Ejercicios de clase
TEMA 5 – Cola de Prioridad y Montículo Binario

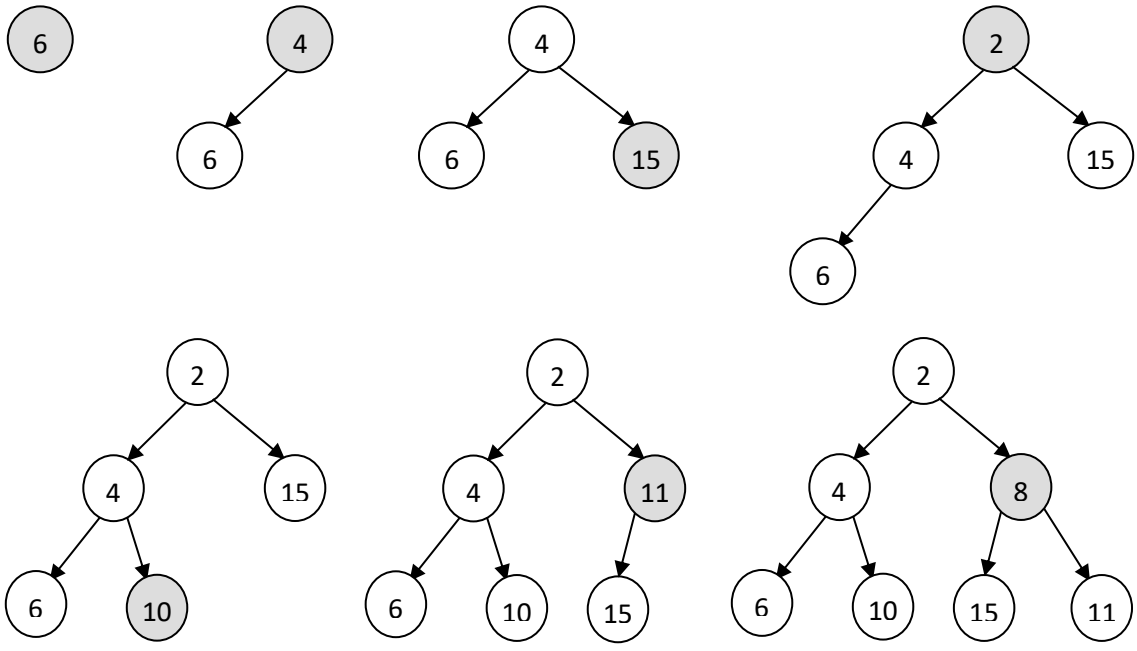
Ejercicio 1

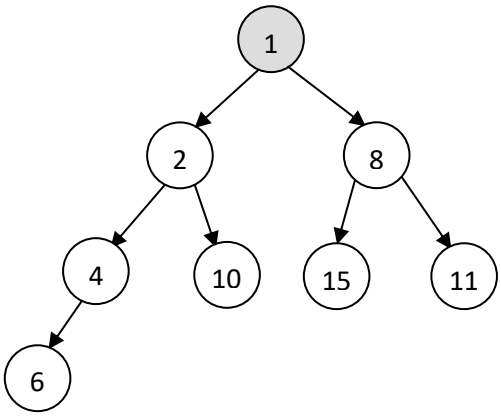
Hacer una traza de insertar a partir de un montículo vacío los siguientes valores: 6, 4, 15, 2, 10, 11, 8, 1, 13, 7, 9, 12, 5, 3, 14.

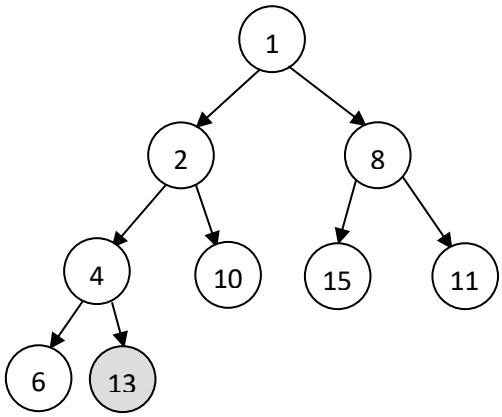
SOLUCIÓN:

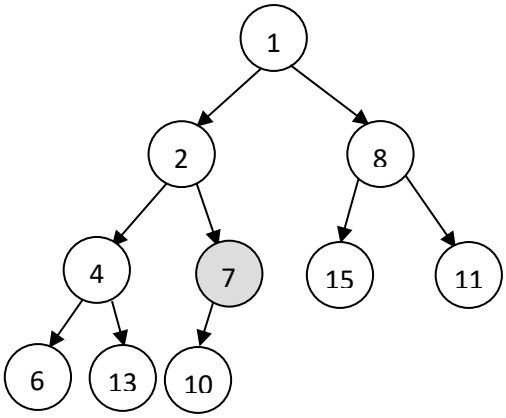
Este ejercicio ya lo habéis realizado justo después de estudiar la operación de inserción. Fíjate que en las figuras se refleja el estado final del heap después de hacer la inserción. Pero tened en cuenta cómo se hace:

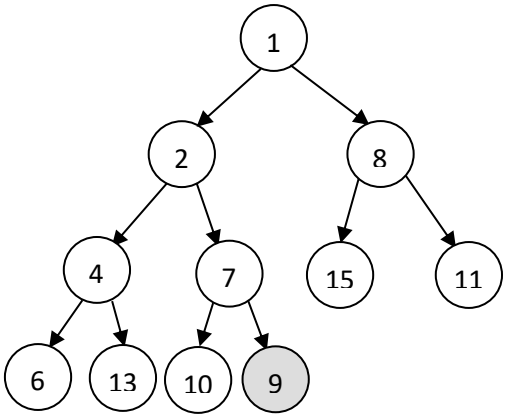
- Primero se mete el elemento a insertar como una nueva hoja en el heap.
- Luego va subiendo, intercambiando la posición con el padre, hasta llegar al sitio que le corresponde para cumplir la condición de orden (el padre debe ser menor que sus hijos). No perdáis de vista que lo que se dibuja como un árbol en realidad es un array con todos los elementos llenos desde la posición 1 hasta la última.
- El resultado final sería: 1,2,3,4,7,8,5,6,13,10,9,15,12,11,14. Fijaos que, en general, no hay ninguna relación de orden entre las partes izquierdas y las derechas de los subárboles.

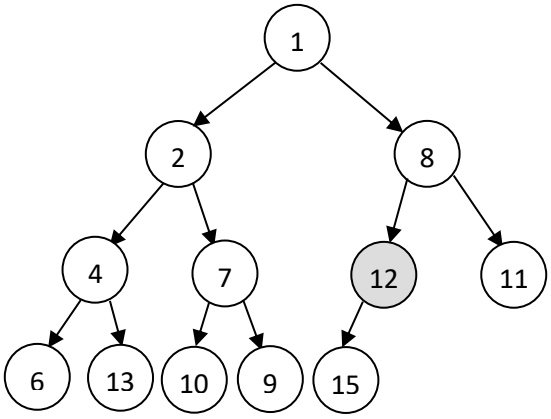


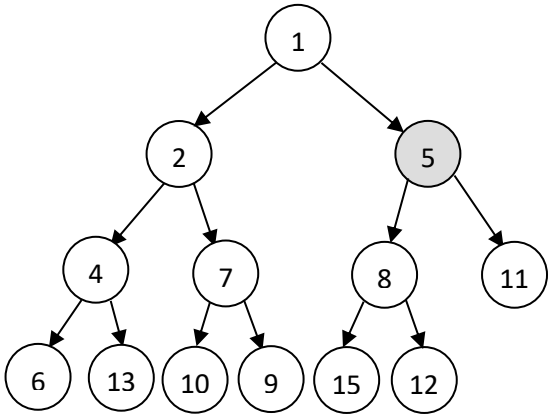

1
2
8
4
10
15
11
6

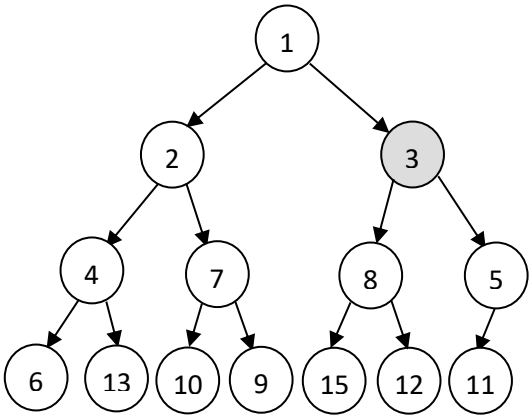

1
2
8
4
10
15
11
6
13

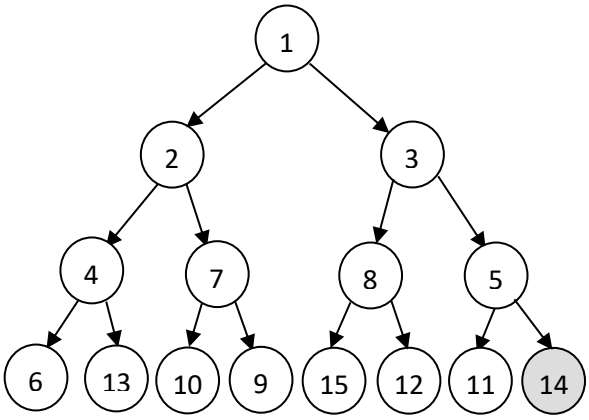

1
2
8
4
7
15
11
6
13
10


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

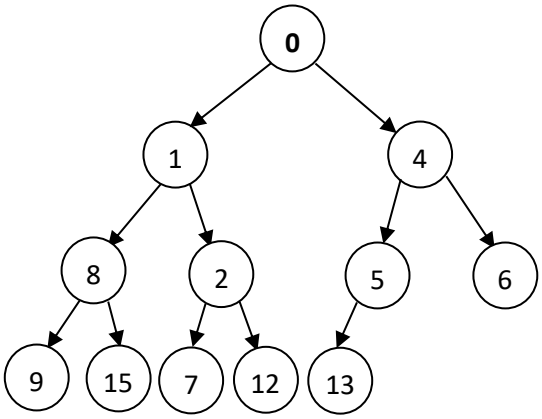
Ejercicio 2

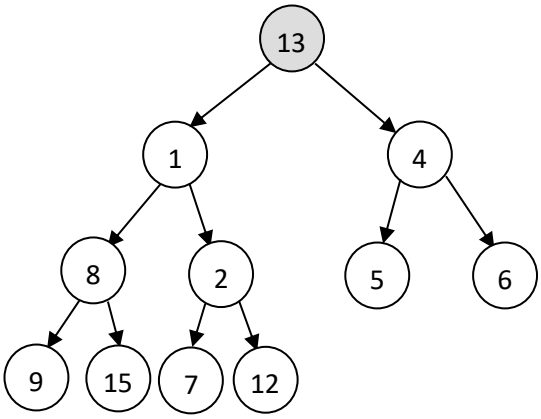
Hacer una traza de *eliminarMin* sobre el Montículo Binario [0; 1; 4; 8; 2; 5; 6; 9; 15; 7; 12; 13].

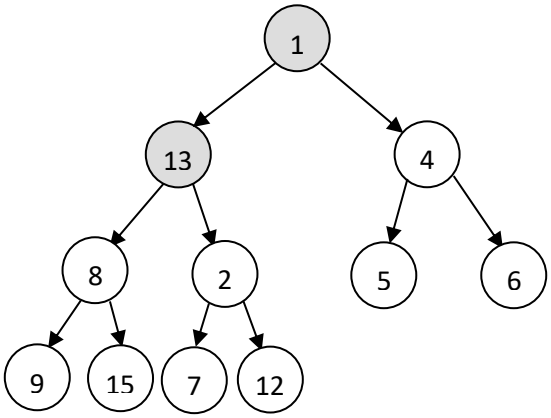
SOLUCIÓN:

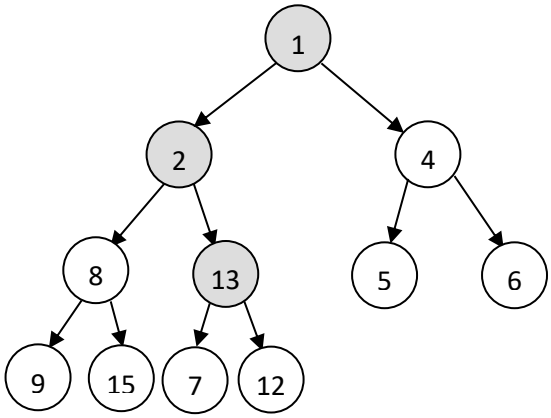
Este ejercicio ya lo hemos comentado en clase. Para eliminar el mínimo lo que se hace es machacar el valor de la raíz con el del último elemento (en este caso el 13), y luego se decrementa la talla del heap.

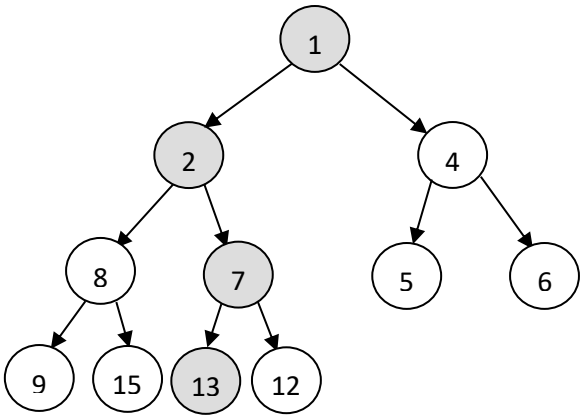
A continuación hay que hundir el 13 hasta la posición que le corresponda. En cada paso (ver figuras) se intercambia con el menor de sus hijos para que el menor quede arriba y se mantenga la propiedad de orden del heap.











Ejercicio 3

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

SOLUCIÓN:

Este problema es sencillo. El máximo seguro que está en una de las hojas, por lo que es suficiente recorrer las hojas y quedarse con el máximo. La primera hoja estará en la posición talla/2+1 y la última hoja en la posición talla.

```
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 4

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

SOLUCIÓN:

Para eliminar el máximo primero tenemos que encontrarlo. Por eso, como en el ejercicio anterior se recorren las hojas y nos guardamos el índice donde está el máximo. Luego tendrás que tener en cuenta que solo se puede borrar el último elemento del heap (simplemente decrementando la talla, `talla--`), ya que el resultado no permite huecos en el array. La propiedad estructural del heap de que sea un árbol completo (con las hojas "apretadas a la izquierda") obliga a que sólo se elimine la última casilla del array. Esto es importante para muchos problemas.

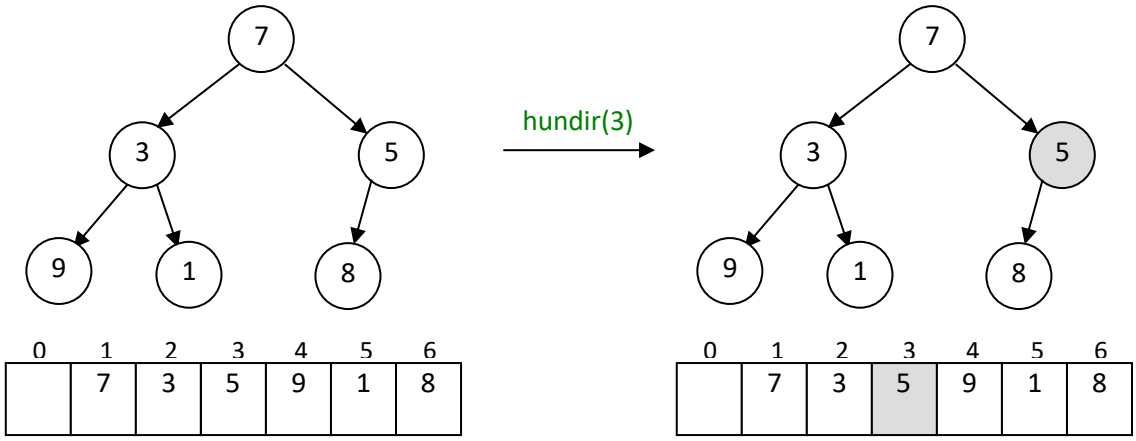
Una vez detectado el índice del máximo lo que se hace es machacar su valor con el del último elemento del array (o sea del heap, es lo mismo), y decrementar la talla. Ahora sólo hay que hacer subir ese elemento hasta donde le toque.

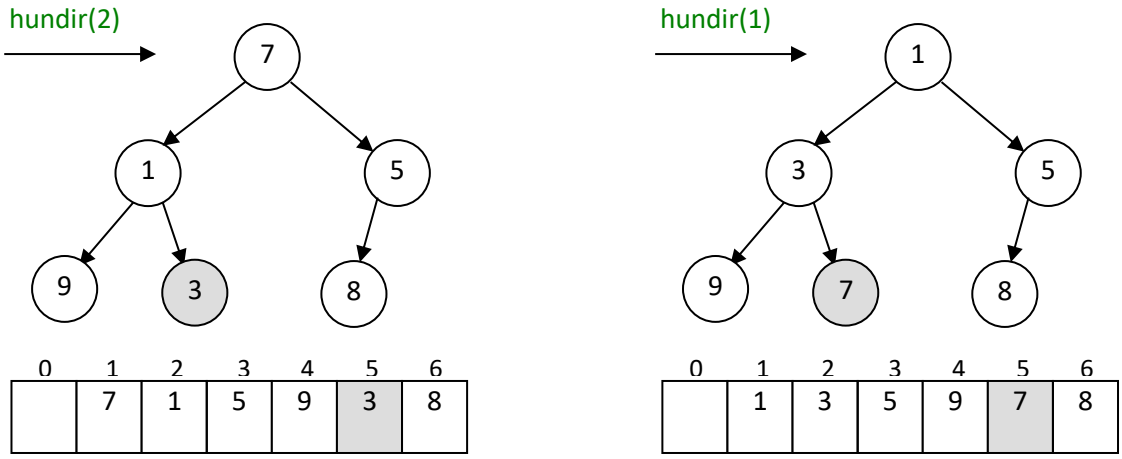
```
public E eliminarMax() {
  if (talla == 0) return null;
  int primeraHoja = talla/2 + 1, posMax = primeraHoja;
  // Buscamos el máximo
  for (int i = primeraHoja + 1; i <= talla; i++)
    if (elArray[posMax].compareTo(elArray[i]) < 0)
      posMax = i;
  // Sustituimos el máximo por el último y reflotamos
  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 5

Hacer una traza del método *arreglarMonticulo* sobre el árbol binario completo [7, 3, 5, 9, 1, 8].

SOLUCIÓN:





Ejercicio 6

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:

Este ejercicio es la explicación del coste de arreglarMonticulo (con un coste O(n)) frente a crear el heap a base de O(n) inserciones, con un coste O(n.log n).

a) Mediante el método *insertar*

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

$T_{insertarDatos}^{\mu}(N) = k_1 * N + k_2 \rightarrow T_{insertarDatos}(N) \in \Theta(N)$

$T_{insertarDatos}^{P}(N) = k_3 * \sum_{i=1}^{N} \lfloor \log_2 i \rfloor + k_4 \rightarrow T_{insertarDatos}(N) \in O(N*\log_2 N)$

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 7

Diséñese un método que compruebe si un dato *x* dado está en un montículo minimal y estúdiese su coste.

SOLUCIÓN:

En este problema se trata de buscar un dato en el heap. Esta es una incómoda operación para esta estructura de datos que no está pensada para esto, porque hay una relación de orden "de padres a hijos", pero no entre "hermanos". Por eso, en principio habría que recorrerlo todo, pero afortunadamente podemos ahorrarnos recorrer partes de heap, ya que si buscamos un elemento que es menor que el dato del nodo que estoy explorando ya sabemos que no estará en el subárbol que cuelga de él. Por poner un caso extremo, si la raíz de heap tiene un 5 y buscamos el 3, sólo con esa comparación ya sabemos que no está.

El algoritmo que se da como solución es un algoritmo recursivo (en heaps suele ser más fácil encontrar soluciones recursivas) que compara el dato buscado con el dato del nodo en que estamos (primero la raíz) y si es mayor entonces buscamos en AMBOS hijos. Se hace un OR para que devuelva true si aparece en alguno de los subárboles.

Como podéis ver el coste sería $\Omega(1)$ si el dato buscado es menor o igual al que hay en la raíz, y O(n) si tenemos que recorrerlo todo (porque no está o porque es el último).

```
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;
}
```

Talla del problema: tamaño del nodo que ocupa la posición *pos*

Instancias significativas:

- *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*

Relaciones de recurrencia:

$T_{buscar}^{M}(talla) = k_1$

$T_{buscar}^{P}(talla = 0) = k_2$

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

Coste asintótico del método:

$T_{buscar}(talla) \in \Omega(1)$

$Tbuscar(talla) \in O(talla)$

Ejercicio 8

Diséñese un método que elimine el dato de la posición *k* de un montículo minimal y estúdiese su coste.

SOLUCIÓN:

La operación "borrar el elemento de la posición k" tampoco es muy cómoda para esta estructura de datos.. De nuevo lo primero que tenéis que pensar es que sólo se puede eliminar la última casilla del heap. Por eso hay que salvar el dato de la última casilla y ponerlo en la posición k (así ya nos hemos cargado el elemento deseado). Luego, no se os olvide se reduce la talla del heap talla--. Finalmente el dato que ha quedado en la posición k no está en su sitio y puede ser que tenga que ascender o descender tal como se ve en la solución.

De nuevo el coste de este algoritmo es $\Omega(1)$ en el mejor caso y $O(\log n)$ en el peor caso.

```
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:

$T_{eliminarK}(talla) \in \Omega(1)$

$T_{eliminarK}(talla) \in O(\log_2 talla)$