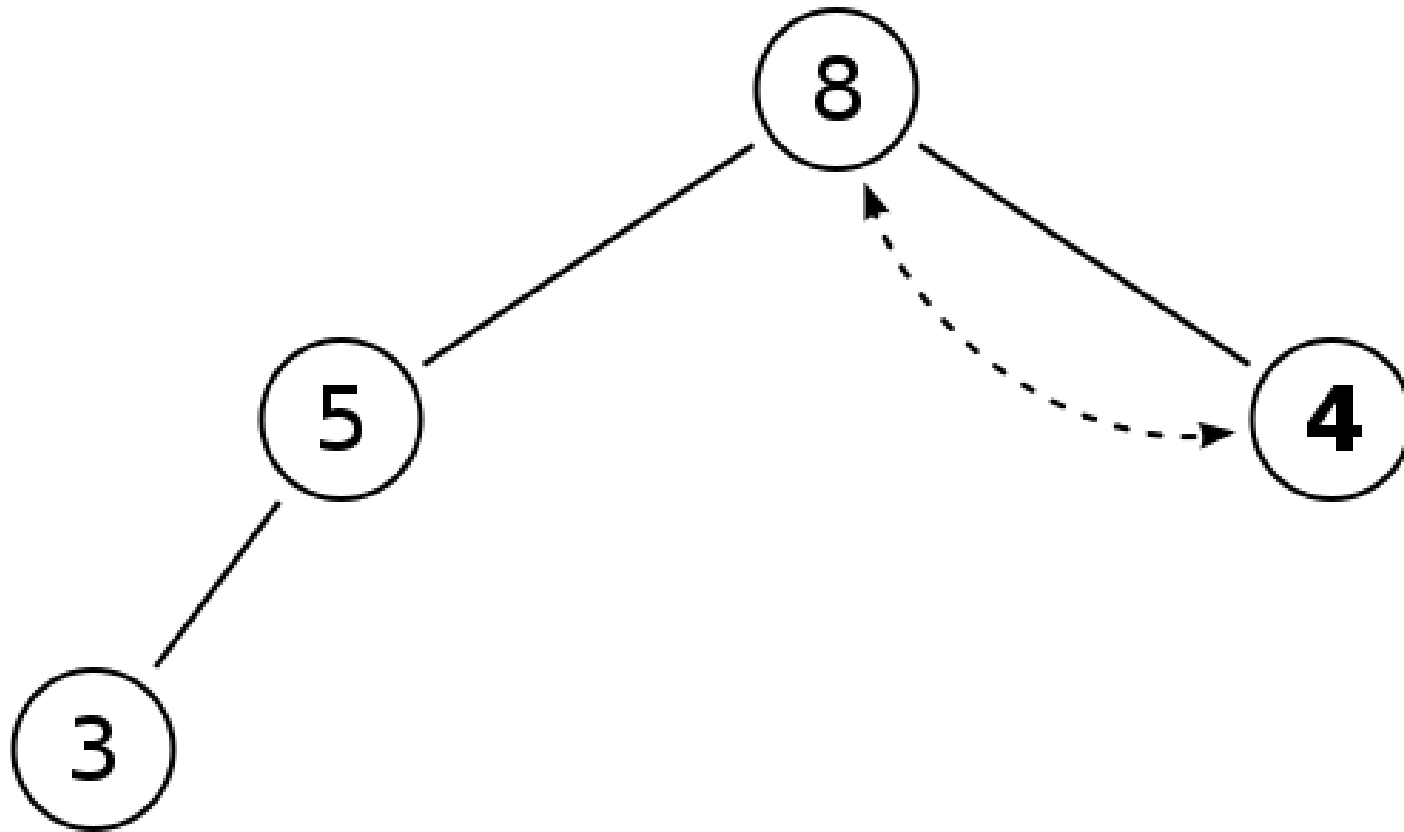
- Delete-max on the following max-heap



# Heap operations - Delete



# Heap operations - Delete



# Heap operations - Create

- A heap could be built by successive insertions. This approach requires  $O(n \log n)$  time because each insertion takes  $O(\log n)$  time and there are  $n$  elements. However this is not the optimal method.

# Heap operations - Create

- The optimal method starts by arbitrarily putting the elements on a binary tree, respecting the BT shape property. Then starting from the lowest level and moving upwards, shift the root of each subtree downward as in the deletion algorithm until the heap property is restored.
- More specifically if all the subtrees starting at some height  $h$  (measured from the bottom) have already been "heapified", the trees at height  $h + 1$  can be heapified by sending their root down along the path of maximum valued children when building a max-heap, or minimum valued children when building a min-heap.

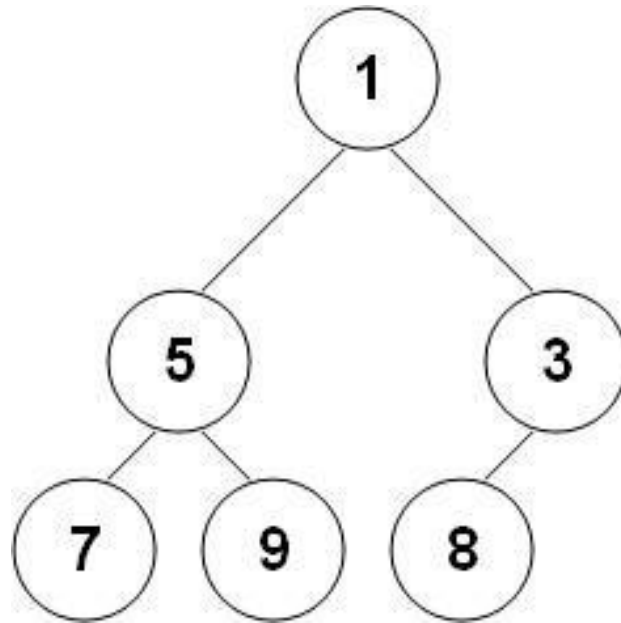
# Implementations of heaps







- Heaps are commonly implemented with an array. Any binary tree can be stored in an array, but because a heap is always an almost complete binary tree, it can be stored compactly. No space is required for pointers; instead, the parent and children of each node can be found by arithmetic on array indices.

# Implementations of heaps

- Let  $n$  be the number of elements in the heap and  $i$  be an arbitrary valid index of the array storing the heap. If the tree root is at index 1, with valid indices 1 through  $n$ , then each element  $a[i]$  has:
  - children  $a[2i]$  and  $a[2i+1]$
  - parent  $a[\text{floor}(i/2)]$ .

# Implementations of heaps



<b>Node</b>						
<b>Index</b>	1	2	3	4	5	6



# The ADT heap

**const**

maxsize := {some suitable constant}

**class** [max|min]heap

private content: **array**[1..maxsize] of node

privste last: integer

{methods...}

**endclass**

node: elementtype

# Variants

- Binomial heap
- Fibonacci heap
- Beap – Biparental heap

# Heaps

- Running times of operations on binary heaps

Operation	Average	Worst case
find (min or max)	1	1
insert	1	$O(\log n)$
delete (min or max)	$O(\log n)$	$O(\log n)$
create	$O(n)$	$O(n)$

# Heapsort

- Heapsort is a comparison-based sorting algorithm to create a sorted array (or list)
- Although somewhat slower in practice on most machines than a well implemented quicksort, it has the advantage of a more favorable worst-case  $O(n \log n)$  runtime.

# Heapsort

- Heapsort begins by building a heap out of the data set.
- The largest value (in a max-heap) or the smallest value (in a min-heap) are extracted until none remain, the values having been extracted in sorted order. The heap's invariant is preserved after each extraction, so the only cost is that of extraction.
- Heapsort uses three heap operations: *create*, *find-max* (or *find-min*) and *delete-max* (or *delete-min*).

# Heapsort

```
func heapsort (a: elementtype[]) ret b: elementtype[]
    //assuming that a binary min-heap is used
    var    sorted: elementtype[]
           h: heap
           index: integer
    h := create(a)
    index := 1
    while not empty(h)
        sorted[i] := find-min(h)
        delete-min(h)
        index := index+1
    endwhile
    return sorted
endfunc
```

# Heapsort

6 5 3 1 8 7 2 4

# Priority queues

- A *priority queue* is an ADT which is like a regular queue or stack, but additionally, each element is associated with a "priority"
- Common operations on a priority queue are INSERT, DELETEMIN (or DELETEMAX), and MAKENULL



# The ADT priority queue

```
const maxsize := {some suitable constant}
```

```
node = record  
    element: elementtype  
    priority: integer  
endrecord
```

```
class priorityQueue  
    content: array[1..maxsize] of node  
    last: integer  
    {methods...}  
endclass
```

# Implementations of priority queues

## Naive implementations

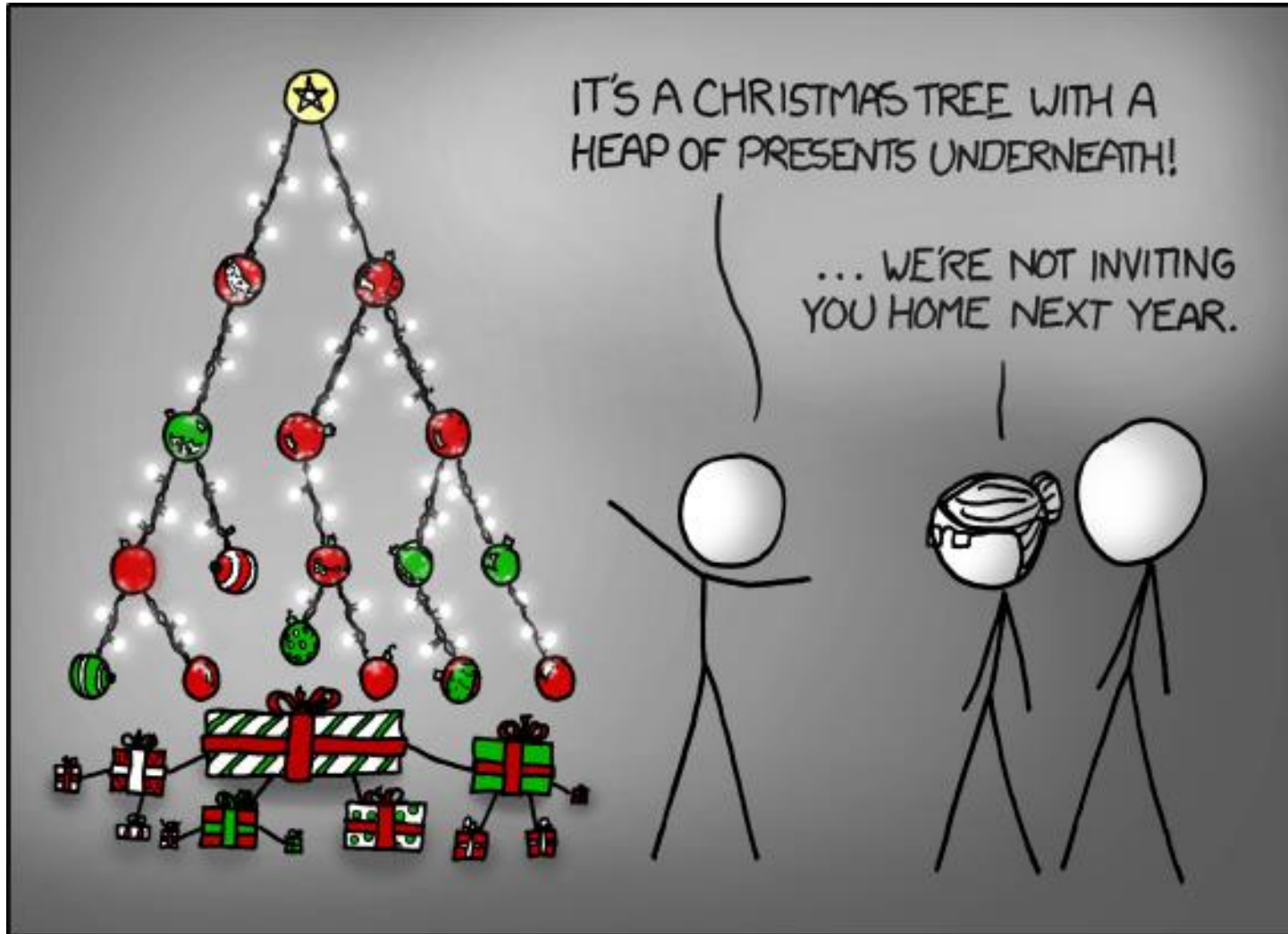
- There are a variety of simple, usually inefficient, ways to implement a priority queue. They provide an analogy to help one understand what a priority queue is. For instance, one can keep all the elements in an unsorted list. Whenever the highest-priority element is requested, search through all elements for the one with the highest priority. ( $O(1)$  insertion time,  $O(n)$  pull time due to search)

# Implementations of priority queues

## Usual implementation

- To get better performance, priority queues typically *use a heap* as their backbone, giving  $O(\log n)$  performance for inserts and removals, and  $O(n)$  to build initially.

# Trees & Heaps



# Heaps / Priority queues

Luis de Marcos Ortega

[luis.demarcos@uah.es](mailto:luis.demarcos@uah.es)