Ejresueltost12

Estructuras de datos y algoritmos (Universitat Politecnica de Valencia)

TEMA 12

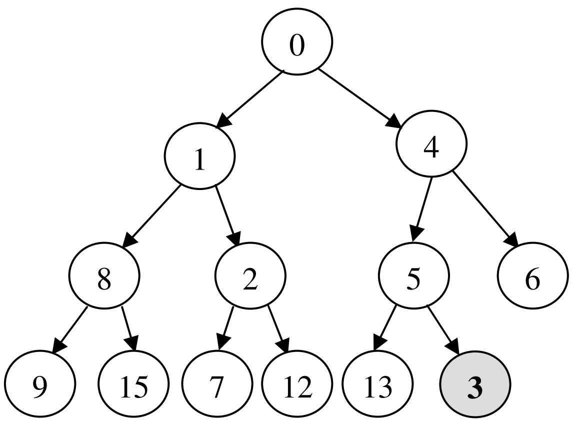
EDAs de búsqueda y su jerarquía Java

Montículo Binario: implementación de Cola de Prioridad y Ordenación Rápida

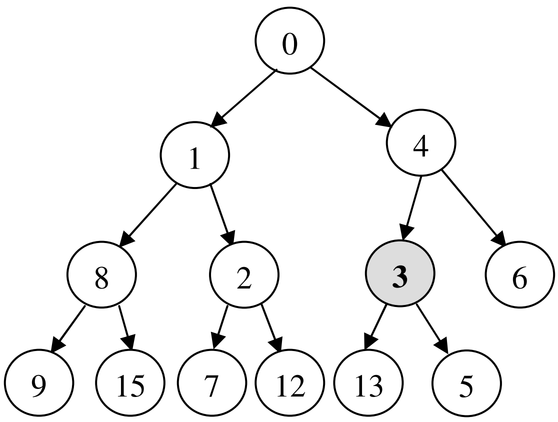
EJERCICIOS RESUELTOS

Ejercicio 1.- Hacer una traza de insertar el valor 3 sobre el *Montículo Binario* [0,1,4,8,2,5,6,9,15,7,12,13]

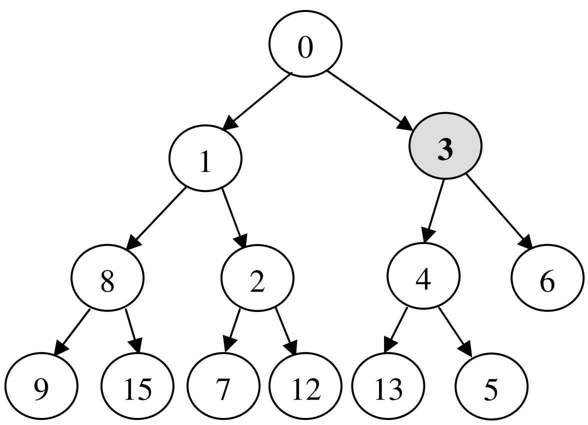
Solución:



Insertamos el dato 3 al final



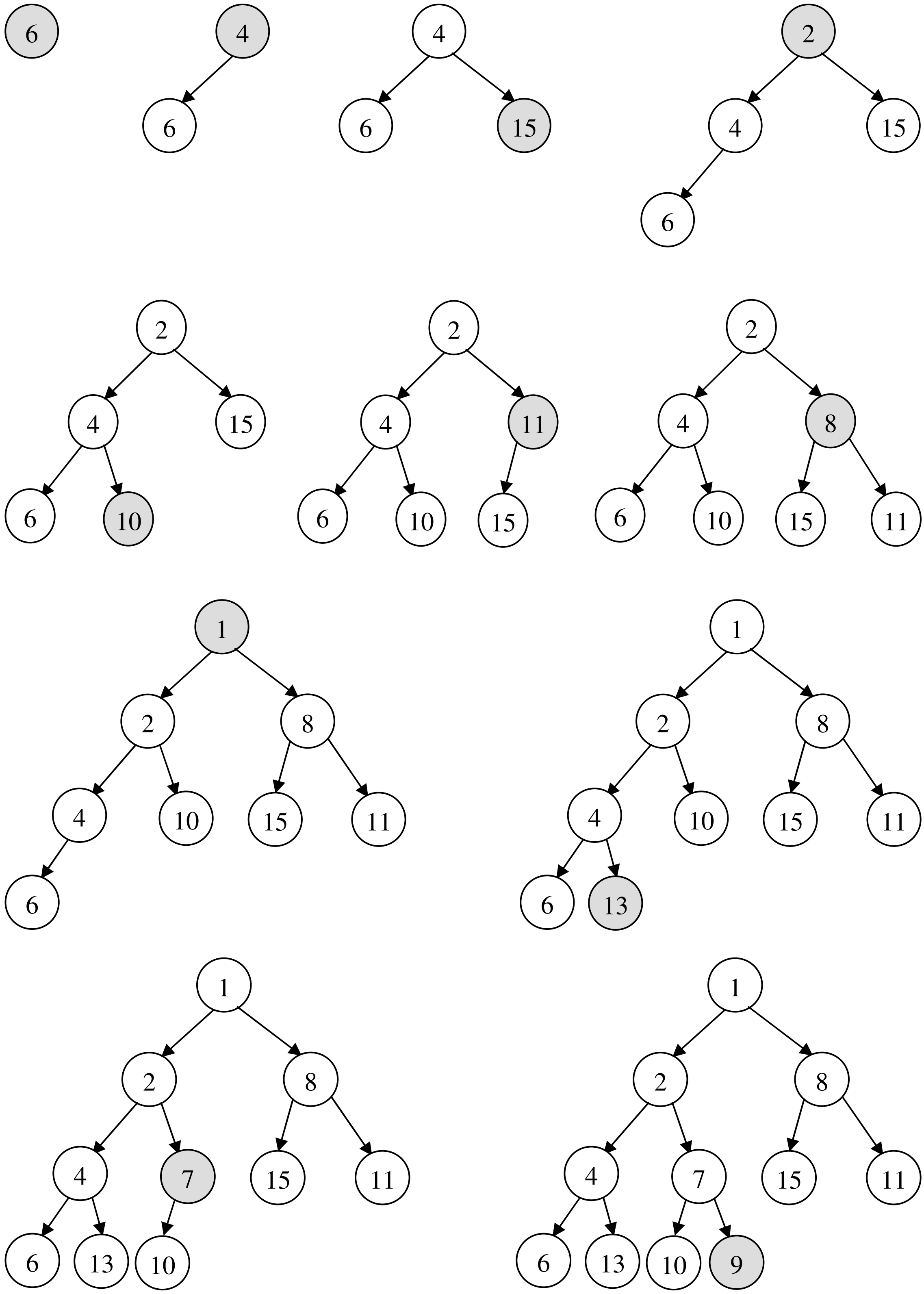
Mientras el nuevo nodo sea menor que su padre se va reflotando

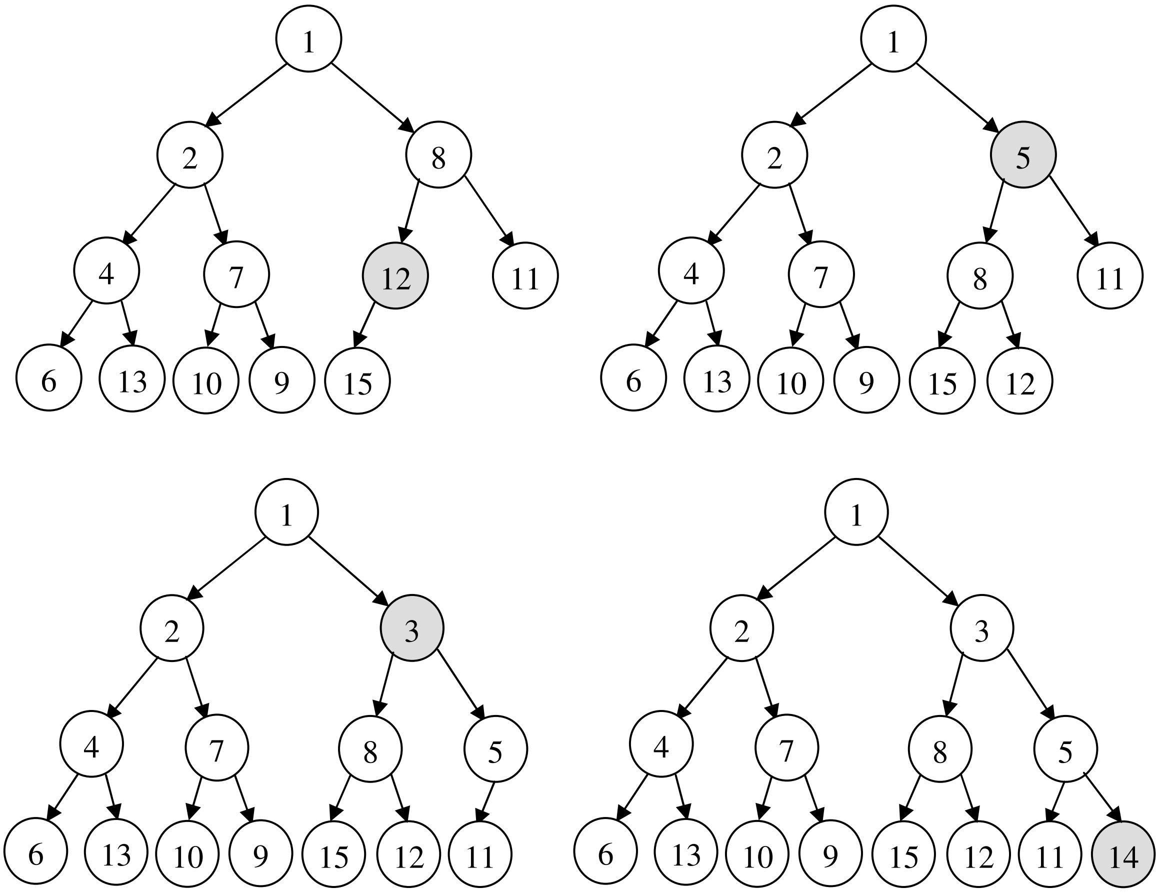


El montículo ya cumple la propiedad de orden

Ejercicio 2.- 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

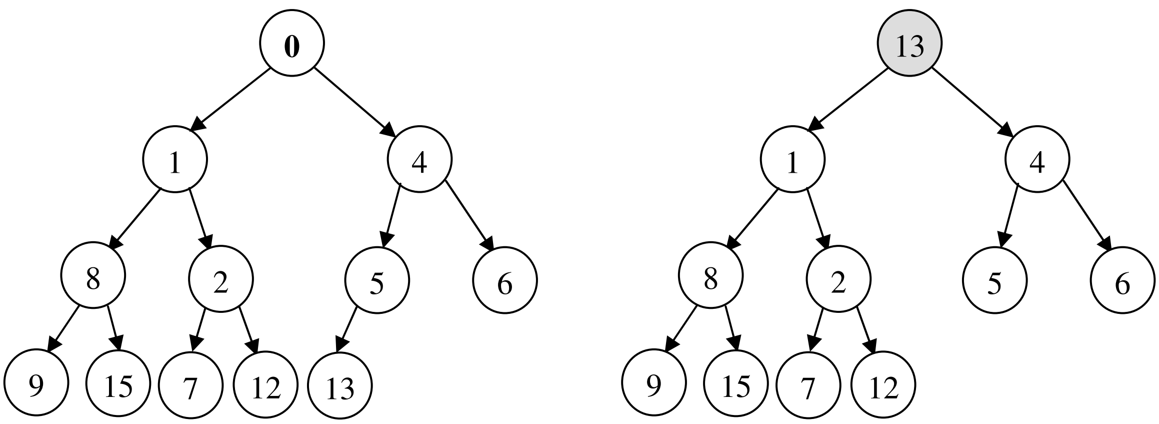
<u>Solución:</u>

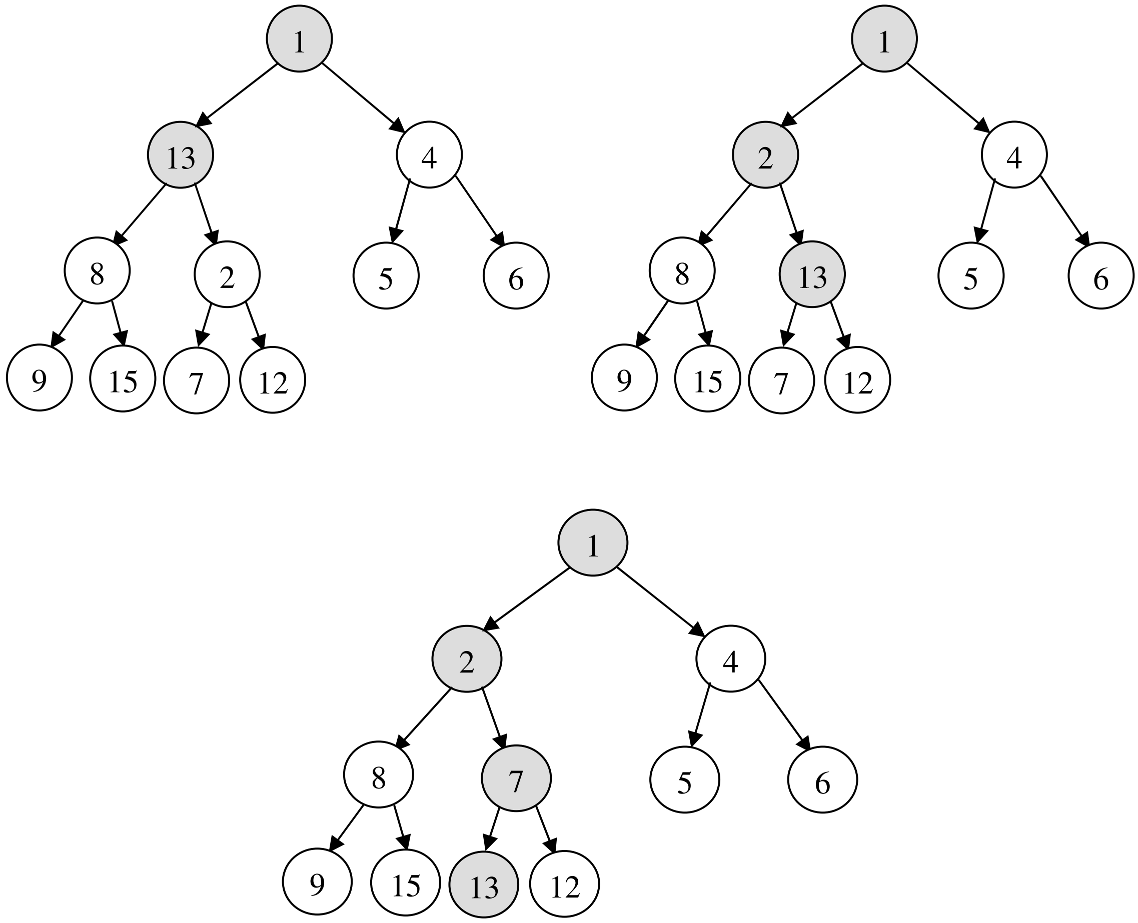




Ejercicio 3.- Hacer una traza de *eliminarMin* sobre el *Montículo Binario* [0; 1; 4; 8; 2; 5; 6; 9; 15; 7; 12; 13]

<u>Solución:</u>





Ejercicio 4.- Hacer una traza de *hundir*(3) sobre el *Montículo Binario* [0; 1; 4; 8; 2; 5; 6; 9; 15; 7; 12; 13]

<u>Solución:</u>

0	1	2	3	4	5	6	7	8	9	10	11	12
	0	1	4	8	2	5	6	9	15	7	12	13
			hueco			*hijo izq*	*hijo der*					

Ya se cumple la propiedad de orden, así que *hundir(3)* no modifica el Montículo.

Ejercicio 5.- Dada una colección de *Integer* representada como un *Montículo Minimal*, escribir un método que obtenga su máximo con el mínimo número de comparaciones.

Solución:

```
public E maximo() {
  if (talla == 0) return null;
  int primeraHoja = talla/2 + 1;
  E max = elArray[primeraHoja];
  for (int i = primeraHoja + 1; i <= talla; i++)
    if (max.compareTo(elArray[i]) < 0) max = elArray[i];
  return max;
}
```

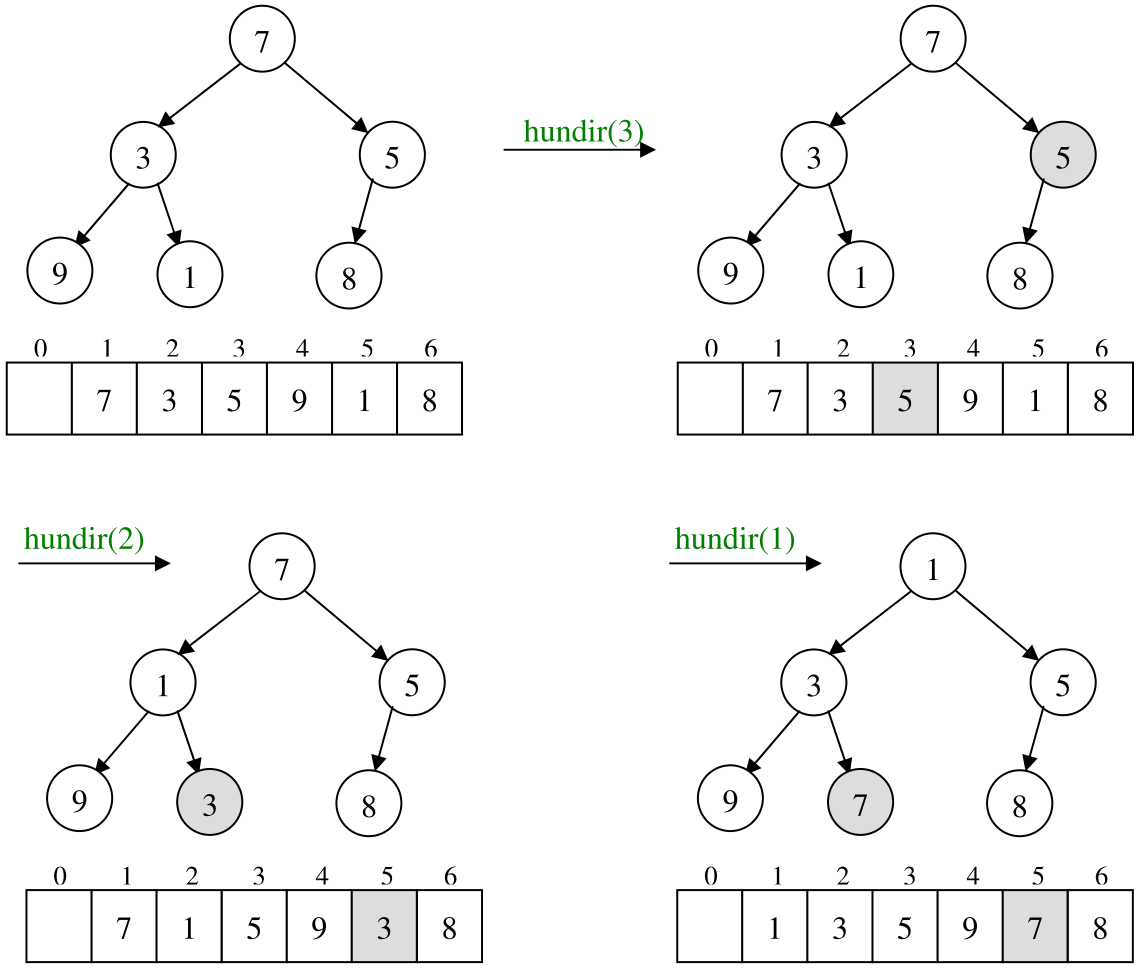
Ejercicio 6.- Diseña una función, *eliminarMax*, que elimine el máximo en un *Montículo Minimal*.

Solución:

```
public E eliminarMax() {
  if (talla == 0) return null;
  int primeraHoja = talla/2 + 1, posMax = primeraHoja;
  // Buscamos el máximo
  for (int i = primeraHoja + 1; i <= talla; i++)
    if (elArray[posMax].compareTo(elArray[i]) < 0)
      posMax = i;
  // Sustituimos el máximo por el último y reflotamos
  E max = elArray[posMax];
  E ult = elArray[talla--];
  while (posMax > 1 && ult.compareTo(elArray[posMax/2]) < 0) {
    elArray[posMax] = elArray[posMax/2];
    posMax = posMax/2;
  }
  elArray[posMax] = ult;
  return max;
}
```

Ejercicio 7.- Hacer una traza del método *arreglarMonticulo* sobre el árbol binario completo <7, 3, 5, 9, 1, 8>

Solución:



Ejercicio 8.- 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?

Solución:

a) Mediante el método *insertar*

Para cada dato llamamos al método insertar. El coste, por lo tanto, será:

$$T_{insertarDatos}^{\mu}(N) = k_1 * N \qquad \rightarrow T_{insertarDatos}(N) \in \Theta(N)$$

$$T_{insertarDatos}^{P}(N) = k_2 * \sum_{i=1}^{N} \lfloor \log_2 i \rfloor \qquad \rightarrow T_{insertarDatos}(N) \in O(N*\log_2 N)$$

b) Se puede hacer de forma más eficiente con el método `arreglarMonticulo`:

$T_{arreglarMonticulo}(N) \in O(N)$

Demostración: en el peor caso, el método *arreglarMonticulo* tiene el coste correspondiente a la suma de las alturas de todos los nodos del árbol. Para establecer este cálculo se recurre al siguiente teorema:

Teorema: Dado un Árbol Binario 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.

Puesto que un Árbol Binario Completo tiene entre 2^H y $N = 2^{H+1} - 1$ nodos, también la suma de la alturas de sus nodos está acotada por O(N).

Ejercicio 9.- Diséñese un método que compruebe si un dato *x* dado está en un *Heap Minimal* y estúdiese su coste.

<u>Solución:</u>

```
public boolean buscar(E x) {
  return buscar(x, 1);
}

private boolean buscar(E x, int pos) {
  boolean encontrado = false;
  if (pos <= talla) {
    int res = x.compareTo(elArray[pos]);
    if (res == 0) encontrado = true;
    else if (res > 0)
      encontrado = buscar(x, pos*2) || buscar(x, pos*2+1);
  }
  return encontrado;
}
```

<u>Talla del problema</u>: tamaño del nodo que ocupa la posición *pos*

<u>Instancias significativas</u>:

- *Mejor caso*: el objeto 'x' es igual o menor que el mínimo del *Heap*.
- *Peor caso*: el objeto 'x' es mayor que todos los datos del *Heap*

<u>Relaciones de recurrencia</u>:
$T_{buscar}^{M}(talla) = k_1$

$T_{buscar}^{P}(talla = 0) = k_2$
$T_{buscar}^{P}(talla > 0) = 2 * T_{buscar}^{P}(talla/2) + k_3$

Coste asintótico del método:
$T_{buscar}(talla) \in \Omega(1)$
$T_{buscar}(talla) \in O(talla)$

Ejercicio 10.- Diséñese un método que elimine el dato de la posición *k* de un *Heap Minimal* y estúdiese su coste.

Solución:

```
public E eliminarK(int k) {
  E dato = elArray[k];
  E ult = elArray[talla--];
  if (ult.compareTo(dato) < 0) {
    // El dato a borrar es mayor que el último → reflotamos
    while (k > 1 && ult.compareTo(elArray[k/2]) < 0) {
      elArray[k] = elArray[k/2];
      k = k/2;
    }
    elArray[k] = ult;
  } else {
    // El dato a borrar es menor que el último → hundimos
    elArray[k] = ult;
    hundir(k);
  }
  return dato;
}
```

Talla del problema: número de elementos del montículo

Instancias significativas:

- *Mejor caso*: al sustituir el elemento *k* por el último no se viola la propiedad de orden.
- *Peor caso*: cuando *k* = 1 y el último elemento del *Heap* es el máximo

Ecuaciones de coste para el método:
$T_{eliminarK}^{M}(talla) = k_1$
$T_{eliminarK}^{P}(talla) = T_{hundir}^{P}(talla) + k_2 = (k_3 * \log_2 talla + k_4) + k_2$

Coste asintótico del método:
$T_{eliminarK}(talla) \in \Omega(1)$
$T_{eliminarK}(talla) \in O(\log_2 talla)$