Laboratorio 4 – HeapSort

Este laboratorio tiene dos objetivos fundamentales:

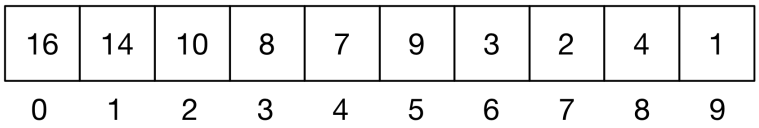
1. Presentar una estructura arborescente denominada Heap.

2. Utilizarla para implementar el algoritmo de ordenación HeapSort.

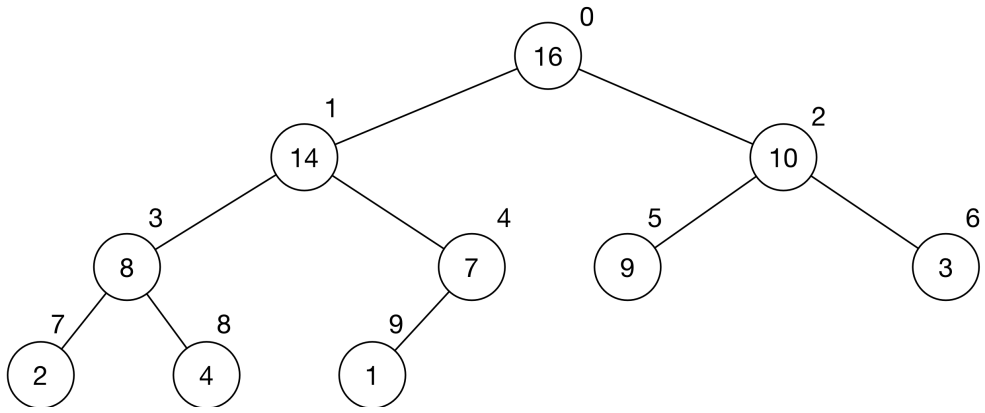
La estructura de datos Heap

La estructura de datos denominada Heap<E> consiste en un array (nosotros usaremos un ArrayList<E>) que podemos ver como un árbol binario casi completo.

Por ejemplo, el ArrayList<Integer> siguiente



puede verse como el siguiente árbol binario



Fijaos en que todos los niveles del árbol están completos, excepto el último (de ahí la calificación de casi completo).

La estructura del árbol viene inducido por los índices del vector, de manera que, dado un índice, podemos calcular el índice correspondiente a su padre, a su hijo izquierdo y a su hijo derecho. Además, también podemos implementar métodos para saber si un nodo tiene padre, hijo izquierdo o hijo derecho.

Los heaps, además de esta visión en forma de árbol sobre un ArrayList<E>, tienen una propiedad adicional: el valor que contiene un nodo siempre es mayor igual al valor de sus hijos (caso de los denominados max-heaps, que

serán los que usaremos en esta práctica; también existe la versión dual denominada min-heaps).

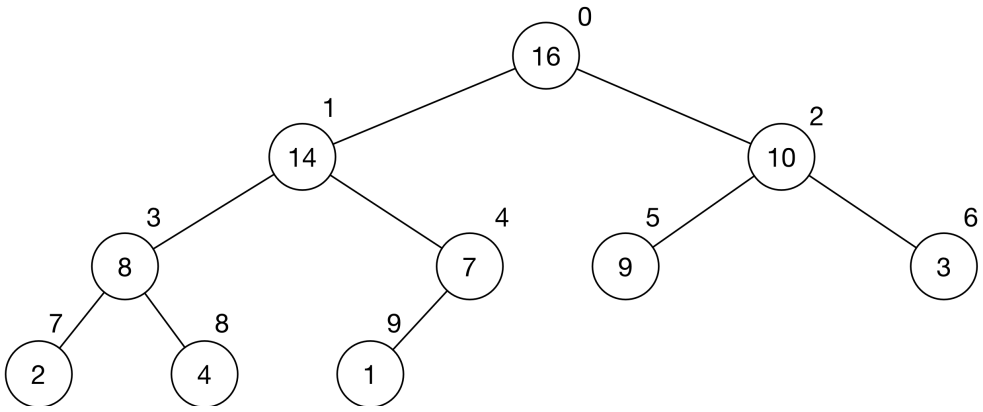
En concreto, en el ejemplo anterior, puede verse que dicha propiedad se cumple.

Es evidente que en un max-heap, la propiedad anterior garantiza que el nodo con valor máximo está en la raíz del árbol y, por tanto, acceder a él se puede realizar en tiempo constante O(1).

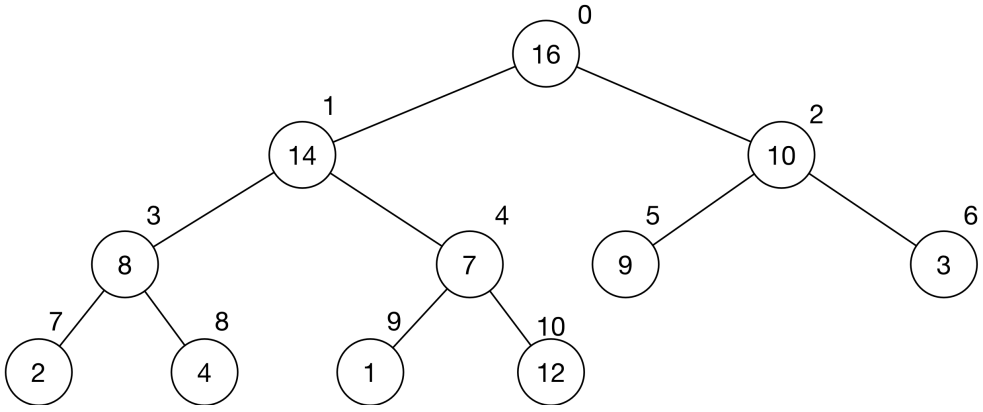
Existen dos operaciones sobre un max-heap que deben ser tenidas en cuenta y que, ayudarán en la implementación del algoritmo HeapSort, aunque no aparezcan explícitamente implementadas en su código. Estas operaciones son las de añadir un elemento al max-heap y eliminar su raíz.

Añadir un elemento

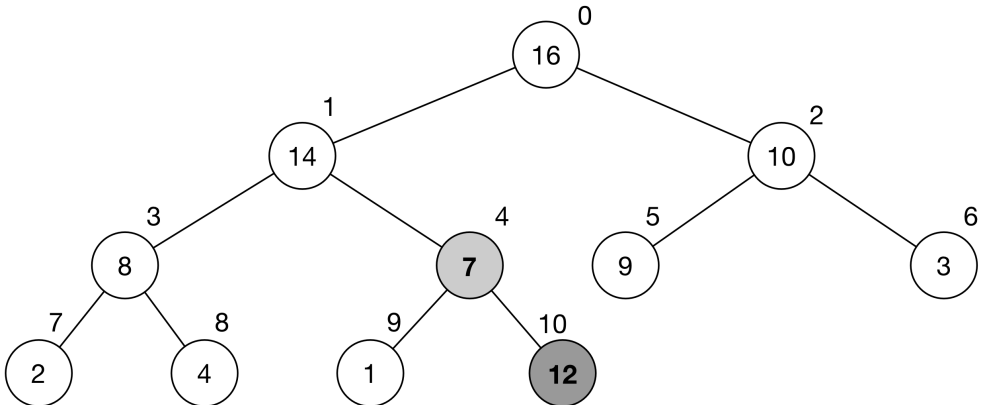
Supongamos que al heap



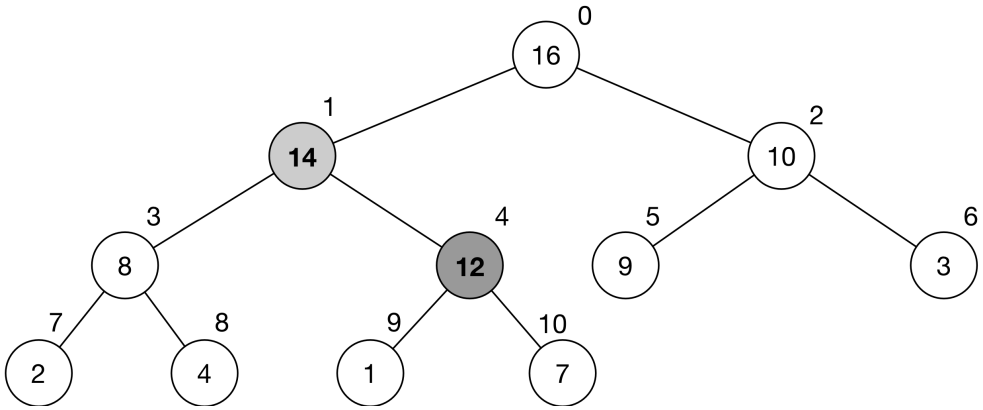
le queremos añadir un elemento de valor 12. Lo añadiríamos al final del ArrayList e iríamos, partiendo de él, intentado ver si está en el lugar que le corresponde o ha de subir.



Lo comparemos con su padre:



y, como es mayor que su padre, para mantener la propiedad de ser un max-heap, lo hemos de intercambiar con él y seguir comprobando hacia arriba:



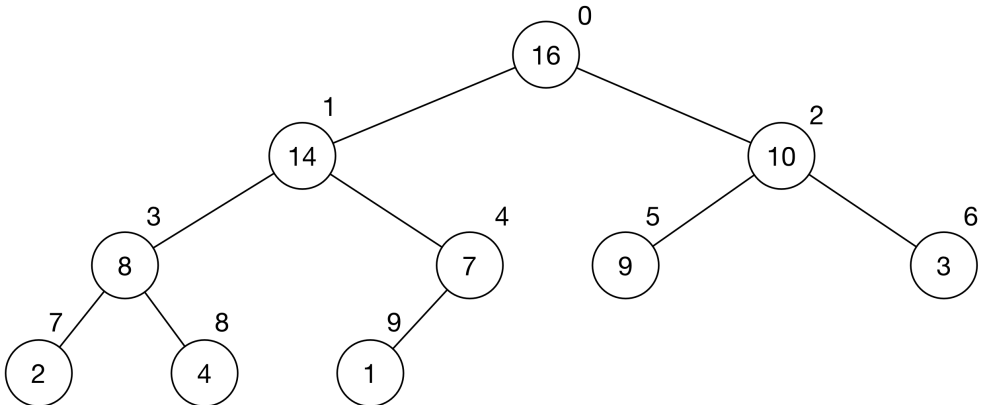
Ahora fijaos en que 12 ya no es mayor que 14 con lo que ya hemos acabado la inserción y tenemos garantizado que el ArrayList vuelve a cumplir la propiedad de ser un max-heap.

En el peor de los casos, recorreremos todo el camino desde una hoja hasta la raíz, que tiene longitud proporcional al logaritmo del número de nodos.

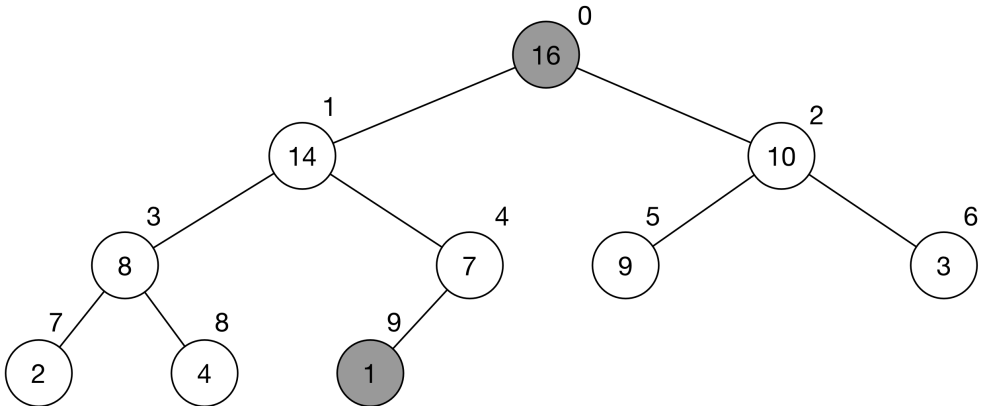
Eliminar la raíz

Vamos ahora a definir la operación que devuelve el mayor elemento (que estará en la raíz del árbol) y lo elimina del mismo, obviamente, manteniendo la estructura de max-heap.

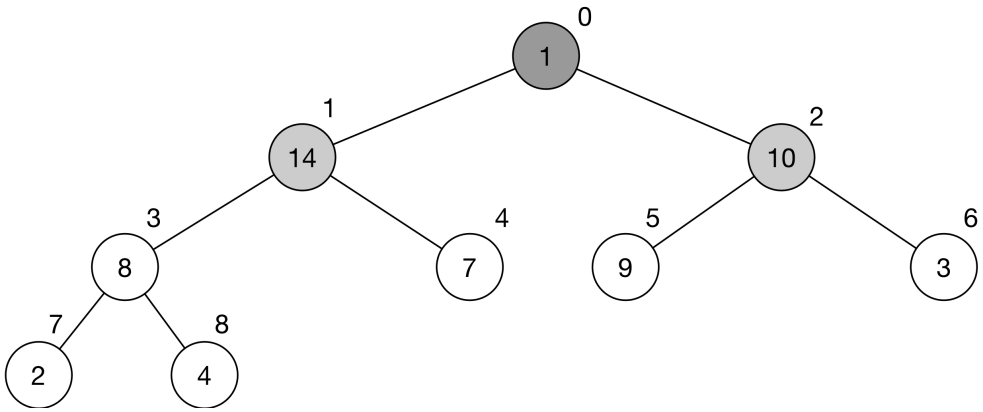
Vamos a ver el caso complicado en el que el heap sí tiene elementos. Por ejemplo, si volvemos a partir del heap



devolveríamos el valor correspondiente al nodo raíz y ahora deberíamos arreglar el max-heap. ¿Cómo lo hacemos? La idea es sustituir la raíz por el único nodo que, si lo quitamos, no nos deja un hueco en el árbol (que ha de ser casi completo): el último nodo.

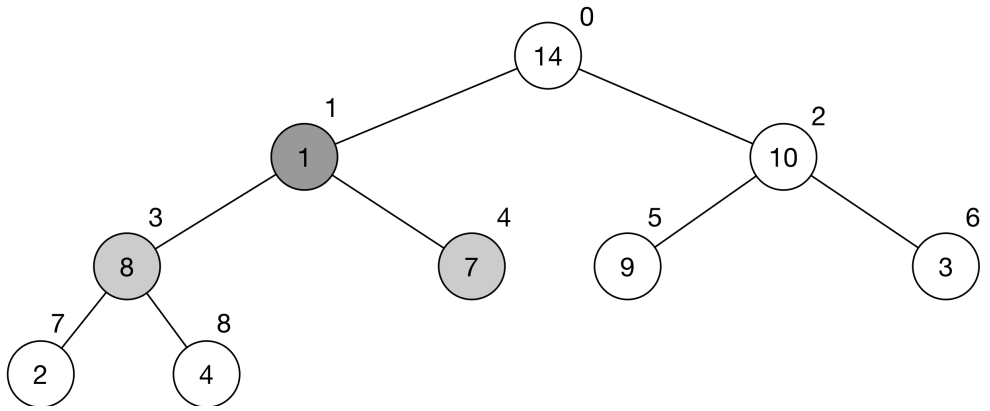


Colocamos el valor correspondiente al último nodo en la raíz, y eliminamos el último elemento del ArrayList. Por tanto, lo que nos queda es:

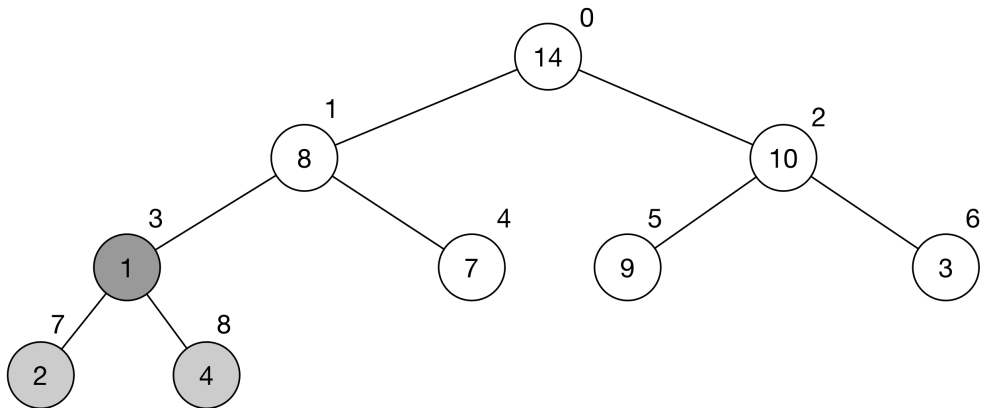


Como podéis observar, el árbol sigue siendo casi completo, pero se ha perdido la propiedad de ser un max-heap. ¿Cómo podemos comprobar eso? Partimos del nodo raíz, y lo comparamos con sus hijos existentes y buscamos el que sea mayor de todos. En nuestro caso el mayor entre 1, 14 y 10, es 14.

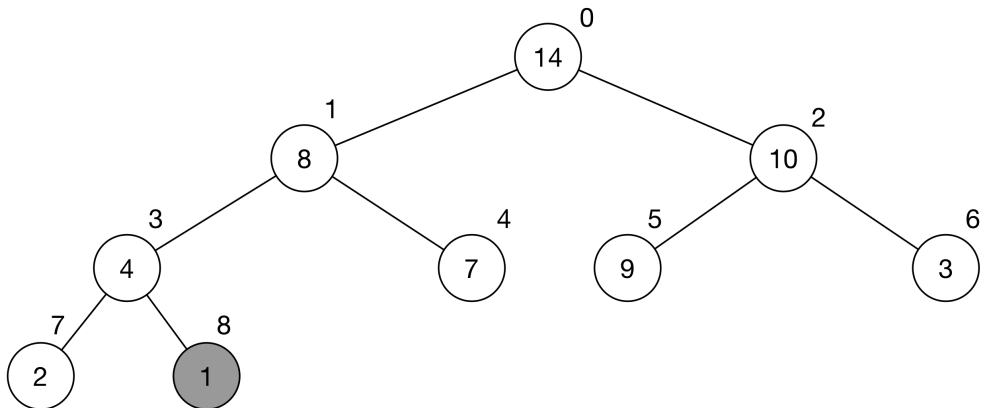
Intercambiamos el nodo raíz por este mayor y seguimos arreglando por el subárbol que hemos modificado:



Ahora, hemos de escoger el mayor entre 1, 8 y 7, que es 8. Por lo que intercambiamos el 1 y el 8, quedando de la siguiente manera:



Ahora se ha de escoger el máximo entre 1, 2 y 4, que es 4, por lo que queda:



El nodo ya no tiene hijos, por lo que no hay que arreglar nada y ya tenemos garantizado que se trata de nuevo de un max-heap.

Nuevamente sucede que, en el peor de los casos, el número de nodos a arreglar como máximo, es proporcional al logaritmo del número total de nodos.

Obviamente, si en un paso del camino no se ha tenido que hacer ninguna modificación, ya se puede dar por acabado el proceso de extracción, ya que el árbol ya estará arreglado.

Implementación del algoritmo HeapSort

El algoritmo de ordenación HeapSort se caracteriza por utilizar como técnica de diseño, el uso de una estructura de datos para gestionar la información, concretamente (tal y como su nombre indica) un max-heap, como el que se ha descrito previamente.

¿Cómo funciona el algoritmo? Una vez recibe, en nuestro caso, un *ArrayList<E>* con los elementos a ordenar, el algoritmo se basa en la realización de dos pasos.

La idea es considerar que el ArrayList<E> está dividido en dos partes:

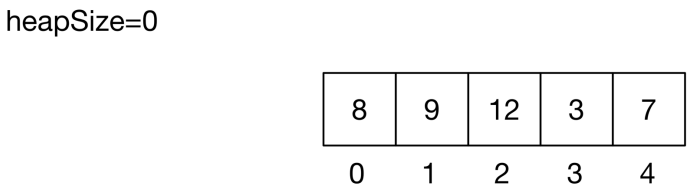
- Un prefijo que contiene elementos organizados como un max-heap
- Un sufijo que contiene el resto de elementos del ArrayList<E>

En un primer paso se irá insertando el primer elemento del sufijo en el max-heap, hasta que ya no queden nuevos elementos a insertar.

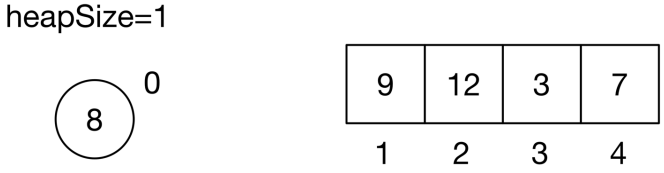
El siguiente paso consistirá en eliminar la raíz del max-heap (que será su elemento máximo) e insertarlo como nuevo primer elemento del sufijo. De esta manera, al final, tendremos los elementos del vector ordenado.

Veamos un ejemplo sobre un ArrayList<Integer> de 5 posiciones (para simplificar los diagramas lo mostramos como si fuera un array).

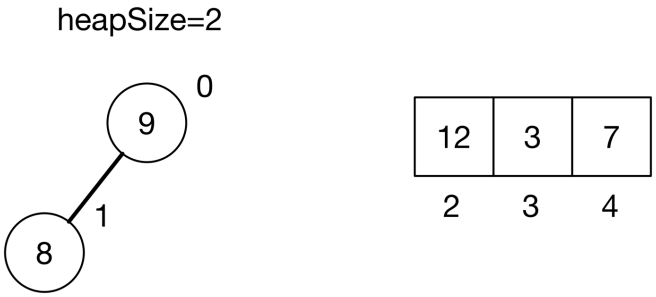
Inicialmente, el max-heap está vacío (tiene 0 elementos) y todos los elementos están en el sufijo:



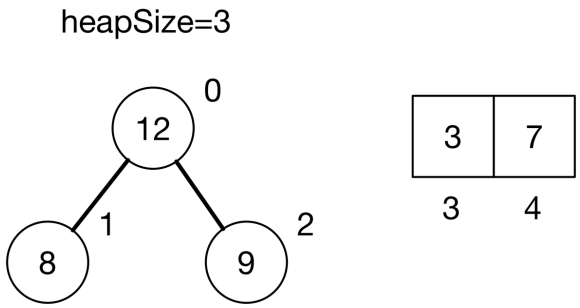
Insertamos el primer elemento del sufijo, el 8, en el max-heap, es decir:



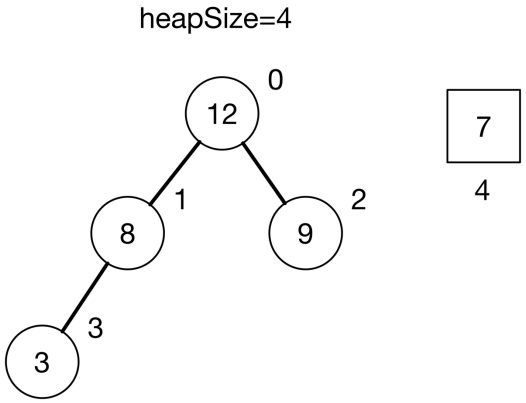
Hacemos lo mismo para el 9, que es el primer elemento del sufijo (ya que el heap ahora tiene un elemento) y nos queda:



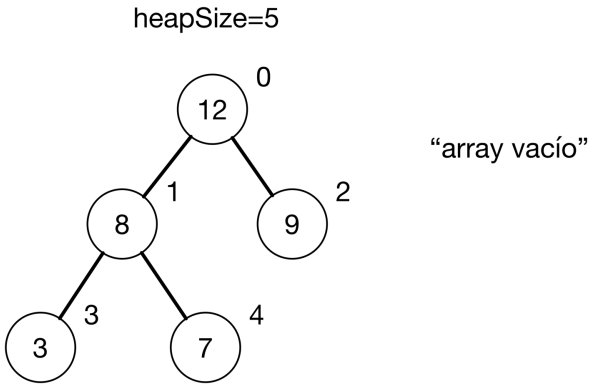
Añadimos el 12 al max-heap:



Ahora el 3:

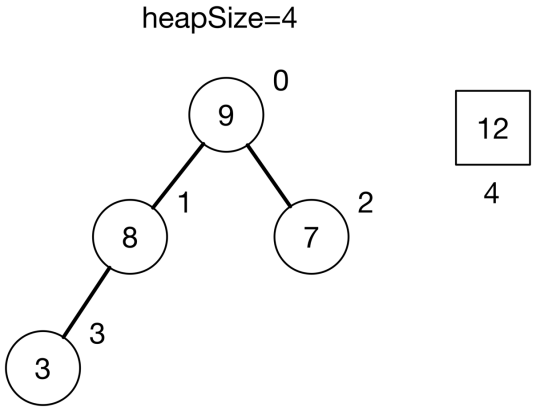


Y, finalmente el 7:

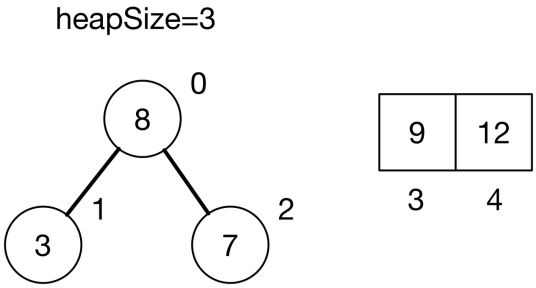


Ahora empieza el segundo paso que consiste en ir desmontando el max-heap, pasando la raíz como el primer elemento del sufijo.

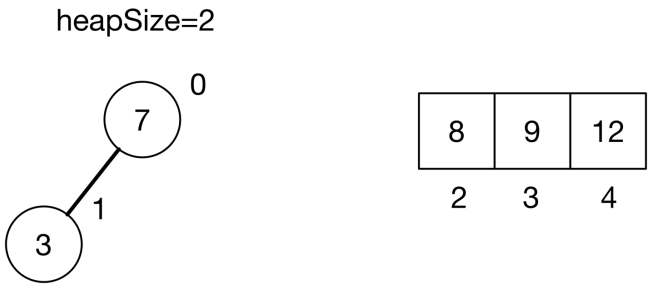
Comenzamos por la raíz, el 12, que es el máximo elemento y que, por tanto, será el último de la lista ordenada, es decir:



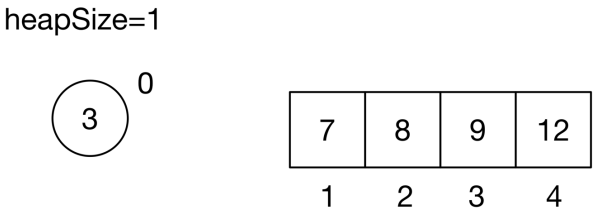
Ahora eliminamos de nuevo la raíz, el 9, que se añadirá como el primer elemento del sufijo, es decir:



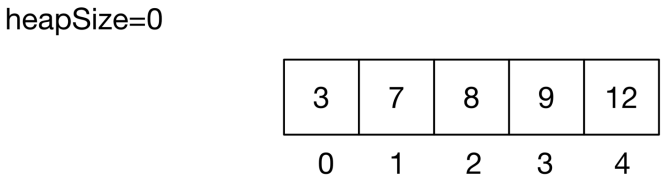
Hacemos lo propio con la nueva raíz, es decir, el 8:



Y ahora con el 7:



Finalmente, el 3, generando la estructura que podemos considerar como:



Y que, por tanto, contiene el ArrayList<Integer> ordenado.

Un esbozo de la clase a implementar sería:

```
public class HeapSort {

    private static class Heap<E> {

        private final ArrayList<E> elements;
        private final Comparator<? super E> comparator;

        private int heapSize = 0;

        ¿?

        private static int parent(int index) { … }
        private static int left(int index) { … }
        private static int right(int index) { … }

        private boolean hasLeft(int index) { … }
        private boolean hasRight(int index) { … }
        private boolesn hasParent(int index) { … }
    }

    public static <E> void sort(
            ArrayList<E> list,
            Comparator<? super E> cmp){

        ¿?
    }

    public static <E extends Comparable<? super E>> void sort(
            ArrayList<E> list) {

        ¿?
    }
}
```

Otras consideraciones

No olvidéis compilar usando la opción –Xlint:unchecked para que el compilador os avise de errores en el uso de genéricos.

Entrega

El resultado obtenido debe ser entregado por medio de un proyecto IntelliJ por cada grupo, comprimido y con el nombre "Lab4_NombreIntegrantes".

Además, se debe entregar un documento de texto, máximo de cuatro páginas, en el cual se explique el funcionamiento de la implementación realizada.

Consideraciones de Evaluación

A la hora de evaluar el laboratorio se tendrán en cuenta los siguientes aspectos:

- El código ofrece una solución al problema planteado.
- Realización de test con JUnit 4.0.
- Explicación adecuada del trabajo realizado, es decir, esfuerzo aplicado al documento de texto solicitado.
- Calidad y limpieza del código.