
Estruturas de Informação

Priority Queues - Heaps

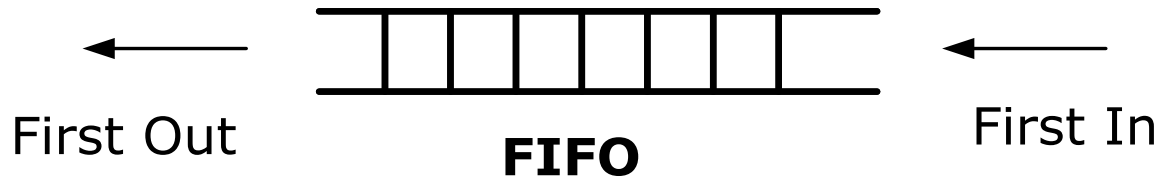
Fátima Rodrigues

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

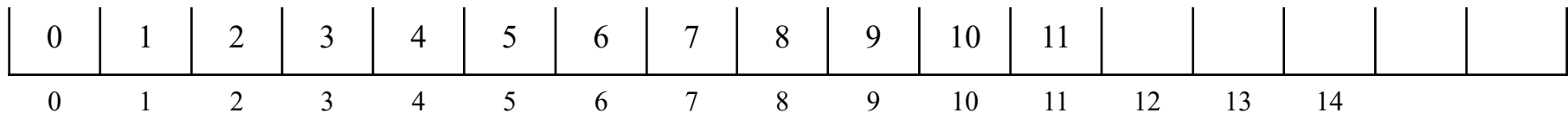
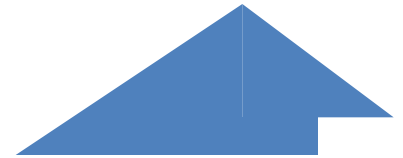
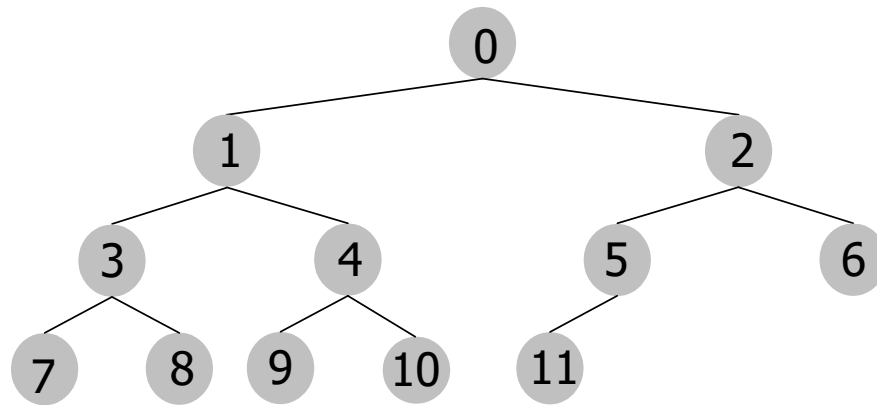
	Array	Sorted Array	Sorted List
insert(k,v)	$O(1)$	$O(n)$	$O(n)$
min()	$O(n)$	$O(1)$	$O(1)$
removeMin()	$O(n^2)$	$O(n)$	$O(1)$

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

Alternative → Heap

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

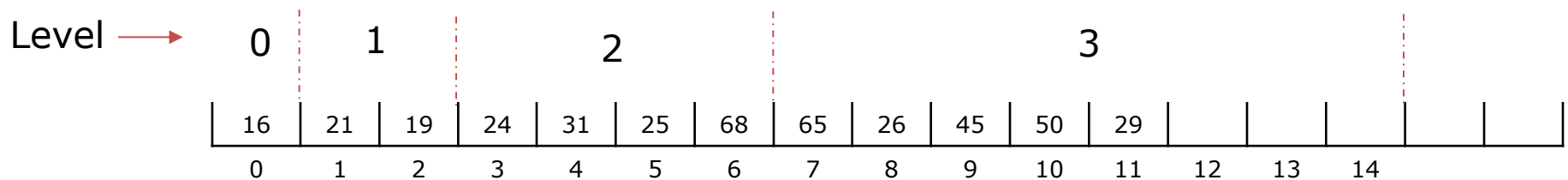
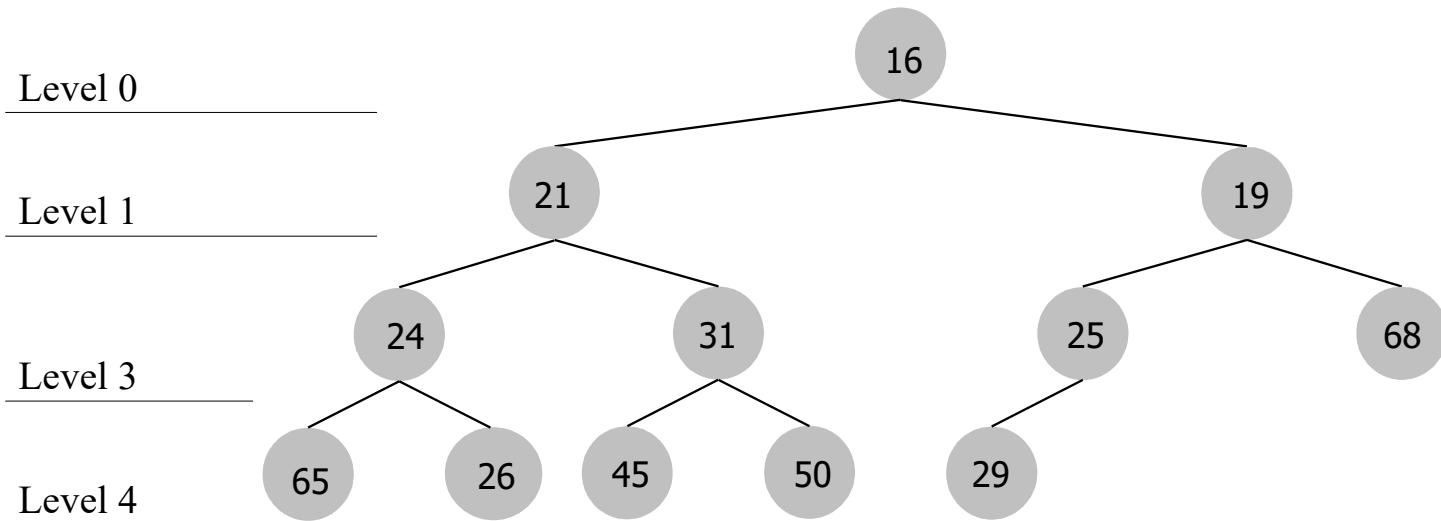


A complete binary tree can be efficiently stored in a vector - the vector does not contain empty cells and all elements are contiguous

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?

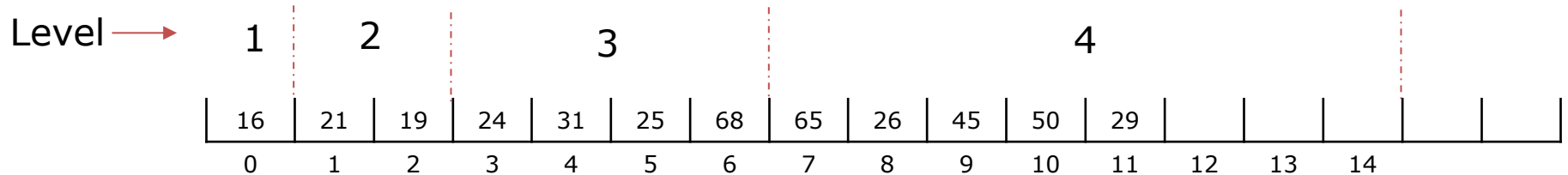
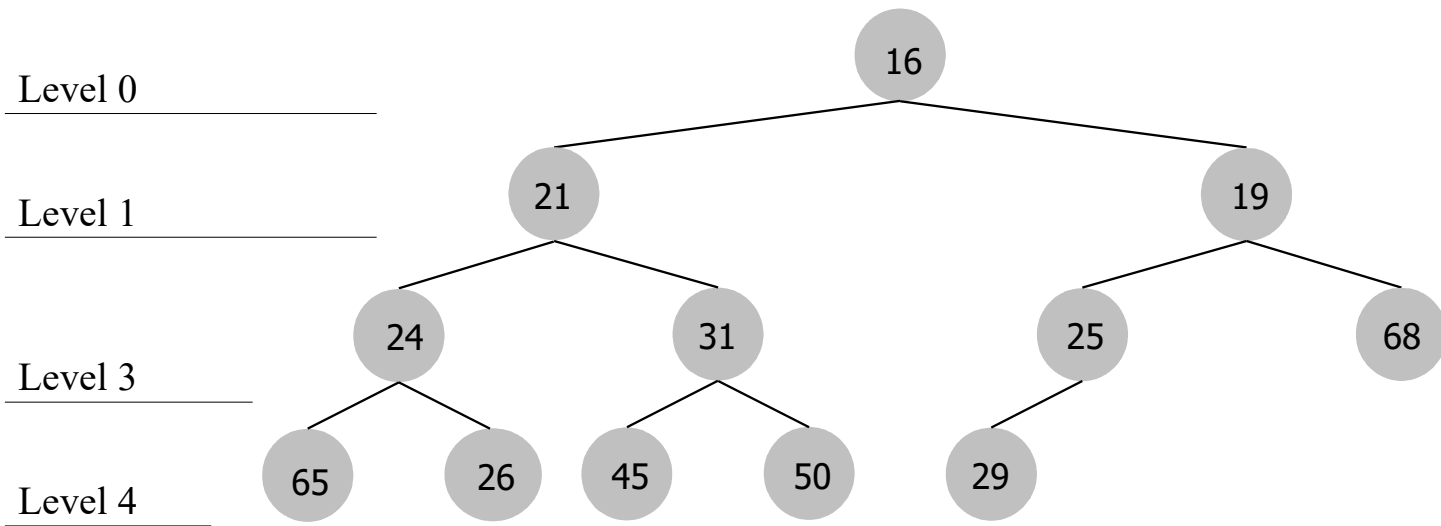
- Being the root at index 0
- Its direct descendants are in the indices 1 and 2
- The descendants of the element at index 3 are at indexes: 7 and 8
-

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

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

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

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

Array-based representation of binary trees

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

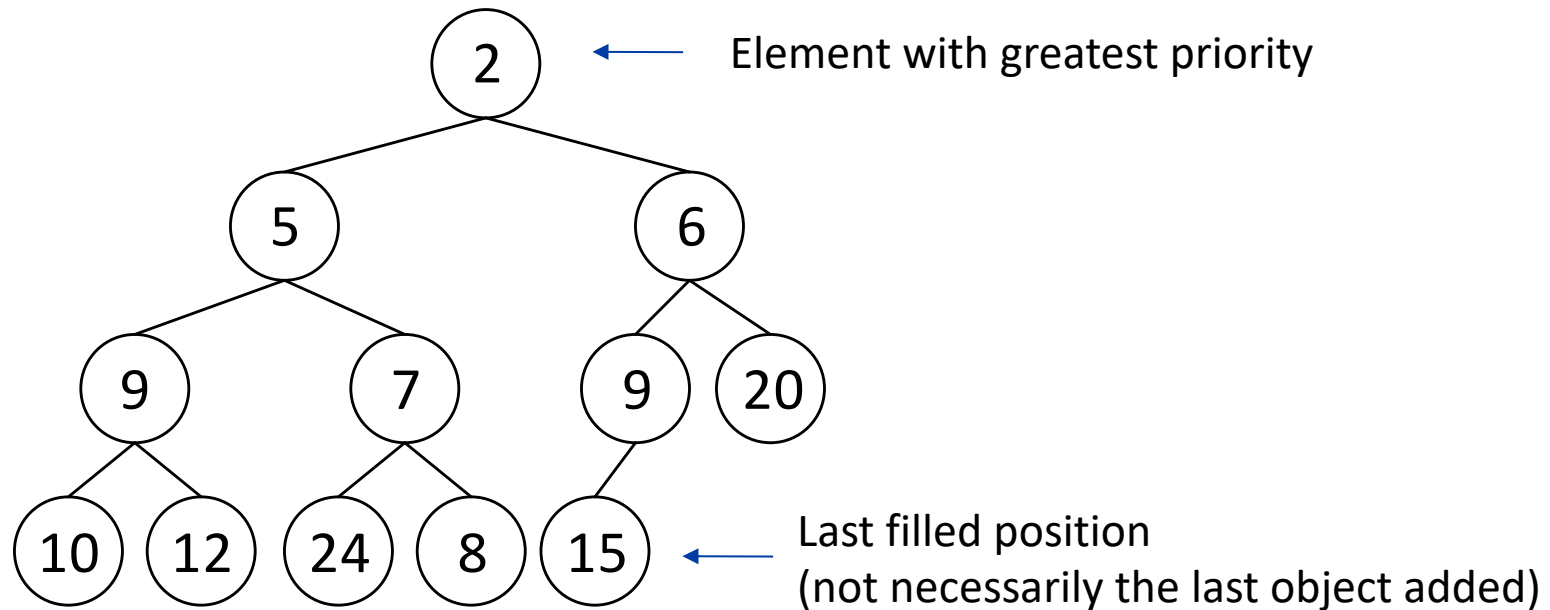
With an array implementation

- 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

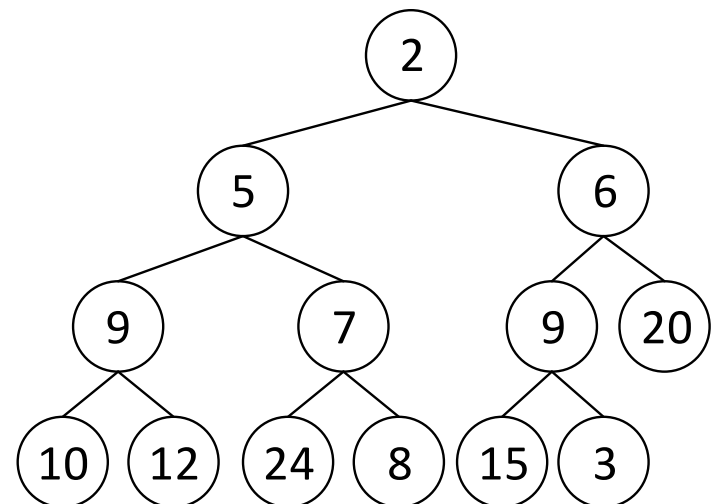
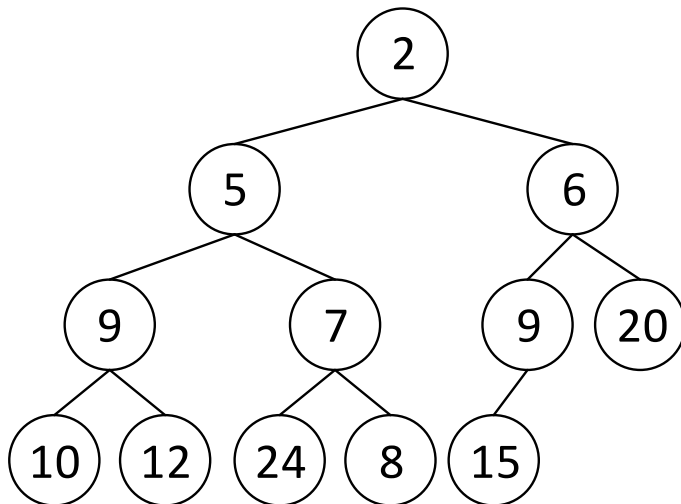
- Two distinct entries in a priority queue can have the same key
- Keys in a priority queue can be arbitrary objects on which an order is defined
- A generic priority queue uses an external comparator object

Heap – Insertion

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

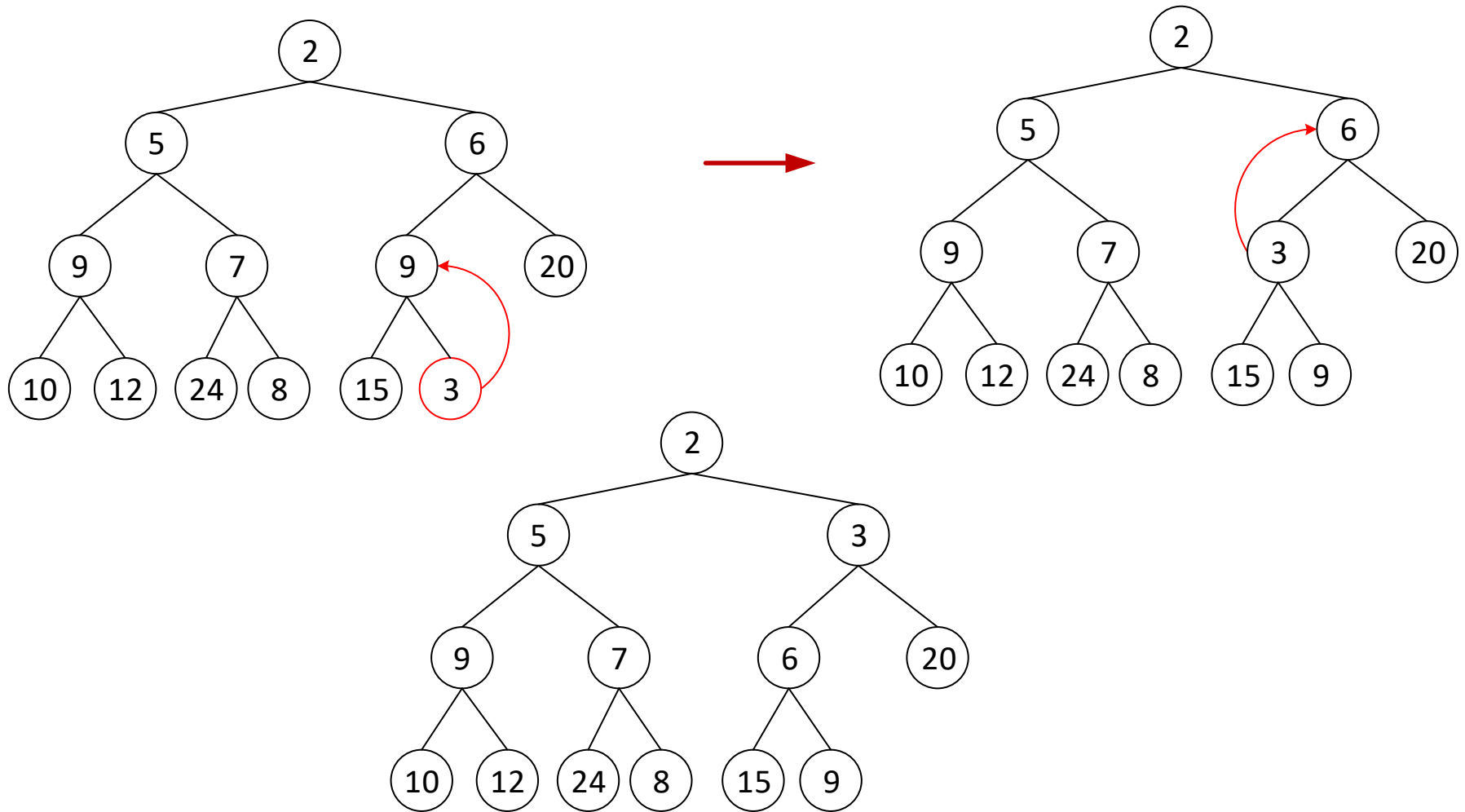
1. the tree remains **complete** - the new element is inserted on the **last level** of the tree the **rightmost** possible
2. the tree remains **orderly** - Fix it by **percolating up**

Insertion of element 3:

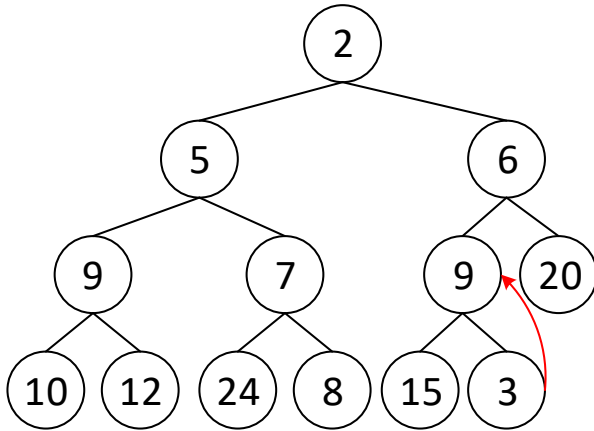


Heap – Insertion: Percolate up

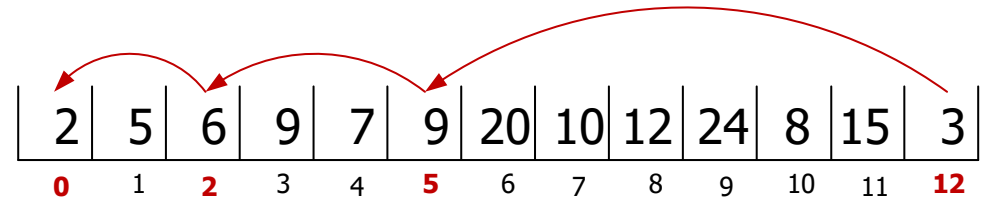
To maintain the tree ordered the new element must be put in the correct place



Percolate up



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



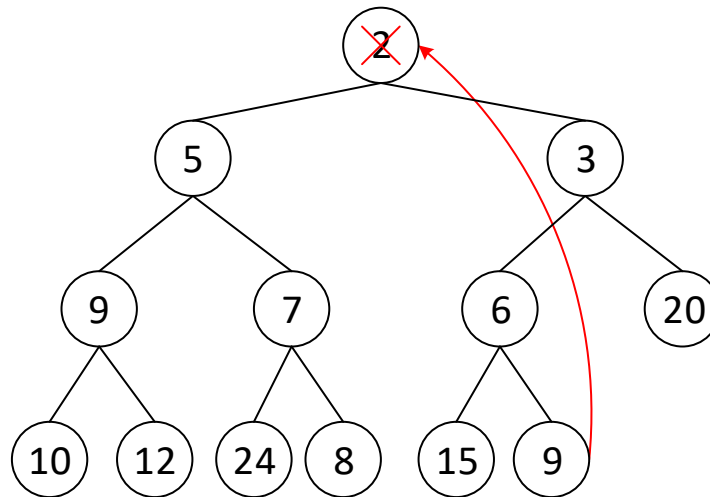
```
Algorithm percolateUp (int i){  
    ind = (i-1)/2;  
    while (ind >= 0 && vector[i] < vector[ind]){  
        swap(vector[i], vector[ind])  
        i = ind  
        ind = (i-1)/2  
    }  
}
```

Complexity(?)

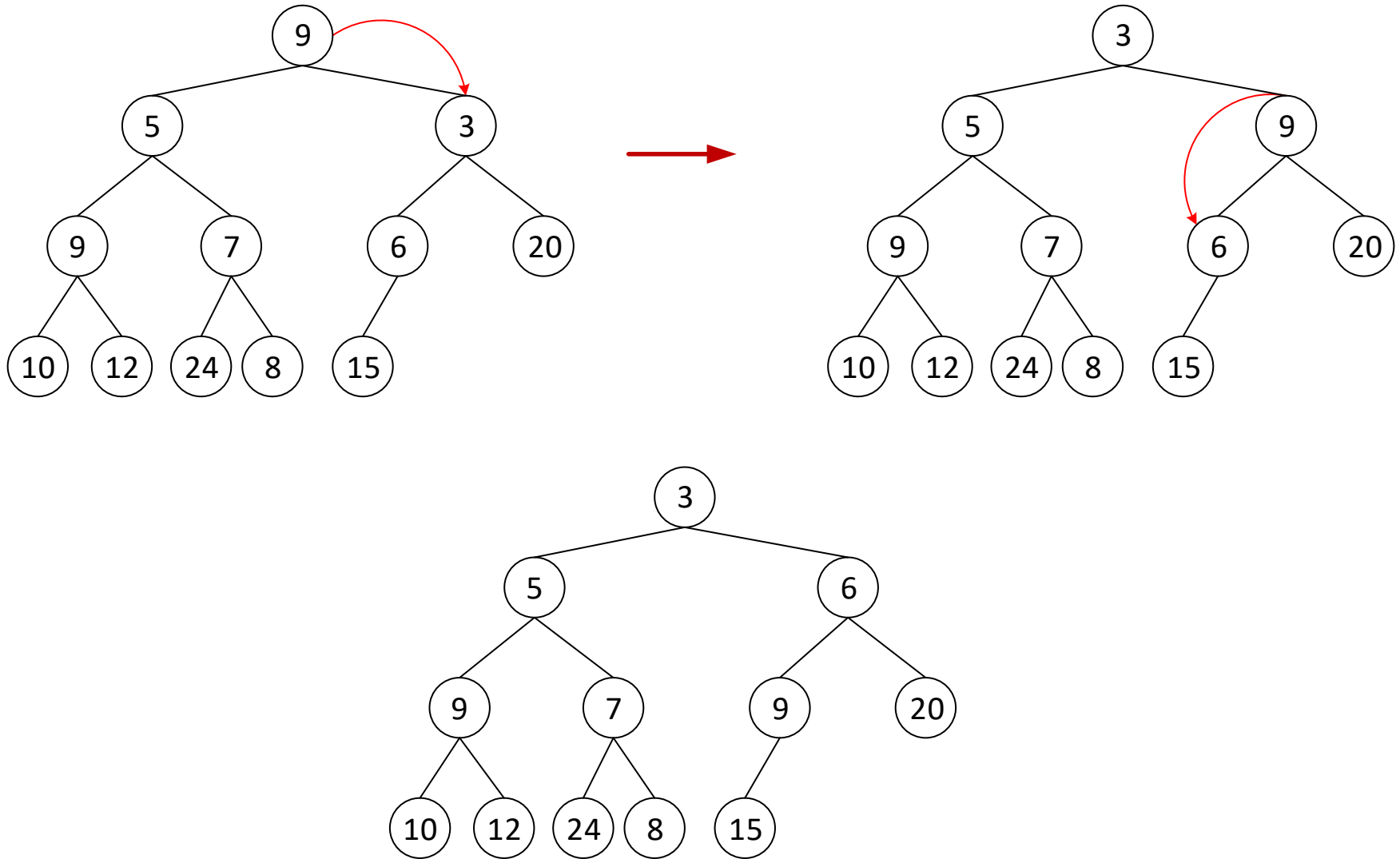
Heap – RemoveMin

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

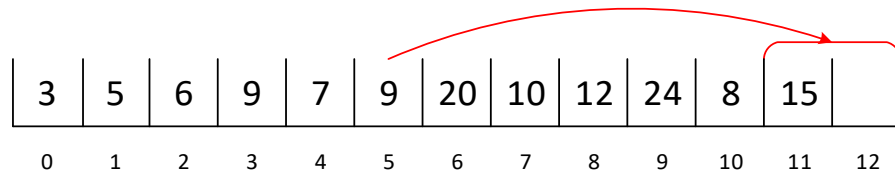
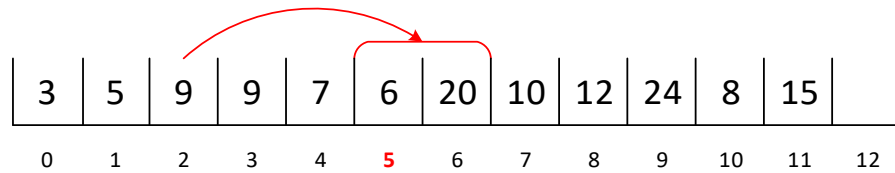
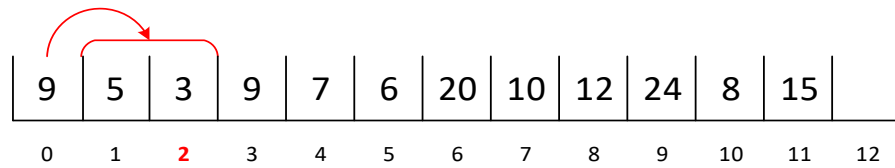
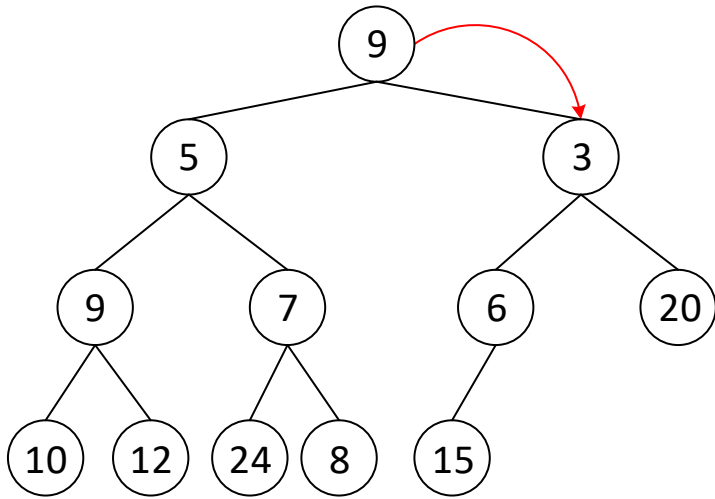
1. the tree remains **complete** – root is replaced by the rightmost element in the last level
2. the tree remains orderly – fix it by **percolating down**



Heap – RemoveMin: Percolate down



Percolate down



Percolate down

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

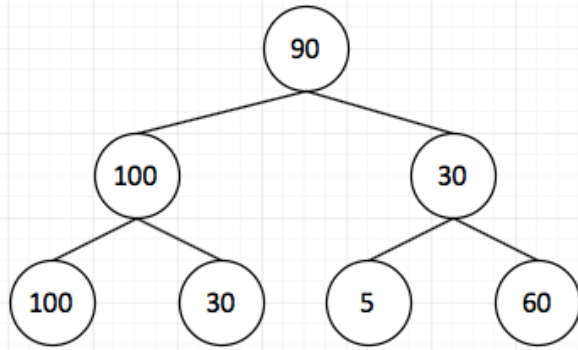
        if (vector[i] > vector[smallindex]) {
            swap(vector[i],vector[smallindex]) //change the elem by the
            i = smallindex                      //child with the highest
            indLeft = 2×i+1                     //priority
            indRight = 2×i+2
        }
        else
      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

Complexity(?)

Batch Bottom-Up Heap Construction

- If we start with an initially empty heap, n successive calls to the insert operation will run in $O(n \log n)$ time, in the worst case
- However, if all n key-value pairs to be stored in the heap **are given in advance**, there is an alternative **batch bottom-up** construction method more efficient
- Intuitively, the **batch bottom-up heap construction** performs a **single percolate-down operation at each internal node** of the tree, rather than a single percolate-up operation from each

Batch Bottom-Up Heap Construction



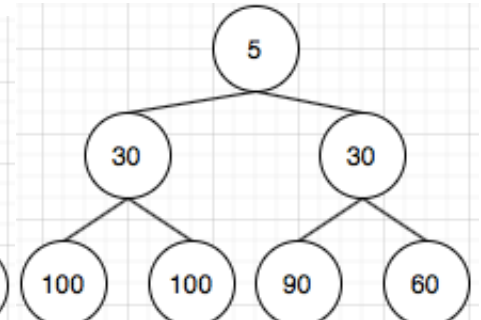
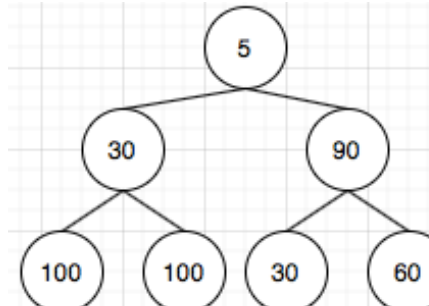
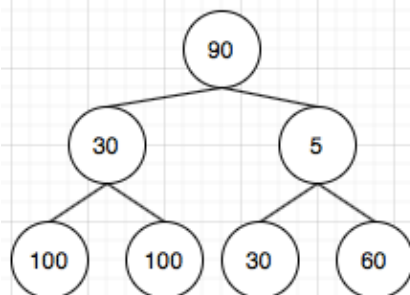
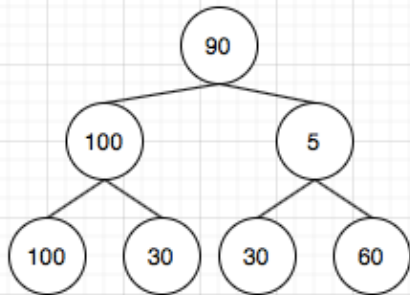
| 90 | 100 | 30 | 100 | 30 | 5 | 60 |
0 1 2 3 4 5 6



Start index

The bottom-up heap construction:

- Starts at the parent of last entry
- Performs percolate-down operation of each internal node of the tree until reaches the root



Asymptotic Analysis of Bottom-Up Heap Construction

- Bottom-up heap construction is asymptotically faster than incrementally inserting n entries into an initially empty heap
- The primary cost of the bottom-up heap construction is due to the percolate-down steps performed at each non-leaf position
- Since more nodes are closer to the bottom of a tree than the top, the **sum of the percolate-down paths is linear**
- Bottom-up construction of a heap with n entries takes **$O(n)$ time**, assuming two keys can be compared in **$O(1)$ time**

Priority Queue - Performance Evaluation

	Array	Sorted Array	Sorted List	Heap
insert(k,v)	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$
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:

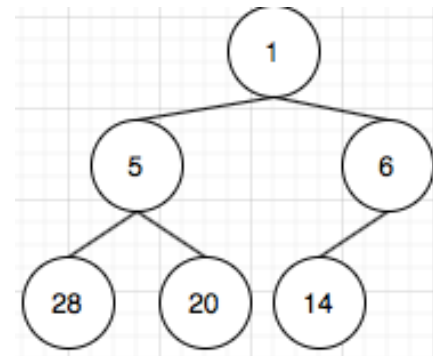
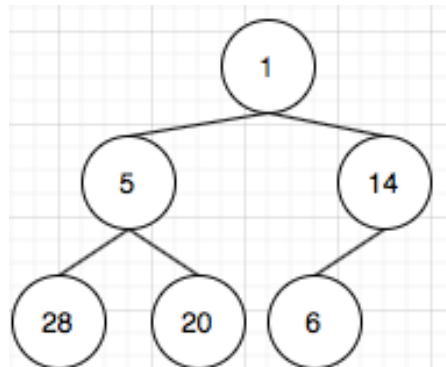
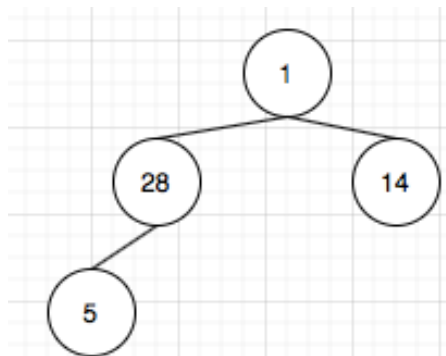
1. Build a heap using the elements of the collection
2. Extract all elements from heap and insert them into the collection

```
Algorithm heapSort (ArrayList<E> vector) {  
    for (int i = 0; i < vector.size(); i++)  
        insert(i,v[i]);  
    for (int i = 0; i < vector.size(); i++)  
        v[i] = removeMin()  
}
```

1 | 28 | 14 | 5 | 20 | 6

1 | 5 | 6 | 28 | 20 | 14

1 | 5 | 6 | 14 | 20 | 28



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