Tema 5

Cola de Prioridad y Montículo Binario. Heapsort

Objetivos

1. Presentar una implementación eficiente del modelo Cola de Prioridad
2. Representación contigua (ímplicita) de los datos mediante un array: Montículo Binario (Binary Heap)
3. Diseño del método genérico de ordenación HeapSort sobre la base de esta implementación

Tema 5

Cola de Prioridad y Montículo Binario. Heapsort

Contenidos

1. El modelo Cola de Prioridad: definición, coste estimado y ejemplos de uso
2. La Implementación de Cola de Prioridad: definición, propiedades y operaciones del Montículo Binario
3. Esquema básico de la clase Java `MonticuloBinario`
4. *Heap Sort*

Tema 5

Cola de Prioridad y Montículo Binario. Heapsort

Bibliografía

- Weiss, M.A. ”Estructuras de datos en Java”, Adisson-Wesley, 2000 (apartados del 1 al 5 del capítulo 20)
- Sahni, S. Data Structures, Algorithms, and Applications in Java. McGraw-Hill, 2000. Capítulo 13, secciones 1 a 4.
- Michael T. Goodrich and Roberto Tamassia. Data Structures and Algorithms in Java (4th edition). John Wiley & Sons, Inc., 2010. Capítulo 8, secciones 1 a 3.

1. Cola de Prioridad

Una Cola de Prioridad es una colección homogénea de datos en la cual las operaciones características son aquellas que permiten acceder al dato de máxima prioridad.

```
package librerias.estructurasDeDatos.modelos;
public interface ColaPrioridad<E extends Comparable <E>> {
    // Añade e a la cola de prioridad
    void insertar(E e);
    // SII !esVacia() devuelve el dato de máxima prioridad
    E recuperarMin();
    // SII !esVacia() devuelve y elimina el dato máx prioridad
    E eliminarMin();
    // Devuelve true si la cola está vacía
    boolean esVacia();
}
```

1. Es indispensable que cualquier dato de la colección, *d1*, sea `Comparable` con otro, *d2*, en base a su prioridad:

   d1* < *d2* SII la prioridad de *d1* es mayor estricta que la de *d2

2. El dato de **mayor prioridad** es el máximo (o mínimo).
3. A igualdad de prioridades, acceso FIFO

1. Cola de Prioridad - *Ejemplos de uso*

Existen múltiples aplicaciones que usan una Cola de Prioridad:

- Simulación dirigida por eventos discretos, como la de las urgencias de un hospital
- Implementación de la lista de espera de una compañía aérea
- Gestión de procesos de un SO
- Problemas de optimización basados en algoritmos voraces, como la obtención de Caminos Mínimos y Árbol de Recubrimiento Mínimo de un Grafo

2. Montículo (*Heap*) Binario - *Definición*

Un Montículo Binario es un Árbol Binario (AB) tal que

1. **PROPIEDAD ESTRUCTURAL:** es un AB **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 un array

2. **PROPIEDAD DE ORDEN:** es un AB en cuya Raíz se sitúa, siempre el dato de máxima prioridad

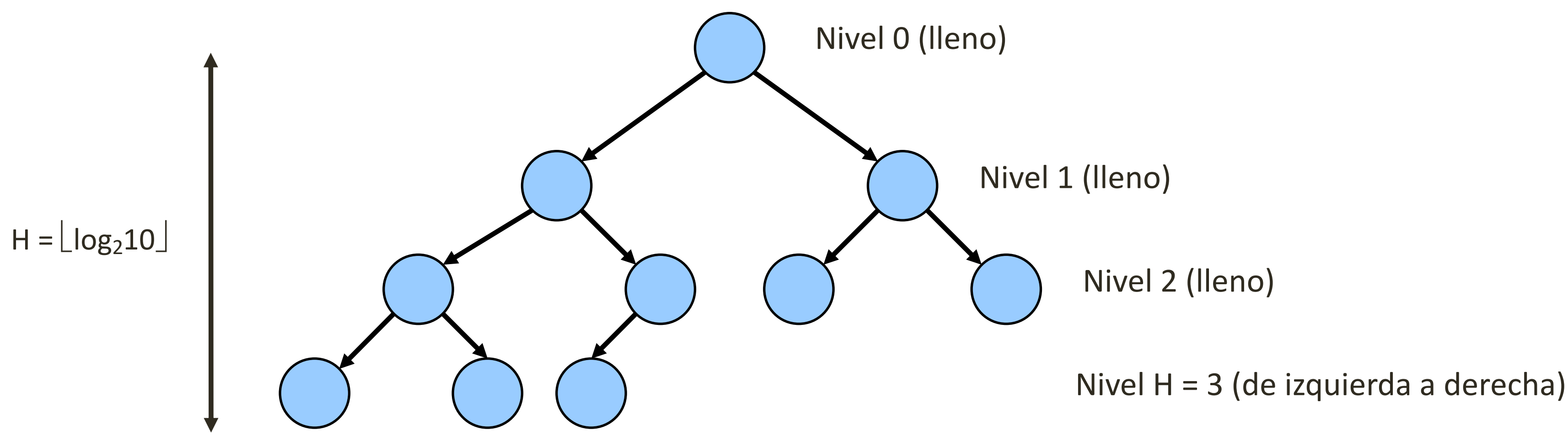
- En un minHeap, el dato de un nodo es siempre menor o igual que el de sus hijos
- En un maxHeap, es siempre mayor o igual

Todo subárbol de un Heap es también un Heap.

2. Montículo (*Heap*) Binario – *Representación*

Un AB Completo es aquel que tiene todos sus niveles llenos a excepción, quizás, del último, que en tal caso debe de tener situados todos sus nodos tan a la izquierda como sea posible

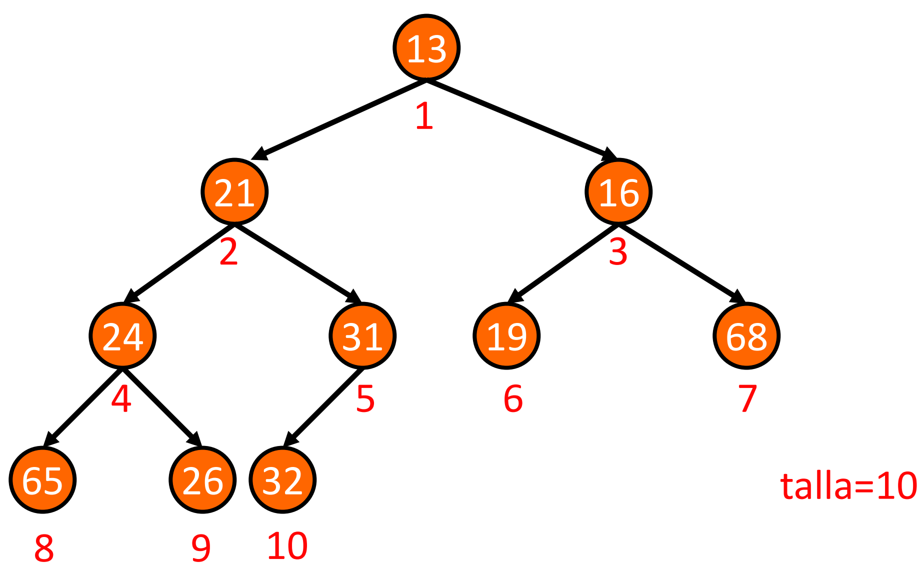
La altura de un Heap de N elementos es $\lfloor \log_2 N \rfloor$



2. Montículo (*Heap*) Binario-*Representación (cont.)*

Un árbol binario completo admite una representación implícita sobre un array

- Se almacena los nodos en un array según su recorrido por niveles
- **La raíz se guarda en la posición 1 del array**; el Hijo Izquierdo de la Raíz en la posición 2; el el Hijo Derecho de la Raíz en la posición 3; etc. (La posición 0 se deja libre, lo que facilita el cálculo de los hijos de un nodo)

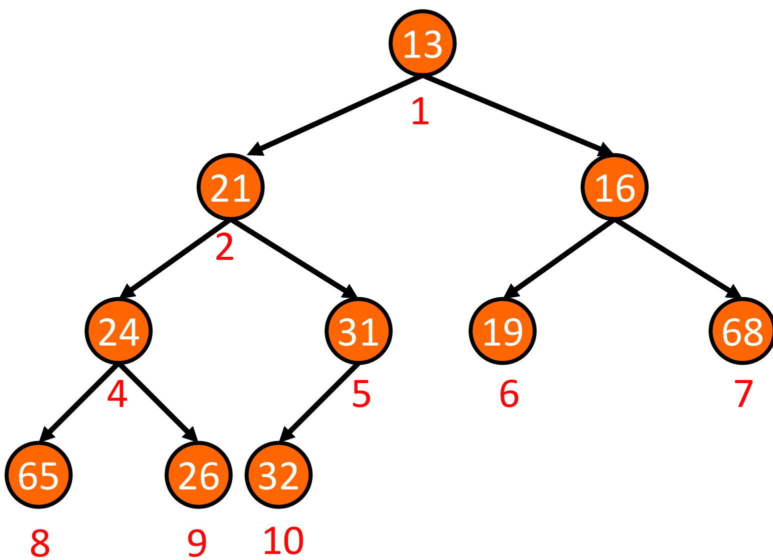


	1	2	3	4	5	6	7	8	9	10	
	13	21	16	24	31	19	68	65	26	32	....

* E[] elArray
* int talla

2. Montículo (*Heap*) Binario-*Representación (cont.)*

- elArray[1] representa a su Nodo Raíz
- si elArray[i] representa a su i-ésimo Nodo (Por Niveles)
  - Su Hijo Izquierdo es elArray[2i], si $2i \leq$ talla
  - Su Hijo Derecho es elArray[2i+1], si $2i + 1 \leq$ talla
  - Su Padre es elArray[i/2], excepto para i = 1



talla=10

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

* E[] elArray
* int talla

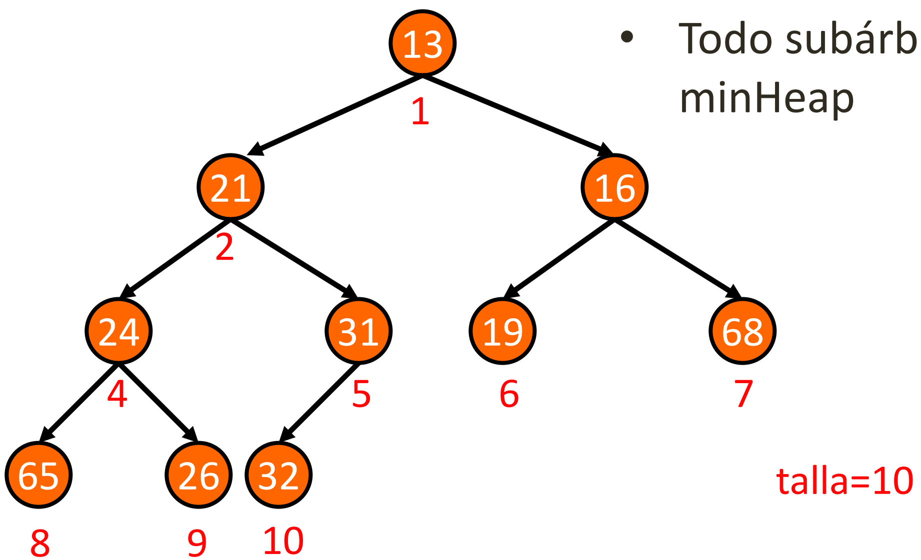
2. Montículo (*Heap*) Binario-*Representación minHeap (cont.)*

- Propiedad de orden en un minHeap:

  elArray[padre(i)] <= elArray[i],   2<=i<=talla

Propiedades:

- Todo camino desde la raíz a una hoja es una secuencia ordenada (13, 21, 31, 32; 13, 16, 68)
- La raíz es el nodo de valor mínimo
- Todo subárbol de un minHeap es también un minHeap



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

* E[] elArray
* int talla

2. Montículo (*Heap*) Binario-*Ejercicio*

Suponiendo que no hay elementos repetidos y que estamos hablando de un minHeap:

a) ¿Dónde estará el mínimo?
b) ¿Dónde estará el máximo?
c) ¿Cualquier elemento de una hoja será mayor que los elementos de los nodos internos?
d) ¿Un minHeap es un vector ordenado de forma creciente?
e) ¿Es un minHeap la siguiente secuencia: {1, 5, 12, 7, 17, 14, 13, 28, 6, 18}

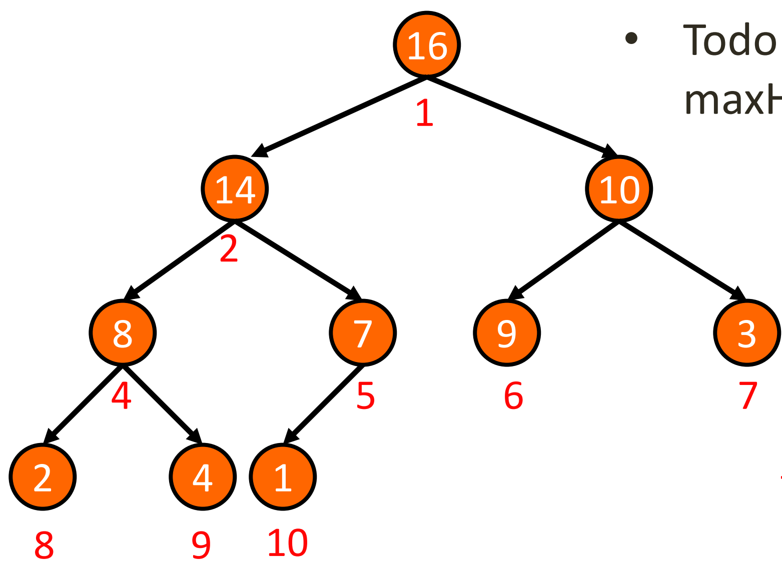
2. Montículo (*Heap*) Binario-*Representación maxHeap (cont.)*

- Propiedad de orden en un maxHeap:

  elArray[padre(i)] >= elArray[i],   2<=i<=talla

Propiedades:

- Todo camino desde la raíz a una hoja es una secuencia ordenada (16, 14, 8, 2; 16, 10, 9)
- La raíz es el nodo de valor máximo
- Todo subárbol de un maxHeap es también un maxHeap



talla=10

	1	2	3	4	5	6	7	8	9	10	
	16	14	10	8	7	9	3	2	4	1	....

* E[] elArray
* int talla

2. Montículo (*Heap*) Binario-*Ejercicio*

Suponiendo que no hay elementos repetidos y que estamos hablando de un maxHeap:

a) ¿Dónde estará el máximo?
b) ¿Dónde estará el mínimo?
c) ¿Cualquier elemento de una hoja será menor que los elementos de los nodos internos?
d) ¿Un maxHeap es un vector ordenado de forma decreciente?
e) ¿Es un maxHeap la siguiente secuencia: {23, 17, 14, 6, 13, 10, 1, 5, 7, 12}

2. Montículo (*Heap*) Binario - *Operaciones, implementación y coste estimado*

- **Operaciones *kernel***: las operaciones de una Cola de Prioridad
  - Insertar un nuevo elemento *e* en un Heap (*add*): insertar(e)
  - Comprobar si un Heap está vacío (*isEmpty*): esVaci**a**()
  - Devolver, **SIN** eliminar, el mínimo de un Heap (*peek*): recuperarMin()
  - Devolver **Y** eliminar, el mínimo de un Heap (*poll*): eliminarMin()

- **Especificación y Esquema algorítmico de las operaciones modificadoras:**

  PreCondición: el árbol es un AB Completo y cumple propiedad de orden del Heap

  1. Realizar la operación sobre un AB Completo **Y** comprobar que el Árbol resultante es también un AB Completo.
  2. Comprobar si el Árbol resultante cumple la propiedad de orden del Heap; si NO lo hace, restaurarla mediante las operaciones pertinentes.

  PostCondición: el árbol es un AB Completo y cumple propiedad de orden del Heap

- **Coste promedio estimado:** para una gran mayoría de las operaciones, al menos aquellas que requieren el acceso a uno de sus datos, varía entre O(1) y O(log N), **por lo que en cualquier caso es sublineal**, siendo N el número de elementos del heap.

3. La clase Java MonticuloBinario - *Esquema: atributos y métodos*

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

```
package librerias.estructurasDeDatos.jerarquicos;
import  librerias.estructurasDeDatos.modelos.*;
public class MonticuloBinario<E extends Comparable<E>>
    implements ColaPrioridad<E> {
    protected E[] elArray; protected static final int CAPACIDAD_INICIAL= …;
    protected int talla;
    public MonticuloBinario() { … }
    public boolean esVacia() ) { … }
    public void insertar(E e) { … }
    protected void duplicarArray ) { … }
    public E eliminarMin() { … }
    public E recuperarMin() { … }
    public String toString() { … }
    …
}
```

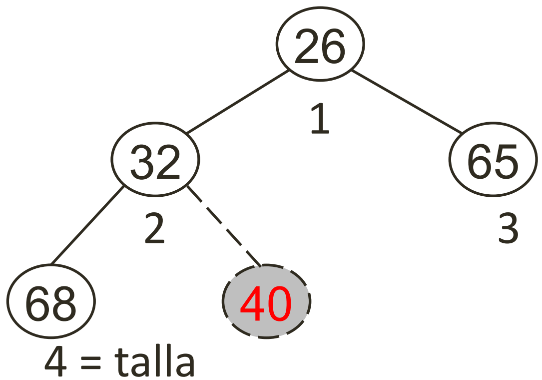
3. La clase Java MonticuloBinario – *Métodos constructor vacío, esVacia y recuperarMin*

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

    /** crea un Heap vacío */
    @SuppressWarnings("unchecked")
    public MonticuloBinario() {
        elArray = (E[]) new Comparable[CAPACIDAD_INICIAL];
        talla = 0;
    }

    /** comprueba si un Heap es vacío en Θ(1) */
    public boolean esVacia() { return talla == 0; }

    /** devuelve el mínimo de un Heap en Θ(1) */
    public E recuperarMin() { return elArray[1]; }
    …
}
```

3. La clase Java `MonticuloBinario`

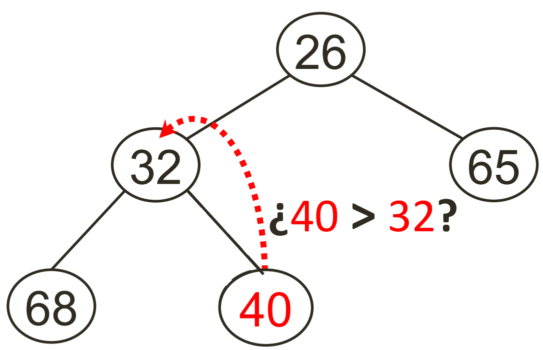
- *Método insertar(e): algoritmo en 2 pasos, ejemplo con e = 40*

Paso 1: se inserta el nuevo elemento en la primera posición disponible del vector: elArray[talla+1]



Si *e* se inserta Por Niveles, el AB sigue siendo Completo → talla **+ 1**

Paso 2: se reflota sobre sus antecesores hasta que no viole la propiedad de orden



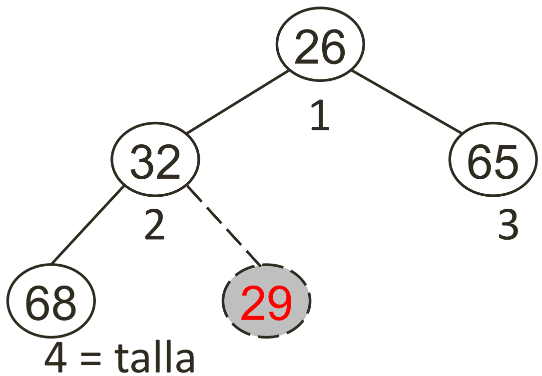
Como 40 **>** 32, si *e* se inserta en *talla+1* el AB sigue siendo Heap → posIns = talla **+ 1**

elArray[posIns] = e;

3. La clase Java `MonticuloBinario` –

Método insertar(e): algoritmo en 2 pasos, ejemplo con e = 29

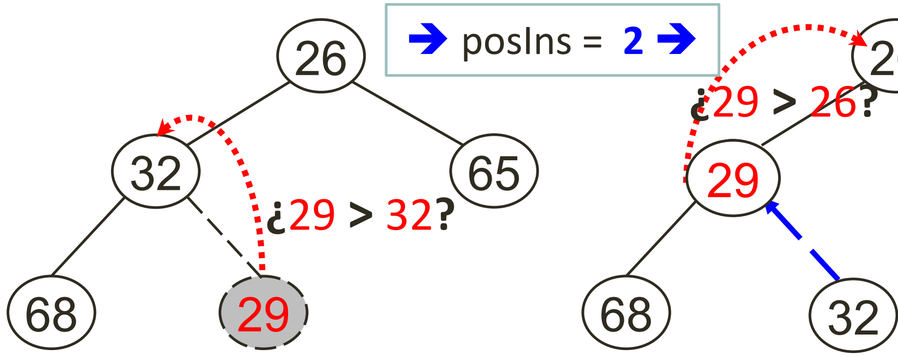
Paso 1: se inserta el nuevo elemento en la primera posición disponible del vector: elArray[talla+1]



Si *e* se inserta Por Niveles, el AB sigue siendo Completo ➔ posIns = talla **+ 1**

Paso 2: se reflota sobre sus antecesores hasta que no viole la propiedad de orden

➔ posIns = **2** ➔



posIns = **2**

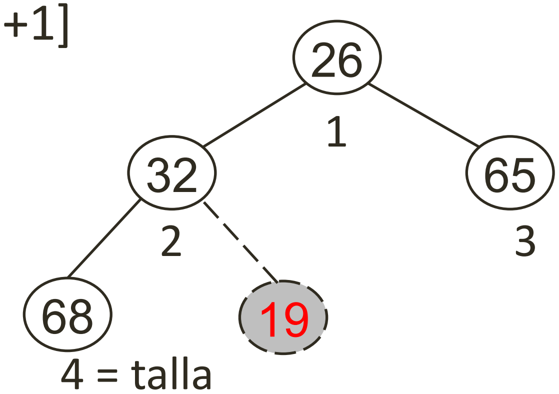
REFLOTADO *e* hasta posIns = **2**

elArray[posIns] = e;

3. La clase Java `MonticuloBinario` –

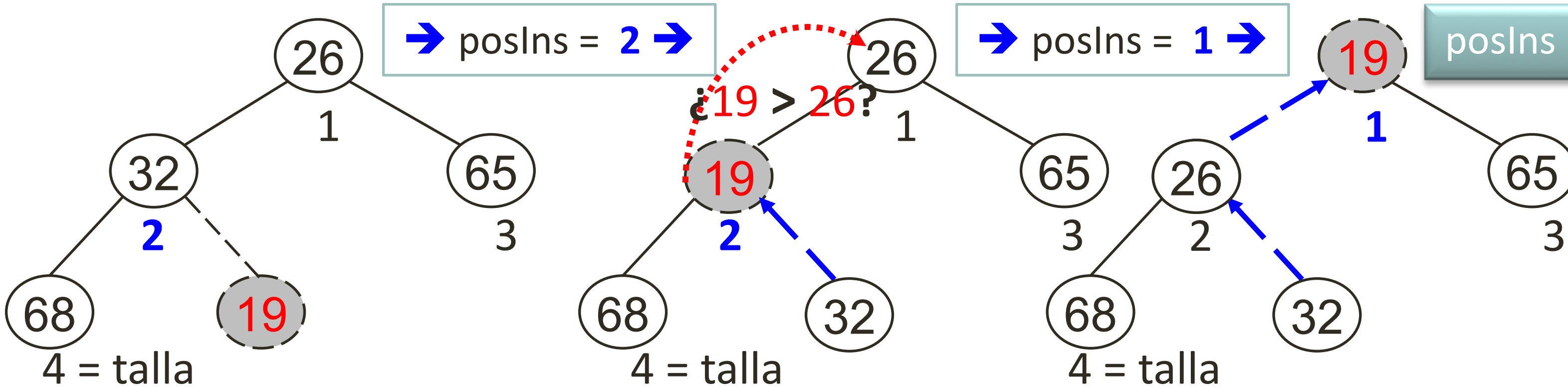
Método insertar(e): algoritmo en 2 pasos, ejemplo con e = 19

Paso 1: se inserta el nuevo elemento en la primera posición disponible del vector: elArray[talla+1]



Si *e* se inserta Por Niveles, el AB sigue siendo Completo ➔ posIns = talla **+ 1**

Paso 2: se reflota sobre sus antecesores hasta que no viole la propiedad de orden



elArray[posIns] = e;

3. La clase Java MonticuloBinario -

Método insertar(e): código

```
/** inserta e en un Heap */
public void insertar(e) {
    if (talla == elArray.length - 1) duplicarArray(); //espacio para nuevo dato?
    // posIns es la posición donde insertaremos e
    int posIns = ++talla;
    // reflotamos hasta que no viole la propiedad de heap
    while (posIns > 1  && e.compareTo(elArray[posIns/2]) < 0) {

            elArray[posIns] = elArray[posIns / 2];
            posIns = posIns / 2;
    }
    // ya tenemos la posición del nuevo dato: insertamos

    elArray[posIns] = e;

}
```

3. La clase Java `MonticuloBinario` - *Método insertar(e): análisis de su coste*

Talla del problema: es el número de elementos del heap N

- Caso Mejor: el elemento e a insertar es mayor que su padre (requiere una única comparación): *e.compareTo(elArray[++talla/2]) >= 0)*)

$$T^{m}_{insertar}(N) \in \Omega(1)$$

- Caso Peor: e es el nuevo mínimo, es necesario *reflotar posIns* hasta la Raíz, i.e. $\lfloor \log_2 N \rfloor$ veces

$$T^{p}_{insertar}(N) \in O(\log_2 N)$$

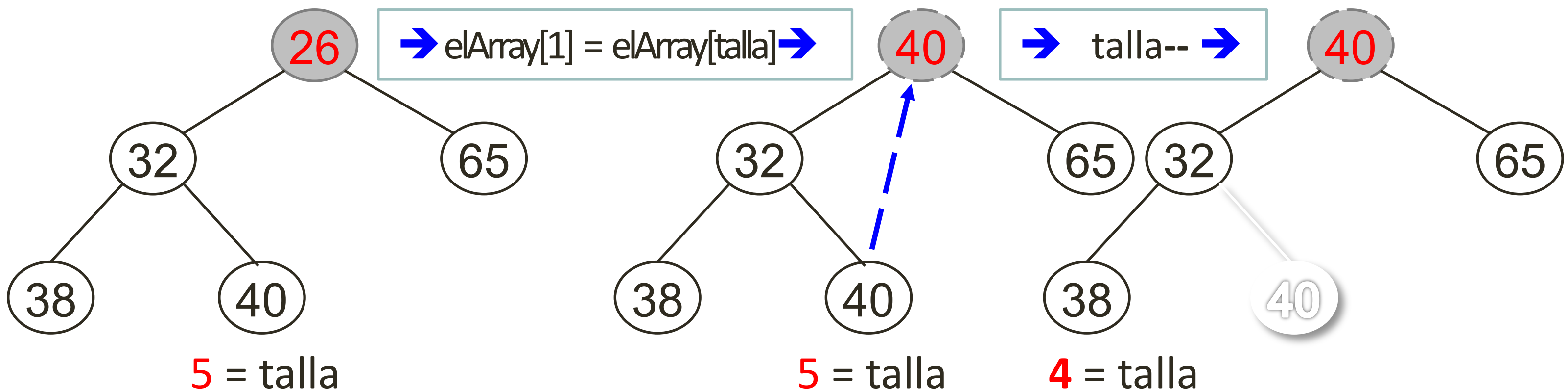
Caso promedio: $T^{\mu}_{insertar}(N) \in \Theta(1)$!!!

Se ha demostrado que, en promedio, se requieren 2.6 comparaciones por inserción (coste constante!!!!)

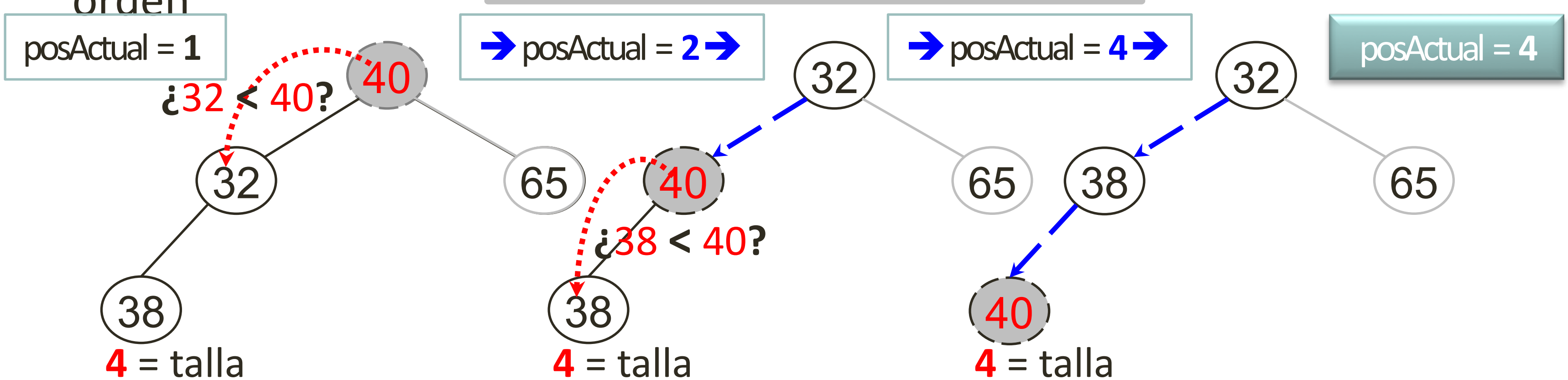
Ejercicio: haz una traza de insertar a partir de un minHeap vacío los siguientes elementos: 6, 4, 15, 2, 10, 11, 8, 1, 13, 7, 9, 12, 5, 3, 14.

3. La clase Java MonticuloBinario – *Método eliminarMin(): algoritmo en 2 pasos, ejemplo 1*

Paso 1: borrar el mínimo del Heap, i.e. *elArray[1]*. El dato del nodo raíz se sustituye por el último elemento del heap.

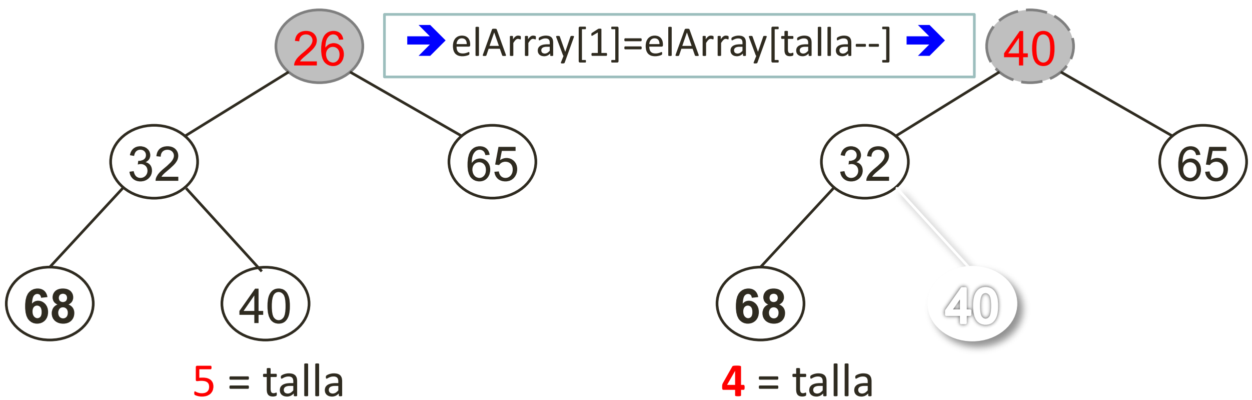


Paso 2: la nueva raíz se hunde a través de sus hijos hasta no violar la propiedad de orden

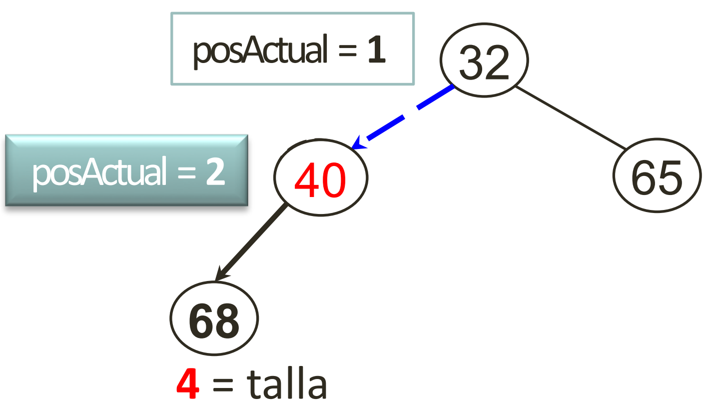


3. La clase Java MonticuloBinario – *Método eliminarMin(): algoritmo en 2 pasos, ejemplo 2*

Paso 1: borrar el mínimo del Heap, i.e. *elArray[1]*. El dato del nodo raíz se sustituye por el último elemento del heap.



Paso 2: la nueva raíz se hunde a través de sus hijos hasta no violar la propiedad de orden



3. La clase Java `MonticuloBinario` –

Método eliminarMin(): código

```
/** recupera y elimina el mínimo de un Heap */
public E eliminarMin() {
   E elMinimo = elArray[1];
   //PASO1-Borrar mínimo del Heap (sustituye raíz por último elemento)
   elArray[1] = elArray[talla--];
   // PASO2-Hundir nueva raíz hasta que no viole propiedad de orden
   hundir(1);
   return elMinimo;
}
```

3. La clase Java `MonticuloBinario` –

Método hundir (heapify), que usa eliminarMin()

- Hunde un nodo a través del heap hasta que no viole la propiedad de orden

```
protected void hundir(int pos) {
    posActual = pos;
    E aHundir = elArray[posActual];
    int hijo = posActual * 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(aHundir) < 0) { //hundimos
            elArray[posActual] = elArray[hijo];
            posActual = hijo;  hijo = posActual * 2;
        }
        else { esHeap = true; } //ya se cumple propiedad heap
    }
    elArray[posActual] = aHundir;
}
```

3. La clase Java `MonticuloBinario` –

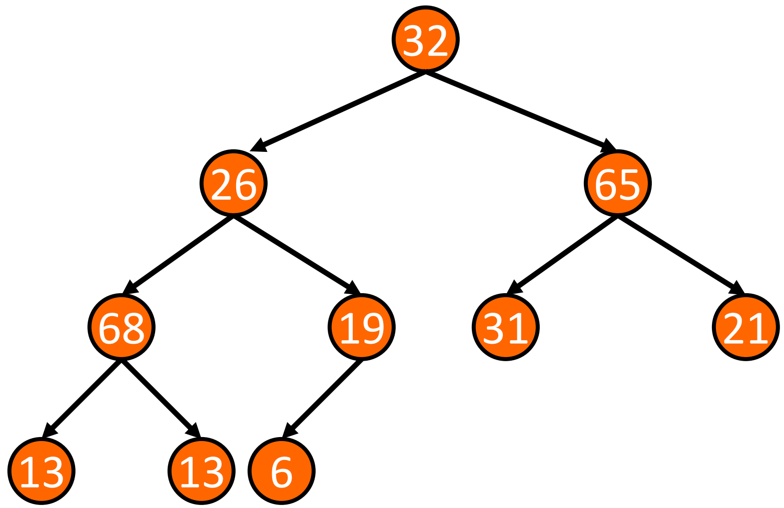
Método eliminarMin(): análisis de su coste

El coste promedio y en el peor de los casos de eliminarMin es logarítmico con el número de elementos.

Ejercicio: haz una traza de eliminarMin sobre el Heap [0,1,4,8,2,5,6,9,15,7,12,13]

3. La clase Java `MonticuloBinario` - *Método arreglarMonticulo() iterativo (buildHeap)*

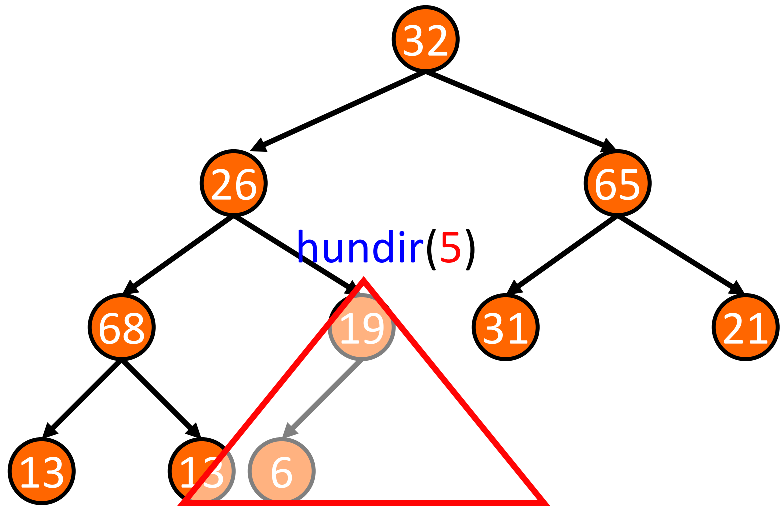
Construir maxHeap a partir del vector: 32, 26, 65, 68, 19, 31, 21, 13, 13, 6



- 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

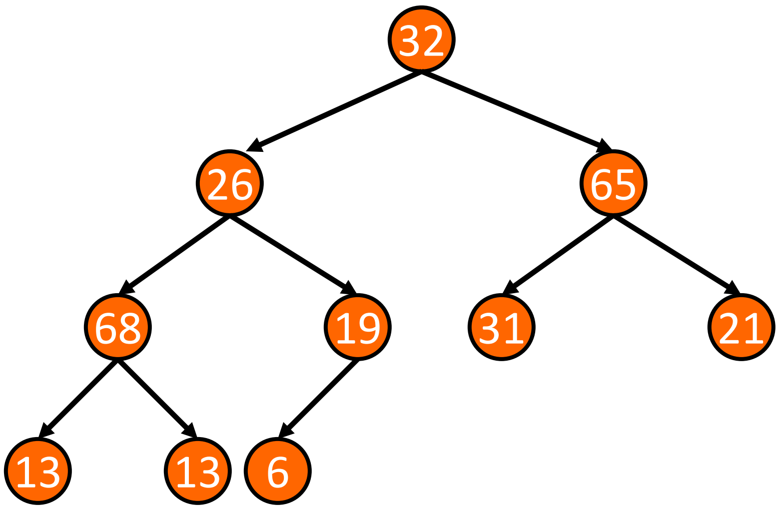
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



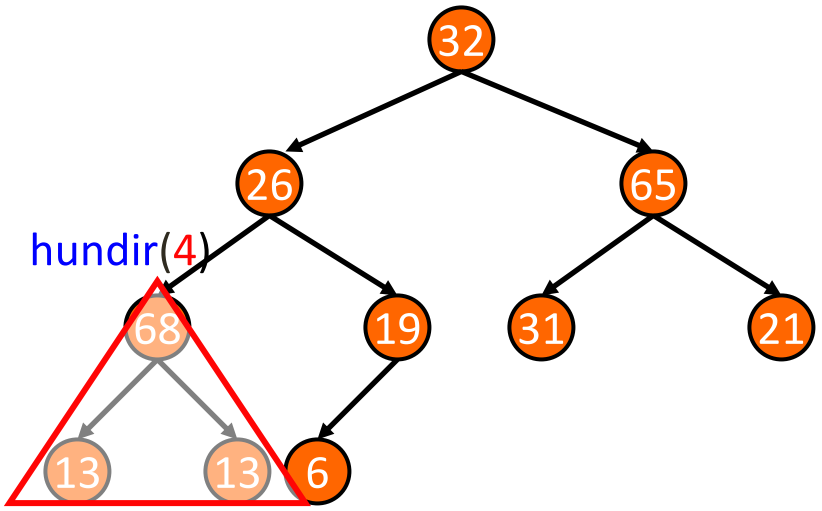
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



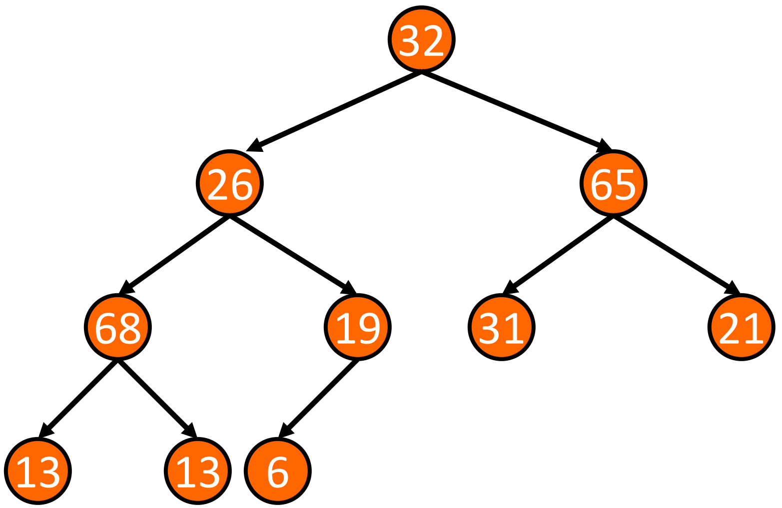
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



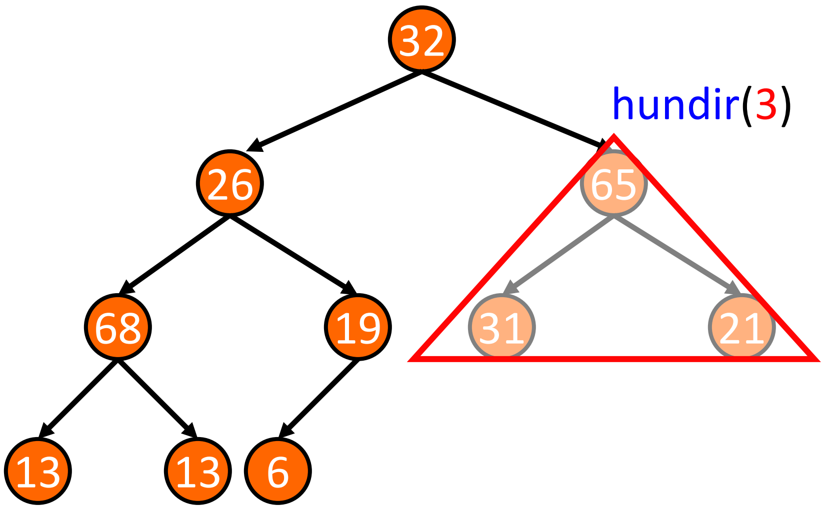
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



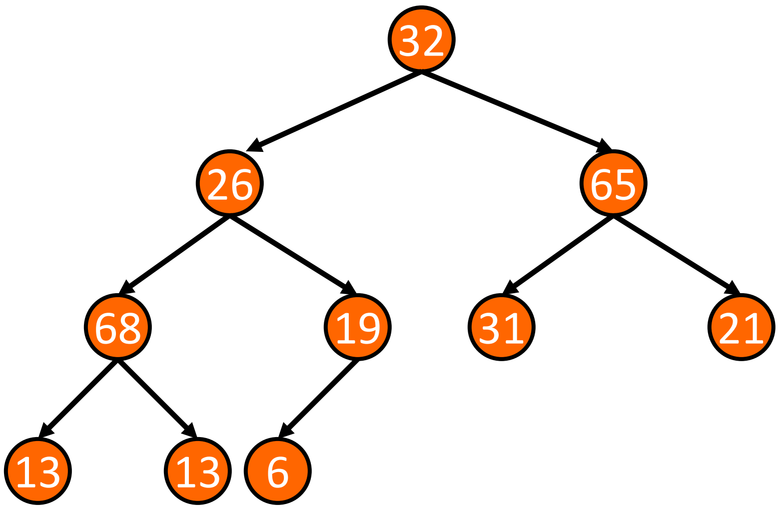
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



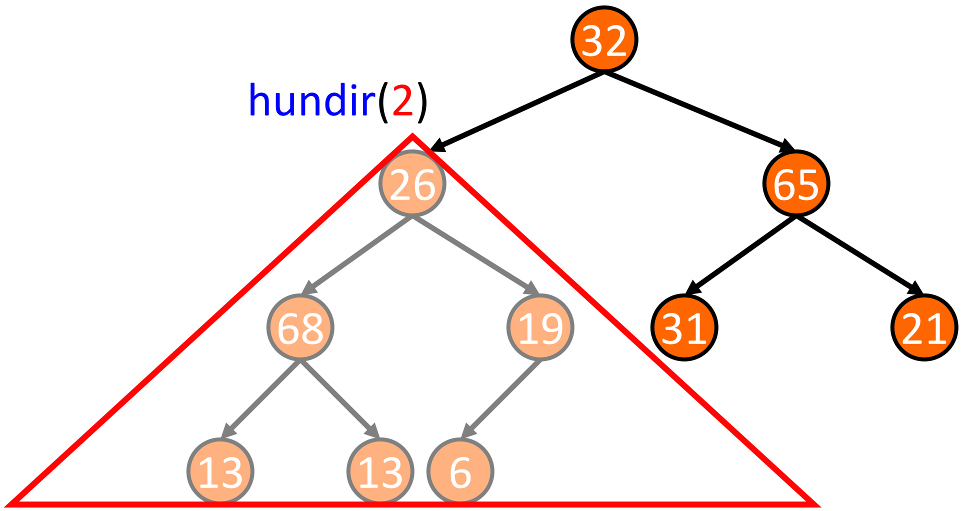
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



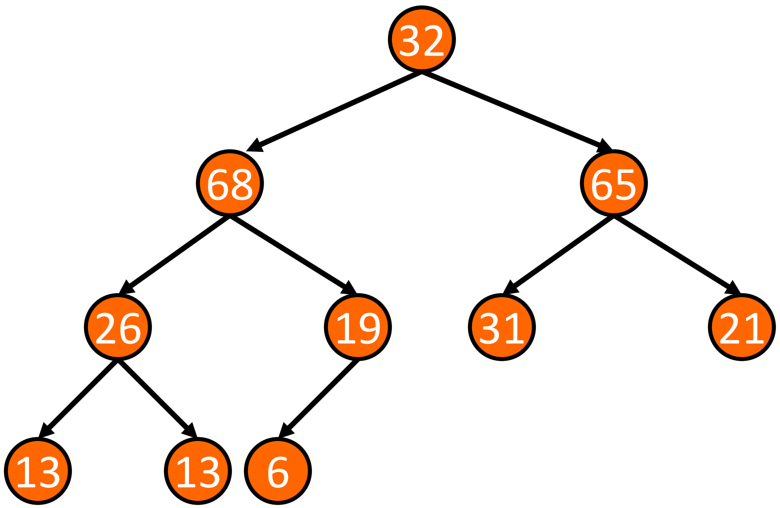
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



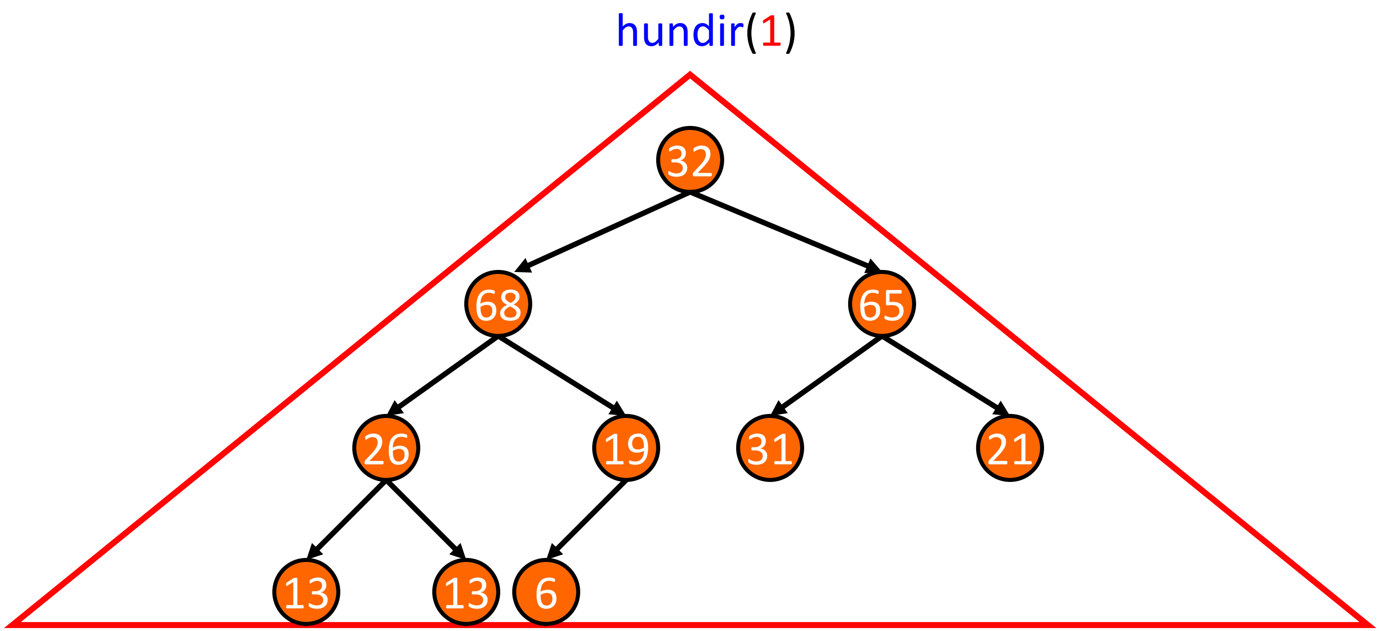
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



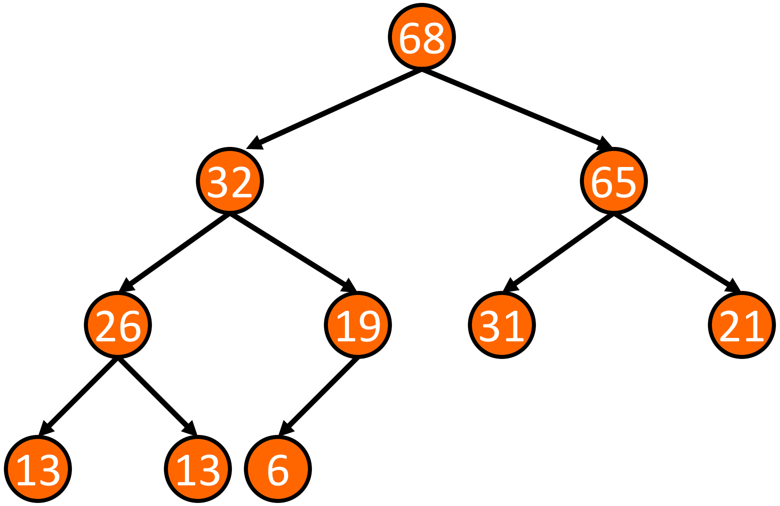
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



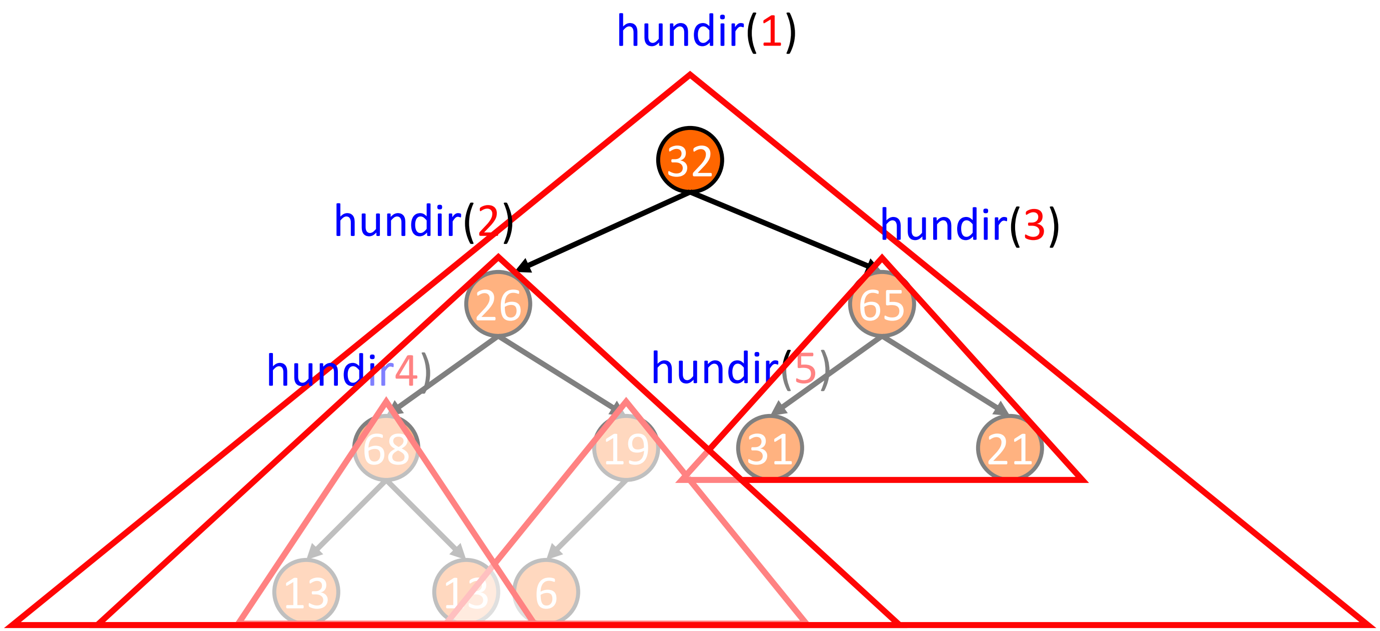
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



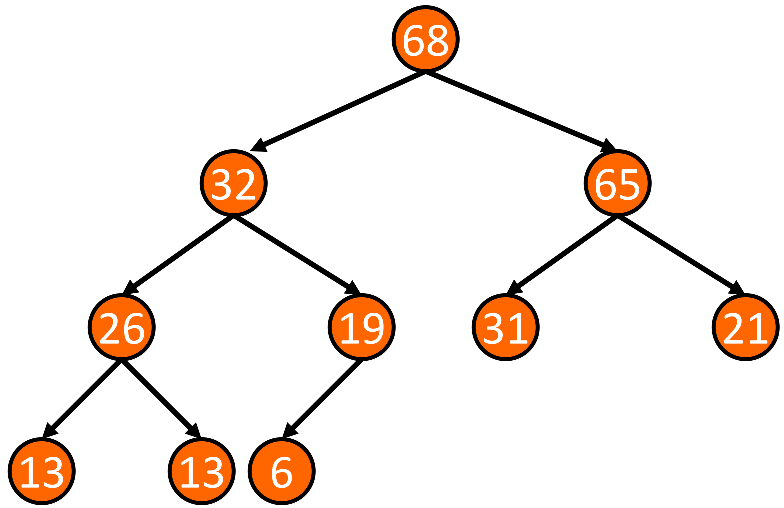
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)




```
/*  Restablece la propiedad de orden de un Heap */
//  "hunde" Por-Niveles  y Descendente los nodos Internos
//  de elArray, pues las Hojas ya son Heaps
public void arreglar() {
    for (int i = talla / 2; i > 0; i --) {
        hundir(i);
    }
}
```

3. La clase Java MonticuloBinario - *Método arreglarMonticulo() iterativo (buildHeap)*




```
/*  Restablece la propiedad de orden de un Heap */
//  "hunde" Por-Niveles  y Descendente los nodos Internos
//  de elArray, pues las Hojas ya son Heaps
public void arreglar() {
    for (int i = talla / 2; i > 0; i --) {
        hundir(i);
    }
}
```

3. La clase Java `MonticuloBinario` - *Método arreglarMonticulo() iterativo (buildHeap) cont.*

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

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

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

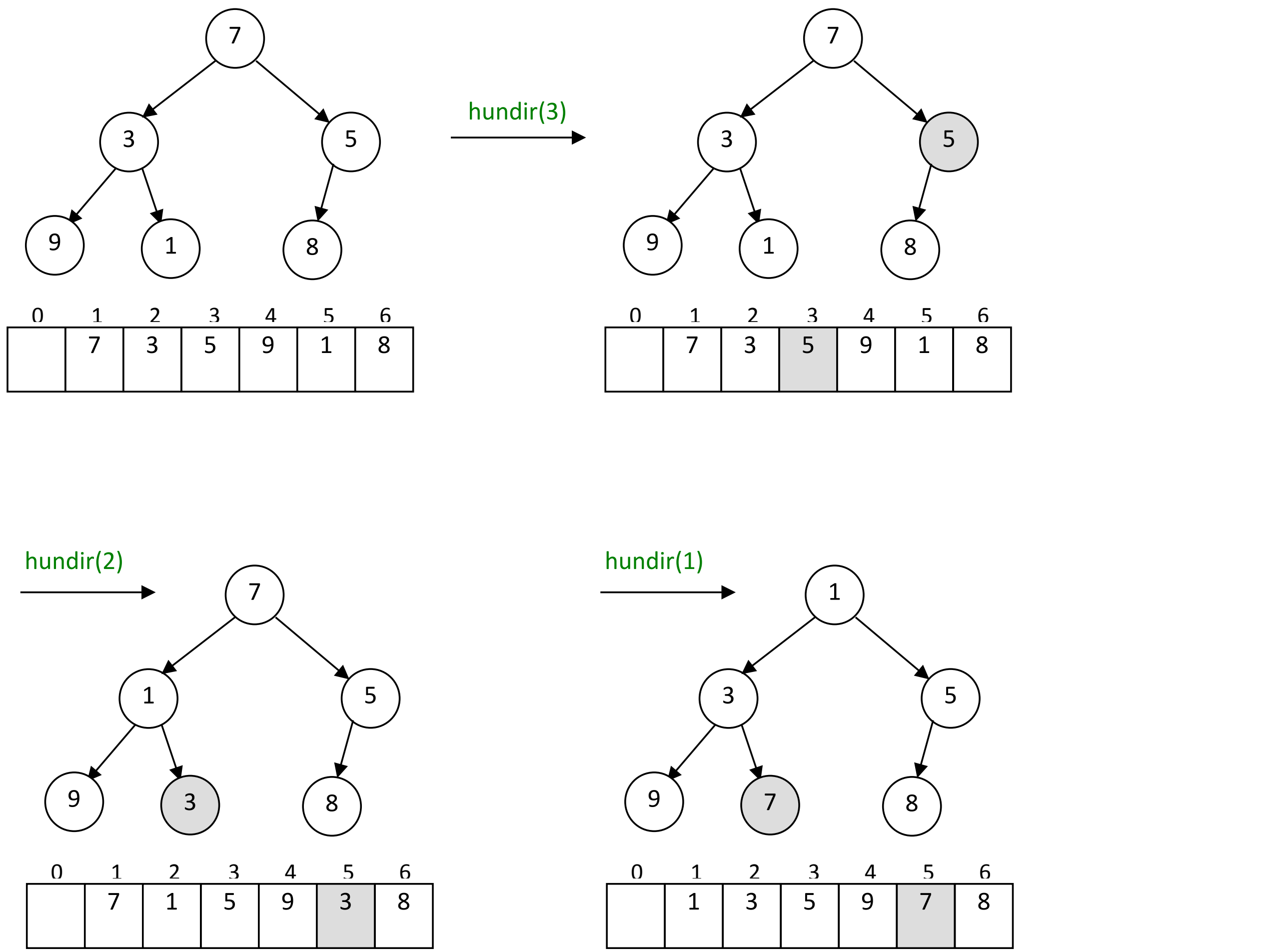
3. La clase Java `MonticuloBinario` – *buildHeap - Ejemplo*

Ejercicio: Hacer una traza del método arreglarMonticulo sobre el árbol binario completo [7, 3, 5, 9, 1, 8] para obtener un minHeap.

3. La clase Java `MonticuloBinario` –

buildHeap - Ejemplo

Ejercicio: Hacer una traza del método arreglarMonticulo sobre el árbol binario completo [7, 3, 5, 9, 1, 8] para obtener un minHeap



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
  - <u>Primer paso</u>: almacenar todos los elementos del vector a ordenar en un montículo (heap)
  - <u>Segundo paso</u>: 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*

```java
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 $O(N + k*\log_2 N)$