# Tema 5

## Cola de Prioridad y Montículo Binario

# Objetivos

- Presentar una implementación eficiente del modelo *Cola de Prioridad*
- Aprender los conceptos básicos de árboles y árboles binarios
- Representación contigua (implícita) de los datos mediante un array
  - Esta implementación recibe el nombre de *Montículo Binario* o *Binary Heap*
- Diseño del método genérico de ordenación *Heap Sort* en base a esta implementación

# Contenidos

1. El modelo Cola de Prioridad
2. Montículo Binario
    - Características
    - Propiedades
    - Representación implícita de un Árbol Binario completo
3. La clase *MonticuloBinario*
4. Ordenación rápida según *Heap Sort*

# 1. Introducción
## *El modelo Cola de Prioridad*

- La Cola de Prioridad es un modelo para una colección de datos en el que las operaciones características son aquellas que permiten acceder al dato de mayor prioridad:

```
public interface ColaPrioridad<E extends Comparable<E>> {
  // Añade x a la cola
  void insertar(E x);
  // SII !esVacia(): devuelve el dato de mayor prioridad
  E recuperarMin();
  // SII !esVacia(): devuelve y elimina el dato de mayor prioridad
  E eliminarMin();
  // Devuelve true si la cola está vacía
  boolean esVacia();
}
```

# 2. Montículo Binario
## *Características*

- Implementación basada en un sencillo *array*
- El coste promedio de *insertar* es constante y logarítmico en el peor de los casos
- El coste promedio y en el peor de los casos de *eliminarMin* es logarítmico
- El coste de *recuperarMin* es constante

# 2. Montículo Binario
## *Propiedades*

**Propiedad estructural:** un *heap* es un árbol binario completo

- Su altura es, a lo sumo, $\lfloor \log_2 N \rfloor$
- Se asegura entonces un coste logarítmico en el peor caso si los algoritmos implican la exploración de una rama entera
- Los árboles binarios completos permiten una representación implícita sobre *array*

**Propiedad de orden**:

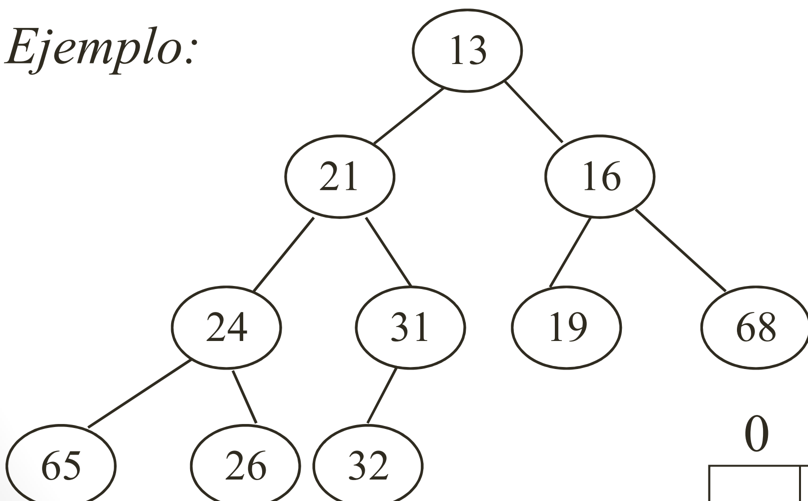
- En un heap (minimal), el dato de un nodo es siempre menor o igual que el de sus hijos

# 2. Montículo Binario

## *Representación implícita/contigua de un árbol binario completo*

- Se almacena en un *array* según su **recorrido por niveles**
- El nodo raíz se sitúa en la posición **1** (la 0 se deja libre, lo que facilita el cálculo de los hijos de un nodo)

*Ejemplo:*




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

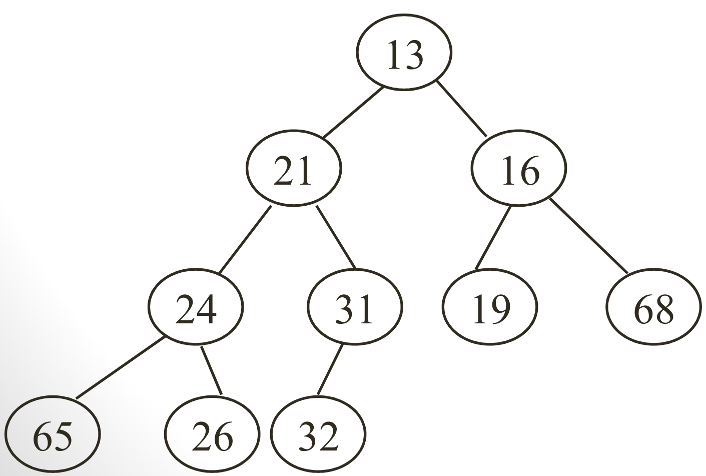
*talla* = 10

# 2. Montículo Binario

## *Representación implícita*

- Para el nodo i-ésimo:
  - Su hijo izquierdo está en la posición 2*i (si $2*i \le$ talla)
  - Su hijo derecho está en la posición 2*i+1 (si $2*i+1 \le$ talla)
  - Su padre está en la posición i/2 (si $i \ne 1$)

*¿Cómo se calcularían los hijos y el padre de un nodo si se utilizara la posición cero del array?*



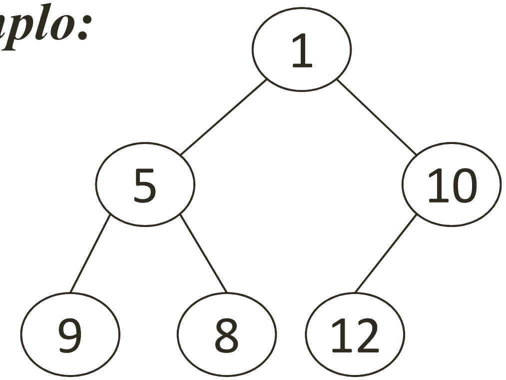

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 13 | 21 | 16 | 24 | 31 | 19 | 68 | 65 | 26 | 32 |

*talla* = 10

# 2. Montículo Binario
## *Propiedades*

- Todo camino desde la raíz a una hoja es una secuencia ordenada:
  - 1, 5, 9
  - 1, 5, 8
  - 1, 10, 12
- La raíz del árbol es el nodo de valor mínimo (o máximo en un Montículo Binario Maximal o Max-Heap)
- Todo subárbol de un Heap es también un Heap

***Ejemplo:***




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 5 | 10 | 9 | 8 | 12 | | … |

*talla* = 6

# 3. La clase *MonticuloBinario*

## *Atributos y constructor*

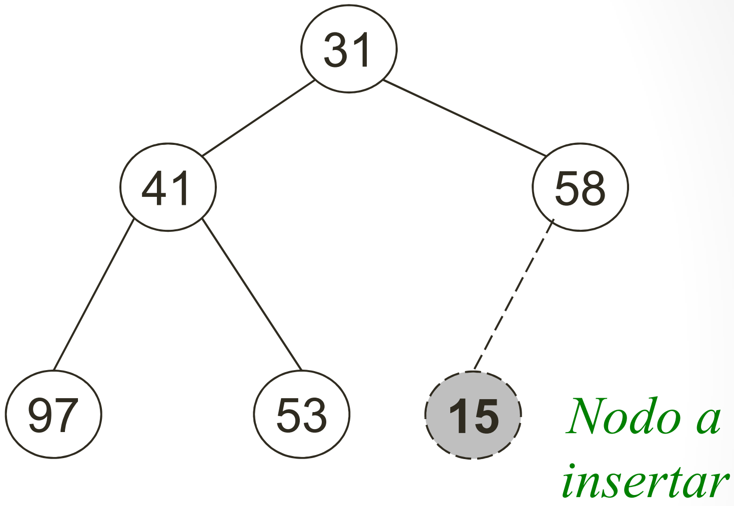
```
public class MonticuloBinario<E extends Comparable<E>>
      implements ColaPrioridad<E> {
  // Atributos
  protected static final int CAPACIDAD_INICIAL = 50;
  protected E elArray[];
  protected int talla;

  // Constructor de un heap minimal vacío
  @SuppressWarnings("unchecked")
  public MonticuloBinario() {
    talla = 0;
    elArray = (E[]) new Comparable[CAPACIDAD_INICIAL];
  }
```

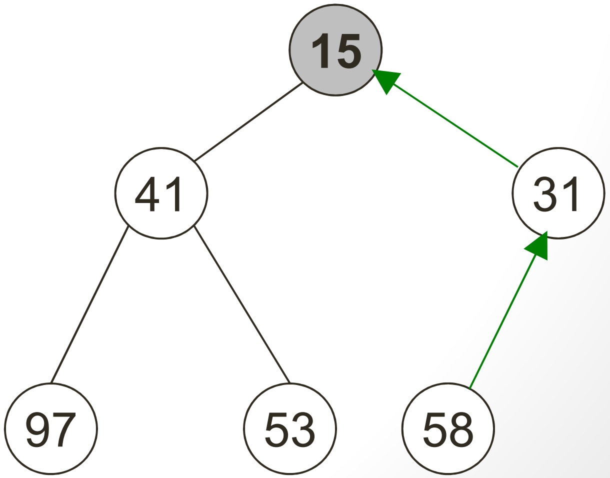
# 3. La clase *MonticuloBinario*

## *El método insertar (1/3)*

- **<u>Paso 1:</u>** se inserta el nuevo elemento en la primera posición disponible del vector: `elArray[talla + 1]`




- **<u>Paso 2:</u>** se reflota sobre sus antecesores hasta que no viole la propiedad de orden



# 3. La clase *MonticuloBinario*

## *El método insertar (2/3)*

```
public void insertar(E x) {
  // ¿hay espacio en el array para el nuevo dato?
  if (talla == elArray.length - 1) duplicarArray();
  // Incrementamos la talla e insertamos x
  elArray[++talla] = x;
  // Reflotamos hasta que no viole la propiedad de orden
  reflotar(talla);
}
protected void reflotar(int hueco) {
  E aux = elArray[hueco];
  while (hueco > 1 && aux.compareTo(elArray[hueco/2]) < 0) {
    elArray[hueco] = elArray[hueco/2];
    hueco = hueco/2;
  }
  elArray[hueco] = aux;
}
```

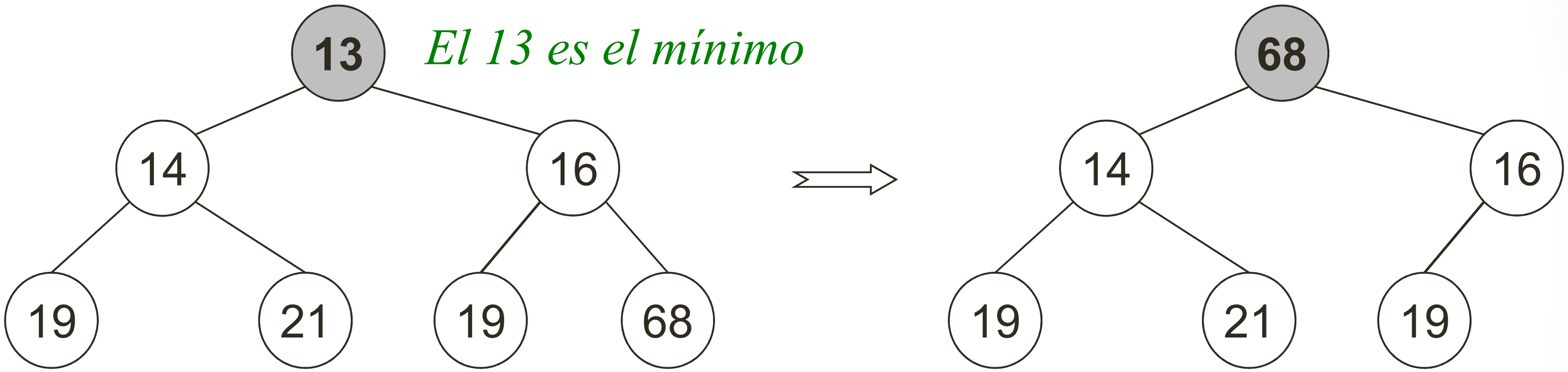
# 3. La clase *MonticuloBinario*
## *El método insertar (3/3)*

- El coste es $O(\log_2 N)$ si el elemento añadido es el nuevo mínimo
- El caso más favorable es cuando el elemento a insertar es mayor que su padre (requiere por lo tanto una única comparación)
- Se ha demostrado que, en promedio, se requieren 2.6 comparaciones para llevar a cabo una inserción (coste constante)

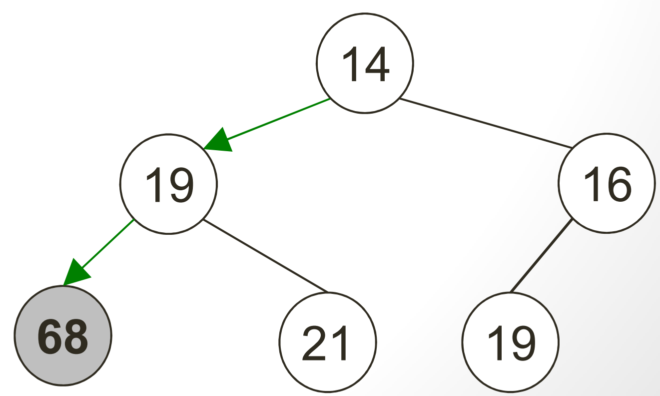
# 3. La clase *MonticuloBinario*

## *El método eliminarMin - funcionamiento (1/3)*

- **<u>Paso 1:</u>** el mínimo está en el nodo raíz. El dato del nodo raíz se sustituye entonces por el último elemento del Heap




- **<u>Paso 2:</u>** la nueva raíz se hunde a través de sus hijos hasta no violar la propiedad de orden:



# 3. La clase *MonticuloBinario*

## *El método eliminarMin – hundir (2/3)*

- El método *hundir* (*heapify)* hunde un nodo a través del *heap* hasta que no se viole la propiedad de orden:

```
private void hundir(int hueco) {
  E aux = elArray[hueco];
  int hijo = hueco * 2;
  boolean esHeap = false;
  while (hijo <= talla && !esHeap) {
    if (hijo != talla &&
          elArray[hijo+1].compareTo(elArray[hijo]) < 0)
          hijo++;                // Elegimos el menor de los hijos
    if (elArray[hijo].compareTo(aux) < 0) {       // Hundimos
      elArray[hueco] = elArray[hijo];
      hueco = hijo;
      hijo = hueco*2;
    } else esHeap = true;      // Ya se cumple la prop. de orden
  }
  elArray[hueco] = aux;
}
```

# 3. La clase *MonticuloBinario*

## *El método eliminarMin (3/3)*

```
// SII !esVacia(): devuelve y elimina el dato de menor prioridad
public E eliminarMin() {
  E elMinimo = recuperarMin();
  // Sustituimos el raíz por el último elemento
  elArray[1] = elArray[talla--];
  // Se hunde la nueva raíz hasta que no viole la propiedad de orden
  hundir(1);
  return elMinimo;
}

// SII !esVacia(): devuelve el dato de menor prioridad
public E recuperarMin() {
  return elArray[1];
}
```

# 3. La clase *MonticuloBinario*

## *El método arreglarMonticulo (build-heap)*

- Restablece la propiedad de orden a partir de un Árbol Binario Completo para obtener un Montículo Binario
- Se basa en hundir los nodos en orden inverso al recorrido por niveles

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

# 3. La clase *MonticuloBinario*

## *El método arreglarMonticulo (build-heap)*

- Tiene una complejidad temporal lineal:
  - Las hojas tienen altura 0 y la raíz altura $\lfloor \log_2 n \rfloor$
  - Hay $2^{\lfloor \log_2 n \rfloor - h}$ nodos a una altura *h*
  - El coste de hundir un nodo de altura *h* es $\Theta(h)$
  - $T_{arreglarMonticulo}(n) = \sum_{h=0}^{\lfloor \log_2 n \rfloor} h \cdot 2^{\lfloor \log_2 n \rfloor - h} = \sum_{h=0}^{\lfloor \log_2 n \rfloor} h \cdot \frac{2^{\lfloor \log_2 n \rfloor}}{2^h} \leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} h \cdot \frac{2^{\log_2 n}}{2^h} = \sum_{h=0}^{\lfloor \log_2 n \rfloor} h \cdot \frac{n}{2^h} = \mathrm{n} \cdot \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \leq \mathrm{n} \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} = 2 \cdot n \in \Theta(\mathrm{n})$

# 4. HeapSort

## *Ordenación rápida según HeapSort*

- El coste de HeapSort es $O(N*\log_2 N)$
  - *QuickSort* tiene un coste $O(N^2)$ en el peor de los casos
  - *MergeSort* requiere un vector auxiliar
- Este algoritmo de ordenación se basa en las propiedades de los Heaps
  - Primer paso: almacenar todos los elementos del vector a ordenar en un montículo (heap)
  - Segundo paso: extraer el elemento raíz del montículo (el mínimo) en sucesivas iteraciones, obteniendo el conjunto ordenado

# 4. HeapSort

## *Inserción de los datos del vector en el Heap*

- La forma más eficiente de insertar los elementos de un vector en un *Heap* es mediante el método *arreglarMonticulo*:

```
@SuppressWarnings("unchecked")      // Constructor a partir de un vector
public MonticuloBinario(E v[]) {
    talla = v.length;                // Copiamos los datos del vector
    elArray = (E[]) new Comparable[talla+1];
    System.arraycopy(v, 0, elArray, 1, talla);
    arreglarMonticulo();             // Arreglamos la propiedad de orden
}
```

- El coste de este constructor es O(N), siendo N la talla del vector

# 4. HeapSort
## *Algoritmo*

```
public class Ordenacion {
  public static <E extends Comparable<E>> void heapSort(E v[]) {
    // Creamos el heap a partir del vector
    MonticuloBinario<E> heap = new MonticuloBinario<E>(v);
    // Vamos extrayendo los datos del heap de forma ordenada
    for (int i = 0; i < v.length; i++)
      v[i] = heap.eliminarMin();
  }
}
```

- Coste HeapSort = coste constructor + N * coste de *eliminarMin*

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

- HeapSort puede modificarse fácilmente para ordenar sólo los *k* primeros elementos del vector con coste $O(N + k*\log_2 N)$

# Bibliografía


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