Laboratorio 4 – HeapSort

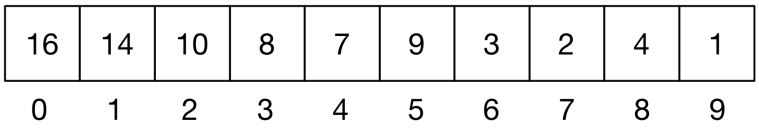
Este laboratorio tiene dos objetivos fundamentales:

1. Presentar una estructura arborescente denominada BinaryHeap<E>.
2. Utilizarla para implementar el algoritmo de ordenación HeapSort.

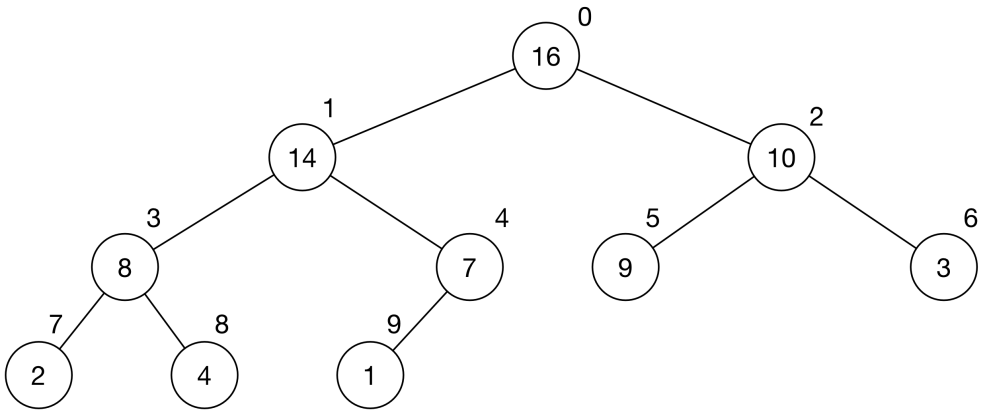
La estructura de datos BinaryHeap

La estructura de datos denominada BinaryHeap<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 inducida por los índices[1] del vector (mostrados como exponentes en los nodos del árbol), de manera que, dado un índice,

[1] Si implementamos el BinaryHeap como una estructura de datos independiente, se suele colocar la raíz en la posición 1 en vez de la 0. Como la práctica va enfocada al algoritmo de HeapSort donde el heap es, de alguna manera, virtual (ya que se trabaja sobre la lista a ordenar), la raíz del heap ocupará la posición 0 del ArrayList<E>.

podemos saber si se trata de la raíz, calcular el índice correspondiente a su padre, a su hijo izquierdo y a su hijo derecho; y, además, también podemos determinar 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.

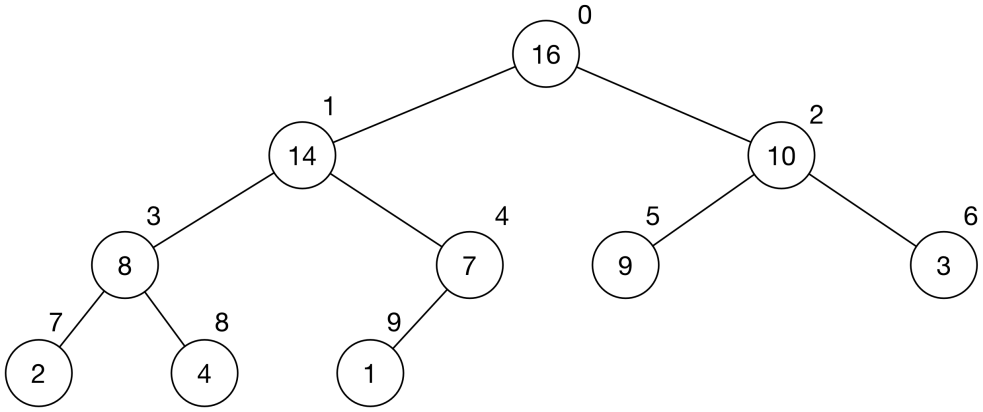
Es evidente que **en un max-heap**, la propiedad anterior garantiza que **el nodo con valor máximo está en la raíz del árbol** y, por tanto, acceder a él se puede realizar en tiempo constante O(1).

Existen dos operaciones fundamentales sobre un max-heap y que, aunque no se implementen explícitamente, serán la base de la implementación del algoritmo HeapSort. Estas operaciones son las de:

- Añadir un elemento al max-heap.
- Eliminar su raíz (el elemento máximo actual).

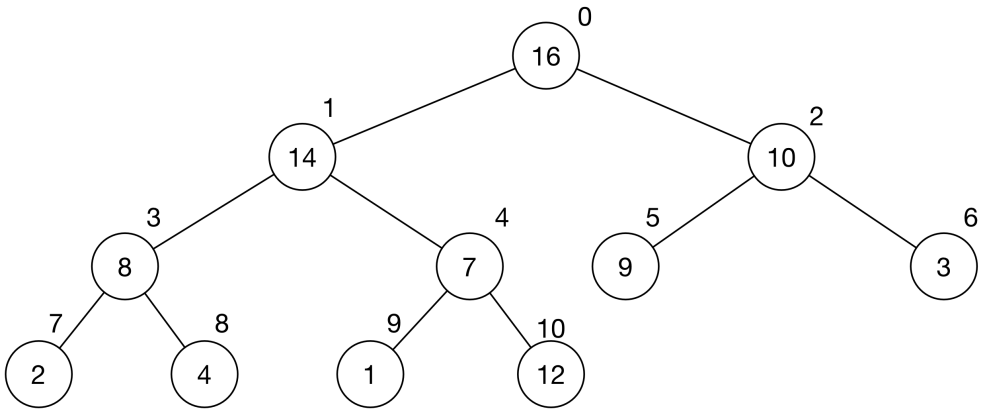
Añadir un elemento

Supongamos que al heap

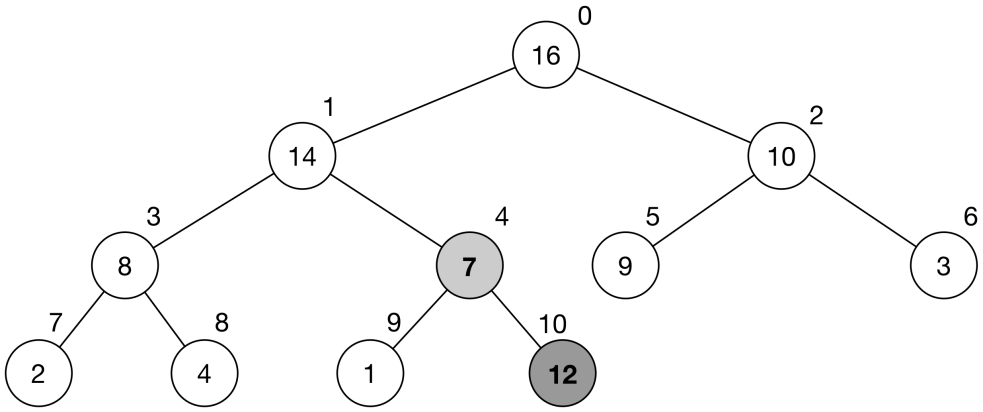


le queremos añadir un elemento de valor 12. Lo añadiríamos al final del ArrayList, ocupando por tanto la posición 10 del mismo, e iríamos, partiendo de él, comprobando si está en el lugar que le corresponde o ha de subir.

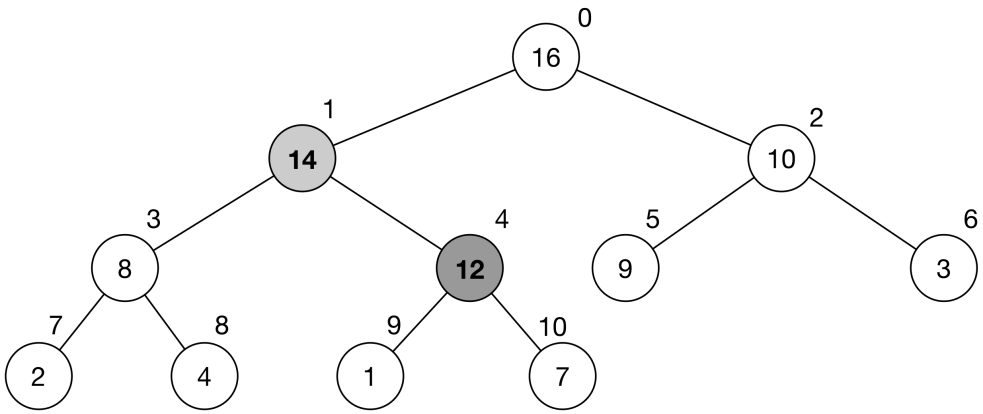
Fijaos en que al haberlo añadido al final del ArrayList, de forma trivial, se mantiene la propiedad de ser un árbol casi completo.



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:



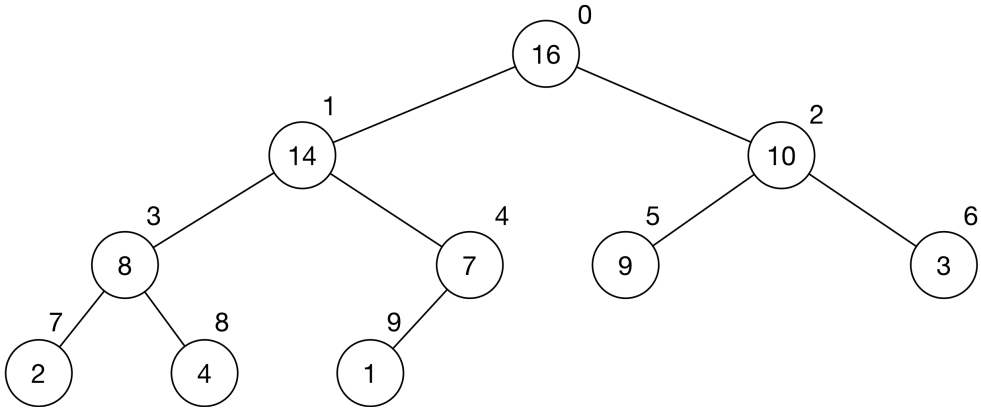
Ahora fijaos en que 12 ya no es mayor que 14 con lo que ya hemos acabado la inserción y tenemos garantizado que el ArrayList vuelve a cumplir la propiedad de ser un max-heap.

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.

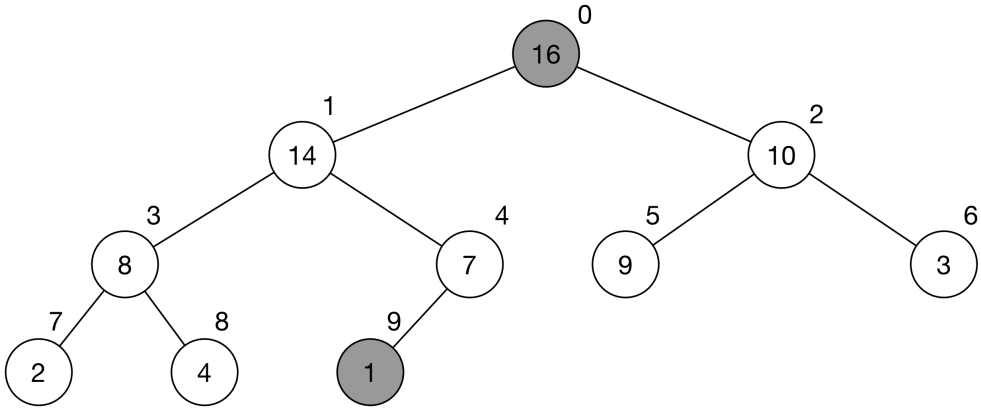
Eliminar la raíz

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.

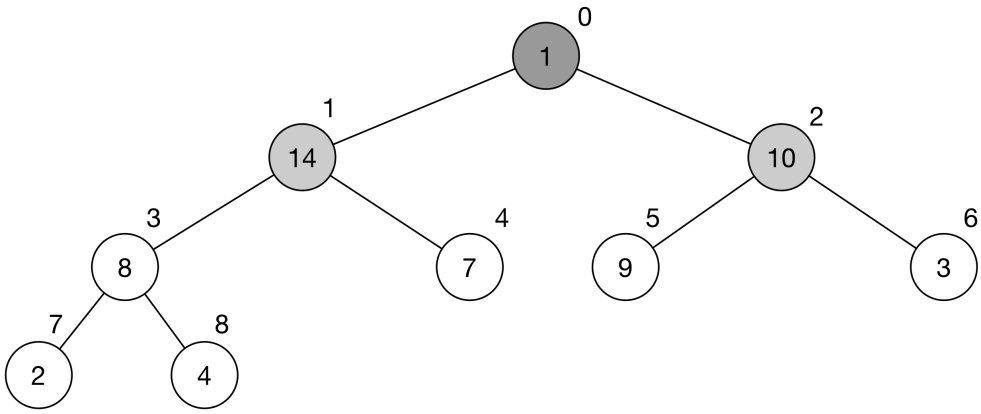
Vamos a ver el caso complicado en el que el heap sí tiene elementos. Por ejemplo, si volvemos a partir del heap



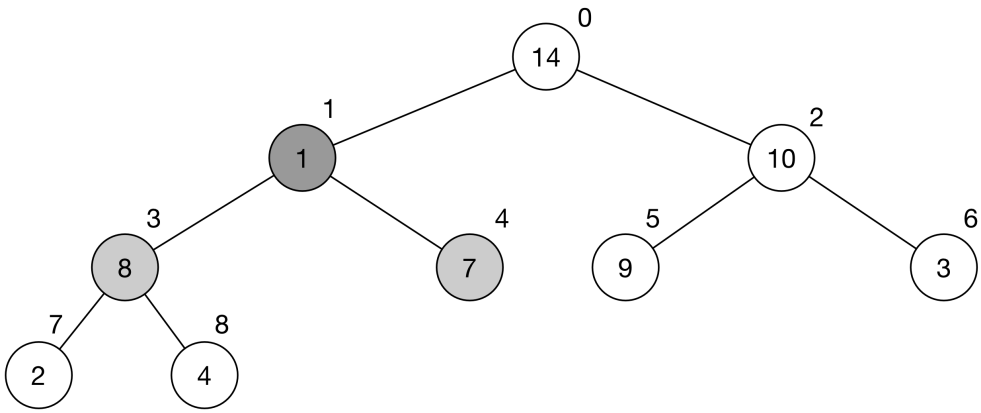
devolveríamos el valor correspondiente al nodo raíz y ahora deberíamos arreglar el max-heap. ¿Cómo lo hacemos? La idea es sustituir la raíz por el único nodo que, si lo quitamos, no nos deja un hueco en el árbol (que ha de ser casi completo): el último nodo.



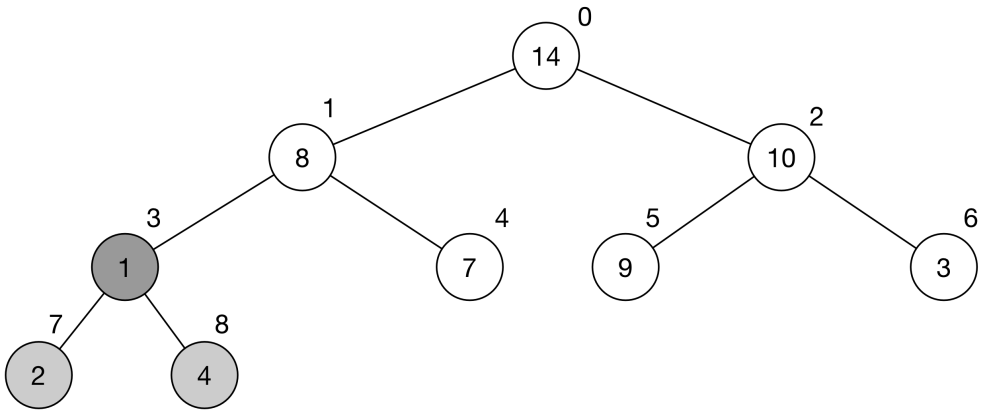
Colocamos el valor correspondiente al último nodo en la raíz, y eliminamos el último elemento del ArrayList. Por tanto, lo que nos queda es:



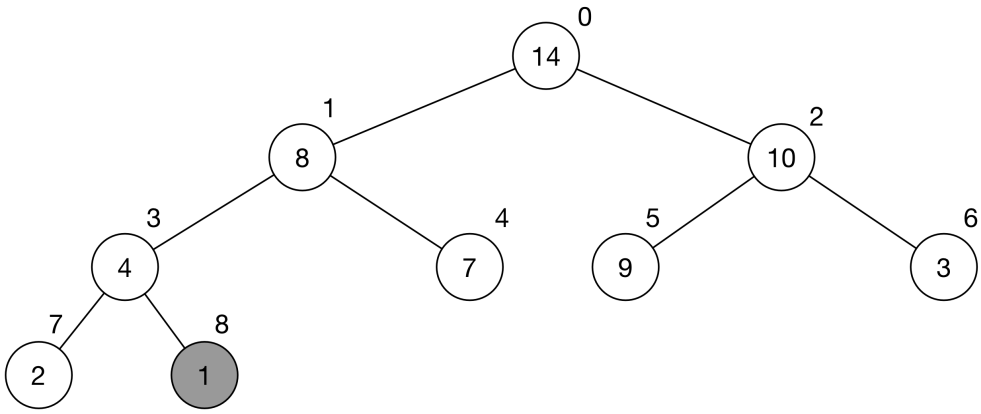
Como podéis observar, el árbol sigue siendo casi completo, pero se ha perdido la propiedad de ser un max-heap. ¿Cómo podemos comprobar eso? Partimos del nodo raíz, y lo comparamos con sus hijos existentes y buscamos el que sea mayor de todos. En nuestro caso el mayor entre 1, 14 y 10, es 14. Intercambiamos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

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

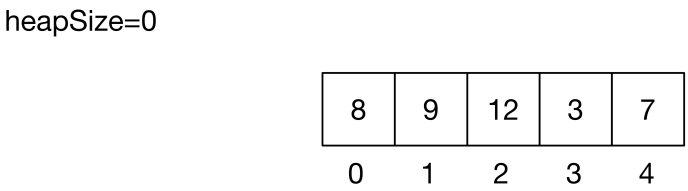
Y, el algoritmo realiza la ordenación en dos fases:

- En la **primera fase**, se irán insertando los sucesivos primeros elementos del sufijo en el max-heap, hasta que ya no queden nuevos elementos a insertar. Fijaos en que, de esta manera, el prefijo (max-heap) irá creciendo (hasta ocupar todo el ArrayList<E>) y el sufijo decreciendo (hasta hacerse vacío).

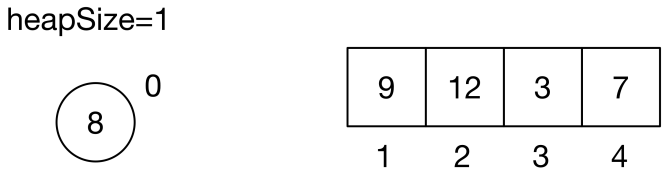
- En la **segunda fase**, se irán eliminando los sucesivos máximos del max-heap para ser insertados como primeros elementos del sufijo. En esta fase, como es fácil ver, el prefijo (max-heap) irá decreciendo (hasta hacerse vacío) y el sufijo acabará ocupando todo el ArrayList<E> que, al final, contendrá todos los elementos y estará ordenado.

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

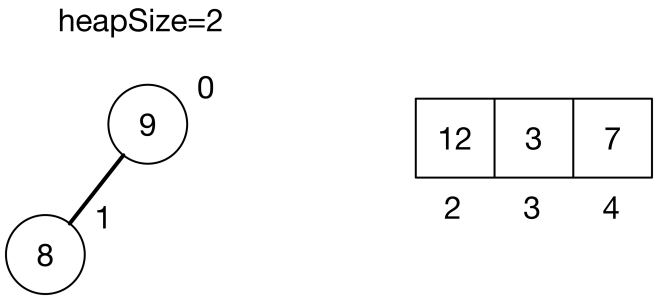
Inicialmente, el max-heap está vacío (tiene 0 elementos) y todos los elementos están en el sufijo:



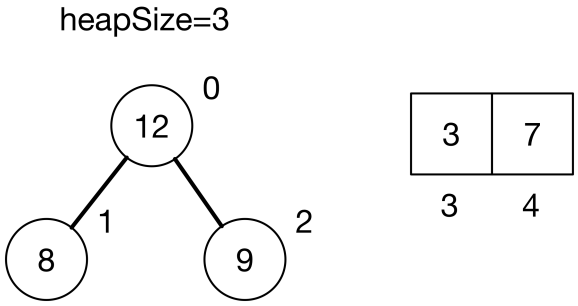
Insertamos el primer elemento del sufijo, el 8, en el max-heap, es decir:



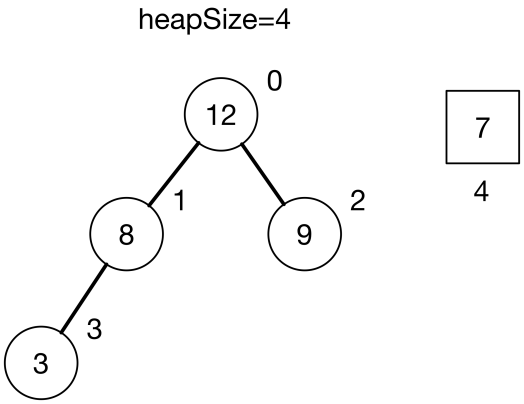
Hacemos lo mismo para el 9, que es el primer elemento del sufijo (ya que el heap ahora tiene un elemento) y nos queda:



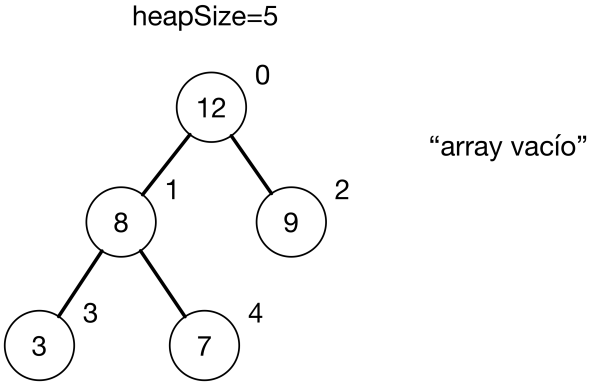
Añadimos el 12 al max-heap:



Ahora el 3:

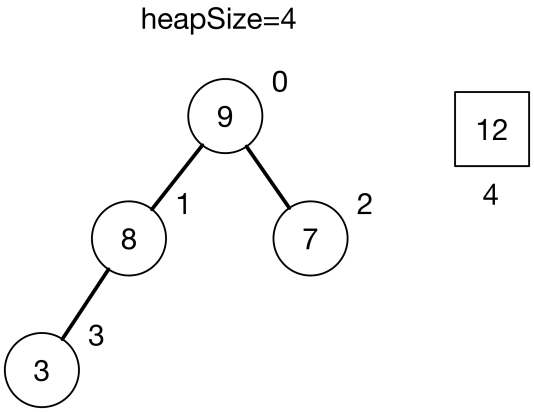


Y, finalmente el 7:

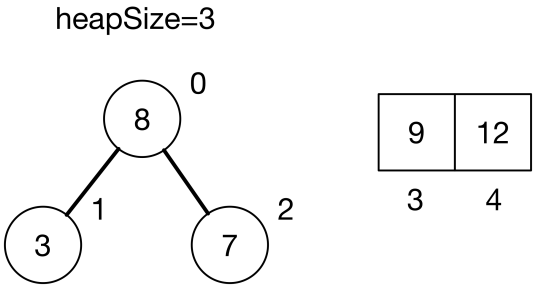


Ahora empieza la segunda fase que consiste en ir desmontando el max-heap, convirtiendo la raíz en el primer elemento del sufijo.

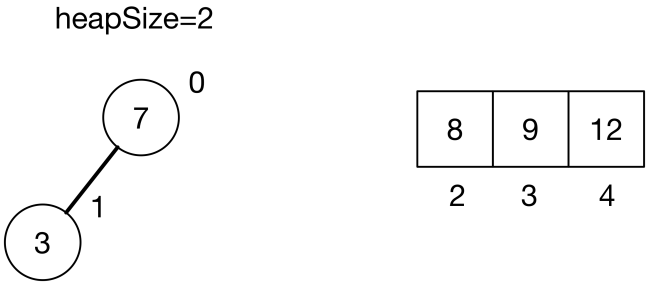
Comenzamos por la raíz, el 12, que es el máximo elemento y que, por tanto, será el último de la lista ordenada, es decir:



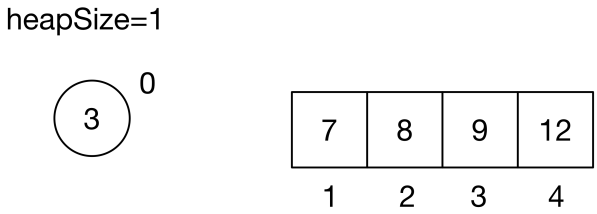
Ahora eliminamos de nuevo la raíz, el 9, que se añadirá como el primer elemento del sufijo, es decir:



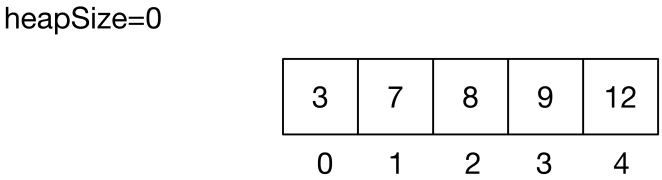
Hacemos lo propio con la nueva raíz, es decir, el 8:



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:

package heapsort;

```
public class HeapSort {

    static class BinaryHeap<E> {

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

        int heapSize = 0;

        ¿?
```

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

        boolean hasLeft(int index) { … }
        boolean hasRight(int index) { … }
        boolean 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) {

        sort(list, ¿?);
    }
}
```

En el segundo método `sort` deberemos, aprovechando que sabemos que la clase `E` implementa `Comparable`, crear una instancia de `Comparator` que use esa ordenación para comparar los elementos del BinaryHeap.

Fijaos en que en modo alguno hemos definido las operaciones que deberá tener el Heap: podremos usar las que más nos convengan para expresar el algoritmo de ordenación de manera que nuestro código, además de correcto, sea legible y entendible. Solamente hemos definido los esqueletos de las funciones que manipulan índices para que, el resto de los métodos de la clase, estén libres de operaciones aritméticas que dificultaran su legibilidad.

MUY IMPORTANTE: La clase BinaryHeap<E>, cuyo único propósito es servir de ayuda al método de ordenación, **no ha de crear ninguna estructura adicional** para guardar los datos del heap ya que **trabajará directamente sobre la lista de elementos a ordenar**.

Otras consideraciones

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

Para facilitar el **testing** de la clase BinaryHeap<E>, se ha definido su visibilidad (y la de sus métodos) como visibilidad por defecto (o de paquete): así se pueden realizar tests, que deberán estar en el mismo paquete `heapsort` (pero en el árbol

de ficheros test) sobre ellas. En este caso, al estar trabajando sobre la misma lista externa, siempre podéis acceder a ella para hacer pruebas (para acceder al otro elemento interesante de la estructura, he definido la visibilidad de `heapSize` como de paquete). Por un lado, así obtengo los beneficios de poder hacer tests pero, por otro, disminuyo el encapsulamiento (la privacidad) de la clase BinaryHeap<E>, lo que podía provocarme problemas.

Java 9, que define el concepto de módulo (y una mayor granularidad a nivel de visibilidades) permitiría, por un lado, definir la clase `BinaryHeap<E>` de manera que: fuera visible y usable desde `HeapSort`, fuera testable y, a su vez, que no se exportara a los clientes de `HeapSort`, de manera que recuperaríamos el encapsulamiento. Si queréis aprender más sobre módulos en Java:

- A Guide to Java 9 Modularity
- Java Modules
- Testing In The Modular World

Entrega

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

Además, se debe entregar un documento de texto, **en formato PDF**, en el cual se explique el funcionamiento de la implementación realizada. Como siempre es conveniente el uso de diagramas para ayudaros en las explicaciones.

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.
- Calidad y limpieza del código.
- Realización de pruebas con JUnit 5.
- Explicación adecuada del trabajo realizado, y funcionamiento de las operaciones sobre el BinaryHeap<E>.

Posibles ampliaciones

- En la práctica se han considerado heaps binarios, pero ello no tiene por qué ser así. Podemos usar la misma idea y definir heaps n-arios en los que cada nodo (excepto uno) tiene n-hijos.

    - Por tanto, una posible ampliación sería implementar la clase NHeap<E> de los heaps n-arios, y con las operaciones de añadir un elemento al heap y eliminar el máximo (ambas con coste logarítmico) y la de acceder al máximo (coste constante).

    - En el caso del HeapSort, se puede comprobar si es más eficiente el heap binario con Heaps ternarios, cuaternarios, etc.

        - Quizás, para hacer más realista la comparación, el caso ternario lo deberíamos particularizar, es decir, no usar la implementación del caso n-ario que usa bucles, sino comparaciones que tengan en cuenta que el número de hijos máximo es tres.

- Otra posible ampliación cosiste en estudiar el código de la clase de Java que implementa las colas con prioridad (java.util.PriorityQueue[2]) que son otro de los usos principales de los heaps.

    - Usando dos PriorityQueues, implementar lo que se conoce como median-heap, un heap que permite mantener acceso en todo momento a la mediana de los elementos que nos van pasando uno a uno, de manera que el acceso a la misma es O(1) y su actualización, cuando llega un nuevo elemento es O(log n)[3]. Una explicación de los median-heaps la podéis encontrar en https://medium.com/@wedneyyuri/how-to-design-a-median-heap-88c80e9db3de

[2] Código de java.util.PriorityQueue en github.

[3] Como siempre amortizando las posibles necesidades de redimensión de arrays.