

---

# Estruturas de Informação

## Priority Queues - Heaps

Fátima Rodrigues

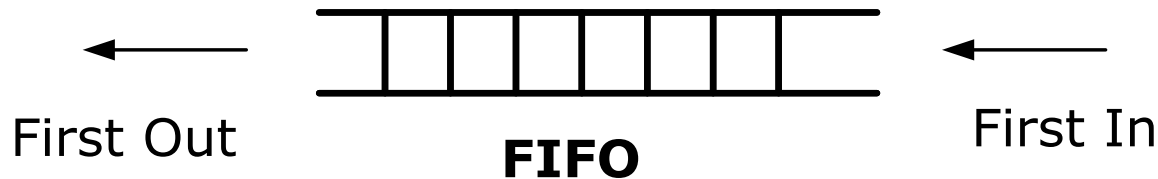
[mfc@isep.ipp.pt](mailto:mfc@isep.ipp.pt)

Departamento de Engenharia Informática (DEI/ISEP)

# Queue

---

- Elements are inserted at one end, and removed from another
- Elements are in the queue in order of arrival



## Queue interface

- `addBack(newElement)`  $O(?)$
- `front()`
- `removeFront()`  $O(?)$
- `isEmpty()`
- `size()`

# Priority Queue

---

Associates a “priority” with each object:

- First element has the highest priority (typically, **lowest value**)

Examples of priority queues:

- to-do list with priorities
- hospital emergency queue
- air-traffic control
- active processes in an OS
- device controller for a shared printer

# Priority Queue

---

- A priority queue is an abstract data type for storing a collection of **prioritized elements**
- The elements in a priority queue have a **priority** provided by its **associated Key** - each entry is a pair (key, value)
- The element with the **minimal key** will be the next to be removed from the queue

## Priority queue interface

- insert(k,v)
- min()
- removeMin()
- size()
- isEmpty()

# Priority Queue - Implementations

---

	<b>Array</b>	<b>Sorted Array</b>	<b>Sorted List</b>
insert(k,v)	$O(1)$	$O(n^2)$	$O(n)$
min()	$O(n)$	$O(1)$	$O(1)$
removeMin()	$O(n^2)$	$O(n)$	$O(1)$

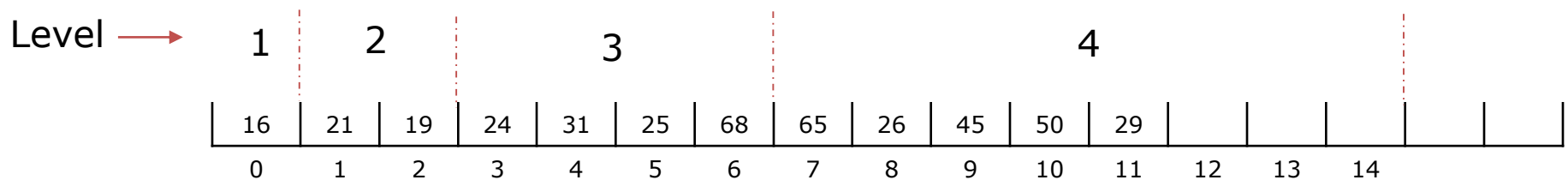
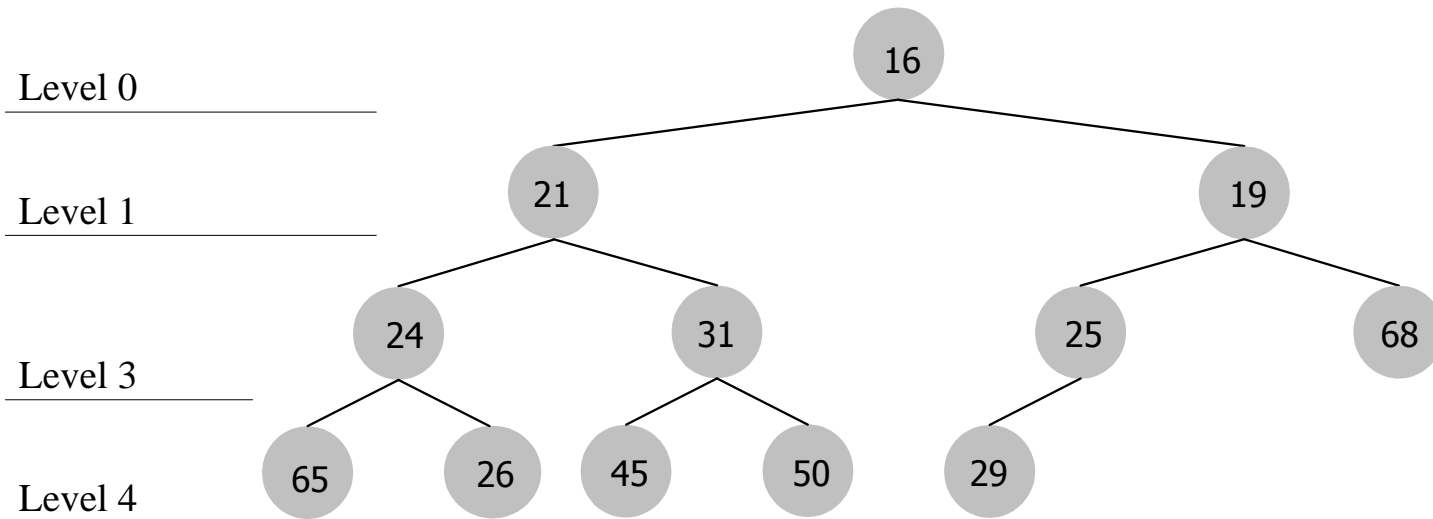
Any implementation using a sequential data structure implies operations with linear complexity

Alternative → Heap

# Binary tree representation with a vector

A binary tree can be represented by a vector:

- Fill up the vector with the tree items ordered by level



# Binary tree representation with a vector

---

How to access the elements?

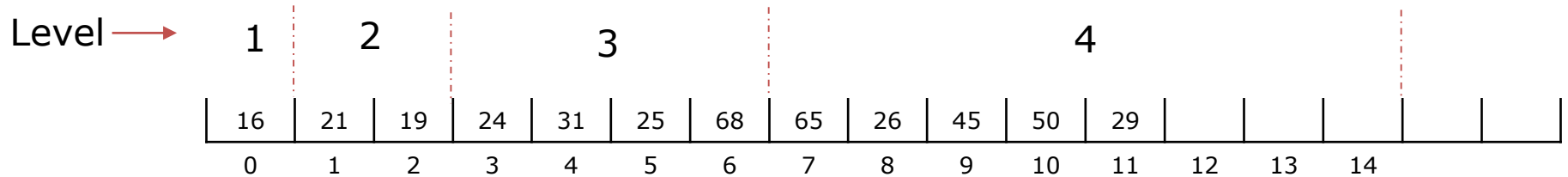
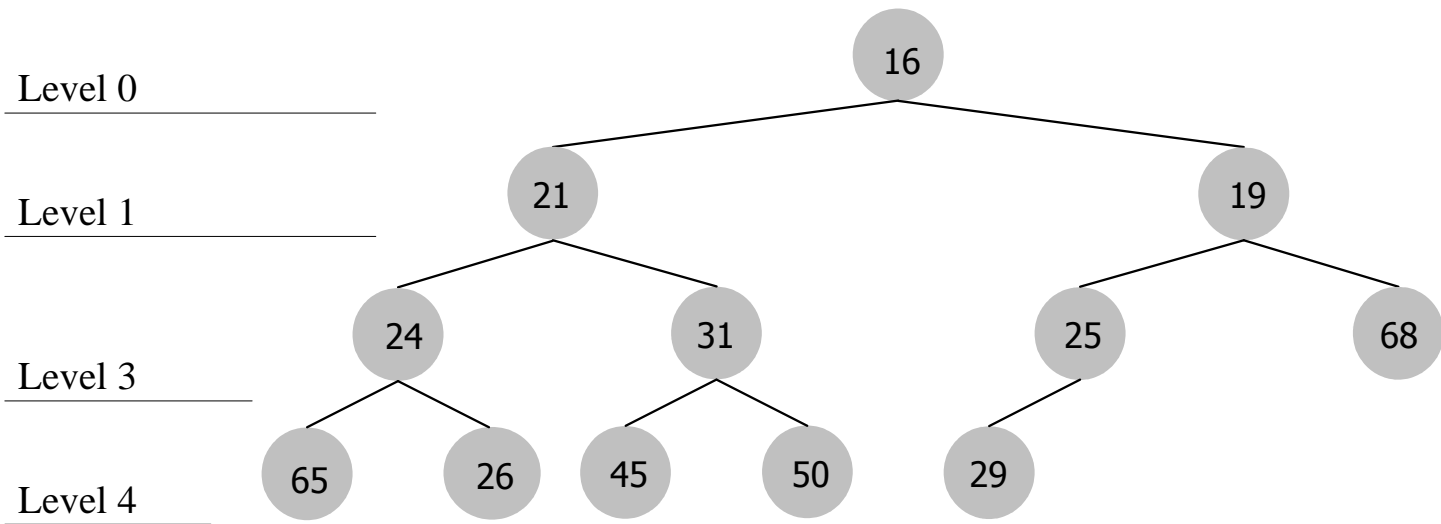
- Being the root at index 0
- Its direct descendants are in the indices 1 and 2
- The descendants of the element at index 3 are at indexes: 7 and 8
- ....

Generally to an element at index  $n$ :

- its **left child** is at index:  $2n+1$
- its **right child** is at index:  $2n+2$
- its **parent** is at index:  $(n-1)/2$

These formulas allow transit between the elements of the different levels of the tree, using an alternative way to references

# Binary tree representation with a vector



Predecessors of the node 50

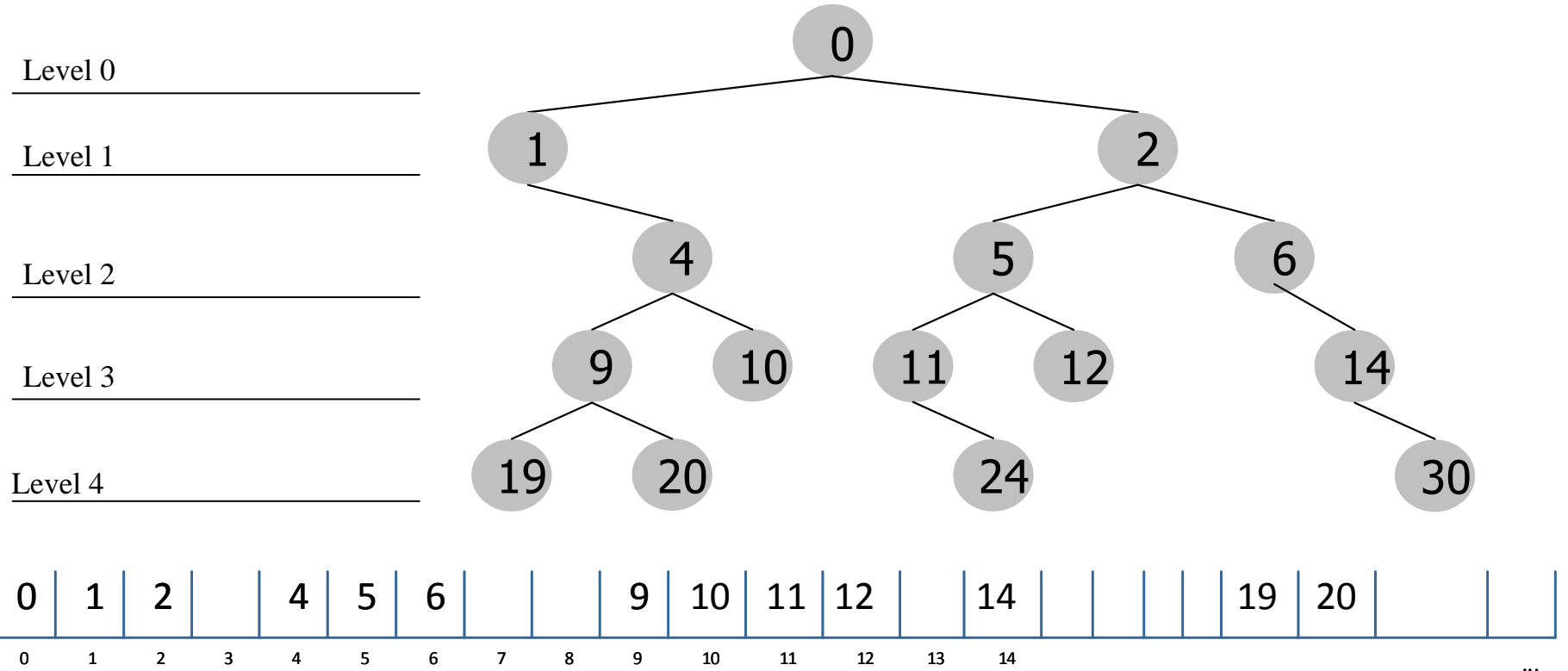
$50 \rightarrow n=10 \rightarrow (n-1)/2 = 4 \rightarrow \text{vect}[4] = 31$

$n=4 \rightarrow (n-1)/2 = 1 \rightarrow \text{vect}[1] = 21$

$n=1 \rightarrow (n-1)/2 = 0 \rightarrow \text{vect}[0] = 16$

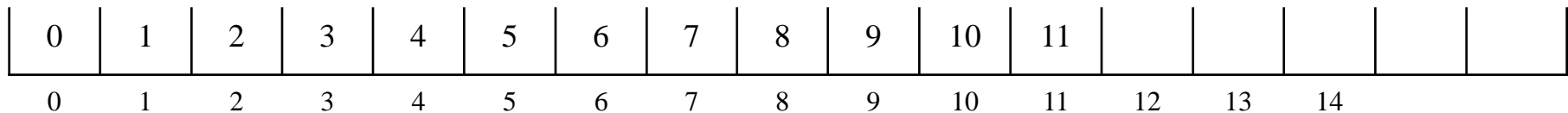
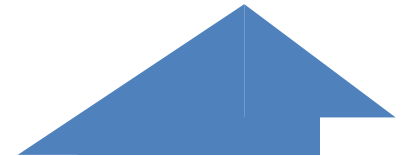
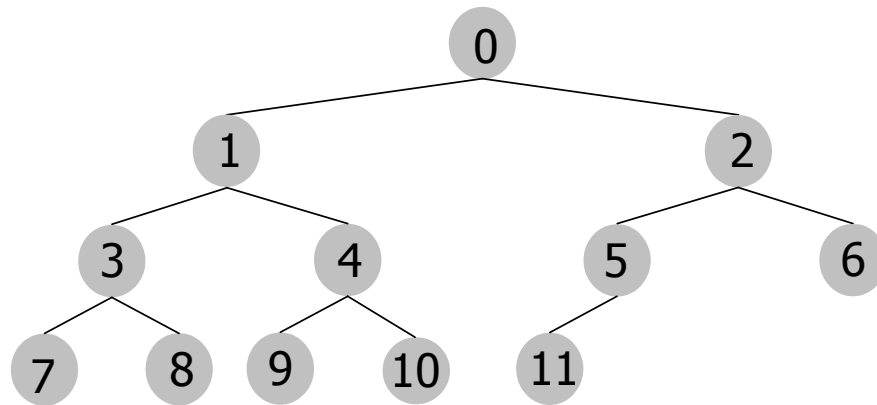


# Vector representation problem



# Complete binary tree

A complete binary tree is a tree where every leaf is at the same depth and the bottom level is filled from left to right



A complete binary tree can be efficiently stored in a vector - the vector does not contain empty cells and all elements are contiguous

# Array-based representation of binary trees

---

An array-based structure is adequate for representing a **complete binary tree**

With an array implementation

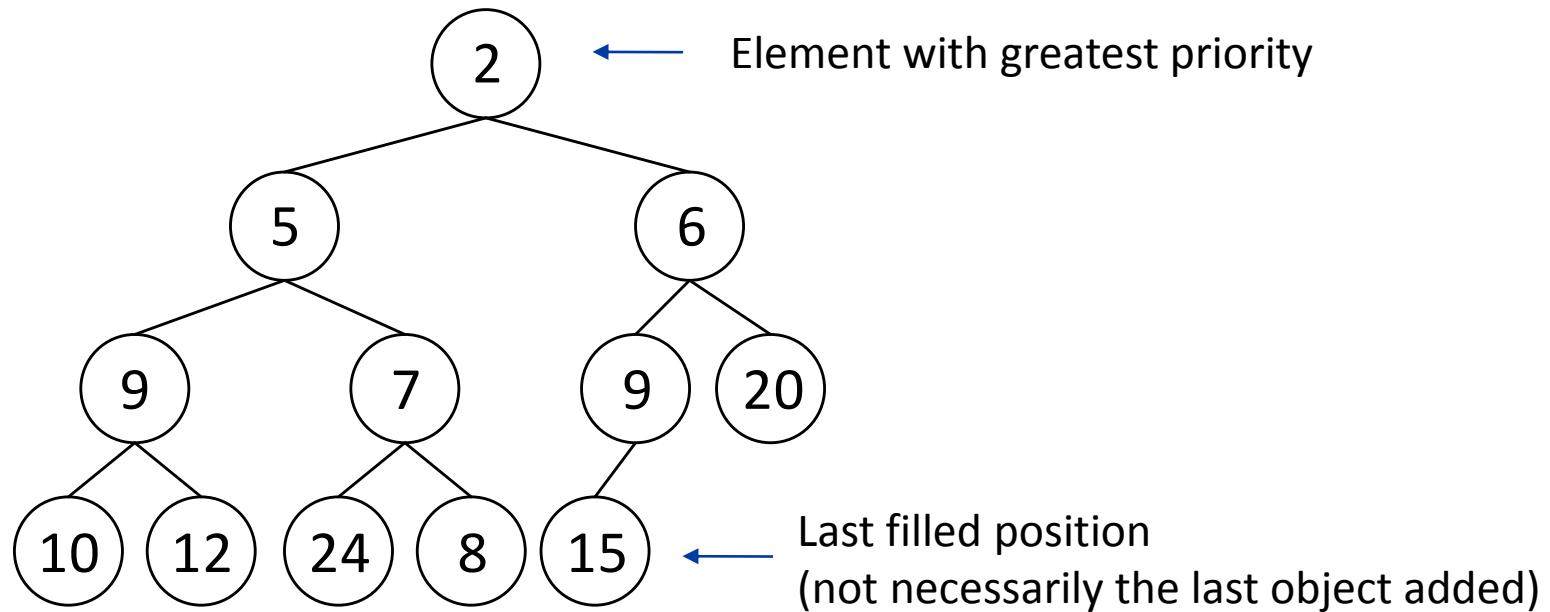
- Operations: size, isEmpty, replace, root, parent, children, left, right, hasLeft, hasRight, isInternal, isExternal, isRoot take  **$O(1)$  time**
- Operations: elements, positions are  **$O(n)$  time**

# Heap

---

Heap is a **complete binary tree** in which every node's value is less or equal its children values

Heap-order implies that each path in the tree is sorted



# Heaps and Priority Queues

---

A heap can be used to efficiently implement a priority queue

Each entry (key, element) is inserted at each node

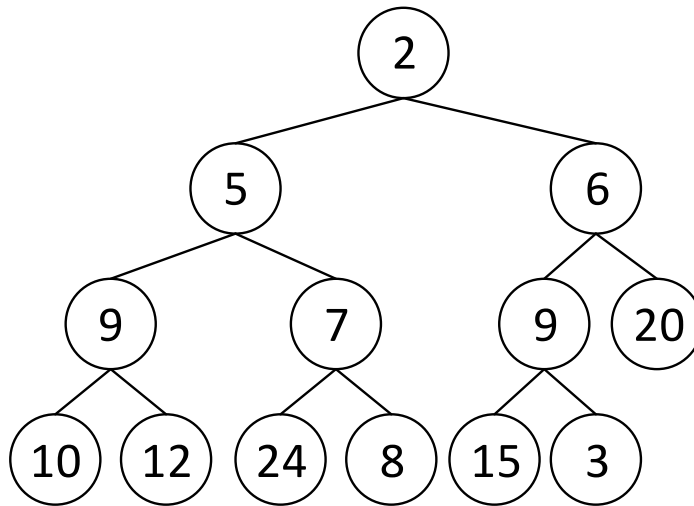
- Two distinct entries in a priority queue can have the same key
- Keys in a priority queue can be arbitrary objects on which an order is defined
- A generic priority queue uses an external comparator object

# Heap – Insertion

---

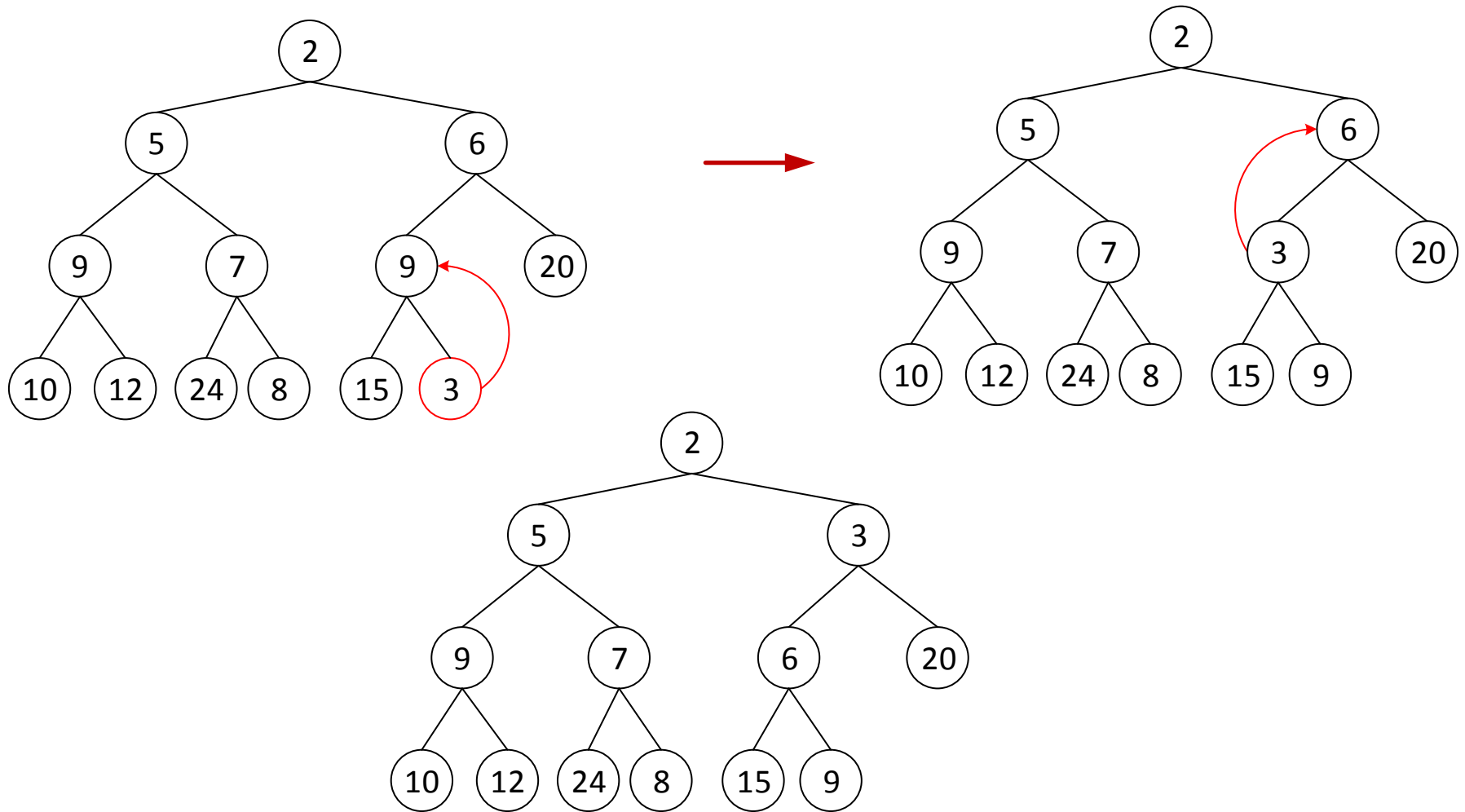
The insertion of an element in a heap must guarantee two properties:

1. the tree remains **complete** - the new element is inserted on the **last level** of the tree the **rightmost** possible
2. the tree remains **orderly** - Fix it by **percolating up**

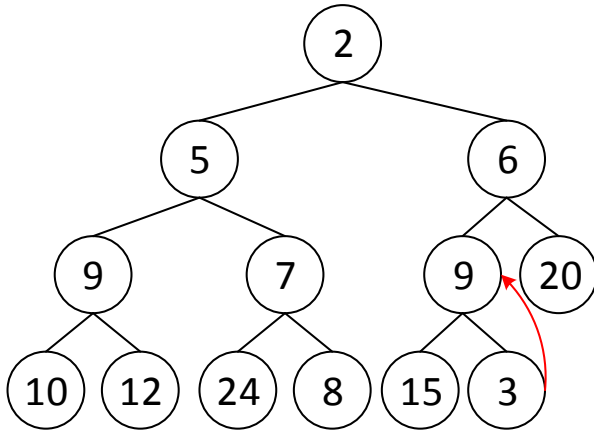


# Heap – Insertion: Percolate up

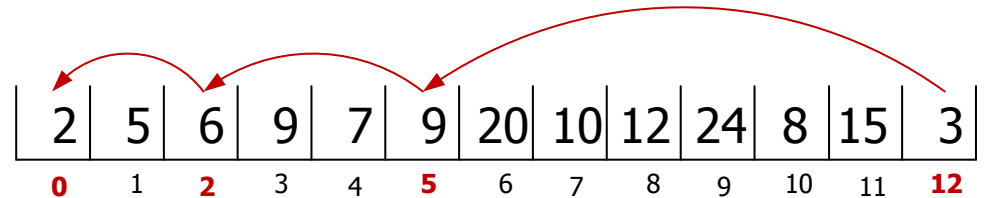
To maintain the tree ordered the new element must be put in the correct place



# Percolate up



Parent's node at index n:  $(n-1)/2$



```
Algorithm percolateUp (int i){  
    ind = (i-1)/2;  
    while (ind >= 0 && vector[i] < vector[ind]){  
        swap(vector[i], vector[ind])  
        i = ind  
        ind = (i-1)/2  
    }  
}
```

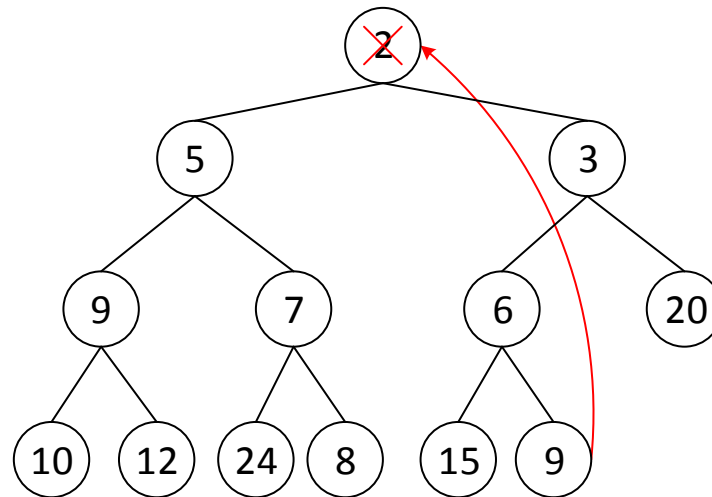
Complexity(?)



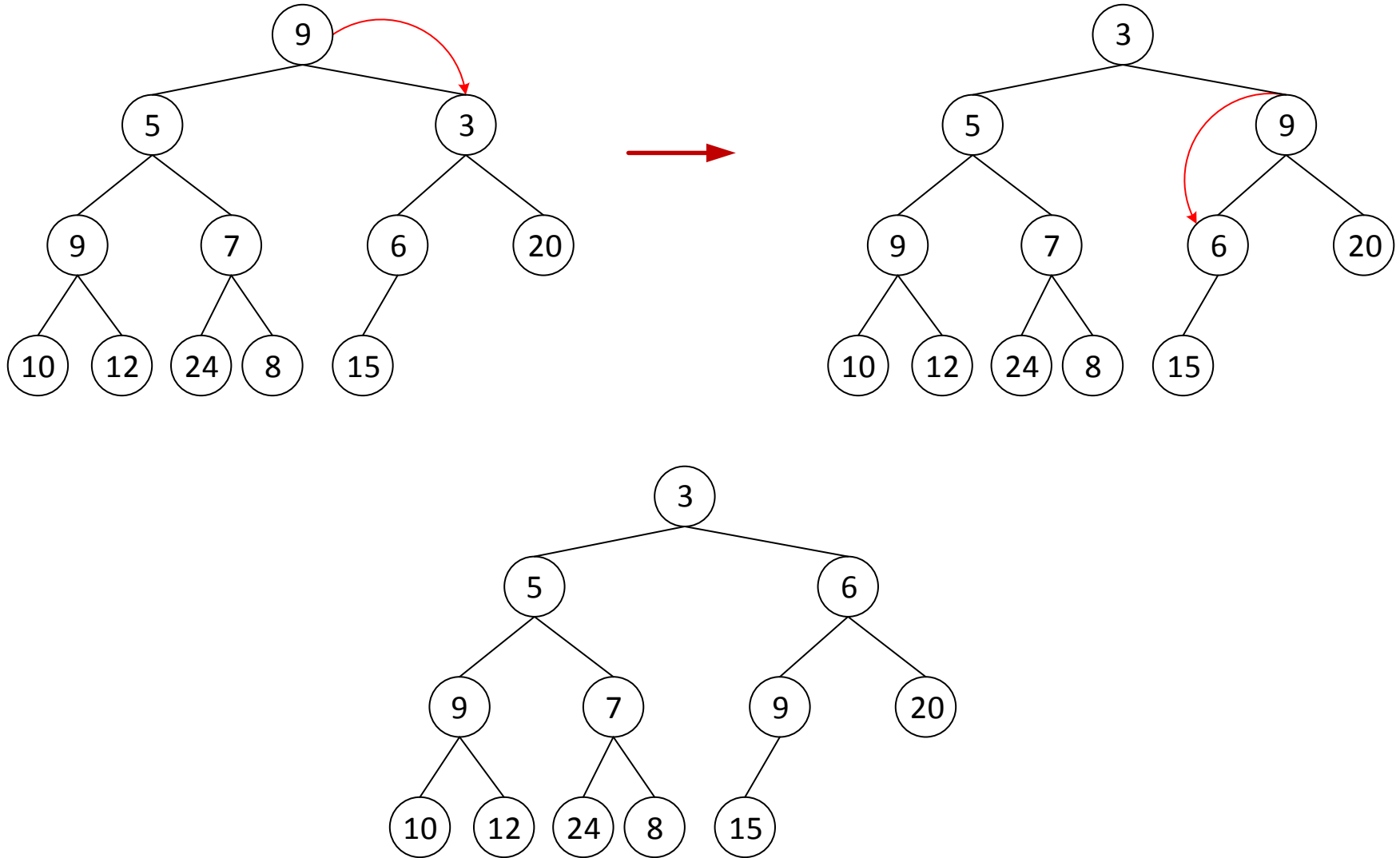
# Heap – RemoveMin

The RemoveMin of an element in a heap must also guarantee the two properties:

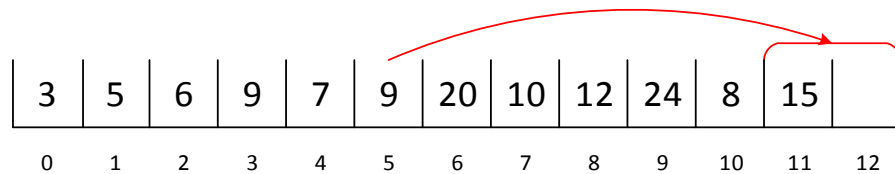
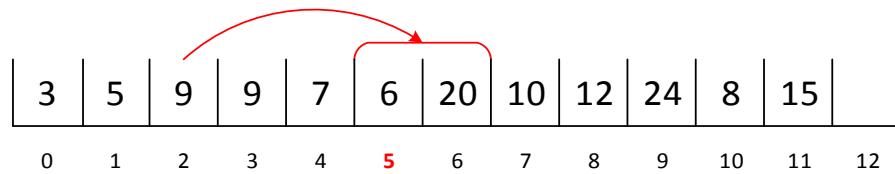
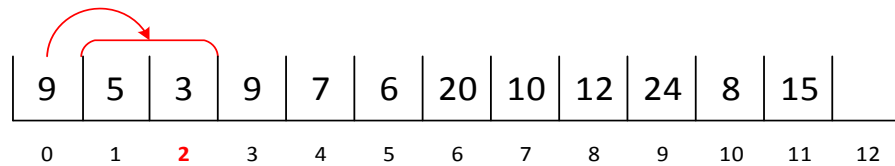
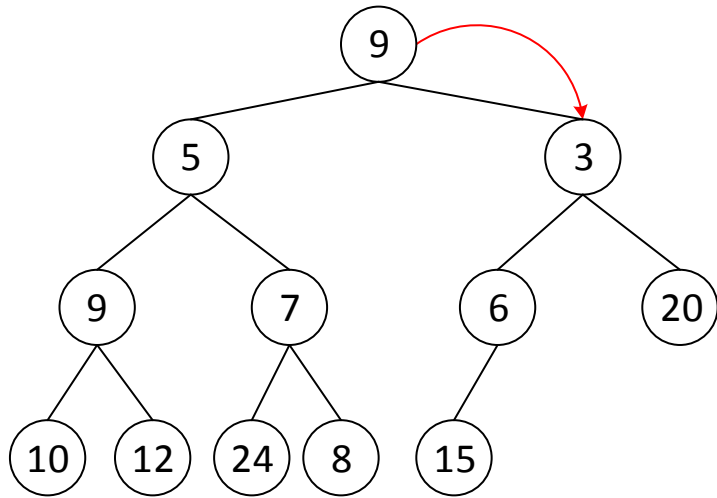
1. the tree remains **complete** – root is replaced by the rightmost element in the last level
2. the tree remains orderly – fix it by **percolating down**



# Heap – RemoveMin: Percolate down



# Percolate down



# Percolate down

---

```
Algorithm percolateDown (int i) {
    indLeft = 2×i+1
    indRight = 2×i+2
    while (indLeft < vector.size()) {
        smallindex = indLeft
        if (indRight < vector.size())
            if (vector[indRight] < vector[indLeft])
                smallindex = indRight

        if (vector[i] > vector[smallindex]) {
            swap(vector[i], vector[smallindex]) //change the elem by the
            i = smallindex                      //child with the highest
            indLeft = 2×i+1                     //priority
            indRight = 2×i+2
        }
        else
            break;
    }
}
```

Complexity(?)

# Priority Queue - Performance Evaluation

---

	Array	Sorted Array	Sorted List	Heap
insert(k,v)	$O(1)$	$O(n^2)$	$O(n)$	$O(\log n)$
min()	$O(n)$	$O(1)$	$O(1)$	$O(1)$
removeMin()	$O(n^2)$	$O(n)$	$O(1)$	$O(\log n)$

The main purpose of a priority queue is rapidly accessing and removing the smallest element!

# HeapSort Algorithm

---

Sort a collection using a Heap:

1. Build a heap using the elements of the collection
2. Extract all elements from heap and insert them into the collection

<code>Algorithm heapSort (ArrayList&lt;E&gt; vector) {</code>	1   28   14   5   20   6
<code>    for (int i = 0; i &lt; vector.size(); i++)</code>	
<code>        insert(i,v[i]);</code>	1   5   6   28   20   14
<code>    for (int i = 0; i &lt; vector.size(); i++)</code>	
<code>        v[i] = removeMin()</code>	1   5   6   14   20   28
<code>}</code>	

Complexity(?)

# Comparison of Sorting Algorithms

---

Algorithm	Best Case	Worst case
SelectionSort	$O(n^2)$	$O(n^2)$
BubbleSort	$O(n)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$
MergeSort	$O(n \log n)$	$O(n \log n)$
QuickSort	$O(n \log n)$	$O(n^2)$
HeapSort	$O(n \log n)$	$O(n \log n)$

# Priority Queue Application: Simulation

---

Original, and one of most important, applications

Discrete event driven simulation:

- Actions represented by “events” – things that have (or will) happen at a given time
- Priority queue maintains list of pending events → Highest priority is the next event
- Event pulled from list is executed → often spawns more events, which are inserted into priority queue
- Loop until everything happens, or until fixed time is reached



# Priority Queue Application: Example

---

Example: Ice cream store

- People arrive
- People order
- People leave

Simulation algorithm:

1. Determine time of each event using random number generator with some distribution
2. Put all events in priority queue based on when it happens
3. Simulation framework pulls minimum (next to happen) and executes the event