Laboratorio 3 – HeapSort (v1)

Este laboratorio tiene dos objetivos fundamentales:

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

Como en laboratorios anteriores, el contenido de esta práctica formará parte del temario, en este caso, del segundo parcial.

La estructura de datos BinaryHeap<E>

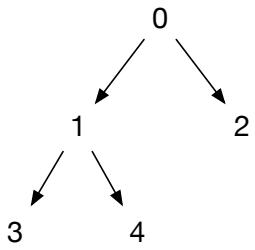
Tal y como se ha comentado en las sesiones de teoría, los árboles, muchas veces no aparecen como estructuras de datos aisladas, con sus operaciones de inserción, acceso y eliminación (por ejemplo, las que existían en las listas), sino que se utilizan en la implementación de otras estructuras de datos o algoritmos, de cara a obtener una mayor eficiencia.

En esta práctica veremos que, usando árboles, podremos encontrar un algoritmo eficiente (de coste $\mathcal{O}(n \log n)$).

Dado un array `E[]` de elementos podemos verlo como un árbol binario casi completo, considerando que los índices, en vez de ser la secuencia

```
0, 1, 2, 3, 4
```

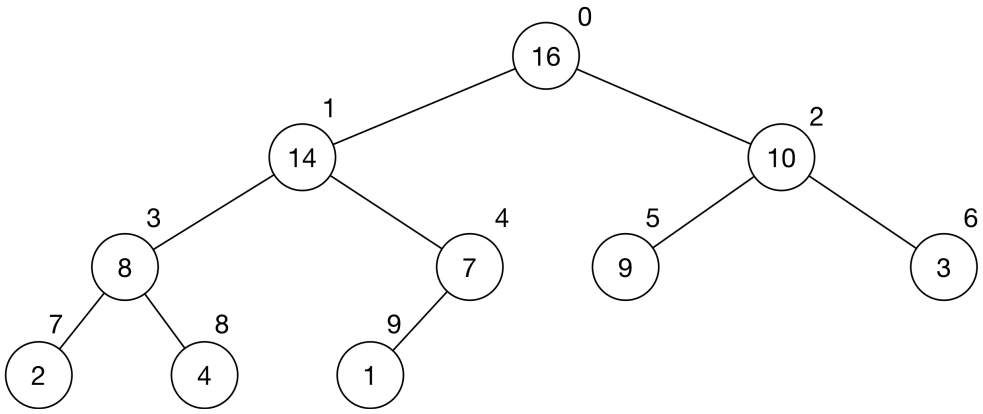
forma el árbol:



Por ejemplo, el `int[]` siguiente

16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9

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).

Como la estructura del árbol viene inducida por los índices[1] del array (mostrados como exponentes en los nodos del árbol), dado un índice, 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 (teniendo en cuenta el tamaño del array).

Los heaps, además de esta visión en forma de árbol sobre un array, tienen una propiedad adicional: **el valor que contiene un nodo siempre es mayor o 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).

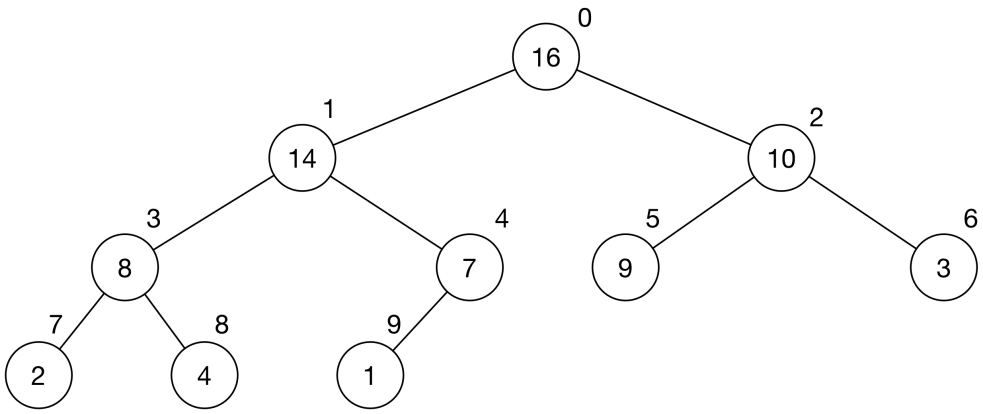
[1] Si implementamos el BinaryHeap<E> como una estructura de datos independiente, se suele colocar la raíz en la posición 1 en vez de la 0. Como la práctica va enfocada al algoritmo de HeapSort donde el heap es, de alguna manera, virtual (ya que se trabaja sobre el array a ordenar), la raíz del heap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

Además de acceder al elemento mayor, existen dos operaciones fundamentales sobre un max-heap, ambas con coste O(log n):

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

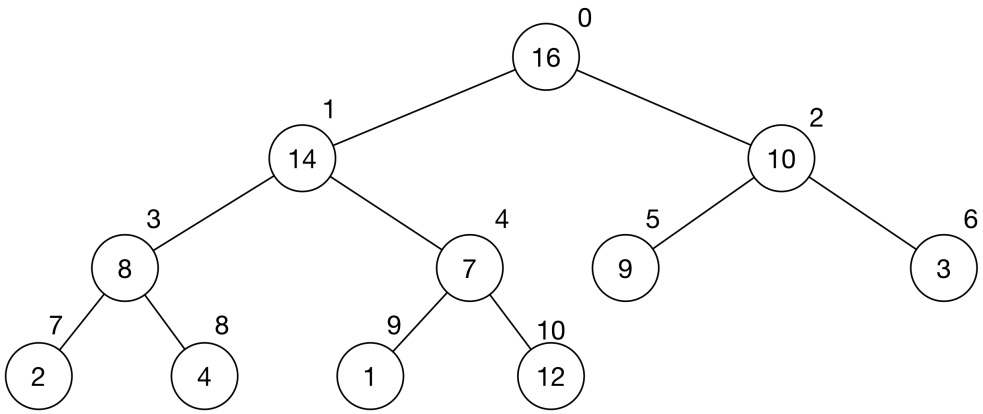
Añadir un elemento

Supongamos que al heap

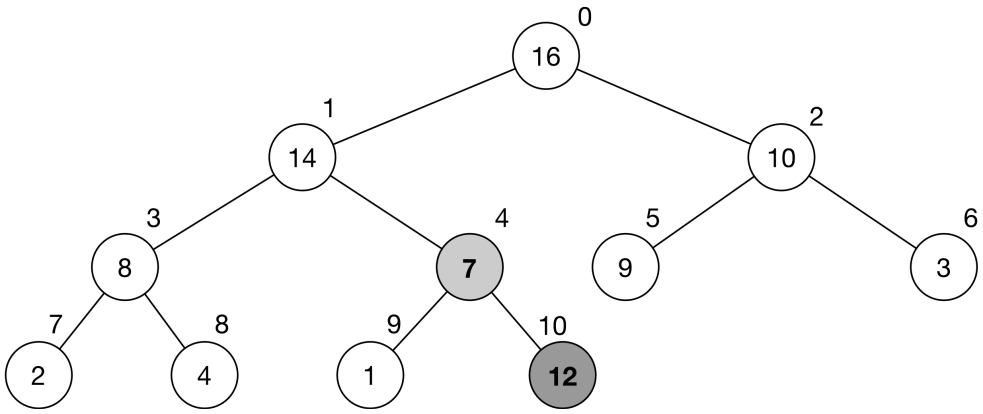


le queremos añadir un elemento de valor 12. Lo añadiríamos al final del array, ocupando por tanto la posición 10 del mismo, e iríamos, partiendo de él, comprobando si está en el lugar que le corresponde o ha de subir. Obviamente, puede darse el caso de que el array ya esté lleno, por lo que, como ya se ha visto en el caso de estructuras secuenciales, se tendría que "redimensionar" el array. En el caso del HeapSort esto no será necesario, pues todo sucederá sobre el array a ordenar, que no ganará ni perderá elementos.

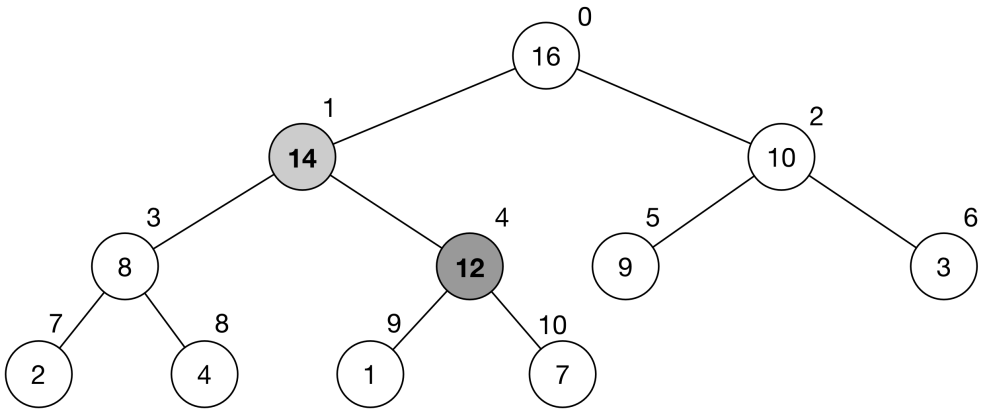
Fijaos en que al haberlo añadido al final del array, de forma trivial, se mantiene la propiedad de ser un árbol casi completo. El problema es que no se cumple el invariante de que un padre siempre es mayor que sus hijos; restablecer este invariante, en tiempo logarítmico, es un elemento básico del trabajo sobre heaps.



Lo comparamos 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 array vuelve a cumplir la propiedad de ser un max-heap.

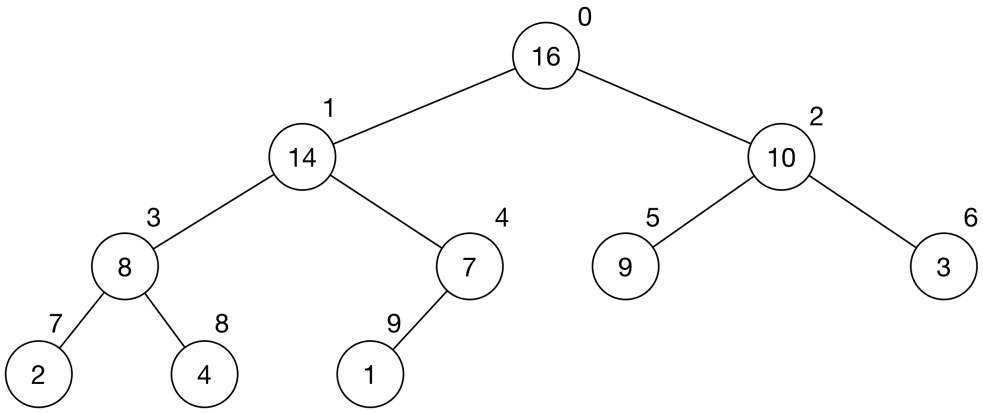
En el peor de los casos, recorreremos todo el camino desde una hoja hasta la raíz, que tiene longitud proporcional al logaritmo del número de nodos, ya que se trata de un árbol casi completo.

NOTA IMPORTANTE: En el caso que nos ocupará, que será el algoritmo de HeapSort, como veremos, el array estará formado por un prefijo, en forma de heap, y un sufijo con el resto de los elementos. Y, en cada paso del bucle, el elemento a insertar en el heap ya estará en la posición correspondiente al "final del array" que se menciona en el apartado.

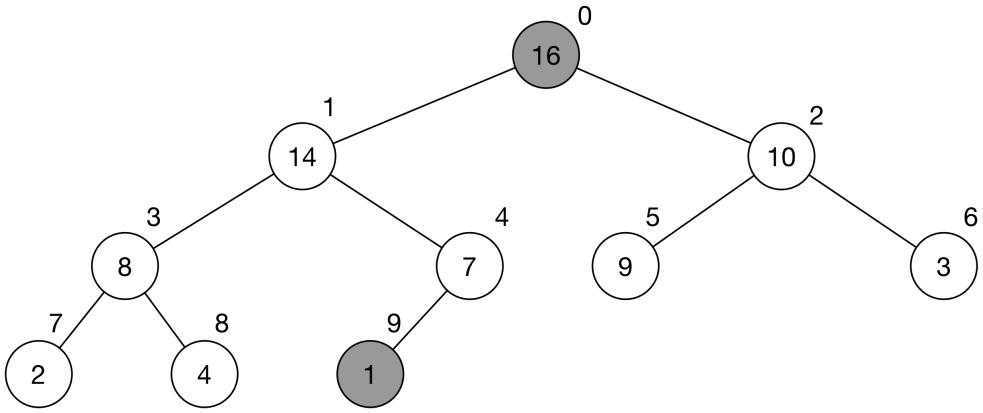
Eliminar la raíz

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.

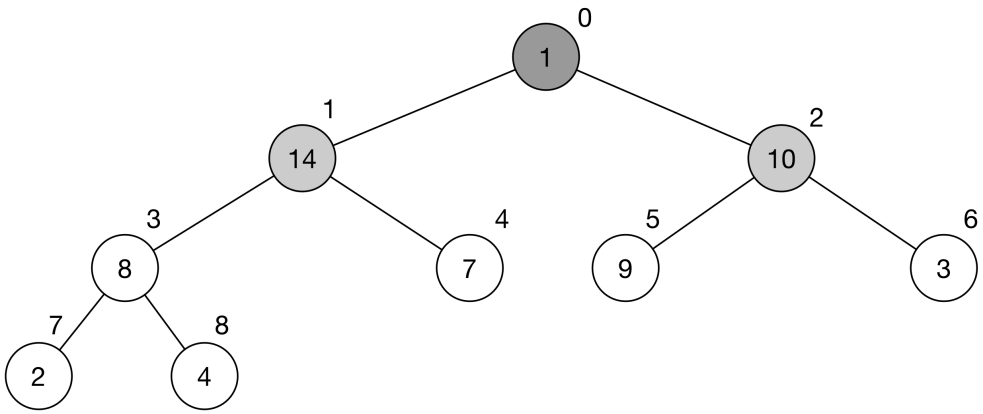
Vamos a ver el caso complicado en el que el heap sí tiene elementos. Por ejemplo, si volvemos a partir del heap



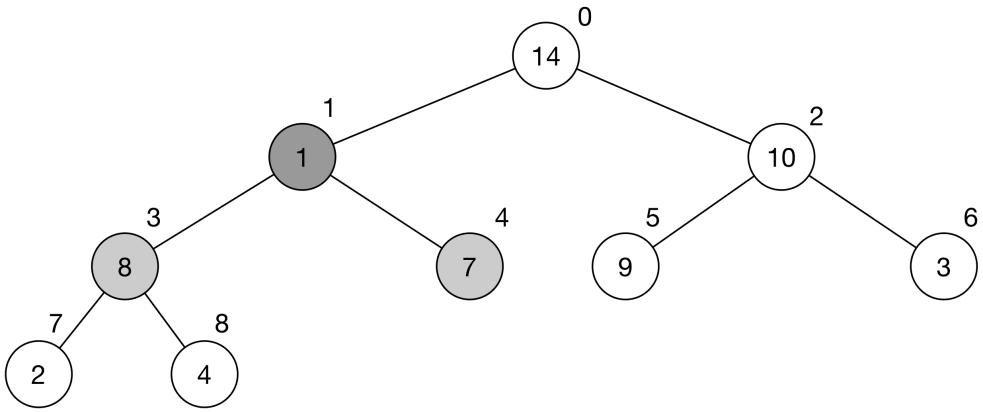
devolveríamos el valor correspondiente al nodo raíz y ahora deberíamos "arreglar" el max-heap. ¿Cómo lo hacemos? La idea es sustituir la raíz por el único nodo que, si lo quitamos, no nos deja un hueco en el árbol (que ha de ser casi completo): el último nodo.



Colocamos el valor correspondiente al último nodo en la raíz, y eliminamos el último elemento del array[2]. Por tanto, lo que nos queda es:

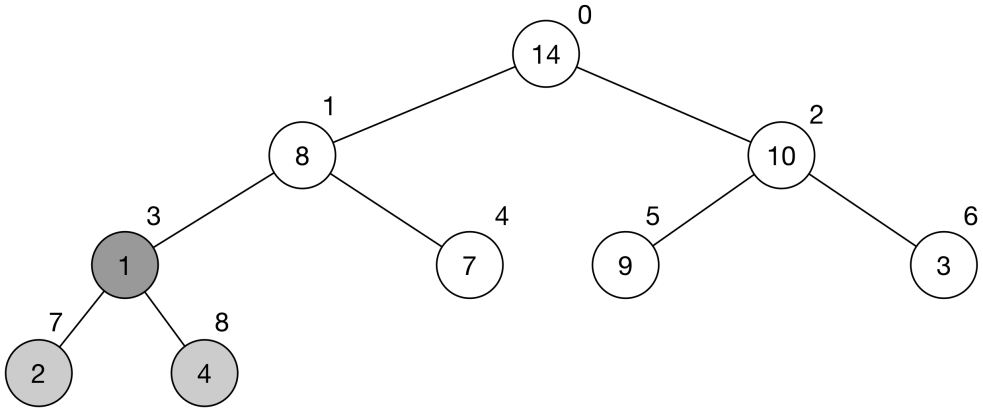


Como podéis observar, el árbol sigue siendo casi completo, pero se ha perdido la propiedad de ser un max-heap. ¿Cómo podemos comprobar eso? Partimos del nodo raíz, y lo comparamos con sus hijos existentes y buscamos que sea mayor de todos. En nuestro caso el mayor entre 1, 14 y 10, que es 14. Intercambiamos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

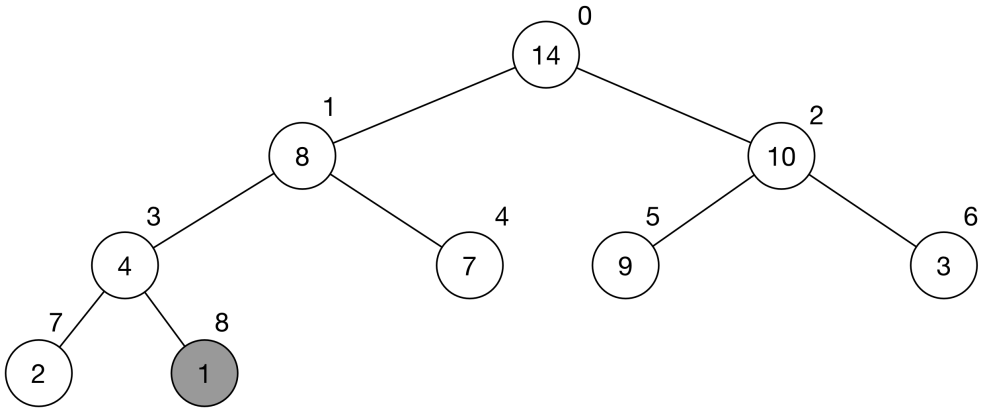


[2] Obviamente, si se tratara de una estructura de heap independiente, tendríamos una variable de instancia size para saber en todo momento el número de elementos reales que hay en el array.

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.

NOTA IMPORTANTE: En el caso que nos ocupará, que será el algoritmo de HeapSort, como veremos el array estará formado por un prefijo, en forma de heap, y un sufijo con el resto de los elementos. Y, en cada paso del bucle, el elemento raíz del heap no desaparecerá, sino que se colocará como primer elemento del sufijo (en la posición que ocupaba el elemento que hemos colocado, provisionalmente, como nueva raíz).

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 `E[]` con los elementos a ordenar, considerar que dicho array 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 `E[]`.

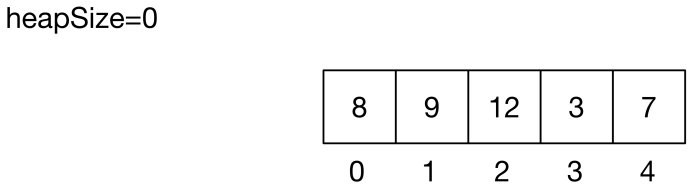
Y, el algoritmo realiza la ordenación en dos fases:

- 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 array) 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 array que, al final, contendrá todos los elementos y estará ordenado.

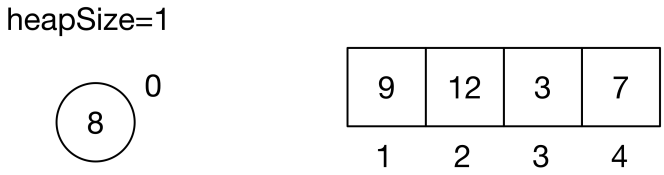
NOTA IMPORTANTE: Durante todo el algoritmo solamente existe un array, del cual el prefijo es el max-heap. No hay que copiar nada en arrays auxiliares, ni redimensionarlo, ni mover más elementos que los derivados de las operaciones de inserción/eliminación en la zona de heap.

Veamos un ejemplo sobre un `Integer[]` de 5 posiciones (para simplificar los diagramas lo mostramos como si fuera un array de `int`s).

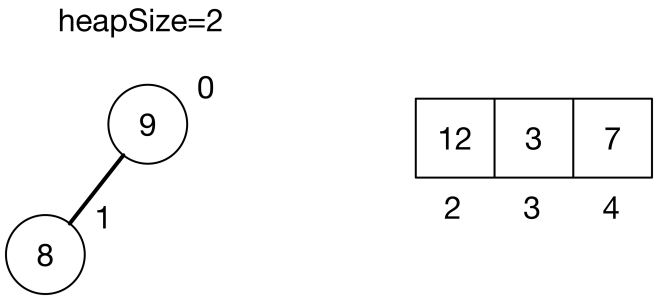
Inicialmente, el max-heap está vacío (tiene 0 elementos) y todos los elementos están en el sufijo. La variable `heapSize` contiene el número de elementos que forman parte del prefijo, es decir, de la zona de heap:



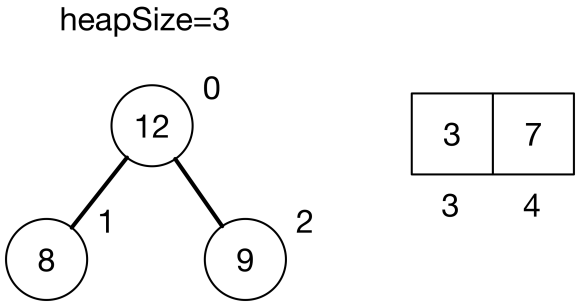
Insertamos el primer elemento del sufijo, el 8, en el max-heap, es decir:



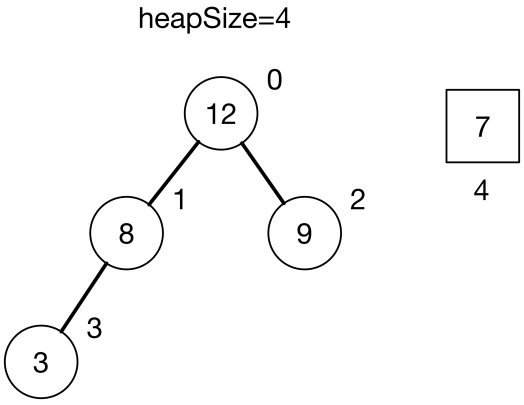
Hacemos lo mismo para el 9, que es el primer elemento del sufijo (ya que el heap ahora tiene un elemento) y nos queda:



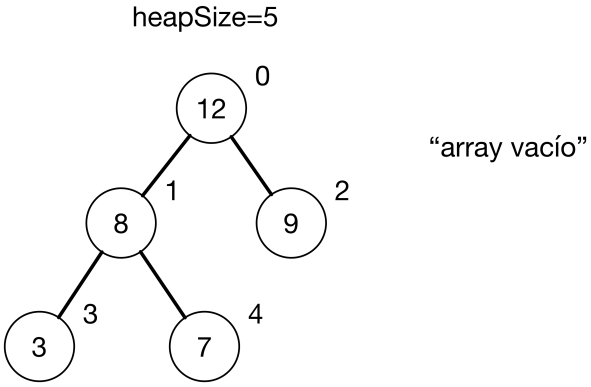
Añadimos el 12 al max-heap:



Ahora el 3:

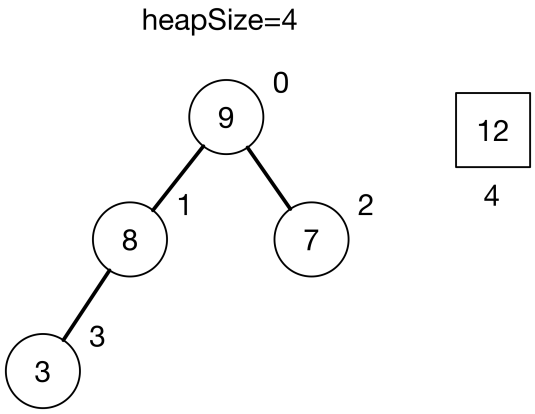


Y, finalmente el 7:

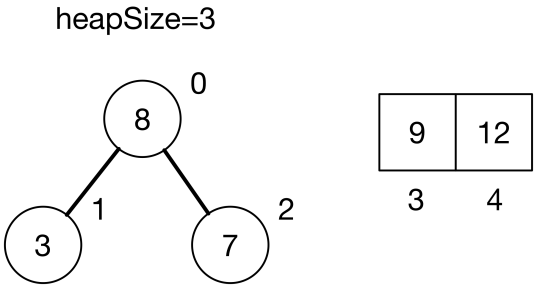


Ahora empieza la segunda fase que consiste en ir desmontando el max-heap, convirtiendo la raíz en el primer elemento del sufijo.

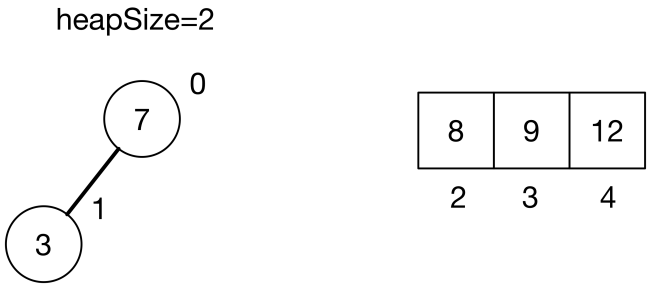
Comenzamos por la raíz, el 12, que es el máximo elemento y que, por tanto, será el último de la lista ordenada, es decir:



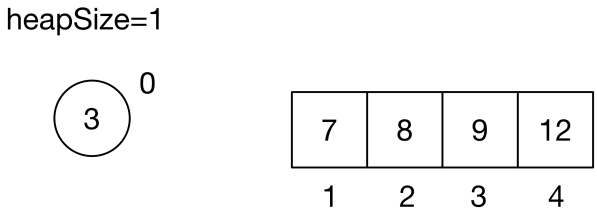
Ahora eliminamos de nuevo la raíz, el 9, que se añadirá como el primer elemento del sufijo, es decir:



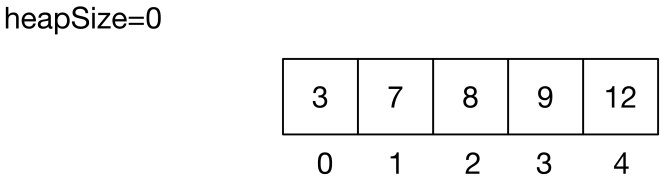
Hacemos lo propio con la nueva raíz, es decir, el 8:



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 {

    public static <E> void sort(
                E[] elements, Comparator<? super E> cmp) {
        new BinaryHeap<>(elements, cmp).heapSort();
    }

    public static <E extends Comparable<? super E>> void sort(
                E[] elements) {
        sort(elements, Comparator.naturalOrder());
    }

    static class BinaryHeap<E> {

        final E[] elements;
        final Comparator<? super E> comparator;
        int heapSize;
```

```
        BinaryHeap(E[] elements, Comparator<? super E> comparator) {
            this.elements = elements;
            this.comparator = comparator;
            this.heapSize = 0;
        }

        void heapSort() { ??? }

        ???
    }
}
```

En el segundo método `sort` creamos, aprovechando que sabemos que la clase `E` implementa `Comparable`, 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 "de forma arbórea" 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 el array de elementos a ordenar.

Consideraciones sobre testing

Visibilidad de las operaciones

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 test, que deberán estar en el mismo paquete `heapsort` (pero en el árbol de ficheros test). Por eso, cuando definàis métodos auxiliares para definir el algoritmo de ordenación, deberéis darles visibilidad por defecto y así podréis testearlos adecuadamente.

Por ejemplo, imaginemos que en la clase BinaryHeap tenemos un método swap para intercambiar dos posiciones del array de elementos. Podríamos comprobar su funcionamiento con el siguiente test:

```
@Test
void swap() {
    Integer[] elements = {1, 2, 3, 4, 5};
    var heap = new HeapSort.BinaryHeap<>(
            elements,
            Comparator.naturalOrder());
    heap.swap(0, 1);
    Integer[] expected = {2, 1, 3, 4, 5};
    assertArrayEquals(expected, elements);
}
```

De esta manera podréis comprobar todos los métodos auxiliares que hayáis definido para manipular el array y así tener certeza de que son correctos antes de usarlos en la implementación del algoritmo de ordenación.

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 testeable 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

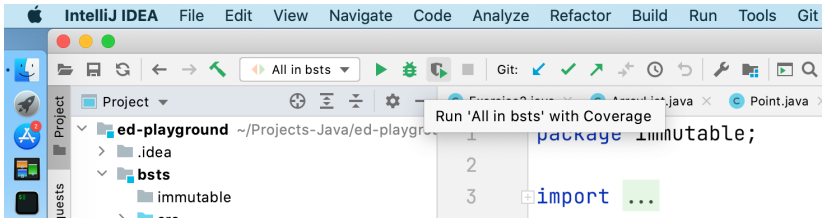
El concepto de cobertura

Muchas veces no es trivial saber si el conjunto de tests que hemos realizado cubre todas las posibilidades. Una herramienta que puede ayudarnos, aunque en casos de código complejo con muchos condicionales y, por tanto, muchos caminos de ejecución, no es suficiente, es la del **análisis de cobertura**. En dicho análisis, IntelliJ nos marcará las líneas de código que han sido ejecutadas por al menos uno de los tests y las que no han sido ejecutadas nunca.

Así que una aproximación, por tanto, consistirá en intentar que todas las líneas de nuestro código hayan sido ejecutadas por al menos un test (ello no garantiza tener cubiertas todas las posibilidades, pero sí hay muchas líneas sin cubrir, o bien tenemos alguna posibilidad no cubierta o ese código es innecesario; en ambos casos algo hemos de hacer).

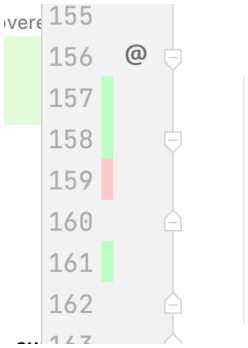
Por ello, al ejecutar el código de pruebas, se realizará con la **opción de cobertura de código**[3] para que IntelliJ indique aquellas líneas que han sido cubiertas con la prueba. Ello no es garantía de haber cubierto todos los casos, pero es una ayuda.

Dicha ejecución puede realizarse para todas las pruebas:



O sobre una en concreto:

Una vez ha sido ejecutado de esta manera, el entorno IntelliJ nos informará de qué líneas de código han sido ejecutadas con la prueba.
La mejor manera de comprobarlo es por medio de la información visual que nos ofrece en el código que ha sido testeado. Aquello marcado con verde representa código cubierto por la prueba y en rojo el que no (**¡Error! No se encuentra el origen de la referencia.**).



Podéis comprobar que el entorno ofrece también información sobre el porcentaje de líneas cubiertas y no cubiertas. Sin embargo, esta información contiene cierto porcentaje relativo a interfaces (en el caso de este laboratorio) que no nos es útil. Por lo tanto, en nuestro caso, es mejor acceder al código

[3] https://www.jetbrains.com/help/idea/code-coverage.html

testeado y comprobar visualmente (indicaciones visuales rojas y verdes) para saber qué código ha sido cubierto por la ejecución.

Entrega

El resultado obtenido debe ser entregado por medio de un **proyecto IntelliJ** por cada grupo, comprimido **en formato ZIP** y con el nombre "Lab3_NombreIntegrantes**.zip**".

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

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

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.
- Explicación adecuada del trabajo realizado, y funcionamiento de las operaciones sobre el `BinaryHeap<E>`.
    - **Añadid ejemplos y diagramas para que se entienda la explicación.**
- Realización de pruebas con `JUnit 5`, especialmente de los diferentes métodos que definís en `BinaryHeap<E>`.
- Calidad y limpieza del código.
    - **En este sentido, existe un caso especial que cabe destacar: para que la práctica sea corregida, el proyecto de IntelliJ debe poderse abrir de forma normal y compilar, como mucho requiriendo configurar una versión de Java adecuada.**
    - **El proyecto que cargaremos será el de la raíz. Aseguraros de no entregar un proyecto dentro de otro proyecto, dentro de otro proyecto.**

Los pesos de cada apartado serán:

- código de heapSort, clase BinaryHeap<E>, etc: 30%
- juegos de pruebas: 30%
- informe: 40%

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 consiste en estudiar el código de la clase de Java que implementa las colas con prioridad (`java.util.PriorityQueue`[4]) 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)[5]. Una explicación de los median-heaps la podéis encontrar en https://medium.com/@wedneyyuri/how-to-design-a-median-heap-88c80e9db3de

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

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