Topic 5

Priority Queue and Heap.

Heap Sort

Aim

- Presentation of an efficient implementation of the model *Priority Queue*
- Representation of the data with an array: *Binary Heap*
- Design of the generic sorting method *Heap Sort*

Contents

1. Introduction

The model Priority Queue

- The Priority Queue (Cola de Prioridad) is a model for a data collection that allows to access to the element of highest priority:

```
public interface ColaPrioridad<E extends Comparable<E>> {
  // Insert x in the queue
  void insertar(E x);
  // IFF !esVacia(): if not empty return the element with highest
priority
  E recuperarMin();
  // IFF !esVacia(): return and delete the element with highest
priority
  E eliminarMin();
  // Return true if the queue is empty
  boolean esVacia();
}
```

2. Heap
Characterisics

- Implementation based on an *array*
- The average cost of *insertar* is constant and logarithmic (worst case)
- The average cost of *eliminarMin* is logarithmic (also in the worst case)
- The cost of *recuperarMin* is constant

2. Heaps
Properties

<u>Structural property:</u> a *heap* is a complete binary tree

- Its height is maximum: $\lfloor \log_2 N \rfloor$
- The cost of the algorithms is in the worst case logarithmic
- The complete binary trees allow for an *array* representation

<u>Sort property</u>:

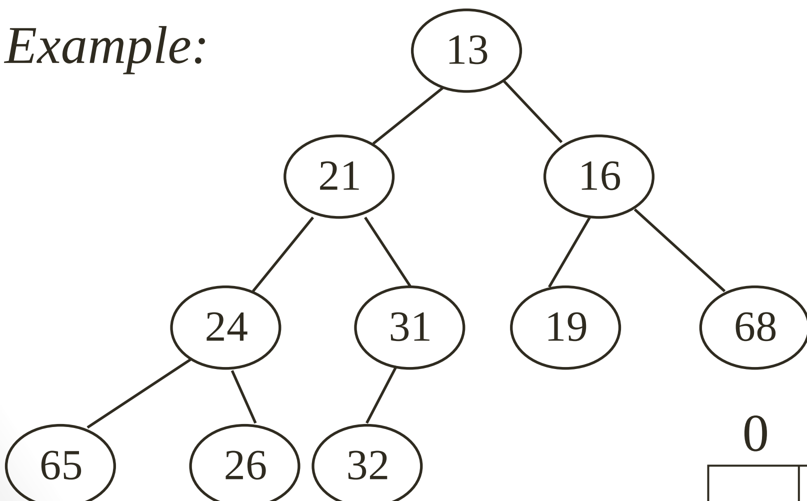
- In a min-heap, the element of a node is always smaller or equal than its children

2. Heaps

Array representation of a complete binary tree

- Data stored in an *array* following the **traversal by level**
- The root is in the position **1**

Example:

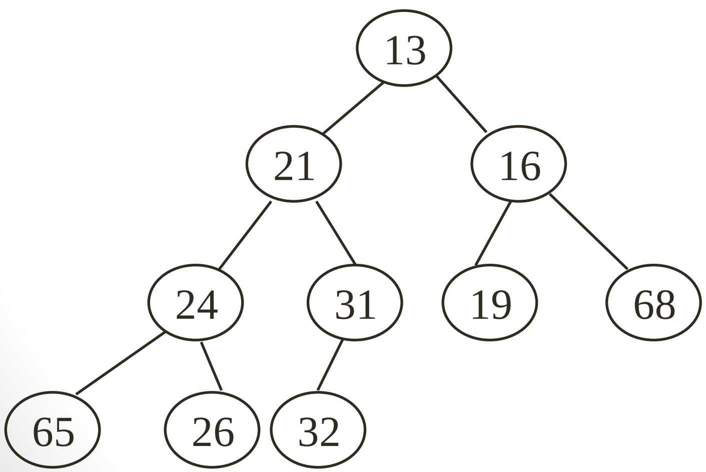


0	1	2	3	4	5	6	7	8	9	10
	13	21	16	24	31	19	68	65	26	32

size = 10

2. Heaps
Implicit representation

- Give the i-th node:
  - Position of its left child: 2*i (if 2*i ≤ size)
  - Position of its right child: 2*i+1 (if 2*i+1 ≤ size)
  - Position of its father: i/2 (if i ≠ 1)



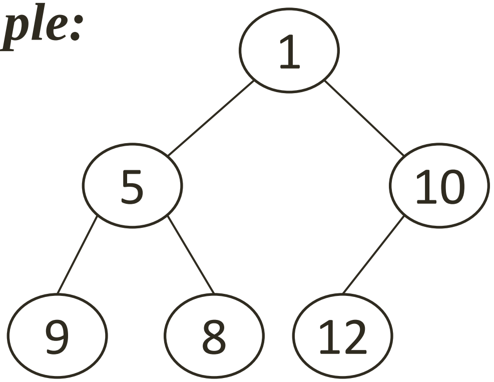
0	1	2	3	4	5	6	7	8	9	10
	13	21	16	24	31	19	68	65	26	32

size = 10

2. Heaps
Properties

- Every path from the root to a leaf is sorted sequence:
  - 1, 5, 9
  - 1, 5, 8
  - 1, 10, 12
- The root is the node with the smallest element (or greatest in a Max-Heap)
- Each subtree of a Heap is also an Heap

Example:



0	1	2	3	4	5	6	7	
	1	5	10	9	8	12		…

size = 6

3. The class *MonticuloBinario*

Attributes and constructor

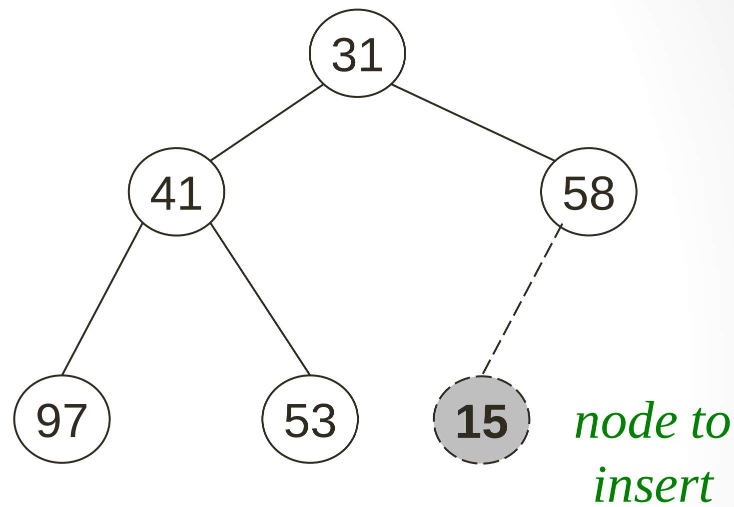
```
public class MonticuloBinario<E extends Comparable<E>>
       implements ColaPrioridad<E> {
  // Attributes
  protected static final int CAPACIDAD_INICIAL = 50;
  protected E elArray[];
  protected int talla;

  // Constructor of an empty min-heap
  @SuppressWarnings(“unchecked”)
  public MonticuloBinario() {
    talla = 0;
    elArray = (E[]) new Comparable[CAPACIDAD_INICIAL];
  }
```

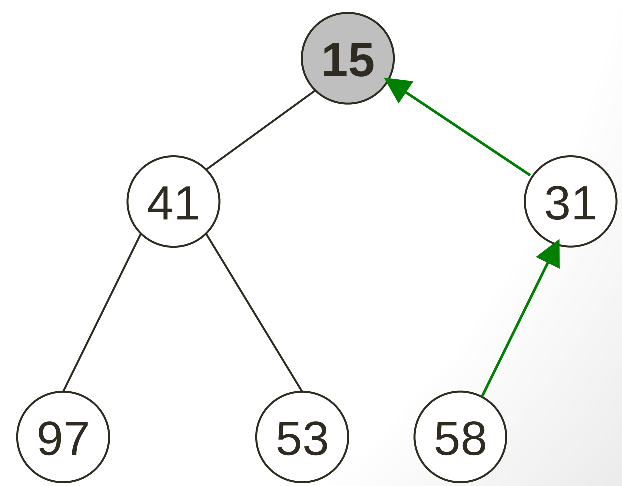
3. The class *MonticuloBinario*

The method insertar (1/3)

- **Step 1:** the new element is inserted in the first available position of the array: `elArray[talla + 1]`



- **Step 2:** the new element is compared with its predecessors and moved in order to have the sort property accomplished



3. The class *MonticuloBinario*

The method insertar (2/3)

```
public void insertar(E x) {
  // do we have space in the array for another element?
  if (talla == elArray.length - 1) duplicarArray();
  // hole is the position where x will be inserted
  int hole = ++talla;
  // it is moved in order to have the sort property
  accomplished
  while (hole > 1 && x.compareTo(elArray[hole/2]) < 0) {
    elArray[hole] = elArray[hole/2];
    hole = hole/2;
  }
  // now we know in what position to insert the new
  element
  elArray[hole] = x;
}
```

3. The class *MonticuloBinario*

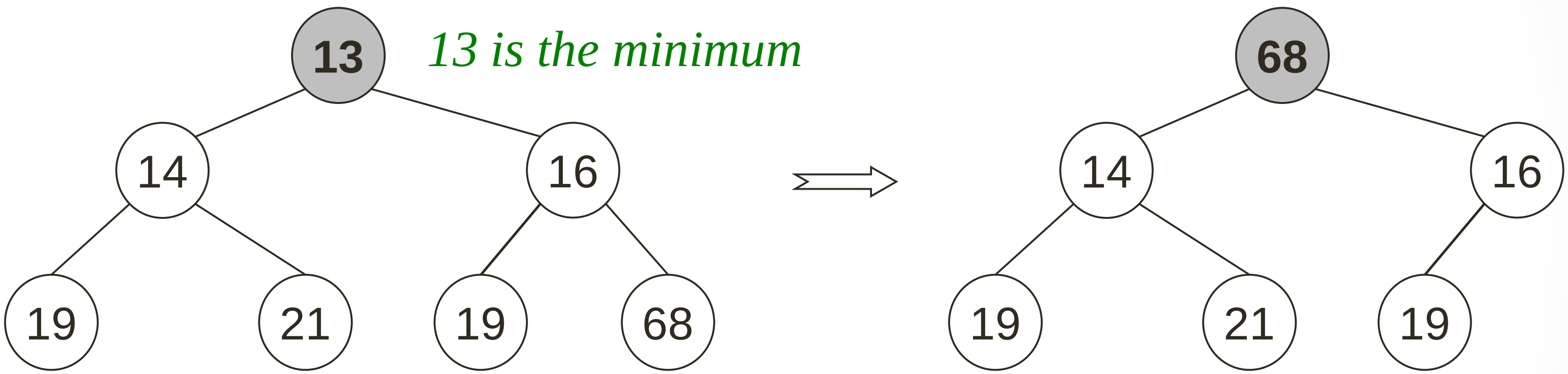
The method insertar (3/3)

- Worst case: the complexity is $O(\log_2 N)$ if the added element is the new minimum
- Best case: when the element to insert is greater than its father (only one comparison)
- It has been empirically proofed that on average 2.6 comparisons are needed to insert a new element (constant complexity)

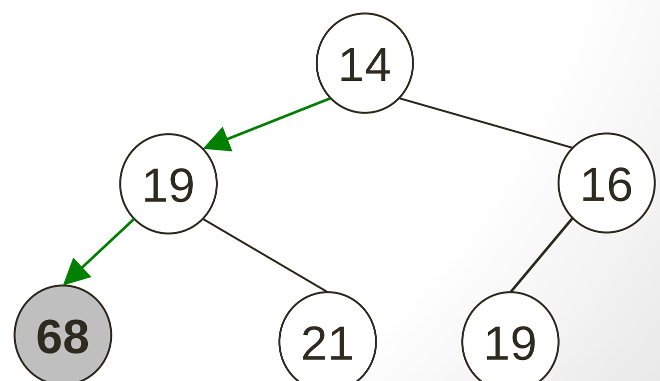
3. The class *MonticuloBinario*

The method eliminarMin (1/3)

- **<u>Step 1:</u>** the minimum is in the root. The root is substituted by the last element of the Heap



- **<u>Step 2:</u>** the new root is moved down via its children in order to accomplish the sort property:



3. The class *MonticuloBinario*

The method eliminarMin – heapify (2/3)

- The method *heapify* (*hundir)*:

```
private void heapify(int hole) {
  E aux = elArray[hole];
  int child = hole * 2;
  boolean isHeap = false;
  while (child <= talla && !isHeap) {
    if (child != talla &&
        elArray[child+1].compareTo(elArray[child]) < 0)
        child++;    // We choose the smalles of the two children
    if (elArray[child].compareTo(aux) < 0) {  // we move down
      elArray[hole] = elArray[child];
      hole = child;
      child = hole*2;
    } else isHeap = true; // The sort property is accomplished
  }
  elArray[hole] = aux;
}
```

3. The class *MonticuloBinario*

The method eliminarMin (3/3)

```java
// IFF !esVacia(): delete and return the smallest element
public E eliminarMin() {
  E theMin = recuperarMin();
  // the root is substituted with the last element
  elArray[1] = elArray[talla--];
  // the new root is moved down (sort property)
  heapify(1);
  return theMin;
}

// IFF!esVacia(): return the smallest element
public E recuperarMin() {
  return elArray[1];
}
```

3. The class *MonticuloBinario*

The method buildHeap (arreglarMonticulo)

- Given a complete binary tree it allows to accomplish the sort property
- It moves down all nodes in an inverse order wrt traversal by levels

```
private void buildHeap() //arreglarMonticulo
{
for (int i = talla / 2; i > 0; i--)
    heapify(i);
}
```

3. The class *MonticuloBinario*

The method buildHeap (arreglarMonticulo)

It has linear time complexity:

Leaves have height 0 and the root height $\lfloor \log_2 n \rfloor$

There are $\lfloor \log_2 n \rfloor - h$ nodes at height *h*

The cost of heapify a node at height *h* is $\Theta(h)$

$$T_{arreglarMonticulo}(n) = \sum_{h=0}^{\lfloor \log_2 n \rfloor} h \cdot 2^{\lfloor \log_2 n \rfloor - h} =$$

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} h \cdot \frac{2^{\lfloor \log_2 n \rfloor}}{2^h} \leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} h \cdot \frac{2^{\log_2 n}}{2^h} = \sum_{h=0}^{\lfloor \log_2 n \rfloor} h \cdot \frac{n}{2^h} =$$

$$n \cdot \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \leq n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} = 2 \cdot n \in \Theta(n)$$

4. HeapSort

Fast sorting with HeapSort

- The cost of HeapSort is $O(N*\log_2 N)$
  - *QuickSort* has a complexity $O(N^2)$ as worst case
  - *MergeSort* needs an auxiliar array
- This algorithm is based on the properties of a heap
  - <u>First step</u>: all the elements of an array to be sorted are stored in a heap
  - <u>Second step</u>: the smallest element is extracted (root) in an iterative way

4. HeapSort

Insertion of the data of an array in a Heap

- The most efficient way to insert the data of an array in a Heap is with the method *arreglarMonticulo (buildHeap)*:

```
@SuppressWarnings("unchecked")     // Constructor
public MonticuloBinario(E v[]) {
  talla = v.length;                // Data is copied
  elArray = (E[]) new Comparable[talla+1];
  System.arraycopy(v, 0, elArray, 1, talla);
  buildHeap();                // Sort property is accomplished
}
```

- The cost of the constructor is O(N), where N is the size of the array

4. HeapSort

Algorithm

```
public class Ordenacion {
  public static <E extends Comparable<E>> void heapSort(E v[]) {
    // Creating the heap from the array
    MonticuloBinario<E> heap = new MonticuloBinario<E>(v);
    // Extracting data from the heap in a sorted way
    for (int i = 0; i < v.length; i++)
      v[i] = heap.eliminarMin();
  }
}
```

- Cost HeapSort = cost constructor + N * cost of *eliminarMin*

  $T_{heapSort}(N) \in O(N) + N^*O(\log_2 N) = O(N^*\log_2 N)$

- Cost HeapSort to sort only the first *k* elements of the array: $O(N + k^*\log_2 N)$

References

- Data structures, algorithms, and applications in Java, *Sahni* (Chapter 13)
- Data Structures and Algorithms in Java (4th edition), *Goodrich y Tamassia* (Chapter 8)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "*Introduction to Algorithms*" (2nd edition). The MIT Press, 2007 (Chapter 6)
- M.A. Weiss. "*Estructuras de Datos en Java*", Adisson-Wesley, 2000 (Sect. 1 – 5 of Chapter 20)
- G. Brassard y P. Bratley . "*Fundamentos de Algoritmia*", Prentice Hall, 2001