Tema 12- Implementación de Cola de Prioridad y Ordenación Rápida según un Montículo Binario

Germán Moltó
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Índice general:

Objetivos y Bibliografía

- Presentar una implementación eficiente del modelo Cola de Prioridad.
- Conocer la Estructura Montículo Binario, implementando sus principales operaciones.
- Retomar el problema de la ordenación eficiente de un array, analizando el método de ordenación genérico Heap Sort.
- Bibliografía básica:
  - Libro de M.A. Weiss, "Estructuras de Datos en Java" (Adisson-Wesley, 2000).
    - Apartados 1 – 5 del capítulo 20

El Modelo Cola de Prioridad

- Colección de Datos que tienen asociada una prioridad que determina el orden en el que se accede a dichos datos.

```
package librerias.estructurasDeDatos.modelos;
public interface ColaPrioridad<E extends Comparable<E>>{
        void insertar(E x);
        E recuperarMin();
        E eliminarMin();
        boolean esVacia();
    }
```

- Precondición: Los métodos recuperarMin() y eliminarMin() se tienen que aplicar sobre ColaPrioridad no vacías.
- La prioridad de los objetos almacenados en la ColaPrioridad se especifica al implementar la interfaz Comparable<E>.

Motivación: Posibles Implementaciones

- Coste Medio de las principales operaciones de ColaPrioridad con diferentes implementaciones subyacentes.

Representación	insertar	recuperarMin	eliminarMin
Lista Enlazada	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$
Lista Enlazada Ordenada	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$
Árbol Binario de Búsqueda (Equilibrado)	$\Theta(\log_2(N))$	$\Theta(\log_2(N))$	$\Theta(\log_2(N))$

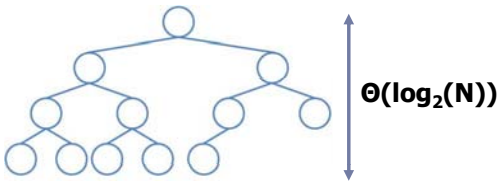
Motivación: Posibles Implementaciones (II)

- ¿Por qué una nueva representación si el ABB ya ofrece costes medios de $\Theta(\log_2(N))$?
- El ABB es una estructura demasiado potente para realizar una búsqueda dinámica restringida.
  - En ColaPrioridad no queremos buscar *cualquier* elemento sino únicamente el elemento mínimo.
  - El ABB, en el peor de los casos, tiene una cota lineal con el número de elementos (árbol degenerado en rama).
- ¿Por qué exigir una propiedad de ordenación fuerte como la del ABB? ¿Qué tal una ordenación parcial?
  - Utilizamos una representación más eficaz para realizar una búsqueda dinámica restringida al dato con mayor prioridad.

El Montículo Binario (I)

- Un Montículo Binario o *Heap* cumple dos propiedades:

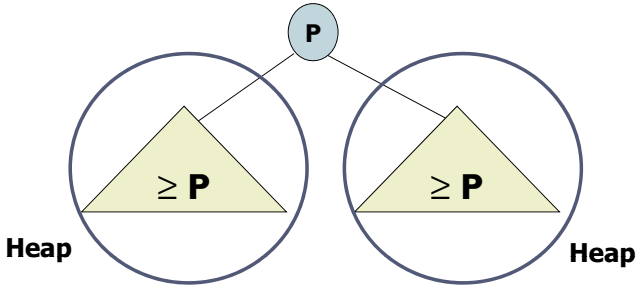
1. Propiedad **Estructural**: Es un Árbol Binario Completo.
   - Árbol Binario Completo:
     - Todos los niveles completos salvo quizás el último, dónde todas sus hojas están lo más a la izquierda posible.
     - Su altura es $\Theta(\log_2(N))$, lo que asegura un coste logarítmico aún en el peor de los casos (exploración del camino desde la raíz hasta una hoja).



El Montículo Binario (II)

2. Propiedad de **Órden**:
   - Para cada nodo X con Padre P se cumple que el dato de P es menor o igual que el dato en X.
   - El dato del Padre nunca es mayor que el Dato de los hijos.
   - Caso especial: Nodo raíz NO tiene padre.

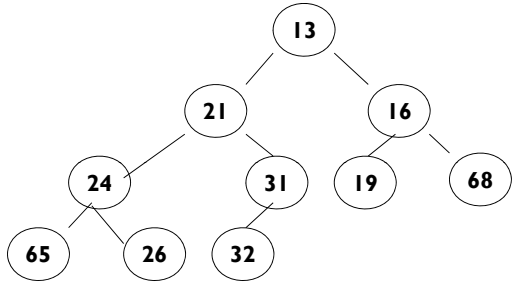
- Nota: Ésta es la definición de Montículo Binario Minimal.



El Montículo Binario: Propiedades

- Propiedades que se derivan de la definición de Heap:

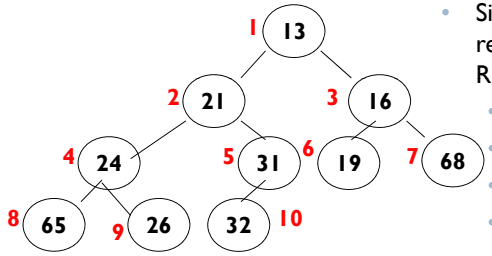
1. Todas las ramas del árbol son secuencias ordenadas.
   - 13, 21, 24, 65
   - 13, 16, 68
   - …
2. La raíz del árbol es el nodo de valor mínimo.
   - No hay nadie por encima de él que pueda tener un menor valor.
3. Todo subárbol de un Montículo Binario es también un Montículo Binario



¿Puede utilizarse la propiedad de orden para listar de forma ordenada sus datos en tiempo lineal?

Montículo Binario: Representación Contigua

- Para representar un Árbol Binario Completo sin ambigüedad, basta con almacenar el resultado de su Recorrido Por Niveles en un array



- Si la i-ésima componente del array representa el i-ésimo Nodo de su Recorrido por Niveles:
  - **Hijo Izquierdo** en **2*i**, si 2*i ≤ N
  - **Hijo Derecho** en **2*i+1**, si 2*i+1 ≤ N
  - **Padre** en **i/2**, si i≠1
  - N es el tamaño del Árbol Binario

$$\forall i : 1 \leq i \leq tallaActual : elArray[i/2] \leq elArray[i]$$

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

Implementación en Java: MonticuloBinario

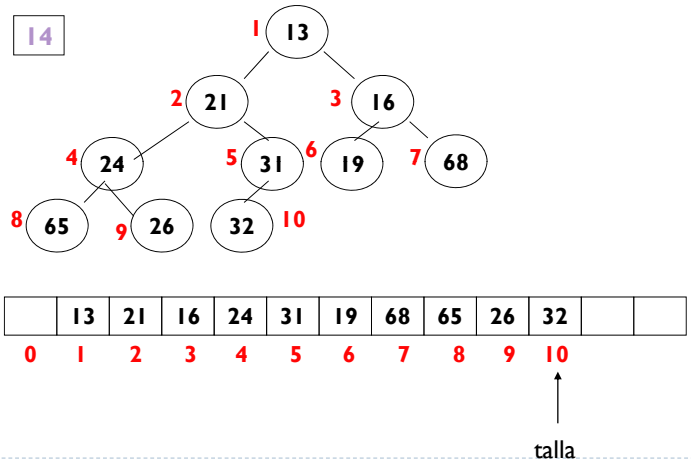
```
package librerias.estructurasDeDatos.jerarquicos;
import librerias.estructurasDeDatos.modelos.*;
public class MonticuloBinario< E extends Comparable<E>>
   implements ColaPrioridad<E> {
    protected E elArray[];
    protected int talla;
    protected static final int CAPACIDAD_POR_DEFECTO = 20;
@SuppressWarnings("unchecked")
public MonticuloBinario(){
  elArray = (E[]) new Comparable[CAPACIDAD_POR_DEFECTO];
  talla = 0;
}
```


Operaciones de MonticuloBinario: insertar

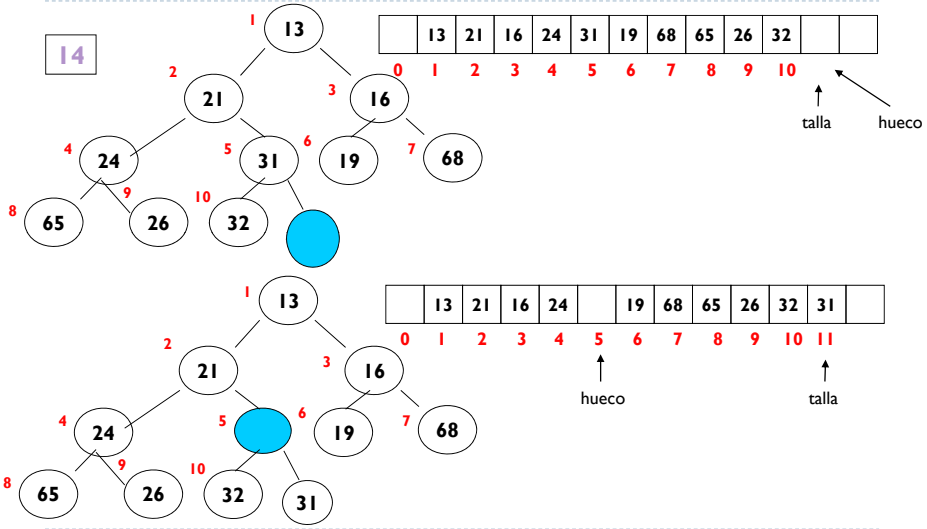
- La propiedad de inserción NO debe de violar la condición de Montículo Binario tras añadir el nuevo elemento.
  - Se debe seguir cumpliendo la propiedad **estructural** y la propiedad de **orden**.
- Para satisfacer la propiedad estructural: Se añade el elemento a la primera posición libre del vector
- Para satisfacer la propiedad de orden: Se reflota el nuevo elemento sobre sus antecesores.
  - El nuevo valor introducido asciende por el Montículo, comparando con el correspondiente padre, hasta encontrar la posición adecuada donde no se viola la propiedad de orden.

Operaciones de MonticuloBinario: insertar (II)

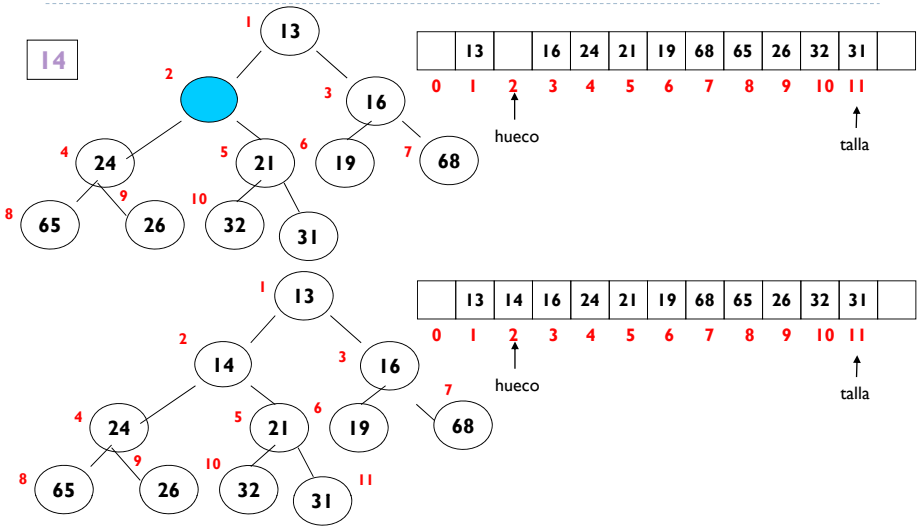
- Insertamos el dato 14 en este Montículo Binario:



Operaciones de MonticuloBinario: insertar (III)



Operaciones de MonticuloBinario: insertar (IV)



Código Java del Método insertar

```
public void insertar(E x){
  if ( talla == elArray.length-1 ) duplicarArray();
  int  hueco = ++talla;
  while ( hueco >1 && x.compareTo(elArray[hueco/2]) < 0 ){
        elArray[hueco] = elArray[hueco/2];
        hueco = hueco/2;
  }
  elArray[hueco] = x;
}
```

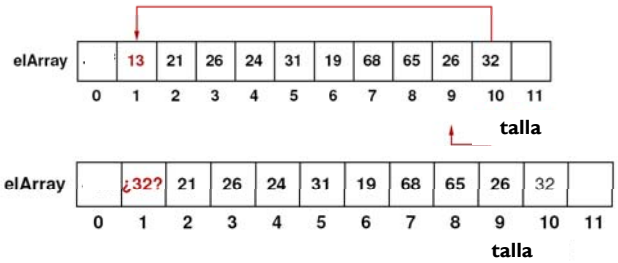
- La comprobación del bucle (hueco>1) impide el acceso a la componente 0 del vector.

Complejidad Temporal de insertar

- Talla del Problema:
  - Número de Nodos del Montículo Binario
- Instancias Significativas:
  - Para observar el mejor y el peor caso, hay que fijar la talla:
    - ¡Nunca un caso mejor es tener un Montículo Binario vacio!
  - Caso Mejor: El nuevo dato a añadir es mayor que el dato de su nuevo nodo padre. $\Omega(1)$
  - Caso Peor: El nuevo dato a añadir es menor que cualquiera de los ya existentes (el nuevo mínimo). Se realizan tantas comparaciones como datos hay en el camino hasta la raíz.
    - El coste, en el peor de los casos, es proporcional a la altura del árbol.
    - Como la propiedad estructural garantiza que es un AB completo: $O(\log_2(N))$

Eliminar el Mínimo de un Montículo Binario

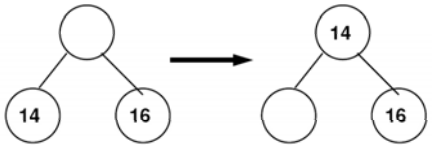
- El mínimo elemento está en la posición 1 del vector.
- Para no violar la propiedad estructural del Montículo, colocamos el dato situado en la última posición ocupada del array en la posicion 1.
  - Se debe actualizar *talla* para reflejar el nuevo cambio.



- Se viola la propiedad de orden debido al valor de la raíz.

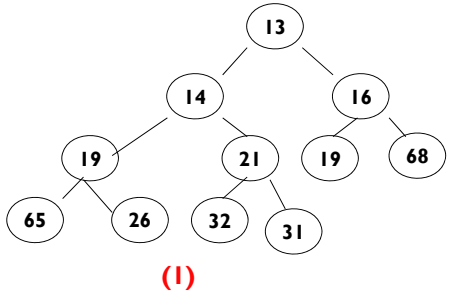
Eliminar el Mínimo de un Montículo Binario (II)

- Para restituir la propiedad de orden al Montículo hay que hundir la posición del hueco hasta encontrar la posición correcta para el dato que se quiere colocar.
  - Analogía con el proceso de *reflotar* visto para insertar, pero ahora el hueco se mueve desde la raíz hacía las hojas: Se debe *hundir*.
  - Hundir es más complejo que reflotar: Hay que considerar los dos posibles hijos.
- **Siempre** se hunde en la dirección del hijo menor:



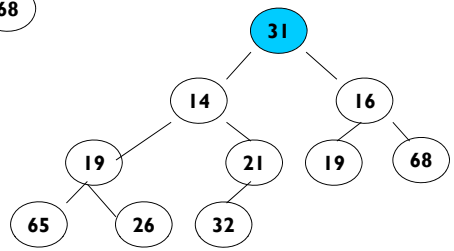
Traza de Eliminar el Mínimo

- Se desea eliminar el mínimo del siguiente Montículo: [13,14,16,19,21,19,68,65,26,32,31]



(1)

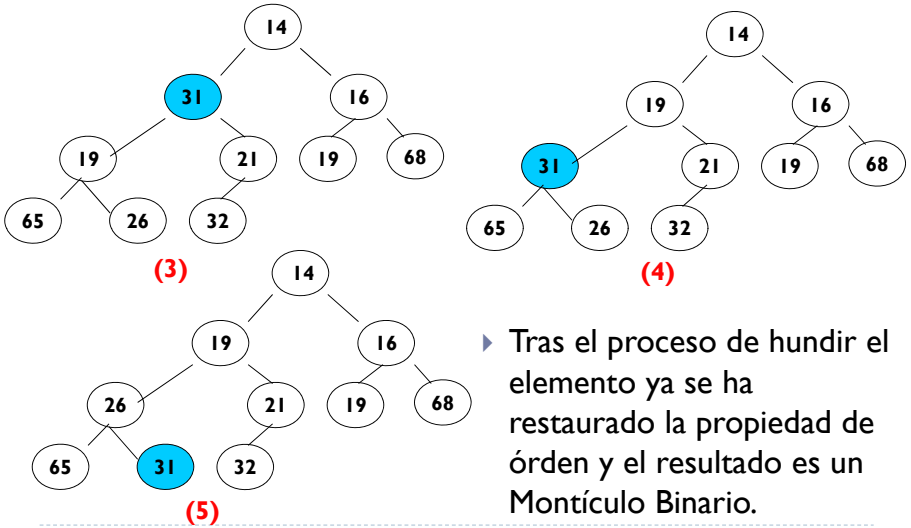
- Representación del Montículo Binario Actual



(2)

- Situamos el elemento 31 en la posición 1 (raíz del árbol), eliminando así el elemento mínimo y manteniendo la propiedad estructural.

Traza de Eliminar el Mínimo (II)



- Tras el proceso de hundir el elemento ya se ha restaurado la propiedad de órden y el resultado es un Montículo Binario.

Código Java del Método eliminarMin

```
public E eliminarMin() {
   E elMinimo = recuperarMin();
   elArray[1] = elArray[talla--];
   hundir(1);
   return elMinimo;
}
```

- Coloca el elemento situado en talla en la posición 1, cumpliendo así la propiedad estructural.
- Hunde el dato situado en la posición 1 hasta su posición adecuada dentro del Montículo Binario para que se cumpla la posición de orden.

Código Java del Método hundir

```
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++;
    if (elArray[hijo].compareTo(aux) < 0 ){
          elArray[hueco] = elArray[hijo];
          hueco = hijo;
          hijo = hueco*2;
       } else esHeap = true;
    }
    elArray[hueco] = aux;
}
```


Resto de Métodos de MonticuloBinario

```
//El mínimo elemento está en la posición 1
public E recuperarMin() {
   return elArray[1];
}
public boolean esVacia(){  return (talla == 0);}
public String toString(){
    String res = "";
    if (talla > 0){
      for (int i = 1 ; i<= talla; i++) res += " " + elArray[i];
    }
    return res;
 }} /* Fin de la clase MonticuloBinario */
```

Inicialización de un Montículo Binario

- Es posible inicializar un Montículo con N datos mediante:
  1. N inserciones.
  2. Colocar los datos en el array y restaurar la propiedad de orden.
- ¿Cuál de las dos estrategias es más eficiente?
  1. Para el primer caso, ejemplificamos con la creación de un Montículo Binario a partir de un fichero de cadenas de caracteres.
  2. Para el segundo caso, ejemplificamos con un método para iniciar el montículo a partir de un vector de elementos compatibles con el tipo de datos del Montículo Binario.

Inicialización Mediante N Inserciones

```
public static MonticuloBinario<String> iniciarMonticulo(String nomfDatos) throws
    FileNotFoundException{
    MonticuloBinario<String> mb = new MonticuloBinario<String>();
    Scanner sc = new Scanner(new File(nomfDatos));
    while (sc.hasNext()){ String dato = sc.next(); mb.insertar(dato); }
    return mb;
 }
 public static void main(String args[]){
  MonticuloBinario<String> mb = null;
   try{
    mb = FactoryMB.iniciarMonticulo("datos.txt");  System.out.println("MB: " + mb);
   }catch(FileNotFoundException ex){
      System.err.println("Fichero inexistente: " + ex);
} }
```


Inicialización Mediante Arreglado de Montículo (II)

2. Cada dato leído se coloca en la primera posición libre del array. Al finalizar, se modifica la estructura para que establezca la propiedad de orden.

```
public void iniciarMonticulo(E v[]){ //En la clase MonticuloBinario
     for (int i = 0; i < v.length; i++){ elArray[i+1] = v[i]; }
      talla = v.length;  arreglarMonticulo();
 }
public static void main(String args[]){ //En otra clase diferente
     String [] v = {"Casa", "Perro", "Adios","Cocina","Zapato"};
     MonticuloBinario<String> mb = new MonticuloBinario<String>();
     mb.iniciarMonticulo(v);  System.out.println("MB: " + mb);
  }
```

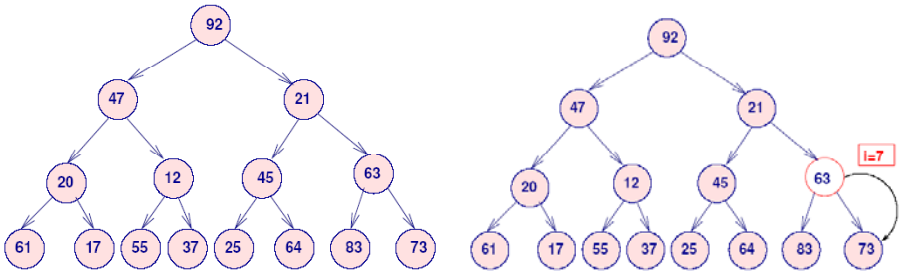

El método arreglarMonticulo

- El método arreglarMonticulo debe restablecer la propiedad de orden sobre un Árbol Binario Completo ya almacenado en un array.
- Estrategia utilizada:
  - Es suficiente con hundir sus nodos en orden inverso al recorrido por niveles realizado para crearlo.
  - No hace falta hundir las hojas ya que éstos son Montículos.
  - Comenzamos por hundir el nodo de mayor índice que NO es una hoja, el cual siempre estará en talla / 2.

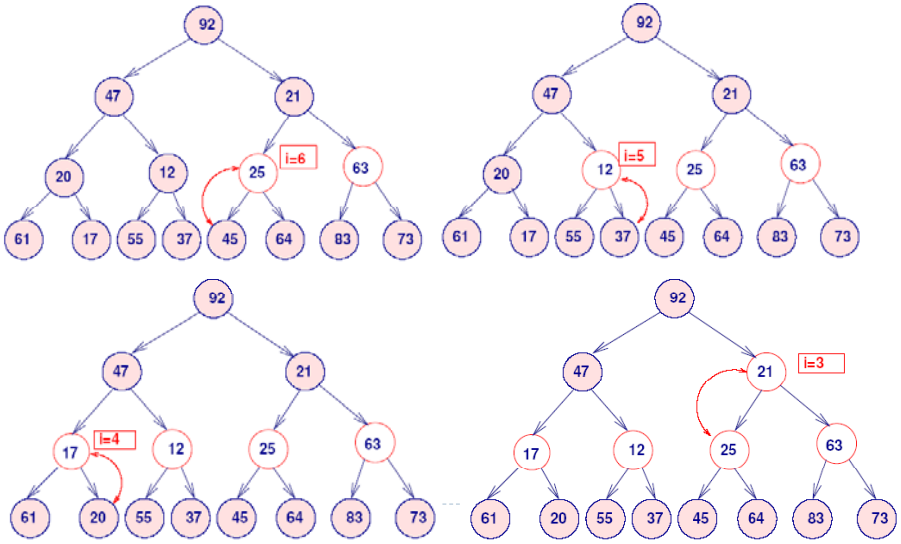
```
private void arreglarMonticulo () {
  for (int i = talla/2; i > 0; i--) hundir(i);
}
```

Traza de arreglarMontículo

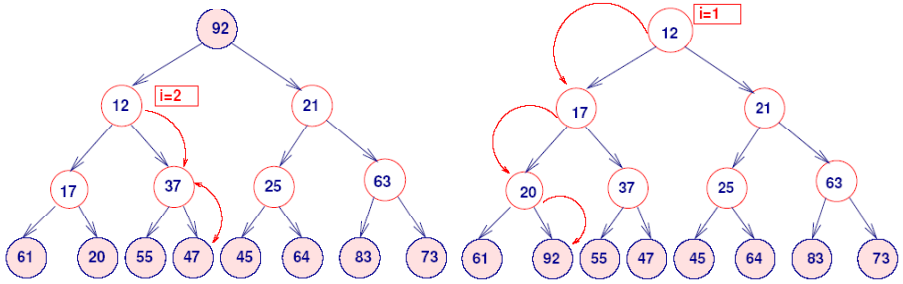
- Traza de arreglarMonticulo sobre los datos: [92,47,21,20,12,45,63,61,17,55,37,25,64,83,73]



Traza de arreglarMontículo (II)



Traza de arreglarMontículo (III)



- El resultado es un Montículo Binario puesto que cumple tanto la propiedad Estructural como la de Orden.

Complejidad Temporal de arreglarMonticulo

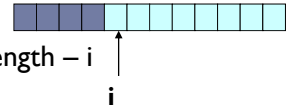
- La complejidad temporal del método arreglarMonticulo depende de la suma de las alturas de todos los nodos.
- Teorema:
  - Dado un AB Lleno de altura H que contiene $N = 2^{H+1}-1$ nodos, se cumple que la suma de las alturas de sus nodos es $N - H - 1$.
- Como un AB Completo tiene entre 2^H y $N = 2^{H+1}-1$ nodos, la suma de las alturas de sus nodos está acotada por **O(N)**
  - Complejidad temporal de arreglarMontículo: **O(N)**
- En cambio, si creamos el Montículo realizando las N inserciones incurrimos en un coste, en el peor de los casos: **$O(N*\log_2(N))$**
- Conclusión: Es más eficiente reconstruir la propiedad de orden a un Montículo que construirlo mediante inserciones.

Objetivos de Heap Sort

- Heap Sort es un método de Ordenación Rápida basada en el principio de Selección.
  - Utiliza un Montículo Binario para estructurar el subarray sobre el que se realiza la Selección.
- Cambio de representación:
  - Heap Sort se implementa sobre un array cuyos datos se almacenan en [0, array.length – 1].
  - Se utiliza un Montículo Binario Maximal (en lugar del Minimal).
- QuickSort, en el peor de los casos tenía un coste **O(N^2)** ya que NO garantizaba que el tamaño de los subproblemas fuera el mismo.
- MergeSort requería el uso de un vector adicional.

Estrategia de Ordenación por Selección

- Ordenar ascendentemente el vector v[0 … v.length – 1]:
  - Se recorre ascendentemente el array v, seleccionando en cada iteración i (0 <= i < v.length) el **menor** elemento de entre los que forman el subarray v[i … v.length – 1].
  - Se coloca el elemento elegido en su posición correcta: *i*.
- De este modo, en cada paso de iteración se cumple que:
  - El subarray v[0 … i-1] ya está ordenado.
  - El subarray v[i … v.length-1] contiene las v.length – i componentes pendientes de ordenar.
  - El proceso de ordenación concluye cuando el subarray ordenado se extiende por todo el array: i == v.length



El método seleccionDirecta

```
private static <T extends Comparable<T>> void seleccionDirecta(T
  v[]){
  for ( int i=0; i < v.length-1; i++ ) {
     int posMinimo = recuperarMinimo(v, i);
     intercambiar(v, i, posMinimo);
  }}
```

¿Cuál es el coste de ordenación por selección directa?

- El método recuperarMinimo devuelve el índice del menor elemento situado entre las componentes [i … v.length – 1] del vector v.
- El método intercambiar intercambia v[i] con v[posMinimo], lo que afecta al contenido del vector v.
- La ordenación por selección directa tiene una complejidad temporal cuadrática con el número de elementos del vector: Θ(N2)

Estrategia Alternativa para Selección Directa

- Ordenar ascendentemente el vector v[0 … v.length – 1]:
  - Se recorre descendentemente el array v, seleccionando en cada iteración i (0 < i <= v.length) el elemento **mayor** de entre los que forman el subarray v[0 … i].
  - Se coloca el elemento elegido en su posición correcta: i.

```
private static <T extends Comparable<T>> void seleccionDirectaMax(T v[]){
    for ( int i = v.length-1; i >0; i-- ) {
        int posMaximo = buscarMaximo(v, i);
        intercambiar(v, i, posMaximo);
    }}
```

- El método *buscarMaximo* devuelve el índice del menor elemento situado entre las componentes [0 … i] del vector v.
- La Complejidad Temporal sigue siendo: Θ(N^2)

Reduciendo el Coste de Selección Directa

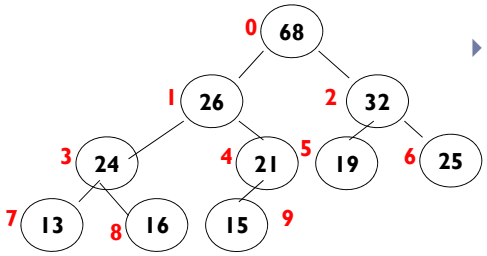
- El coste actual de Selección Directa viene dado por tener que realizar N – 1 veces el recorrido para buscar el máximo elemento, que tiene un coste $\Theta(N)$

- Si la zona del vector que queda pendiente de ordenar se organiza como un Montículo Binario Maximal:
  - El método buscarMáximo pasa a ser *eliminarMax,* que tiene un coste $\Theta(\log_2(N))$.
- La complejidad temporal del *nuevo selección directa* pasaría a ser: $\Theta(N*\log_2(N))$.
- Esta estrategia corresponde al método de Ordenación Rápida según un Montículo: ***Heap Sort***

Estrategia de Ordenación Heap Sort

- Se construye un Montículo Binario Maximal con los elementos del array y, seguidamente, se recorre el vector en sentido descendente, desde el final hasta la posición 0.
  - Para cada posición i:
    1. Se intercambia el valor máximo (por definición de Montículo en posición 0), con el que se encuentra en la posición i.
    2. El valor en v[0] *puede* violar la propiedad de orden. Se debe de hundir ese valor sobre el Montículo que ahora ocupará el subarray v[0 … i-1]
    3. Tratamos la posición anterior, haciendo decrecer en 1 el número de elementos que quedan por ordenar.
- Un MB Maximal es un AB completo donde, para cada nodo X con padre P se cumple que el dato en P es **mayor o igual** que el dato en X. ∀i : 0 < i <= talla : v[i/2] >= v[i]

Montículo Binario Maximal



- Montículo Binario Maximal con N datos a partir de la posición 0 del array.

68	26	32	24	21	19	25	13	16	15	
0	1	2	3	4	5	6	7	8	9	10

talla = 9

- Si la i-ésima componente del array representa el i-ésimo Nodo de su Recorrido por Niveles:
  - **Hijo Izquierdo** en **2*i+1**, si 2*i+1 ≤ N
  - **Hijo Derecho** en **2*i+2**, si 2*i+2 ≤ N
  - **Padre** en **(i-1)/2**, si i≠0

Método arreglarMontículo para un Montículo Binario Maximal

- Trabajamos de manera análoga al método arreglarMonticulo ya definido para un Montículo Binario Minimal.
  - Diseño de un método, privado en la clase Ordenación, análogo a hundir (hundirMax) que trabaje sobre un Montículo Binario Maximal.

private static void **hundirMax**(E v[], int hueco, int numElem);

- numElem indica la posición del último elemento que forma parte del Montículo Binario Maximal.

Método arreglarMontículo para un Montículo Binario Maximal

```
private static <T extends Comparable<T>> void hundirMax(T v[], int hueco,
    int numElem) {
  T aux = v[hueco];
  int hijo = hueco * 2 +1;
  boolean esHeap = false;
  while ( hijo <= numElem && !esHeap ) {
     if ( hijo!=numElem && v[hijo+1].compareTo(v[hijo])>0 ) hijo++;
     if ( v[hijo].compareTo(aux) > 0 ) {
        v[hueco] = v[hijo];
        hueco = hijo;
        hijo = hueco * 2 +1;
      }
     else esHeap = true;
   }
   v[hueco] = aux;}
```


El método Java HeapSort (I)

- A partir de un vector v desordenado que se quiere ordenar ascendentemente, el método HeapSort:
  1. Construye un Montículo Binario Maximal a partir del vector (como arreglarMonticulo).
  2. Bucle principal de ordenación
     - Intercambio del máximo (en la posición 0) con el último elemento para situarlo en su posición ordenada.
     - Hundir el valor de la posición 0, en el MB que resta por ordenar, para restaurar la propiedad de orden (y que el nuevo máximo suba a la raíz).

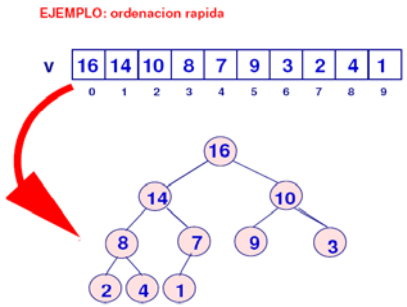
El método Java HeapSort (II)

```
public static <T extends Comparable<T>> void heapSort (T v[]) {
  //Para construir un Montículo Binario a partir de un array (como
  //en arreglarMonticulo)
  for ( int i = (v.length/2)-1; i >= 0; i-- ) hundirMax (v, i, v.length-1);

  for (int i = v.length-1; i > 0; i--) {
    intercambiar(v, 0, i);
    hundirMax(v, 0, i-1);
  }
}
```

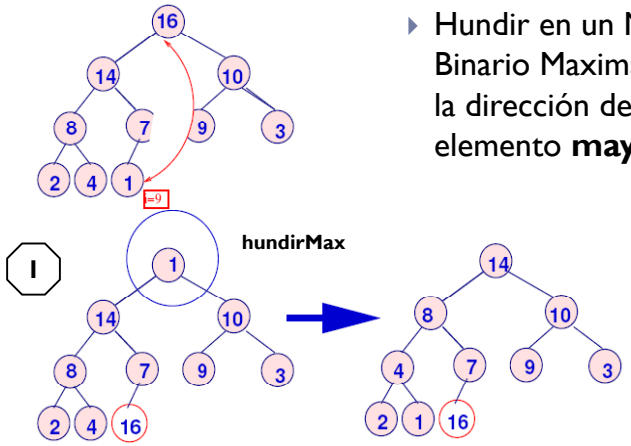

Ejemplo de Ordenación con HeapSort

- Contenido del array y su representación en forma de Árbol después de construir el Montículo Binario



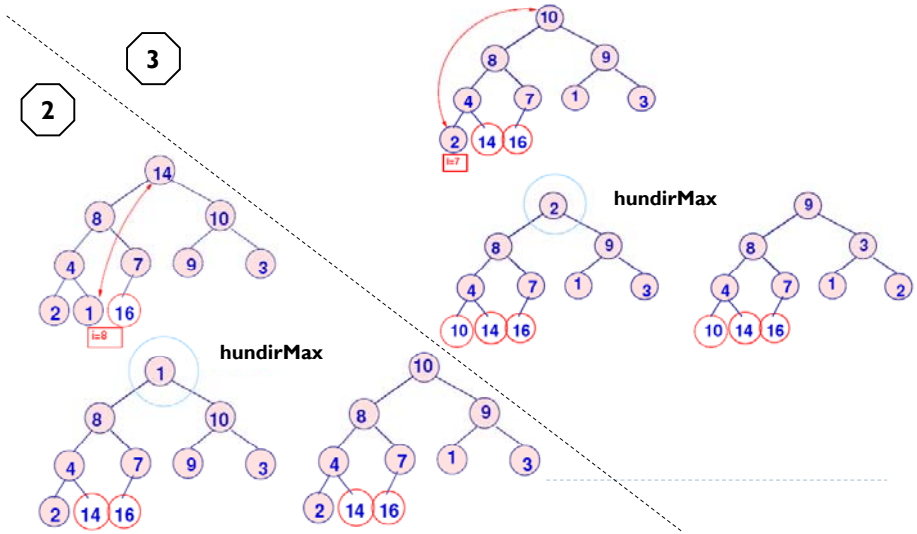
- A continuación, se realiza el bucle de HeapSort desde i = 9 hasta i = 1

Ejemplo de Ordenación con HeapSort (II)

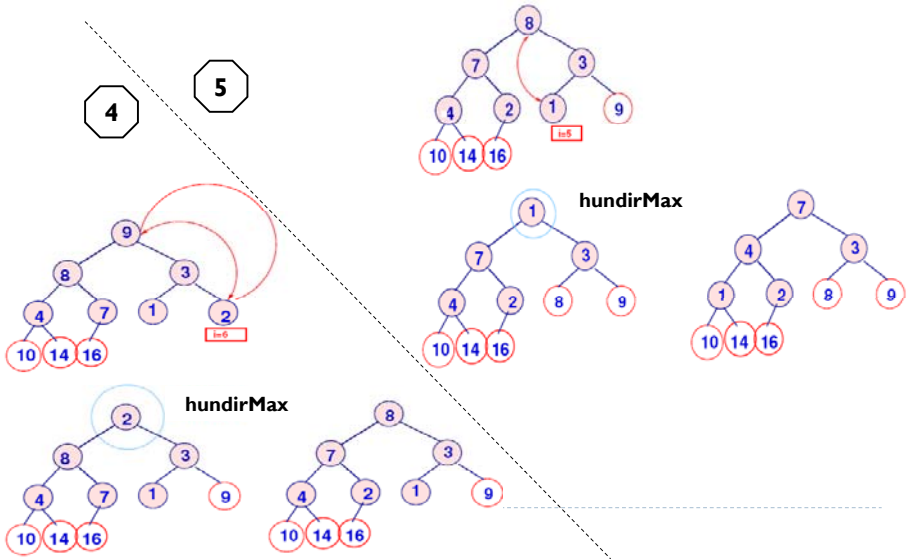


- Hundir en un Montículo Binario Maximal se realiza en la dirección del hijo con el elemento **mayor**.

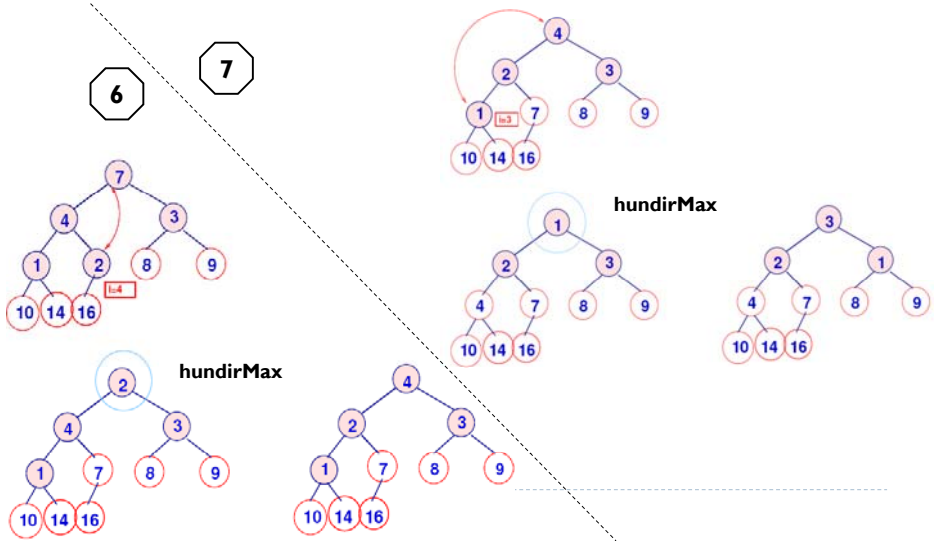
Ejemplo de Ordenación con HeapSort (III)



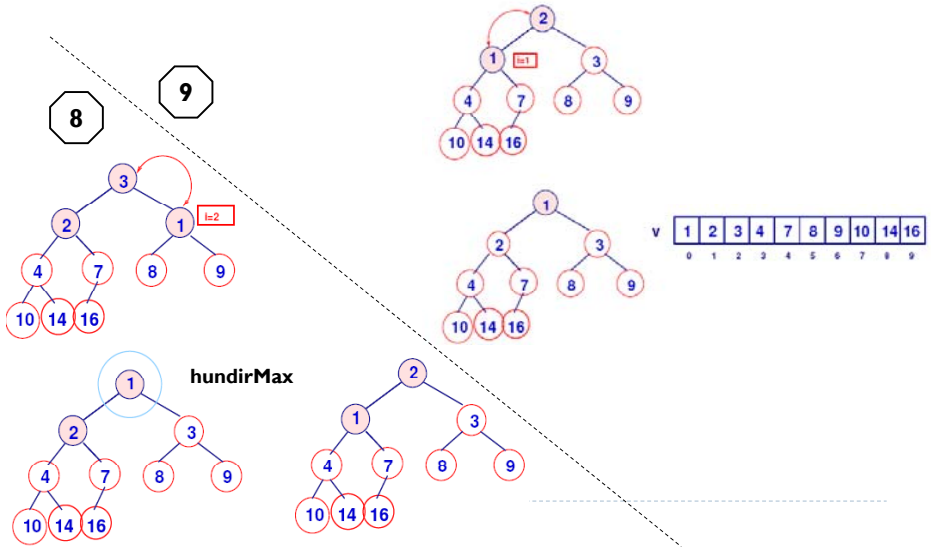
Ejemplo de Ordenación con HeapSort (IV)



Ejemplo de Ordenación con HeapSort (V)



Ejemplo de Ordenación con HeapSort (VI)



Complejidad Temporal de HeapSort

- La Complejidad Temporal de Heap Sort viene dada por el número de veces que se llama a *hundirMax*.
  - *hundirMax* tiene un coste logarítmico con el número de nodos del árbol sobre el que se hunde.
  - El Montículo sobre el que se hunde siempre tiene tamaño i.
  - Coste de cada una de las llamadas a hundirMax: $\Theta(\log_2(i))$
- Sea N el número de datos a ordenar (longitud del array).
- La función de complejidad temporal para heapSort es:

$$\sum_{i=0}^{N-1} \log_2(i) \in \Theta(N * \log_2(N))$$