Soluciones Ejercicios Tema 12

Germán Moltó
gmolto@dsic.upv.es
Estructuras de Datos y Algoritmos
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Traza de Insertar

	0	1	4	8	2	5	6	9	15	7	12	13	
0	1	2	3	4	5	6	7	8	9	10	11	12	

↑ talla (12)

	0	1	4	8	2	5	6	9	15	7	12	13	
0	1	2	3	4	5	6	7	8	9	10	11	12	

↑ talla (12), hueco

- ¿3 <elArray[6] (5) ?

	0	1	4	8	2		6	9	15	7	12	13	5
0	1	2	3	4	5	6	7	8	9	10	11	12	

↑ hueco (6) ↑ talla

Traza de Insertar

- ¿3 < elArray[3] (4) ?

	0	1		8	2	4	6	9	15	7	12	13	5
0	1	2	3	4	5	6	7	8	9	10	11	12	

↑ hueco (3) ↑ talla

- ¿3 < elArray[1] (0) ?

	0	1	3	8	2	4	6	9	15	7	12	13	5
0	1	2	3	4	5	6	7	8	9	10	11	12	

↑ hueco (3) ↑ talla

Traza de Insertar en Montículo Vacio

0	1	2	3	4	5	6	7	8	9	10	11	12	

talla = 0

- insertar(6)

	6												
0	1	2	3	4	5	6	7	8	9	10	11	12	

talla = 1

- insertar(4)

	4	6											
0	1	2	3	4	5	6	7	8	9	10	11	12	

talla = 2

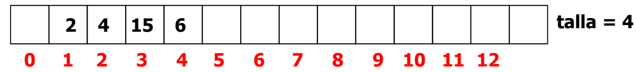
- insertar(15)

	4	6	15										
0	1	2	3	4	5	6	7	8	9	10	11	12	

talla = 3

Traza de Insertar en Montículo Vacio

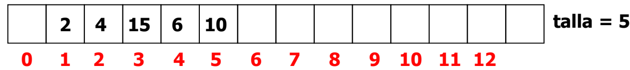
- insertar(2)



	2	4	15	6								
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 4

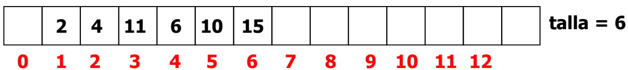
- insertar(10)



	2	4	15	6	10							
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 5

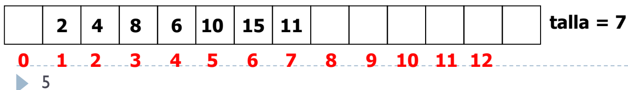
- insertar(11)



	2	4	11	6	10	15						
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 6

- insertar(8)

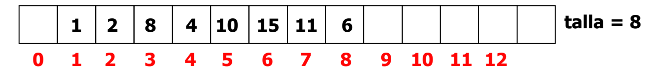


	2	4	8	6	10	15	11					
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 7

Traza de Insertar en Montículo Vacio

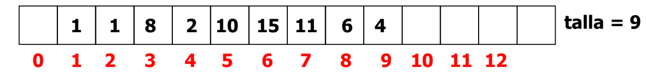
- insertar(1)



	1	2	8	4	10	15	11	6				
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 8

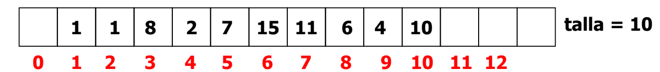
- insertar(1)



	1	1	8	2	10	15	11	6	4			
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 9

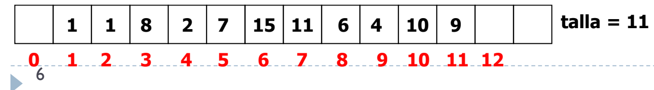
- insertar(7)



	1	1	8	2	7	15	11	6	4	10		
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 10

- insertar(9)

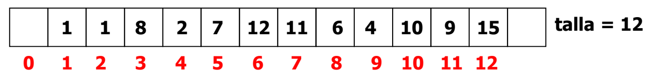


	1	1	8	2	7	15	11	6	4	10	9	
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 11

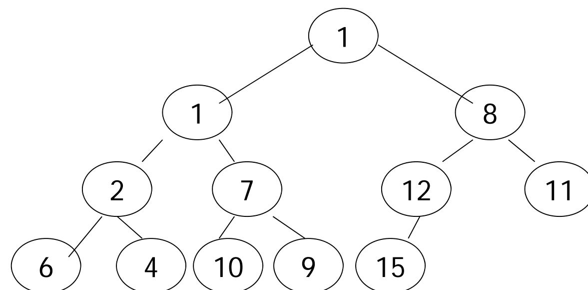
Traza de Insertar en Montículo Vacio

- insertar(12)

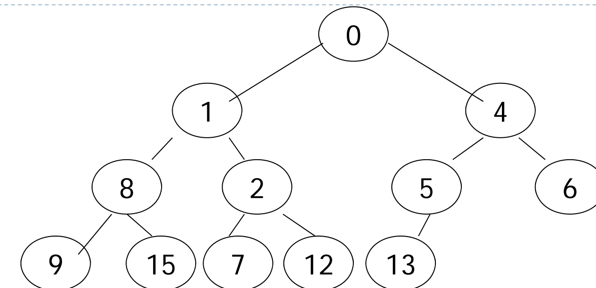


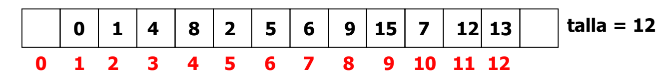
	1	1	8	2	7	12	11	6	4	10	9	15
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 12



Traza de eliminarMin

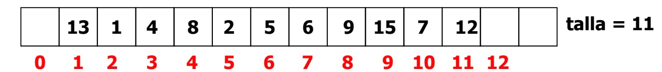




	0	1	4	8	2	5	6	9	15	7	12	13
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 12

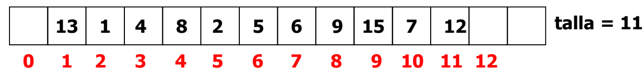
- eliminarMin()



	13	1	4	8	2	5	6	9	15	7	12	
0	1	2	3	4	5	6	7	8	9	10	11	12

talla = 11

Traza de eliminarMin

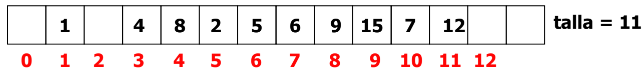


- hueco = 1, hijo = 2, aux = 13

¿elArray[3] < elArray[2]? NO (Selección del mejor hijo)

¿elArray[2] < 13 ? SI (hundo)

elArray[1] = elArray[2]

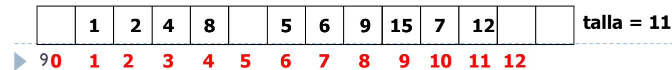


- hueco = 2, hijo = 4, aux = 13

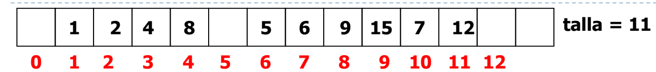
¿elArray[5] < elArray[4]? SI (Selección del mejor hijo)

¿elArray[5] < 13 ? SI (hundo)

elArray[2] = elArray[5]



Traza de eliminarMin

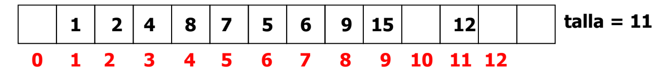


- hueco = 5, hijo = 10, aux = 13

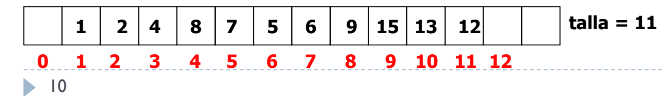
¿elArray[11] < elArray[10]? NO (Selección del mejor hijo)

¿elArray[10] < 13 ? SI (hundo)

elArray[5] = elArray[10]

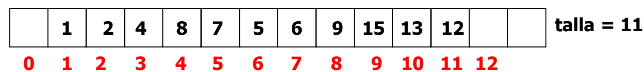


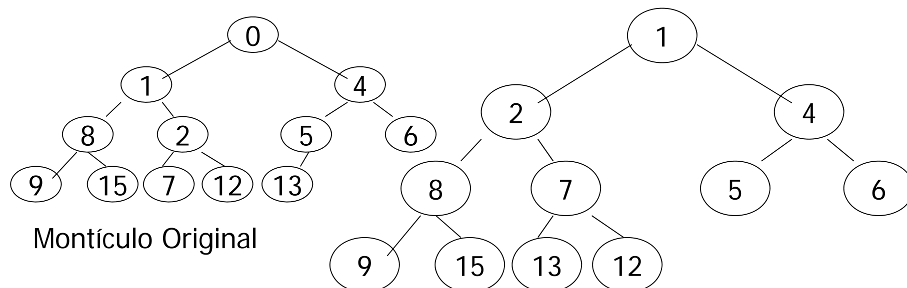
- hueco = 10, hijo = 20, aux = 13
- Salimos del bucle puesto que hijo > talla
- elArray[10] = 13



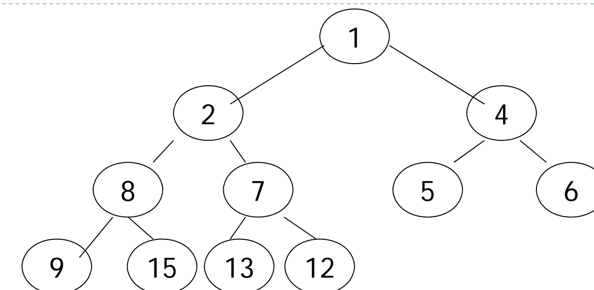
Traza de eliminarMin

- Representamos el resultado en forma de Montículo Binario.





Obtener Máximo de Montículo Minimal



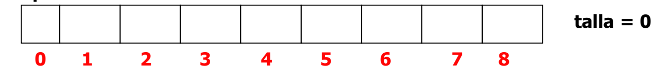
- Dado un Montículo Binario Minimal, sabemos que el máximo elemento estará en una de las hojas, aunque desconocemos cual de ellas lo alberga.
- Buscamos el máximo elemento de las hojas y ese será el máximo elemento del Montículo Minimal.

Obtener Máximo de Montículo Minimal

```
public E buscarMax(){
    int posPrimHoja = (talla / 2) + 1;
    E elMaximo = elArray[posPrimHoja];
    for (int i = posPrimHoja + 1 ; i <= talla ; i++){
        if ( elArray[i].compareTo(elMaximo) > 0)
         elMaximo = elArray[i];
    }
    return elMaximo;
 }
```


Montículo con Datos de Igual Prioridad

- Asumimos que partimos de un Montículo cuya capacidad inicial es suficiente para albergar los datos que se pretende insertar.



- insertar((1,a)) — talla = 1

0	1	2	3	4	5	6	7	8
	(1,a)							

- insertar((2,a)) — talla = 2

0	1	2	3	4	5	6	7	8
	(1,a)	(2,a)						

- insertar((1,b)) — talla = 3

0	1	2	3	4	5	6	7	8
	(1,a)	(2,a)	(1,b)					

Montículo con Datos de Igual Prioridad

- insertar((3,a)) — talla = 4

0	1	2	3	4	5	6	7	8
	(1,a)	(2,a)	(1,b)	(3,a)				

- insertar((4,a)) — talla = 5

0	1	2	3	4	5	6	7	8
	(1,a)	(2,a)	(1,b)	(3,a)	(4,a)			

- insertar((1,c)) — talla = 6

0	1	2	3	4	5	6	7	8
	(1,a)	(2,a)	(1,b)	(3,a)	(4,a)	(1,c)		

- insertar((6,a)) — talla = 7

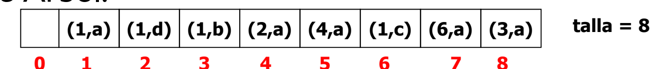
0	1	2	3	4	5	6	7	8
	(1,a)	(2,a)	(1,b)	(3,a)	(4,a)	(1,c)	(6,a)	

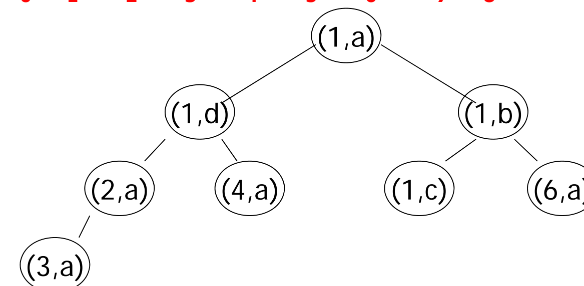
- insertar((1,d)) — talla = 8

0	1	2	3	4	5	6	7	8
	(1,a)	(1,d)	(1,b)	(2,a)	(4,a)	(1,c)	(6,a)	(3,a)

Montículo con Datos de Igual Prioridad

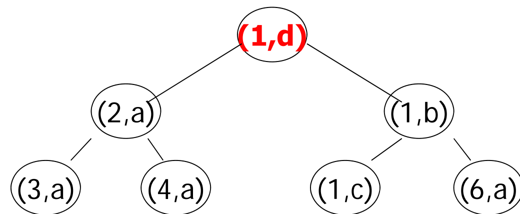
- El Montículo Binario resultante, representado en forma de Árbol:





- eliminarMin() devolverá (1,a) y el Montículo Binario resultante quedará:

Montículo con Datos de Igual Prioridad



- La siguiente llamada a eliminarMin() devolverá (1,d) con lo que ya NO se respeta el orden en el que fueron insertados los datos con prioridad 1.
- Por lo tanto, un Montículo Binario NO se comporta como una Cola para los datos de igual prioridad.

Búsqueda en un Heap Minimal

- La propiedad de ordenación del Heap no es lo suficientemente fuerte para realizar una búsqueda guiada. Podemos realizar una búsqueda secuencial, comenzando por la posición 1.

```
public boolean buscar(E x){
  boolean esta = false; int i = 1;
   while ( i <= talla && !esta){
      if (elArray[i].equals(x)) esta = true;
      else  i++;
     }
   return esta;
  }
```


Búsqueda en un Heap Minimal

- Talla del Problema:
  - Número de elementos del Heap.
- Instancias Significativas:
  - Es un problema de búsqueda. Sí que hay.
  - Caso Mejor: El elemento buscado es el mínimo del conjunto con lo que está situado en la raíz y se encuentra con una comparación.
    - Coste Constante: **Ω(1)**
  - Caso Peor: El elemento buscado NO se encuentra en el Montículo y se deben visitar TODOS los elementos.
    - Coste Lineal con el Número de Elementos: **O(N)**

Búsqueda en un Heap Minimal (II)

- Para este problema es posible realizar una **optimización,** abortando la búsqueda por un subárbol cuando la raíz del subárbol ya es mayor que el elemento a buscar.
  - El coste temporal asintótico NO varía.
- La propiedad de ordenación del Montículo Binario Minimal garantiza que el elemento buscado NO aparecerá en ese subárbol.
- Planteamos un algoritmo recursivo.

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

Búsqueda en un Heap Minimal (II)

```
private boolean buscar(E x, int i){
boolean esta = false;
 if ( i > talla) esta = false;
 else {
    if (elArray[i].equals(x)) esta = true;
    else {
       if ( (2*i <= talla) && elArray[2*i].compareTo(x) <= 0)
                         esta = buscar( x, 2*i);

       if (!esta &&  (2*i + 1 <= talla) &&
              elArray[2*i + 1].compareTo(x) <= 0)
                   esta = buscar(x, 2*i+1);
       }
    }
return esta;
}
```


Búsqueda en un Heap Minimal (II)

- Estilo de implementación alternativo, realizando la comparación de continuación de búsqueda en el caso base:

```
private boolean buscar(E x, int i){
    boolean esta = false;
    if ( i <= talla &&  x.compareTo(elArray[i]) >= 0){
     if (elArray[i].equals(x)) esta = true;
      else {
          esta = buscar(x, 2*i);
          if (!esta) esta = buscar(x, 2*i + 1);
       }
     }
    return esta;
  }
```


Búsqueda en un Heap Minimal (II)

- Talla del Problema:
  - Número de nodos que cuelgan del i-ésimo
- Instancias Significativas:
  - Es un problema de búsqueda. Sí que hay.
  - Caso Mejor: El elemento buscado es el mínimo del conjunto, situado en la raíz, y se encuentra con una comparación. También: el elemento buscado es menor que la raíz del subárbol a visitar.
    - Coste Constante: **Ω(1)**
  - Caso Peor: El elemento buscado es mayor que cualquiera de los del Montículos y se deben visitar TODOS los elementos.
    - Coste Lineal con el Número de Elementos: **O(N)**

Ventajas e Inconvenientes

- Implementación como LEG Ordenada.
  - La inserción en una LEG Ordenada requiere, en promedio, un tiempo **Θ(N)**, dado que hay que buscar la posición adecuada de inserción.
  - Se puede obtener el mínimo con coste constante: **Θ(1)**
  - Admite que los objetos puedan tener cualquier prioridad.
- Implementación como un Array de Colas
  - Únicamente válida cuando hay pocas categorías de prioridad.
  - La inserción tendría un coste constante **Θ(1)** ya que encolamos directamente el elemento en la cola correspondiente.
  - Obtener el mínimo tendría un coste constante **Θ(1)**. Cuando se acaban los elementos de la cola más prioritaria se continúa sacando elementos de las siguientes colas.

Ventajas e Inconvenientes

- Implementación como Montículo Binario:
  - Admite que los objetos puedan tener cualquier prioridad.
  - La inserción se realiza con un coste promedio de **$\Theta((\log_2(N))$**
  - Buscar el mínimo se obtiene con un coste constante (siempre está en la raíz de un Montículo Minimal): **$\Theta(1)$**

Eliminar de Montículo Minimal

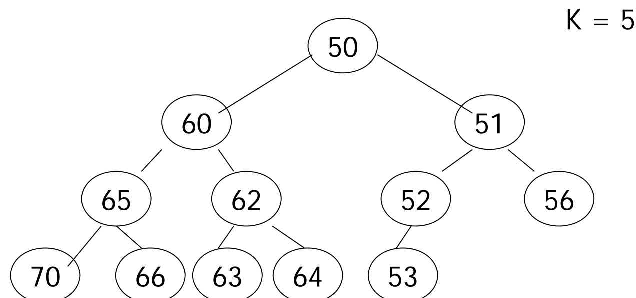
- Solución sencilla pero **poco eficiente:**
- En la posición k del vector ponemos el último elemento y restauramos la propiedad de orden en TODO el montículo.

```
public E borrar(int k) {
   E res = elArray[k];
   elArray[k] = elArray[talla--];
   arreglarMonticulo();
   return res;
}
```

- Coste Temporal: $T_{borrar} \in$ **O(N)**

Eliminar de Montículo Minimal

- Solución **eficiente:**
- Tras el intercambio de valores, suponer que hay un hueco en k. Si array[k] es menor que el padre, reflotar la posición k. Sino, hundirla.



Eliminar de Montículo Minimal

```
public E borrar(int k) {
   E res = elArray[k] ;
   elArray[k] = elArray[talla--];
   int hueco = k;
  E ultimo = elArray[k] ;
   /** reflotar(k) */
   while ( hueco>1 && ultimo.compareTo(elArray[hueco/2]) < 0 ) {
         elArray[hueco] = elArray[hueco/2];
         hueco = hueco / 2;
   }
   elArray[hueco] = ultimo;
   hundir(hueco);
   return res;
}
```

Eliminar de Montículo Minimal

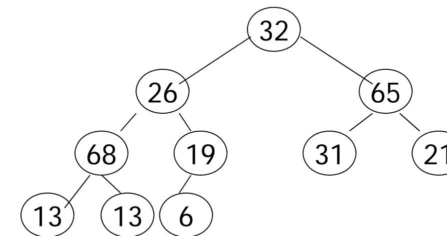
- NOTAS: No hace falta hacer las comparaciones de si debemos reflotar o hundir puesto que internamente los métodos ya la hacen.
- Complejidad Temporal de esta solución:
  - Viene determinada, bien por el proceso de reflotar o por el proceso de hundir.
  - $T_{borrar} \in$ **$O(\log_2(N))$**
- Esta solución es más eficiente que la de restaurar la propiedad de orden en TODO el Montículo Binario.

Inicialización de Montículo Binario

- La solución más eficiente pasa por restaurar la propiedad de orden en el vector en lugar de realizar las N inserciones en el Montículo vacío.

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

talla = 10

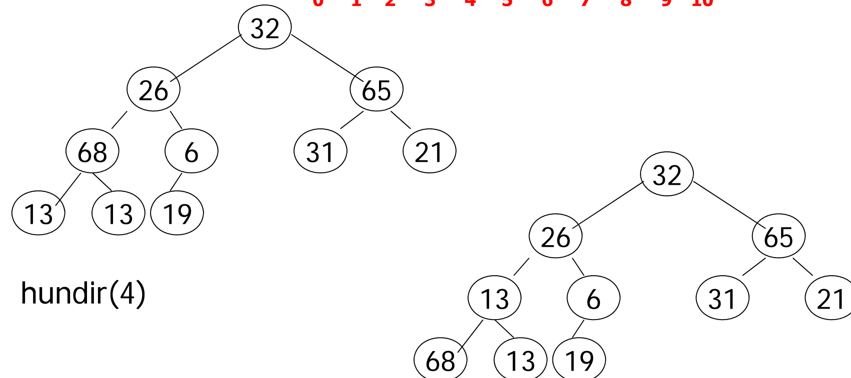


Inicialización de Montículo Binario

- hundir(5)

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

t.a. = 10



- hundir(4)

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

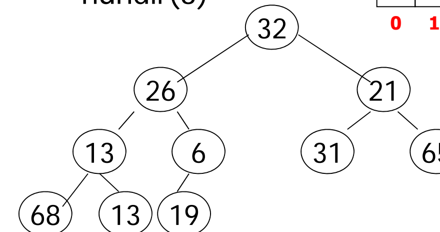
t.a. = 10

Inicialización de Montículo Binario

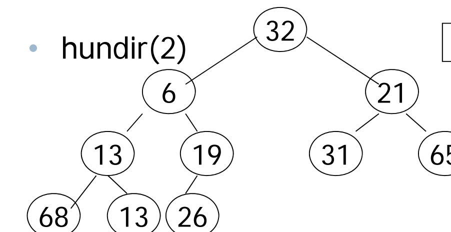
- hundir(3)

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

t.a. = 10



- hundir(2)



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

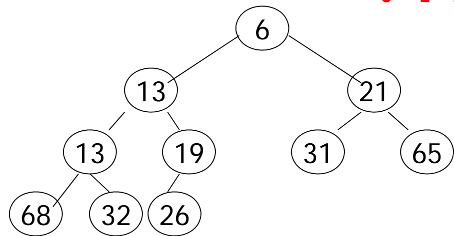
t.a. = 10

Inicialización de Montículo Binario

- hundir(1)

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

t.a. = 10



- Se consigue restaurar la propiedad de orden en el Montículo con un coste **O(N)**
- Si hubiéramos realizado N inserciones en el Montículo hubiéramos incurrido en un coste **$O(N*\log_2(N))$**

Iguales al Mínimo (I)

```
public int igualesAlMinimo()
{
  int res=0;
  if (talla!=0) res=igualesAlMinimo(1);
  return res;
}
```


Iguales al Mínimo (II)

```
protected int igualesAlMinimo(int actual)
{  int res=1; //el actual es igual al mínimo
   if (2*actual <= talla &&
         elArray[2*actual].equals(elArray[actual])
                  res+=igualesAlMinimo(2*actual);
   if (2*actual+1 <= talla &&
         elArray[2*actual+1 ].equals(elArray[actual])
                  res+=igualesAlMinimo(2*actual+1);
   return res;
}
```


Iguales al Mínimo: Coste (I)

- Talla del problema:
  - Tamaño del montículo cuya raíz es el nodo actual.
- Instancias significativas:
  - Mejor caso:
    - El elemento que ocupa el nodo actual no está repetido en el Montículo del que es raíz, por lo que se realizan cero llamadas desde el cuerpo del método.
  - Peor caso:
    - El elemento que ocupa el nodo actual es igual a todos los del Montículo del que es raíz, por lo que cada vez que se invoca al método se producen 2 llamadas recursivas de talla/2 cada una desde su cuerpo hasta alcanzar el caso base de talla 1 (actual es una hoja)

Iguales al Mínimo: Coste (II)

- Ecuaciones de Recurrencia:
  - $T_{igualesAlMinimo}^{M}(x) = k''$
  - $T_{igualesAlMinimo}^{P}(x = 1) = k'$
  - $T_{igualesAlMinimo}^{P}(x > 1) = 2 * T_{igualesAlMinimo}^{P}(x/2) + k$
- Coste Temporal Asintótico:
  - $T_{igualesAlMinimo}(x) \in \Omega(1)$
  - $T_{igualesAlMinimo}(x) \in O(x)$ (por T3 con a > 1)

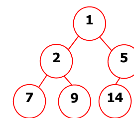
Comprobación de Propiedad de Orden

```
public static <T extends Comparable<T>> boolean
  esHeap(T v[]) {
  for (int i = 2; i < v.length; i++) {
    int padre = i / 2;
    if (v[i].compareTo(v[padre]) < 0) return false;
  }
  return true;
}
```


Resultado Tras Operaciones

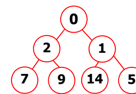
- Situacion inicial:

	1	2	5	7	9	14				
0	1	2	3	4	5	6	7	8	9	10



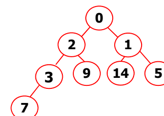
- Tras ejecutar mb.insertar(new Integer(0));

	0	2	1	7	9	14	5			
0	1	2	3	4	5	6	7	8	9	10



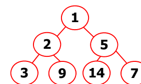
- Tras ejecutar mb.insertar(new Integer(3));

	0	2	1	3	9	14	5	7		
0	1	2	3	4	5	6	7	8	9	10



- Tras ejecutar mb.eliminarMin();

	1	2	5	3	9	14	7	7		
0	1	2	3	4	5	6	7	8	9	10



Sobre el Método Reflotar

- La talla x del problema que resuelve es:
  - x = pos
- Sobre sus instancias significativas, indíquese cuáles de las siguientes afirmaciones son correctas
  - Una instancia del caso mejor es elArray[pos/2] < e
- Relaciones de Recurrencia:
  - Para el peor caso:
    - $T^{P}_{reflotar}(x>1) = 1 * T^{P}_{reflotar}(x/2) + k$;
    - $T^{P}_{reflotar}(x=1) = k'$
  - Para el mejor caso:
    - $T^{M}_{reflotar}(x) = k''$