Colas de prioridad

- Materia Algoritmos y Estructuras de Datos – CB100
- Cátedra Schmidt
- Colas con prioridad - heap – Heapsort

Tipo de Dato Abstracto Cola con Prioridad:

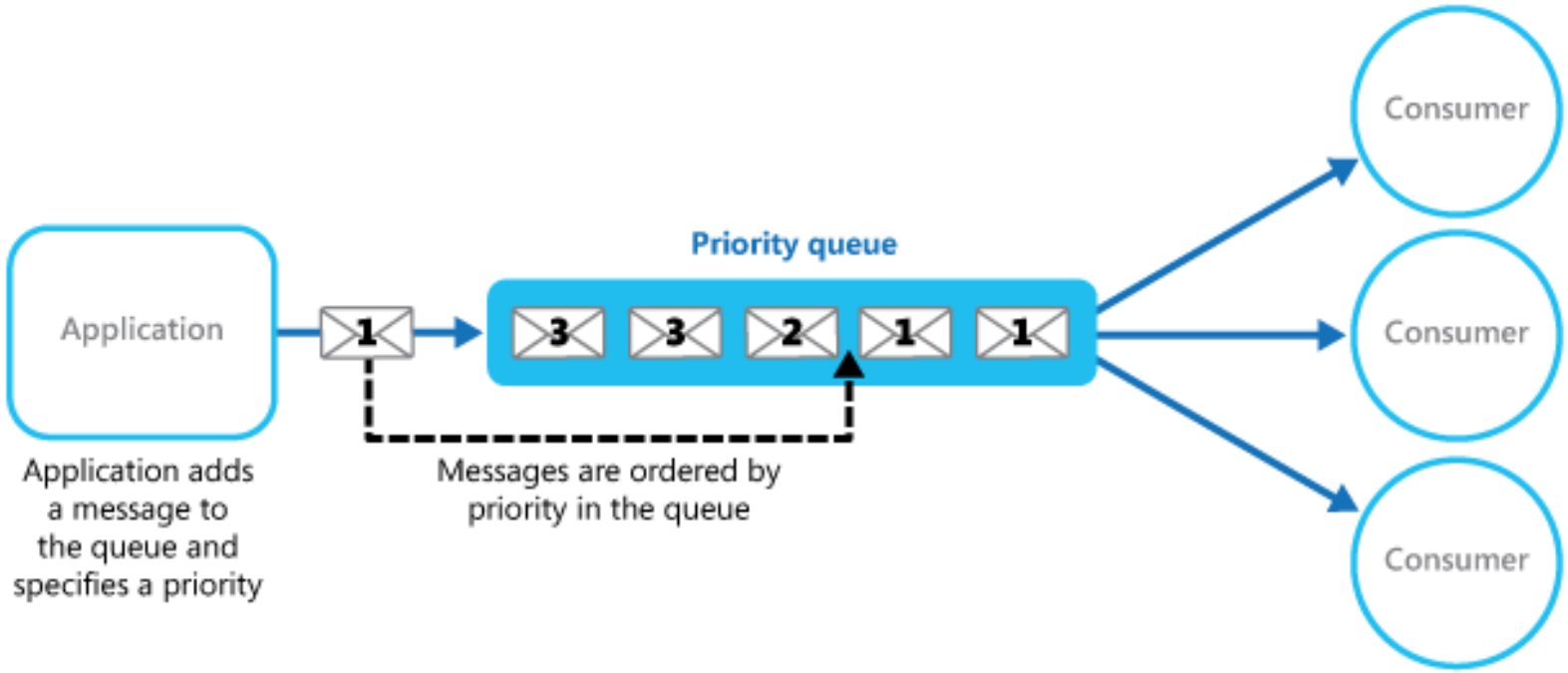
En este TDA (que es un contenedor de elementos del mismo tipo), cada elemento es un par ordenado (clave, prioridad). Las prioridades pueden repetirse.
La primitiva Desacolar de las colas con prioridades funciona como en una cola “común”, eliminando el primer elemento de la cola.
Acolar es diferente de la operación vista en las otras colas. Cada nuevo elemento al ser acolado se ubica delante de aquellos de prioridad inferior, si es que hay elementos con estas características; si no hay elementos de prioridad inferior, se ubicará al final. Es como si se le permitiera a cada elemento que se incorpora a la cola o fila de espera, “colarse” delante de los que son menos importantes.
Lista de primitivas del TDA Cola con Prioridad:

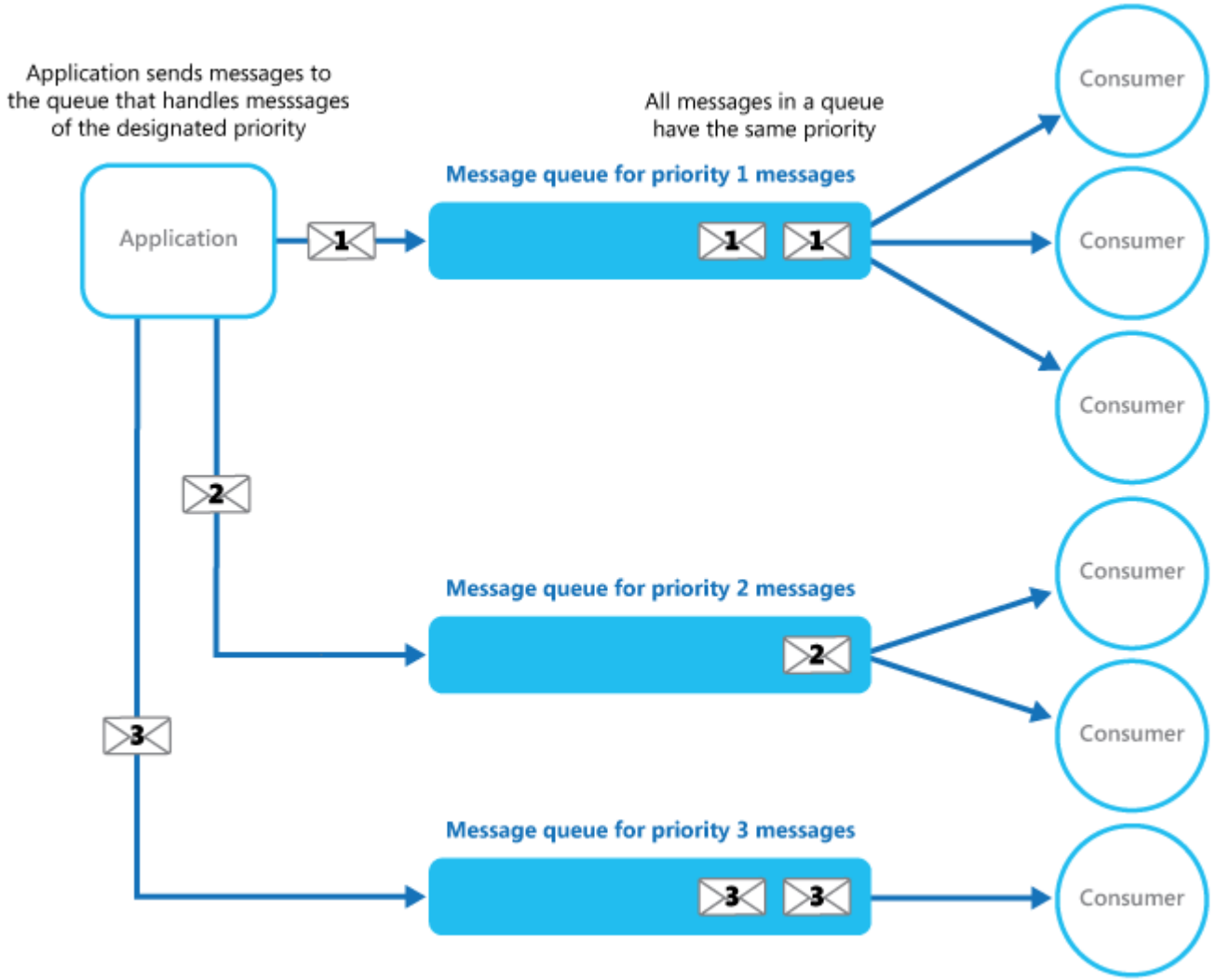
Creación de la Cola con Prioridad:
Precondiciones: ---
Postcondiciones: Cola creada en condiciones de ser usada

Destrucción de la Cola con Prioridad:
Precondiciones: la cola debe existir
Postcondiciones: ----

Alta de un elemento
Precondiciones: la cola debe existir
Postcondiciones: Cola modificada, ahora con un elemento más

Extracción del frente:
Precondiciones:la cola debe existir y no debe estar vacía
Postcondiciones: Cola modificada, ahora con un elemento menos.


Application
1
Priority queue
3
3
2
1
1
Consumer
Consumer
Consumer
Application adds a message to the queue and specifies a priority
Messages are ordered by priority in the queue


Application sends messages to the queue that handles messsages of the designated priority
All messages in a queue have the same priority
Application
Message queue for priority 1 messages
Message queue for priority 2 messages
Message queue for priority 3 messages
Consumer
Consumer
Consumer
Consumer
Consumer
Consumer

Implementaciones posibles:

Una cola con prioridades puede implementarse de múltiples formas: con arrays, con listas ligadas y también con otras estructuras (se deja como ejercicio el planteo de los algoritmos correspondientes a las primitivas indicadas y el análisis de los costes).

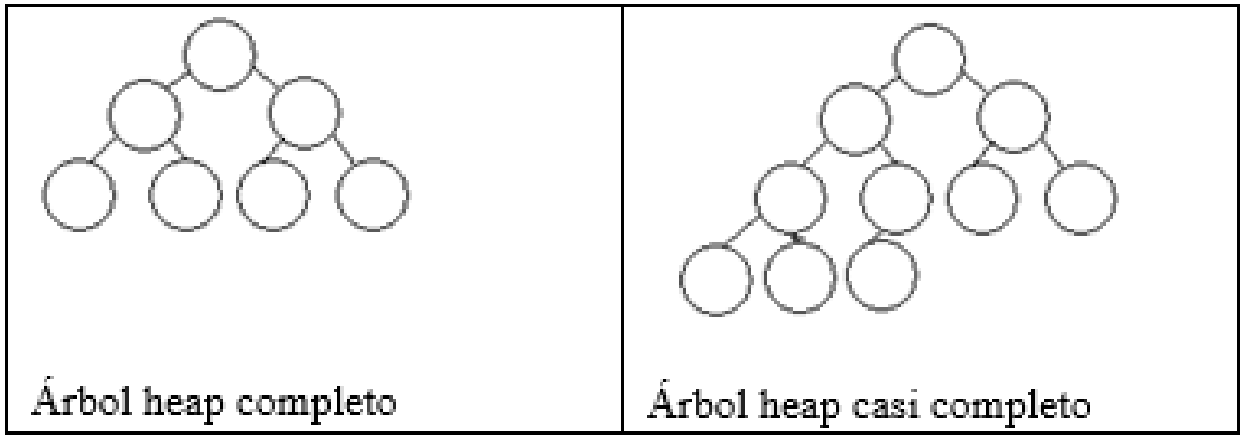
En particular nos interesa la utilización de los árboles heap o montículos para implementar el TDA cola con prioridad, por lo que desarrollaremos ese concepto. (No confundir ”árbol heap” con la zona de memoria dinámica llamada heap).

Árboles heap o montículos:

Un árbol heap conceptualmente es un árbol binario completo o casi completo y parcialmente ordenado.

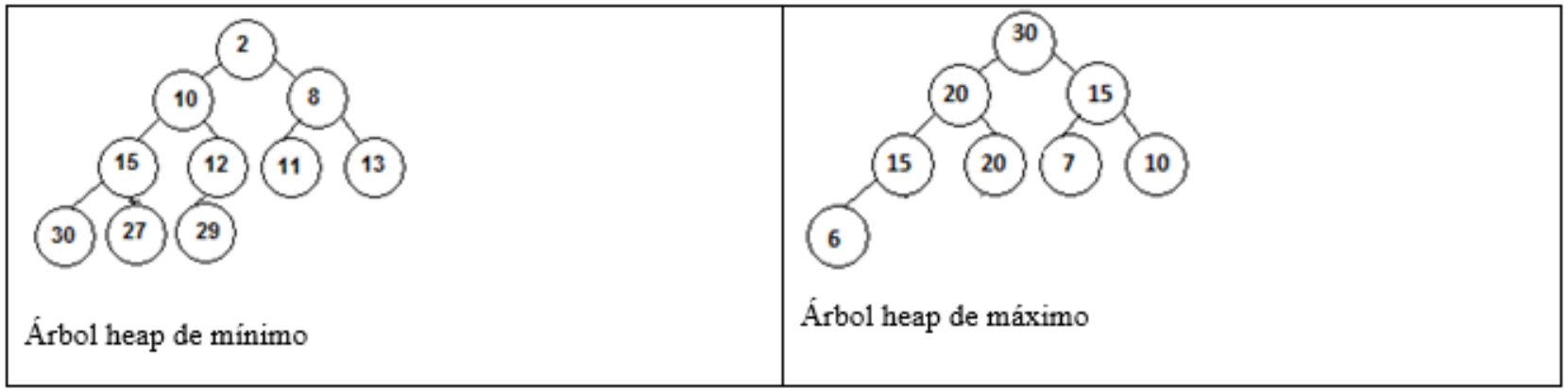
Completo significa que tiene ‘aspecto triangular’, es decir que contiene un nodo en la raíz, dos en el nivel siguiente, y así sucesivamente teniendo todos los niveles ocupados totalmente con una cantidad de nodos que debe ser potencia de 2.

Casi completo significa que tiene todos los niveles menos el de las hojas ‘saturados’; en el nivel de las hojas los nodos existentes deben estar ubicados ‘a izquierda’.
Ejemplo:

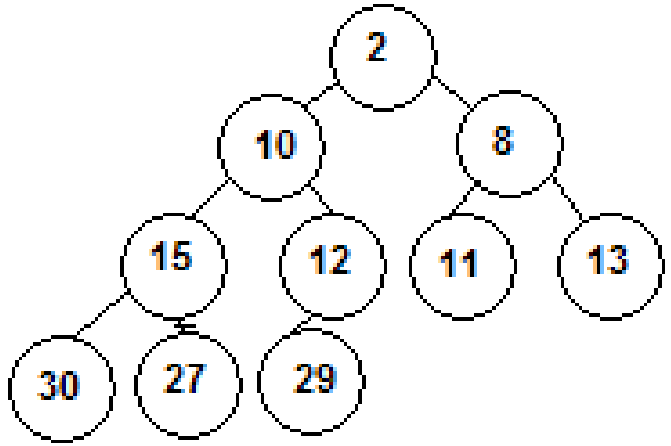


‘Parcialmente ordenado’ significa que para todo nodo se cumple que los valores de sus nodos hijos (si los tuviera) son ambos menores o iguales que el del nodo padre (en este caso el árbol heap es “de máximo”), o bien que para todo nodo se cumple que los valores de sus hijos son ambos mayores o iguales que los de su nodo padre (y se llama árbol heap “de mínimo”). Entre los hijos no se exige ningún orden en particular.

Ejemplo:



Es muy frecuente que un árbol heap se implemente con un array, almacenando los valores por nivel; por ejemplo, para el primer heap es:



2	10	8	15	12	11	13	30	27	29

Considerando que las posiciones del array son 0, 1,..n-1, se verifica que las posiciones de los nodos hijos con respecto a su nodo padre cumplen :

Padre en posición **k** => Hijos en posiciones **2*k+1** y **2*k+2**

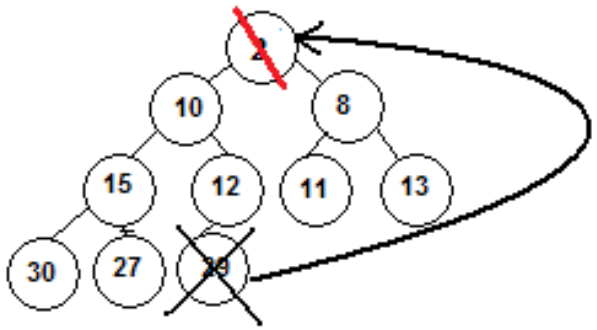
Análogamente, si el hijo está en posición **h**, su padre estará en posición **(h-1)/2 ,** considerando cociente entero.

Un árbol heap o montículo tiene asociados algoritmos básicos para las operaciones básicas que son la baja de la raíz y el alta de un elemento nuevo. Se detallan a continuación.

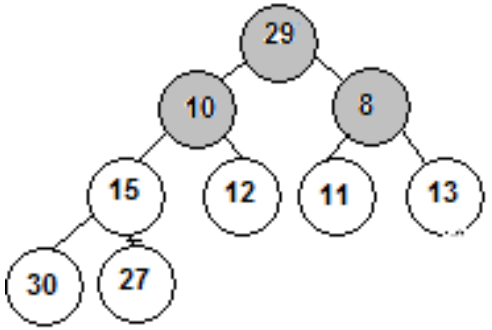
En todos los casos se representarán solamente las prioridades almacenadas.

Baja o remoción de la raíz del heap:

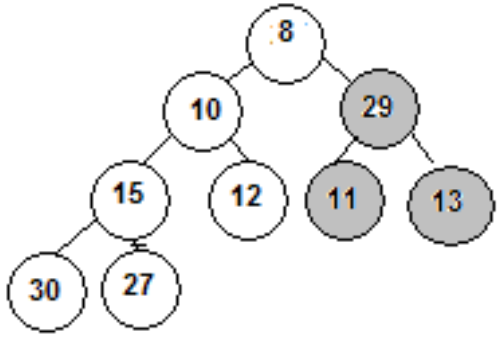
Esta operación se lleva a cabo de la siguiente forma:
Se remueve la raíz. Se reemplaza el nodo removido por la última hoja
Se restaura el heap o montículo hacia abajo, es decir se compara el valor de la nueva raíz con el de su hijo menor (si es un montículo ordenado de esta forma) y se realiza el eventual intercambio, y se prosigue comparando hacia abajo el valor trasladado desde la raíz hasta llegar al nivel de las hojas o hasta ubicar el dato en su posición definitiva.
Ejemplo:
En el montículo del ejemplo anterior eliminamos la raíz y la reemplazamos por la última hoja, reestructurando hacia abajo el árbol. Se muestra la secuencia:



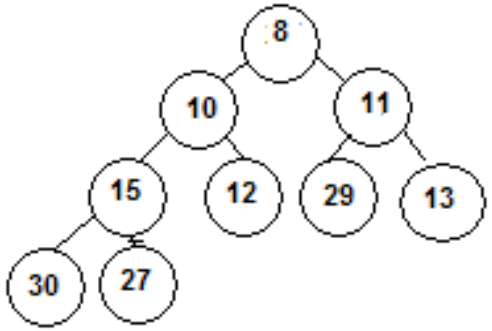
Se reemplaza 2 por 29 y se elimina el nodo hoja con 29.



El menor de los hijos es 8, y es menor que 29.
Se intercambia con él.



Ahora se analiza la terna 29, 11,13. Se intercambia 29 y 13



Árbol heap restaurado.

Coste de la baja de la raíz:

Consideremos una implementación en array. El reemplazo de la clave de la raíz por la última hoja lleva un tiempo constante.

Luego, para restaurar el montículo y ubicar el valor que se ha colocado en la raíz en su posición definitiva se lleva a cabo, a lo sumo para cada par de niveles sucesivos, un par de comparaciones y un intercambio hasta llegar al nivel de las hojas.

Como el árbol está completo o casi completo, su altura es aproximadamente $\log_2 N$, siendo N el número de nodos del heap. Las comparaciones y eventual intercambio tienen coste constante.

Así que, en el peor caso, para restaurar el montículo, el número de ‘pasos’ (entendiendo por paso el par de comparaciones más el eventual intercambio) es aproximadamente $\log_2 N$.

Considerando una implementación en array entonces, se realiza lo siguiente:

2	10	8	15	12	11	13	30	27	29

Situación inicial del heap. N=10

29	10	8	15	12	11	13	30	27	~~29~~

Se reemplaza 2 por 29. N=9

29	10	8	15	12	11	13	30	27	~~29~~

Se compara 29 con sus hijos.

8	10	29	15	12	11	13	30	27	~~29~~

Se intercambia 29 con el menor de sus hijos.

8	10	29	15	12	11	13	30	27	~~29~~

Se analiza ahora la situación de 29 con respecto a sus 'nuevos' hijos'.

8	10	29	15	12	11	13	30	27	~~29~~

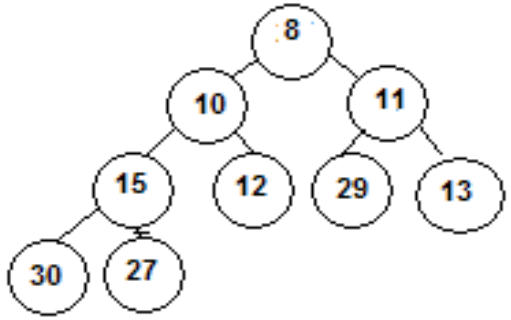
Se intercambia 29 con 11, el menor de sus hijos

8	10	11	15	12	29	13	30	27	~~29~~

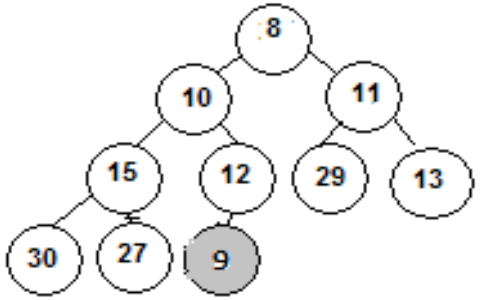
El montículo ha sido restaurado. Ahora N=9

Alta en el heap :

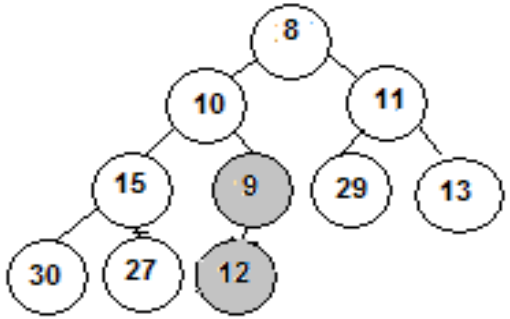
El nuevo elemento se ubica como ‘última hoja’. Luego se restaura el montículo analizando ternas ‘hacia arriba’ hasta ubicar el nuevo elemento en su posición definitiva:
Ejemplo:



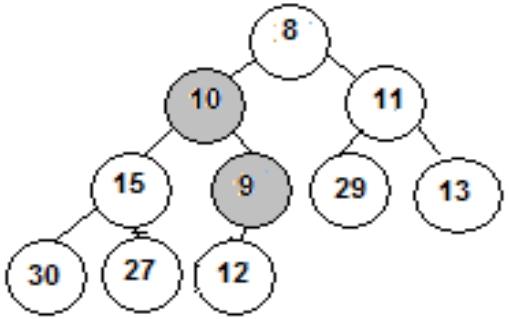
Árbol inicial



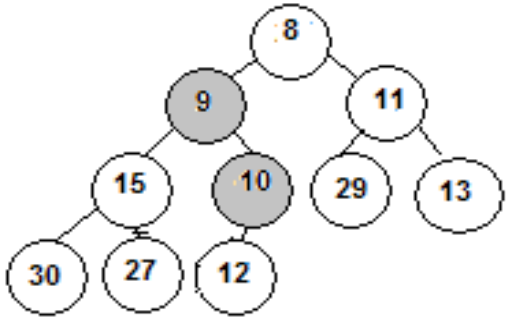
Se agrega 9 al heap como última hoja.



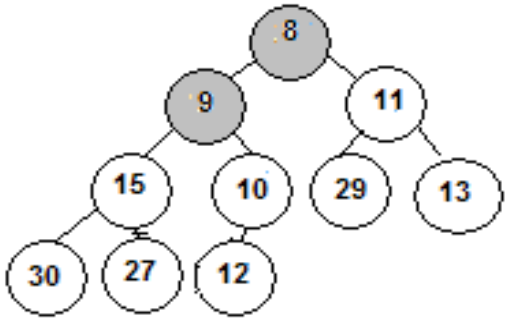
Como 9 es menor que 12, se intercambian entre si.



9 se compara con 10.



9 se intercambia con 10.



9 se compara con 8, pero están ordenados.
El proceso termina.
El heap está restaurado.

Con implementación en un array, es:

8	10	11	15	12	29	13	30	27	9

Se almacena 9 como nueva hoja, es decir, se pone a continuación del último valor almacenado en el array. De esta forma, N, número de elementos del array, pasa de 9 a 10.

8	10	11	15	12	29	13	30	27	9

Se analiza la situación de 9 con respecto a su padre.

8	10	11	15	9	29	13	30	27	12

Como 9 es menor que 12, se intercambian los valores.

8	10	11	15	9	29	13	30	27	12

Ahora se analiza la situación de 9 con respecto a su padre 10

8	9	11	15	10	29	13	30	27	12

Como 9 es menor que 10, se intercambian los valores.

8	9	11	15	10	29	13	30	27	12

Se analiza la situación de 9 con respecto a 8; como se verifica la condición del heap, no se realiza intercambio, y el proceso concluye.

8	9	11	15	10	29	13	30	27	12

El heap o montículo ha sido restaurado.

Coste del alta en el heap:

Considerando una implementación en array, el agregado de una nueva hoja es de coste constante, ya que se agrega un elemento a continuación del último almacenado en el array. Como sucedió en el caso de la extracción de la raíz, al agregar un nuevo elemento, se realizan comparaciones (una en esta operación) y un eventual intercambio desde el nivel de las hojas hasta el nivel de la raíz. En el peor caso, se realizará esto tantas veces como niveles tenga el árbol heap, menos uno.

Es decir que, considerando que la altura del árbol es aproximadamente $\log_2 N$, el coste es O(log N).

Operaciones adicionales en el heap:

Los árboles heaps son utilizados a menudo en el tratamiento de problemas con estrategias “greedy” para organizar el “conjunto de candidatos” que pueden alterar su valor durante el proceso (esto lo veremos en detalle más adelante).

Cuando esto sucede es conveniente considerar una operación más sobre el heap, que habilita una modificación del valor o prioridad de un nodo.

En los problemas en los que esto puede suceder, esa modificación siempre involucra una mejora en prioridad para el nodo, es decir que el nodo en cuestión va a moverse hacia la raíz del árbol, mediante una sucesión de comparaciones y eventuales intercambios con sucesivos nodos “padre”, tal como se hace en un alta, pero quizás considerando un nodo que no esté a nivel hoja sino más arriba.

Primitivas de las colas con prioridad en árboles heap:

El alta de un elemento en el heap permite implementar la primitiva *Acolar* de la cola con prioridades.

La baja de la raíz en el árbol heap permite implementar la primitiva *Desacolar* de la cola con prioridades.

Por lo tanto, el árbol heap es una alternativa eficiente para la implementación de colas con prioridad

Armado del heap sobre un array en coste lineal:

A veces necesitamos construir un heap sobre un array. Esto puede llevarse a cabo con distintos algoritmos, pero mostraremos el más conocido por su eficiencia.

Este algoritmo hace lo siguiente:
Considera el “último nodo” que tiene hijos, y realiza una reestructuración “hacia abajo” del heap (este proceso es conocido como “heapify”). Luego toma el anterior a éste y reestructura también “hacia abajo”. Continúa tomando nodos y reestructurando hacia abajo hasta llegar al nodo raíz, con el cual hace lo mismo.

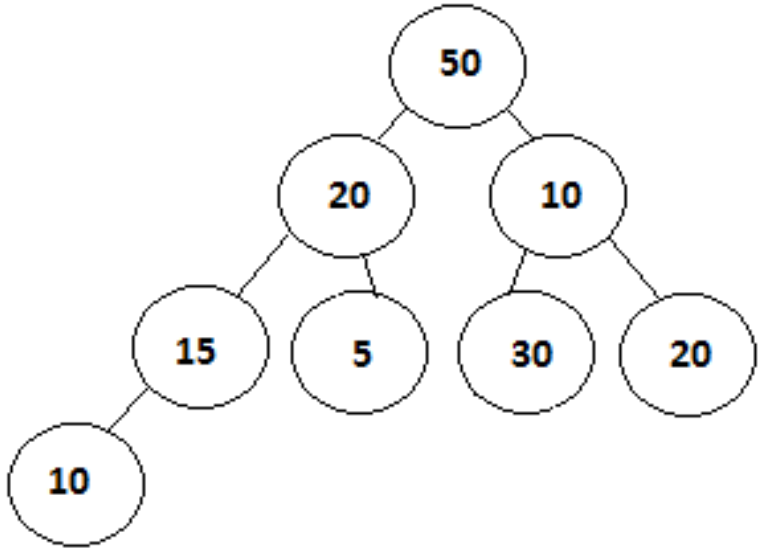
Una versión recursiva del Heapify:

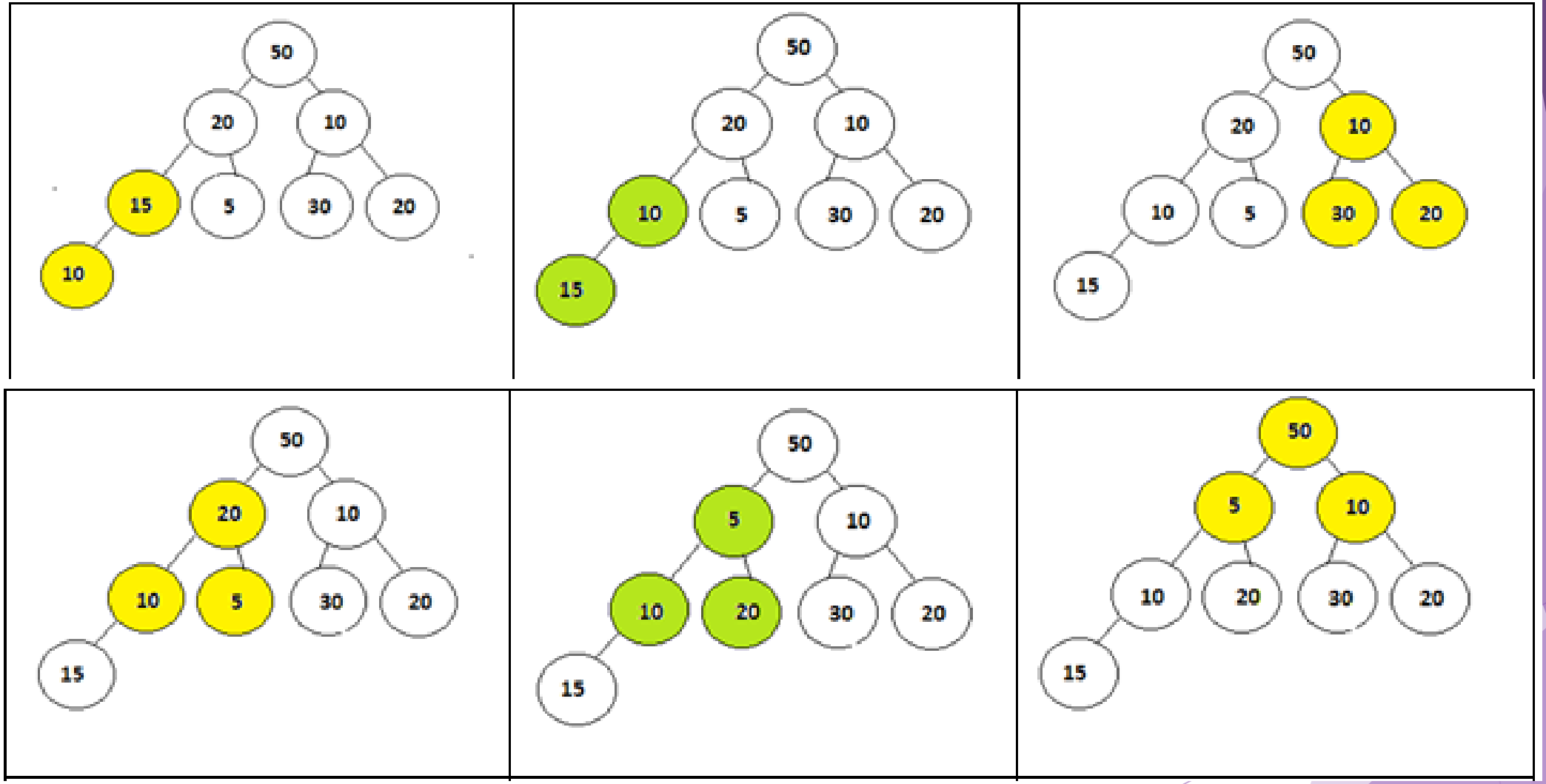
```
//A es el array donde está implementado el heap
//n es el tamaño
//pos es la posición del elemento raíz del subárbol que estamos ordenando

Heapify (A, n, pos)
 {

   t = pos;

  izq = pos * 2 +1;

  der = pos * 2 +2;

 si ((izq < n ) && (A[izq] > A [t]))

     { t = izq ;  }

si ((der > n) && (A[der] < A[t]))

     { t = der ; }

si (t!= pos)

     {
       Swap (A[pos] , A[t]) ;

       Heapify (A, n, t) ;
     }
}
```

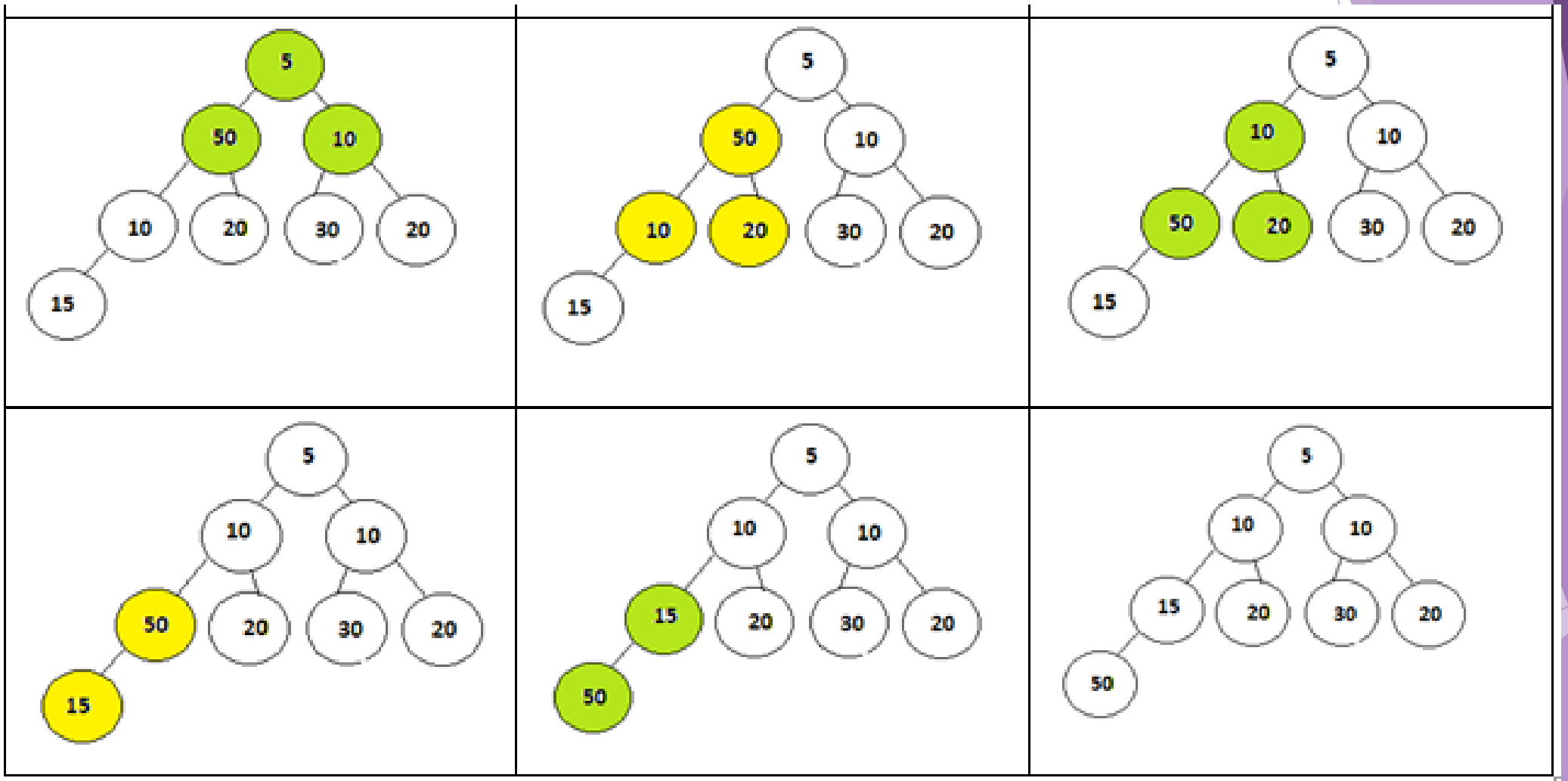
Gráficamente, el proceso de armado de heap aplicando el Heapify desde el último nodo con hijos y recorriendo “hacia atrás” hasta llegar a la raíz, aplicando Heapify en cada nodo, se ejemplifica a continuación.
Consideremos este array

50	20	10	15	5	30	20	10

El cual corresponde a la siguiente representación como árbol:




50
20
10
15
5
30
20
10
50
20
10
10
5
30
20
15
50
20
10
10
5
30
20
15
50
20
10
10
5
30
20
15
50
5
10
10
20
30
20
15
50
5
10
10
20
30
20
15


5
50
10
10
20
30
20
15
5
50
10
10
20
30
20
15
5
10
10
50
20
30
20
15
5
10
10
50
20
30
20
15
5
10
10
15
20
30
20
50
5
10
10
15
20
30
20
50

Como se puede observar, el array queda finalmente así:

5	10	10	15	20	30	20	50

Coste del armado del heap:

Lo que tenemos que considerar es lo siguiente: por tratarse de un árbol completo o casi completo, podemos demostrar que el número de hojas del heap, si n es la cantidad de nodos del mismo, se expresa como techo(n/2).
Entonces, en el peor caso tendremos la mitad de los nodos en el nivel de las hojas, la cuarta parte en el nivel inmediatamente superior, la octava parte en el anterior, y así sucesivamente hasta llegar al nodo raíz, el cual está a una distancia del nivel de las hojas que se expresa de modo logarítmico.
En el proceso de reestructuración "hacia abajo" se comienza por los nodos que tienen hijos, esto es por los que están en el penúltimo nivel. Cada uno de estos nodos puede "bajar" durante el proceso a lo sumo un nivel. Con lo cual tenemos n/4 nodos que pueden descender a un coste de 1, ya que las operaciones involucran comparaciones entre ese nivel y el inferior. Los nodos del siguiente nivel son n/8 y cada uno de ellos puede descender 2 niveles; el coste total involucrado es pues, 2* n/8.
Para el nivel que está por encima, razonando de manera análoga tenemos un coste de 3*n/16.
Así hasta el nodo raíz, para el cual el descenso puede ser de $\log_2 n$ niveles.

Con lo que llegamos a esta expresión del coste del armado del heap:

$$T(n) = \sum_{h=0}^{lg(n)} \frac{n}{2^{h+1}} = n * \sum_{h=0}^{lg(n)} \frac{h}{2^h} = O(n)$$

Nota: Para la demostración, considerar que

$$O(n * \sum_{h=0}^{lg(n)} \frac{h}{2^h}) = O(n * \sum_{h=0}^{\infty} \frac{h}{2^h})$$

Y que $$\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$$

Heapsort:

El árbol heap se utiliza para un método de ordenamiento rápido llamado heapsort.
El método requiere dos etapas.

En la primera construye un heap sobre el array, como vimos en la sección anterior. Este heap debe ser de máximo si se quiere ordenar de forma creciente y de mínimo si se necesita un orden decreciente.

En la segunda etapa, se lleva a cabo el intercambio del primer elemento (posición inicial del heap) con el último, y la reconstrucción del heap en una zona que se disminuye en uno en cada etapa. Esta parte concluye cuando la zona del heap queda reducida a 1.

```
//A es el array a ordenar
// n es el número de elementos

Heapsort (A,  n)
{
   ConstruirHeap (A, n ) // se construye el heap en A sobre las n primeras posiciones

   mientras(n > 1)  // es decir, mientras la zona ocupada por el heap tenga más de 1 posición

  {
    Intercambiar (A, 0, n- 1); // se intercambia el primer elemento con el último de la zona del heap
    n--;                       // la zona del heap disminuye en 1
    Heapify(A, 0, n);          // reconstruir heap desde la posición 0 hasta n (atención: n disminuye en 1)
  }
}
```

Ejemplo:

Este es el array original

3	2	1	9	7	6

Las siguientes son las etapas de la conformación del heap

3	2	6	9	7	1

3	9	6	2	7	1

9	3	6	2	7	1

9	7	6	2	3	1

Ahora ya se tiene un monticulo formado. Se lleva a cabo la segunda parte.

1	7	6	2	3	9
7	1	6	2	3	9
7	3	6	2	1	9
1	3	6	2	7	9
6	3	1	2	7	9
2	3	1	6	7	9
3	2	1	6	7	9
1	2	3	6	7	9
2	1	3	6	7	9
1	2	1	6	7	9

Análisis del coste del Heapsort:

Como hemos dicho, el ordenamiento Heap tiene dos etapas: en la primera se construye un heap sobre el array.
En la segunda etapa, mientras la zona del heap sea mayor que 1, se intercambia el valor de la raíz con la última hoja (si se tiene un heap de máximo, esto manda dicho elemento al final de la zona del heap), se reduce la zona del heap en 1 (con lo cual no se toca la posición del elemento que antes estaba en la raíz) y se restaura el heap desde la primera posición hasta la última de la zona, es decir, se reubica en el lugar correcto la clave que se ha colocado como raíz.

La primera etapa, como hemos visto, puede hacerse en tiempo lineal, esto es tiene coste O(n).

En la segunda etapa, mientras el heap no esté vacío (es decir un número de veces que es O(n)) se realiza un intercambio entre la raíz y la última hoja, se reduce en 1 el tamaño del heap (todo esto de coste constante) y se restaura nuevamente el heap (lo cual puede tener un coste O(log n)). Por tanto esta segunda etapa pertenece a O(n*log n).

Por la regla de la suma, O(n) +O(n*log n) es O(n * log n), que es el coste del Heapsort.

Fin