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*): esVacia**()**
  - 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*

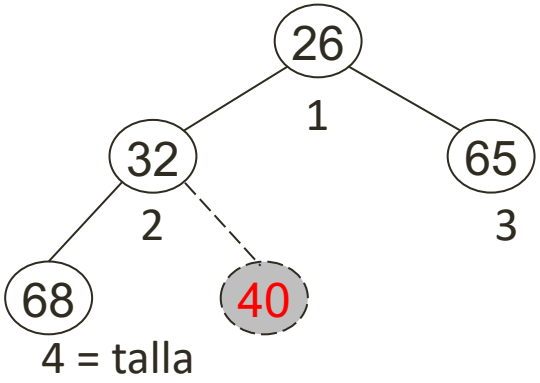
constructor vacío, esVacia y recuperarMin

```
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`

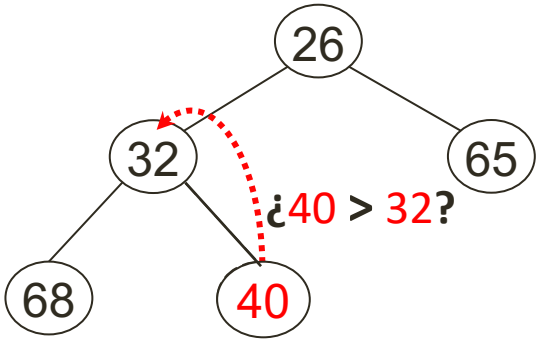
- Método insertar(e): algoritmo en 2 pasos, ejemplo con e = 40

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



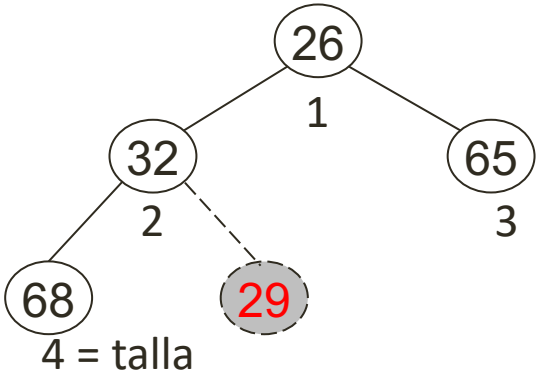
Como 40 **>** 32, si *e* se inserta en *talla+1* el AB sigue siendo Heap➔ posIns = talla **+ 1**

elArray[posIns] = e;

3. La clase Java `MonticuloBinario` –

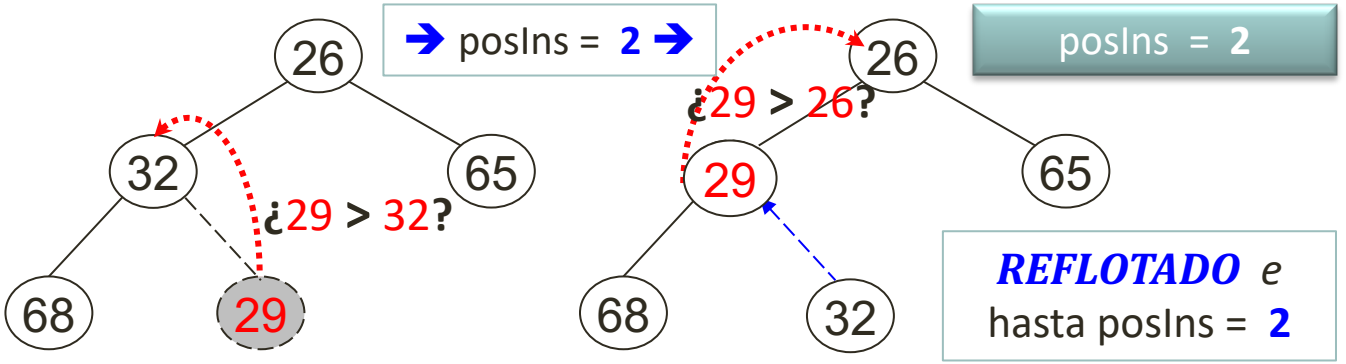
Método insertar(e): algoritmo en 2 pasos, ejemplo con e = 29

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

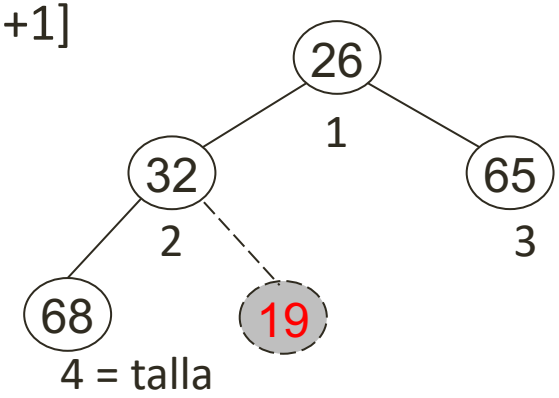


elArray[posIns] = e;

3. La clase Java `MonticuloBinario` –

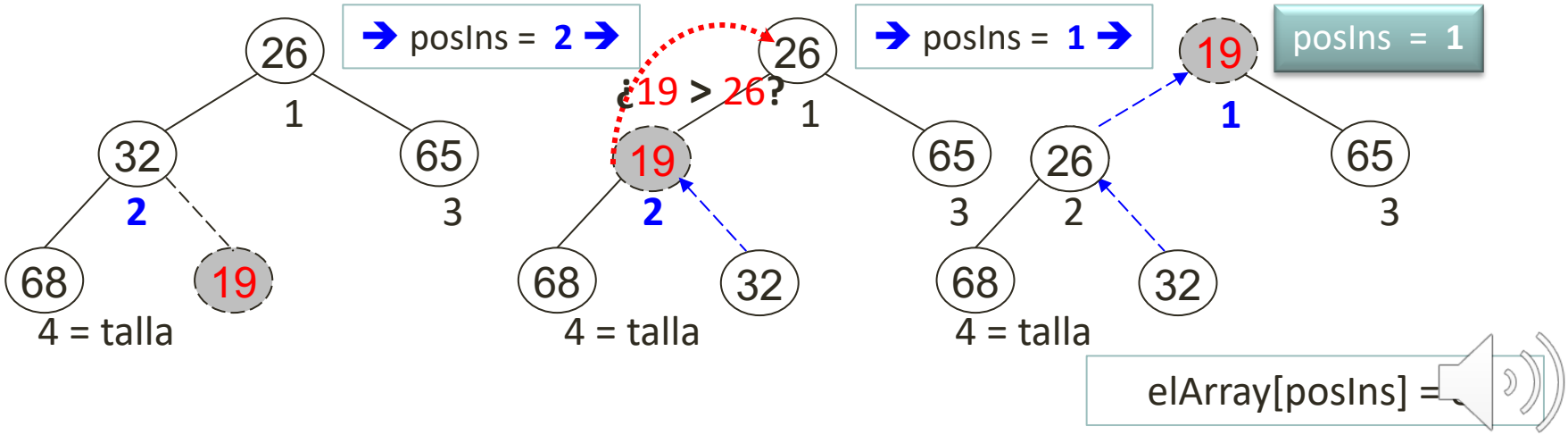
Método insertar(e): algoritmo en 2 pasos, ejemplo con e = 19

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



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!!!!)

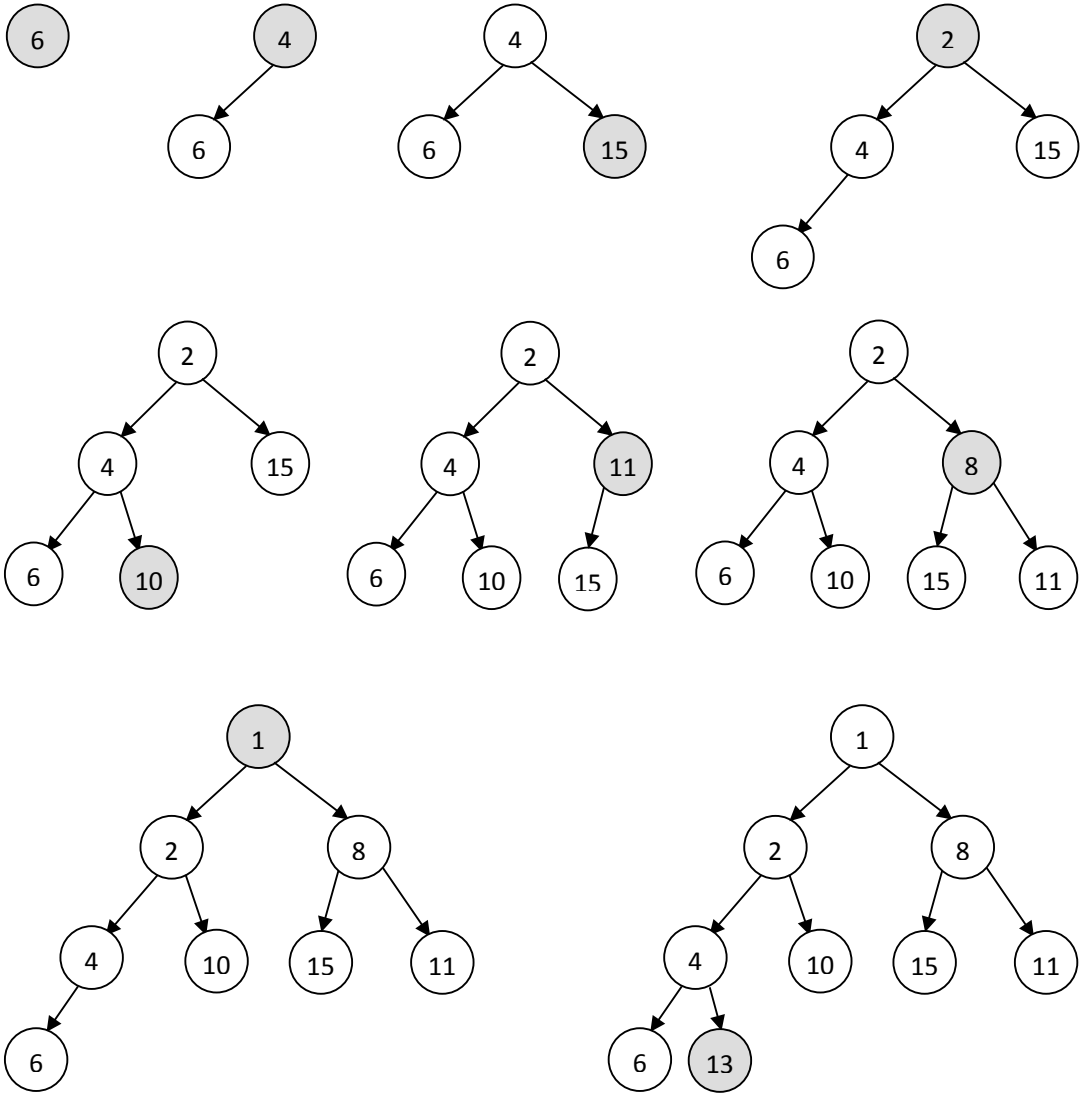
3. La clase Java `MonticuloBinario`-

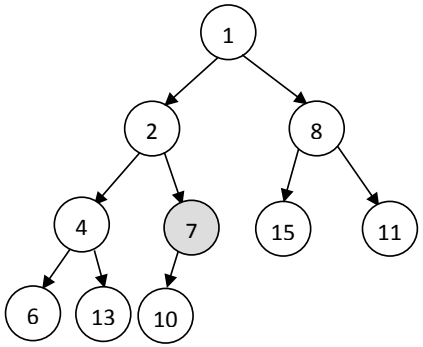
Método insertar(e): ejercicio

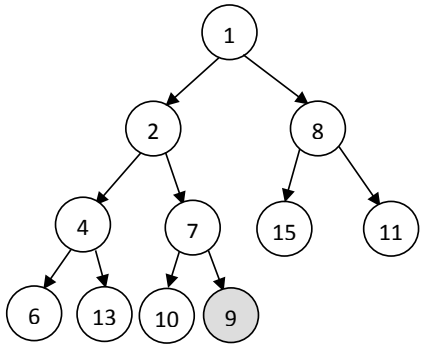
Ejercicio: haz una traza de insertar a partir de un minHeap vacío los siguientes elementos: 6, 4, 15, 2, 10, 11, 8, 1, 13, 7, 9, 12, 5, 3, 14.

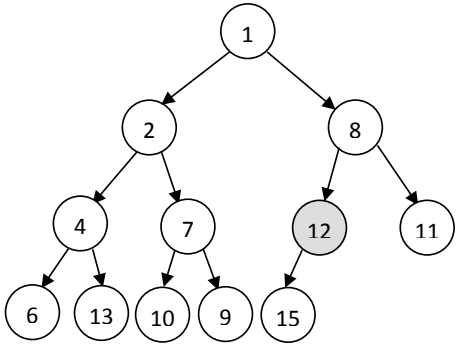
Hacer una traza de insertar a partir de un montículo vacío los siguientes valores: 6, 4, 15, 2, 10, 11, 8, 1, 13, 7, 9, 12, 5, 3, 14.

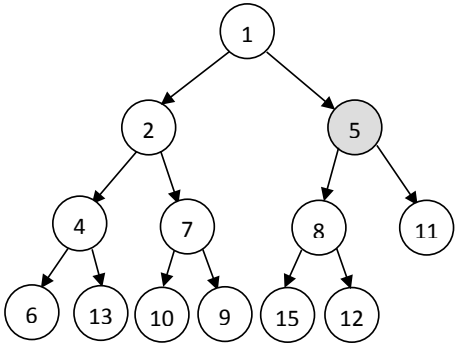
SOLUCIÓN:

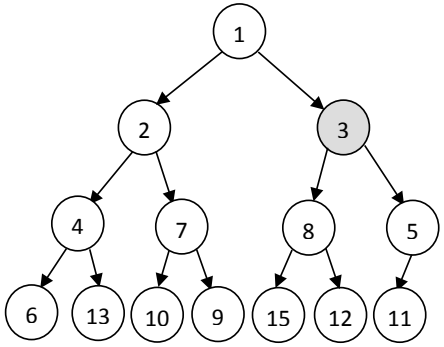


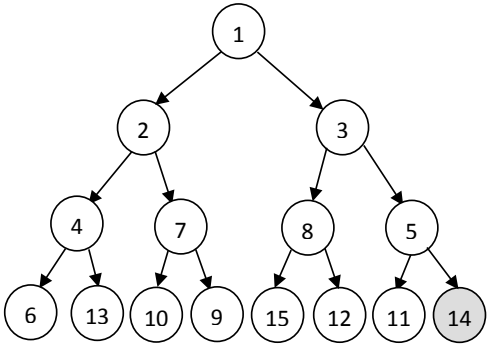

1
2
8
4
7
15
11
6
13
10


1
2
8
4
7
15
11
6
13
10
9


1
2
8
4
7
12
11
6
13
10
9
15


1
2
5
4
7
8
11
6
13
10
9
15
12

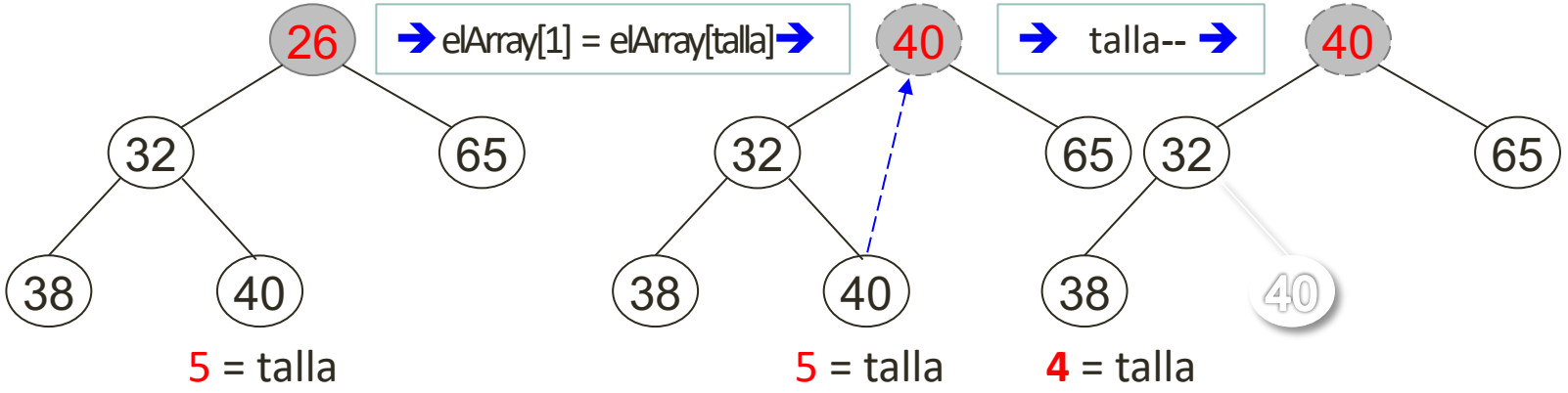

1
2
3
4
7
8
5
6
13
10
9
15
12
11


1
2
3
4
7
8
5
6
13
10
9
15
12
11
14

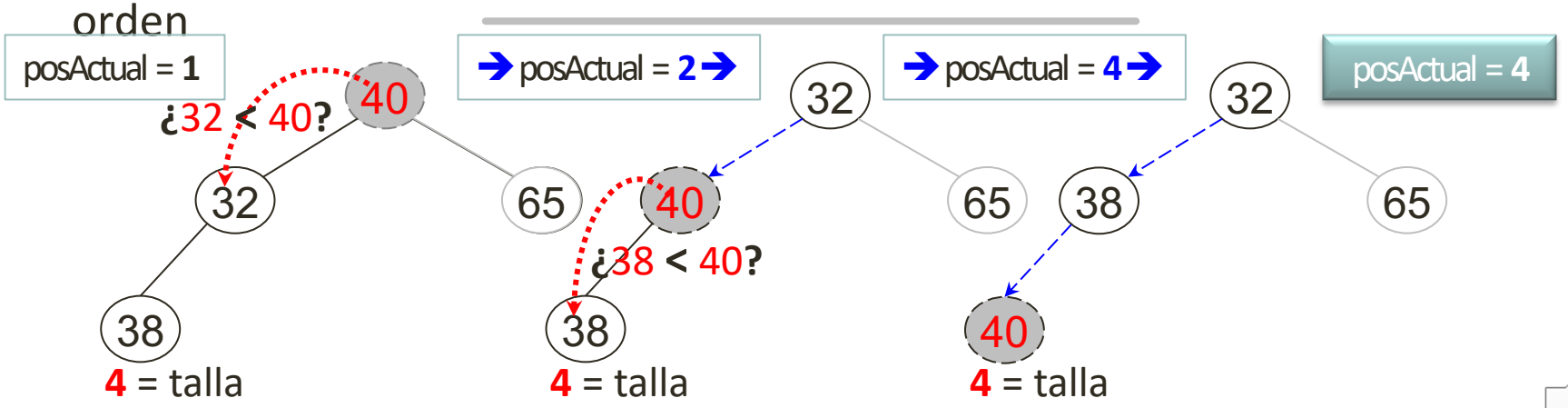
3. La clase Java `MonticuloBinario` – *Método*

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.

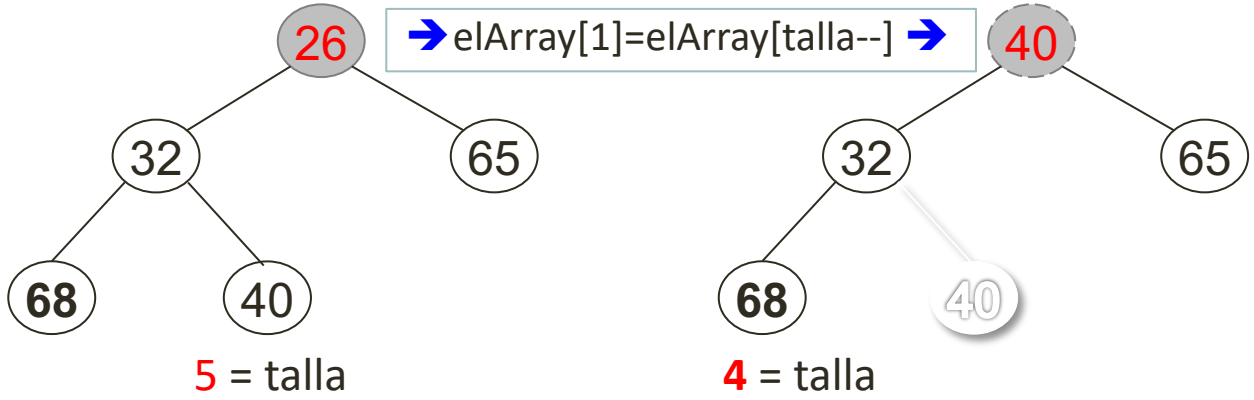


Paso 2: la nueva raíz se hunde a través de sus hijos hasta no violar la propiedad de orden

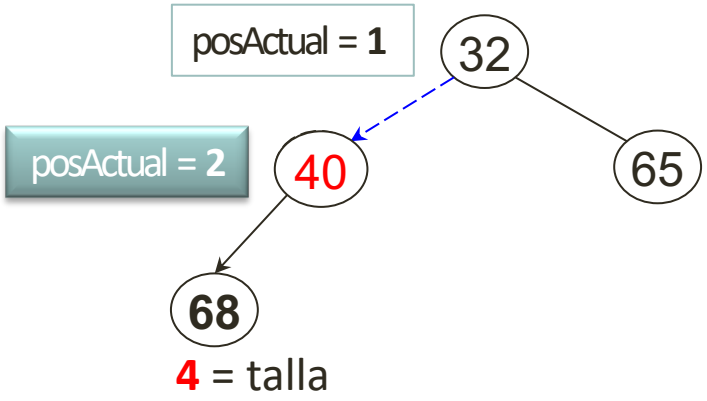


3. La clase Java `MonticuloBinario` – *Método eliminarMin(): algoritmo en 2 pasos, ejemplo 2*

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` –

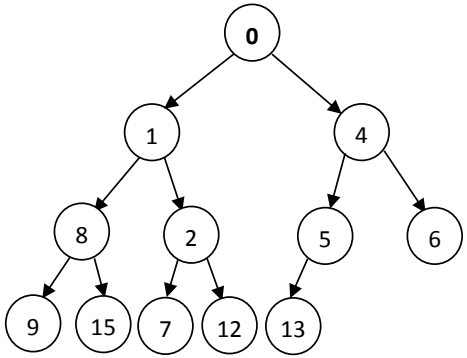
Método eliminarMin(): análisis de su coste

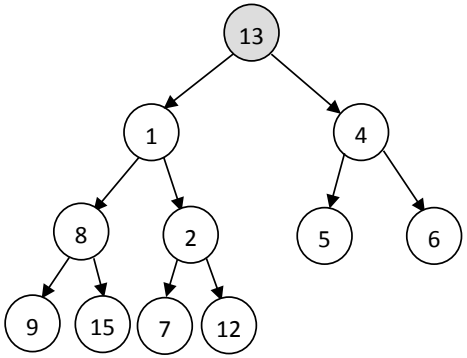
El coste promedio y en el peor de los casos de eliminarMin es logarítmico con el número de elementos.

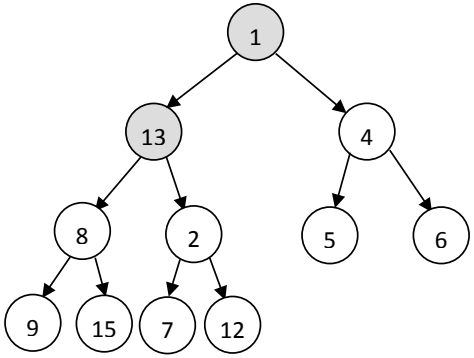
Ejercicio: haz una traza de eliminarMin sobre el Heap [0,1,4,8,2,5,6,9,15,7,12,13]

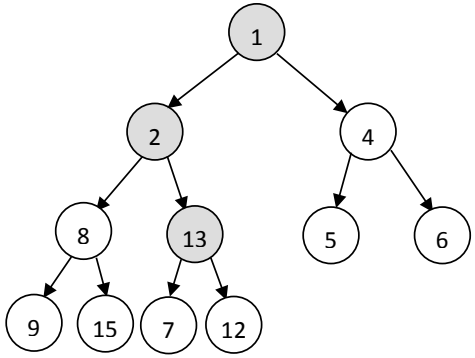
Hacer una traza de *eliminarMin* sobre el Montículo Binario [0; 1; 4; 8; 2; 5; 6; 9; 15; 7; 12; 13].

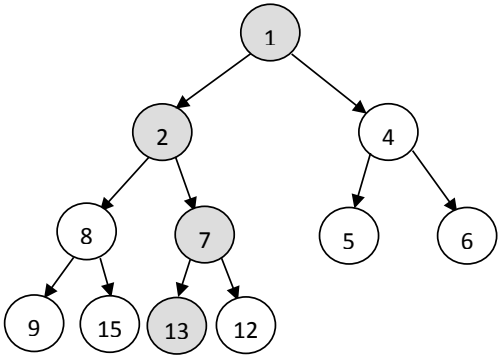
SOLUCIÓN:











Ejercicios

Ejercicio: Escribir un método en la clase MonticuloBinario, que representa un montículo binario minimal, que obtenga su elemento máximo realizando el mínimo número de comparaciones.

Ejercicio: Diseña una función, eliminarMax, que elimine el máximo en un montículo minimal.

Ejercicio: Se quiere insertar los datos de un vector de enteros de talla N en un montículo minimal inicialmente vacío. ¿Qué coste tiene si lo hacemos mediante el método insertar? ¿Cómo puede hacerse de una forma más eficiente?