

Laboratorio 3 – Heaps y Colas con Prioridad

Este laboratorio tiene dos objetivos fundamentales:

1. Presentar una estructura arborescente denominada Heap que, como veremos, puede implementarse en un array o, por comodidad, en un ArrayList.
2. Utilizarla para implementar de forma eficiente una cola con prioridades, es decir, una cola en la que el orden de salida no sólo tenga que ver con el momento de la llegada, sino también con la prioridad asociada a cada elemento.

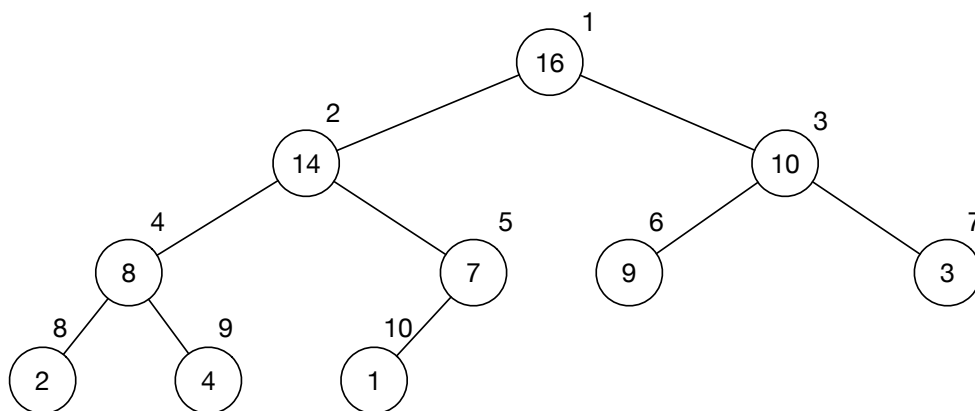
La estructura de datos Heap

La estructura de datos denominada **Heap<E>** consiste en un **array** (nosotros usaremos un **ArrayList<E>**, lo que nos permitirá desentendernos de redimensionados) que podemos ver como un **árbol binario casi completo por la izquierda**.

Por ejemplo, el **ArrayList<Integer>** siguiente

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

en el que, para comodidad en cálculo de índices, la posición 0 la consideramos no usada). Dicho **ArrayList<Integer>** 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), que está ocupado parcialmente por la izquierda.

La estructura del árbol viene inducida por los índices del `ArrayList<E>`

- dado un índice, podemos calcular el índice que le corresponderá a su padre, a su hijo izquierdo y a su hijo derecho.
- también podemos implementar métodos para saber si un nodo tiene padre, hijo izquierdo o hijo derecho, mirando si los índices calculados están dentro de los válidos en el `ArrayList<E>`.

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-heap).

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 $\mathcal{O}(1)$** .

El interés que tiene esta estructura es que **permite implementar operaciones de eliminación del máximo e inserción de nuevos valores en tiempo logarítmico $\mathcal{O}(\log_2 n)$** .

[La cola con Prioridad](#)

Una cola con prioridad es una cola en la que los elementos no se eliminan en orden de llegada, sino en orden decreciente de prioridad (los más prioritarios, primero).

En el caso de elementos con la misma prioridad, el orden de salida de los mismos es el orden de llegada (por ello, además de guardar su prioridad, deberemos guardar una marca de tiempo de llegada).

```
package heaps;
```

```
public interface PriorityQueue<
    P extends Comparable<? super P>, V> {

    void add(P priority, V value);
    V remove();
    V element();
}
```

```
    int size();  
}
```

Vamos por partes, pues la definición parece un poco complicada:

- Tenemos dos parámetros genéricos: **P**, que representa la prioridad, y **V** que representa el valor.
- Como deberemos poder hacer comparaciones por prioridad, necesitaremos que la prioridad **P** implemente la interfaz **Comparable**.
 - Para tener un máximo de flexibilidad, usamos comodines y, por tanto, queda como **Comparable<? Super P>**.

A partir de aquí las operaciones son inmediatas:

- **void add(P priority, V value):**
 - añade el elemento de valor “*value*” con la prioridad “*priority*”.
 - debéis considerar que el valor de prioridad **null** es el más bajo posible.
- **V remove():**
 - retorna el elemento de prioridad máxima que haya llegado en primer lugar (el mismo que uno obtendría con **element()**) y lo elimina de la estructura.
 - en caso de que la cola esté vacía, lanza la excepción no comprobada predefinida: **NoSuchElementException**.
- **V element():**
 - devuelve el valor correspondiente al elemento de prioridad máxima que haya llegado en primer lugar.
 - en caso de que la cola esté vacía, lanza la excepción no comprobada predefinida: **NoSuchElementException**.
- **int size():**
 - devuelve el número de elementos que contiene la cola.

Fijaos en que sería muy fácil implementar la cola con prioridad simplemente con una lista de tríos prioridad-timestamp-valor.

¿Qué tipo de lista? ¿Cómo la organizamos?

Como tipos de lista podemos pensar en:

- **ArrayList**
- **LinkedList**

y como organización,

- ordenada
- desordenada

Aunque, la simplicidad tendría un precio, pues las operaciones serían lineales:

- ArrayList/LinkedList sin ordenar: añadir será $O(1)$, en el caso del ArrayList amortizado; acceder al más prioritario será $O(n)$; eliminar el más prioritario también será $O(n)$.
- ArrayList/LinkedList ordenada: añadir será $O(n)$; acceder al más prioritario $O(1)$; eliminarlo $O(1)$, obviamente, si lo dejamos al final del ArrayList.

En cambio, usar un **ArrayList¹ organizado como heap** permite obtener costes logarítmicos en las operaciones de añadir y eliminar, y coste constante en la operación de acceder.

Es decir, **el heap no constituye una estructura de datos aparte, sino una forma diferente de organizar los elementos en el array.**

Implementación de las colas usando un Heap

El objetivo de esta implementación es usar internamente un ArrayList de tríos prioridad-timestamp-valor que comentábamos con anterioridad, **pero viendo el Array como un heap**. De esta manera podremos obtener implementaciones eficientes de tanto inserciones, extracciones y accesos.

Un esbozo de la clase a implementar sería:

¹ Podríamos conseguir lo mismo con un simple array, pero entonces deberíamos implementar el redimensionado del mismo; un ArrayList nos permite no preocuparnos de ese detalle.

```

package heaps;

import java.util.ArrayList;
import java.util.NoSuchElementException;

public class HeapQueue<P extends Comparable<? super P>, V>
    implements PriorityQueue<P, V> {

    private final ArrayList<Triplet<P, V>> triplets;
    private long nextTimeStamp = 0L;

    static class Triplet<P extends Comparable<? super P>, V>
        implements Comparable<Triplet<P, V>> {

        private final P priority;
        private final long timeStamp;
        private final V value;

        Triplet(P priority, long timeStamp, V value) {
            this.value = value;
            this.priority = priority;
            this.timeStamp = timeStamp;
        }

        @Override
        public int compareTo(Triplet<P, V> other) { ¿? }
    }

    public HeapQueue() { ¿? }

    @Override public void add(P priority, V value,) { ¿? }

    @Override public V remove() { ¿? }

    @Override public V element() { ¿? }

    @Override public int size() { ¿? }

    ¿?
}

```

Vamos poco a poco, pues parece complicado, pero no lo es tanto:

- `HeapQueue<P,V>` tiene los mismos parámetros que la interfaz `PriorityQueue<P,V>` y sus mismas restricciones.
- Internamente usará un `ArrayList` de tríos prioridad-timestamp-valor, implementadas en la clase interna estática `Triplet` que se inicializará con la lista vacía.

- La clase `Triplet` requerirá también que los elementos de tipo `P` (prioridad) sean `Comparables` y, a su vez, implementará la interfaz `Comparable` de las parejas. ¿Para qué?
 - de esta manera, los elementos del `ArrayList` serán comparables
 - implementaremos la comparación entre tríos en función de la comparación entre sus prioridades (en el método `compareTo` que está por completar) teniendo en cuenta tanto sus valores nulos como los valores del `timestamp`.
 - **La correcta implementación de esta comparación es una de las claves de la práctica por lo que os sugiero que testeéis su buen funcionamiento a conciencia.**

Para que la podáis testear con facilidad, la clase `Triplet` se ha definido con visibilidad de paquete (fijaos en que no es privada), por lo que test definidos en el mismo paquete que la clase `HeapQueue` pueden acceder a la misma y crear tripletas y testear el método `compareTo`.

```
package heaps;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertTrue;

public class TripletTest {

    @Test
    void testSomeCompare() {
        var triplet1 = new HeapQueue.Triplet<Integer, Object>(
            1, 1L, new Object());
        var triplet2 = new HeapQueue.Triplet<Integer, Object>(
            2, 2L, new Object());
        assertTrue(triplet2.compareTo(triplet1) > 0);
    }
}
```

Vamos a ver las diferentes operaciones sobre el `heap`.

Operaciones privadas que os simplificarán el código

De cara a poder expresar de forma cómoda los algoritmos de inserción y extracción, es muy conveniente definir operaciones que manipulen los índices en forma de árbol. Por ejemplo:

```
static int parentIndex(int i) { ¿? }

static int leftIndex(int i) { ¿? }
```

```
static int rightIndex(int i) { ¿? }

boolean exists(int index) {
    return 1 <= index && index <= size();
}
```

Las podemos definir con visibilidad por defecto, de manera que sean visibles desde cualquier clase del mismo paquete que la clase `HeapQueue`. Como los test los colocaremos en este mismo paquete, podremos hacer test sobre ellas sin problemas.

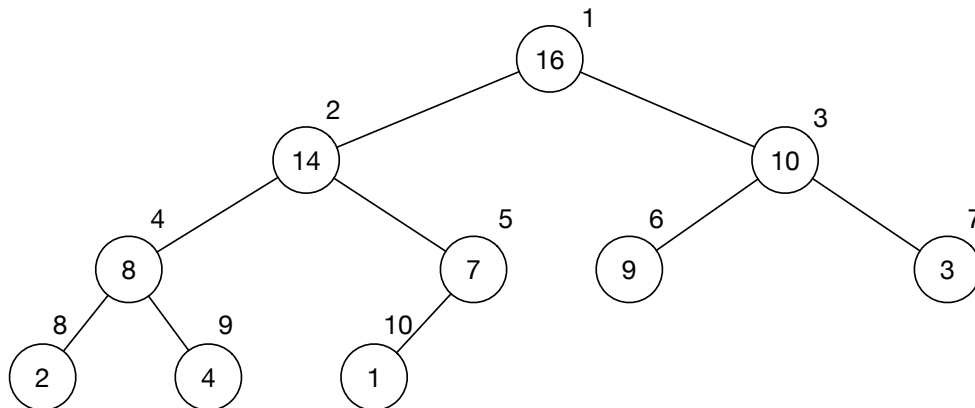
```
void add(P priority, V value,)
```

Para añadir un elemento al heap, crearemos el trío prioridad-timestamp-valor en base a los valores introducidos en la función y el valor actual del `nextTimeStamp`.

El trío generado se añadirá como elemento final del `ArrayList` y, partiendo de él, iremos arreglando los nodos en el camino hacia la raíz que queden desordenados (de hecho, podemos parar en el primer nodo que no debamos arreglar).

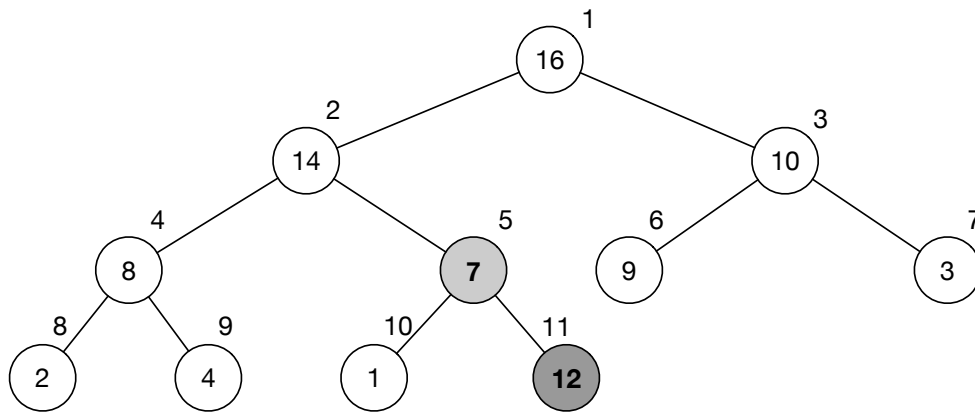
Para mostrar cómo funciona el mecanismo, solamente mostraremos en cada nodo del heap su prioridad.

Supongamos que al heap

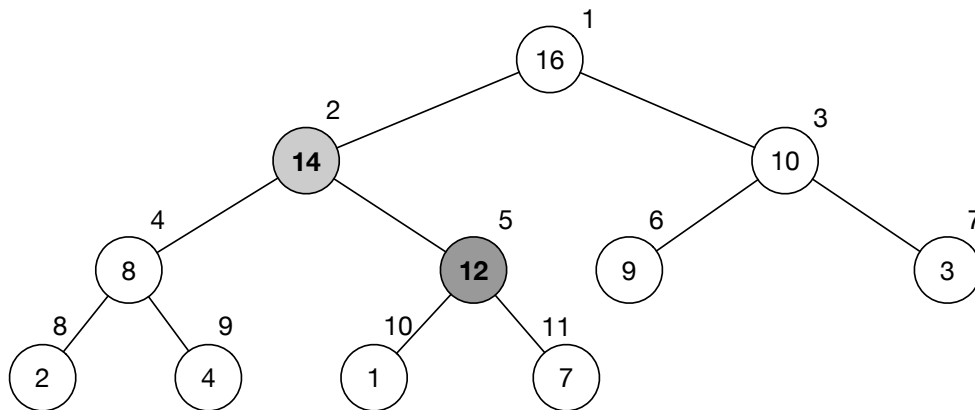


le queremos añadir un valor de prioridad 12 (recordad que en el árbol solamente mostramos las prioridades). Lo añadiríamos al final del `ArrayList` e iríamos, partiendo de él, intentado ver si está en el lugar que le corresponde o ha de subir dentro de la jerarquía dentro del árbol.

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.

El coste logarítmico de la operación procede del hecho de que, en el peor de los casos, deberemos recorrer todos los nodos en el camino desde el nodo añadido hasta la raíz, y en un árbol casi completo dicho camino es proporcional al logaritmo del número de nodos.

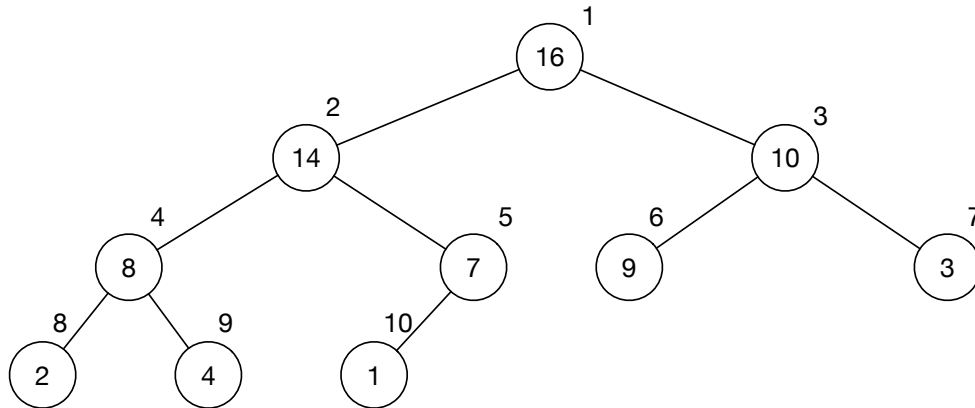
De cara a que se tenga en cuenta el momento de inserción, para decidir empates entre tripletas de la misma prioridad, deberéis incrementar el valor del `nextTimeStamp` en cada inserción.

[V remove\(\)](#)

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

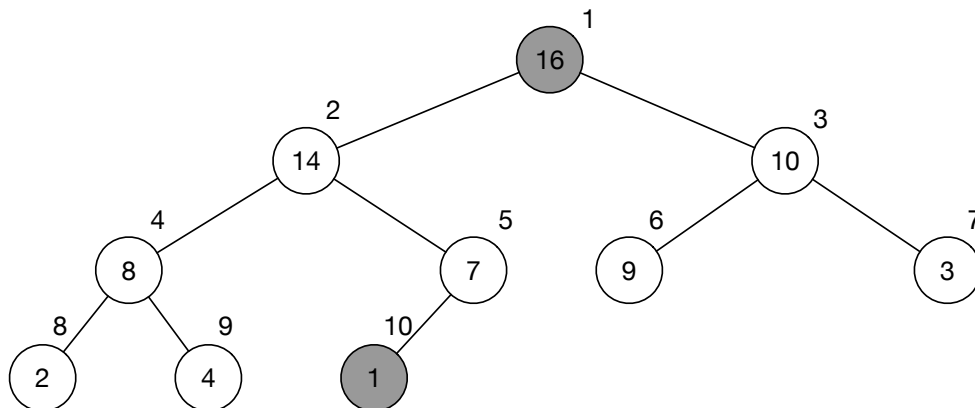
El caso en el que el heap no tenga elementos se lanzará la excepción predefinida y no comprobada denominada `NoSuchElementException`. Esta acción no merece demasiados comentarios adicionales.

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

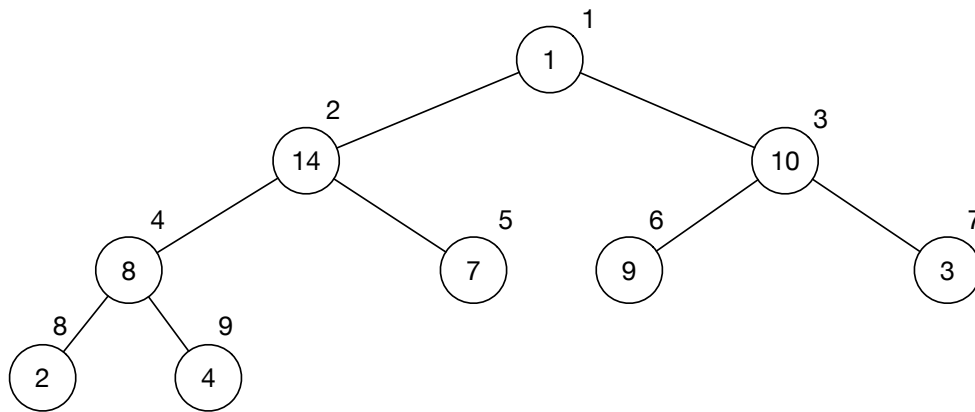


devolveríamos el valor correspondiente al nodo raíz (os recuerdo de nuevo, que en los diagramas solamente aparecen las prioridades) y ahora deberíamos arreglar el max-heap.

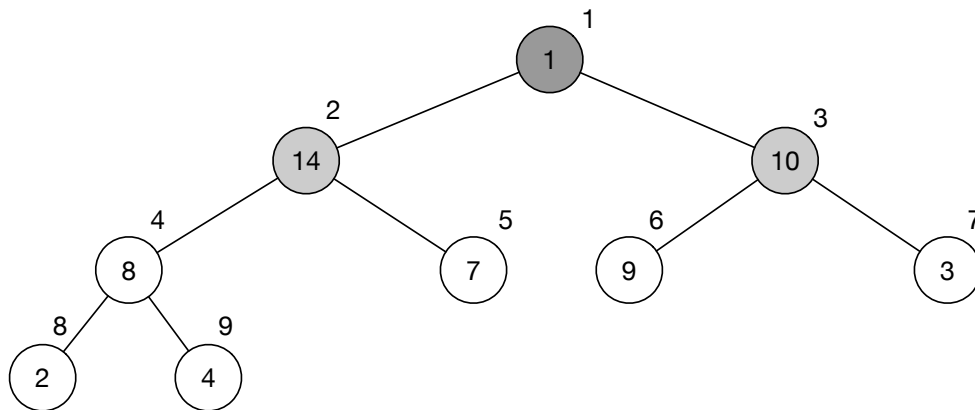
¿Cómo lo hacemos? La idea es sustituir la raíz por el único nodo que, si lo quitamos, no nos deja un hueco en el árbol (que ha de ser casi completo): el último nodo.



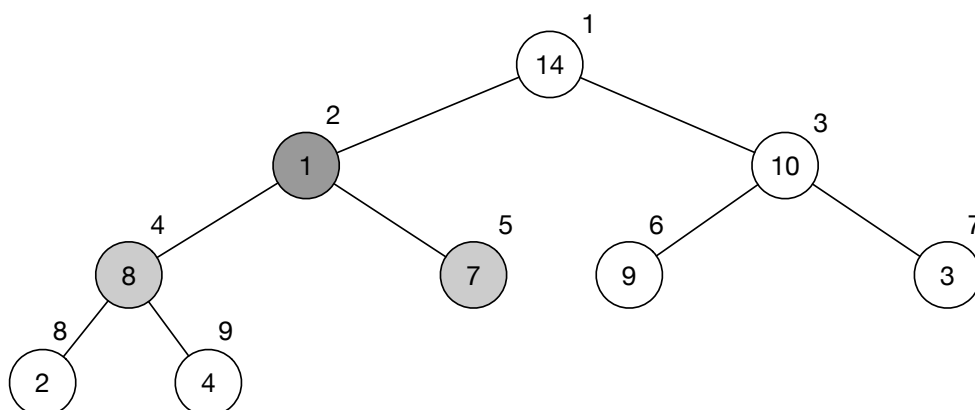
Una vez colocado el valor del último nodo en la raíz (cuyo valor será lo que retornaremos), podemos eliminar el último nodo del árbol, por lo que nos queda:



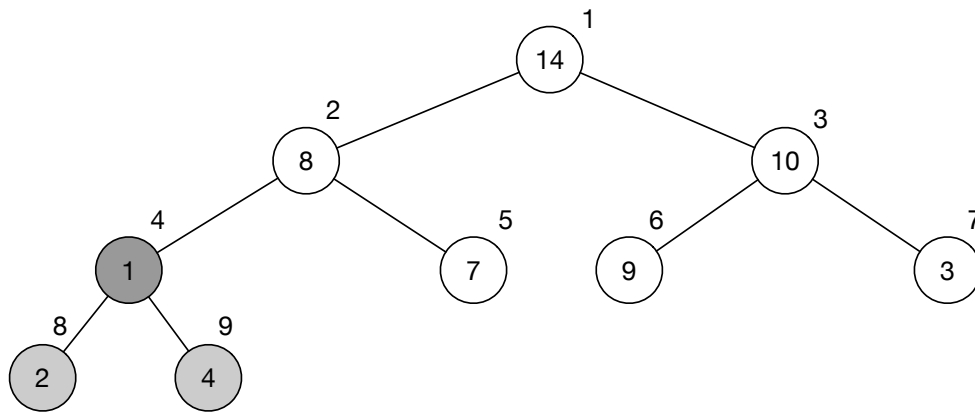
Por tanto, hemos arreglado la forma del árbol, es decir, sigue siendo un árbol completo por la izquierda, pero como se puede ver con los nodos marcados, el trío de elementos marcados no cumple la propiedad de max-heap.



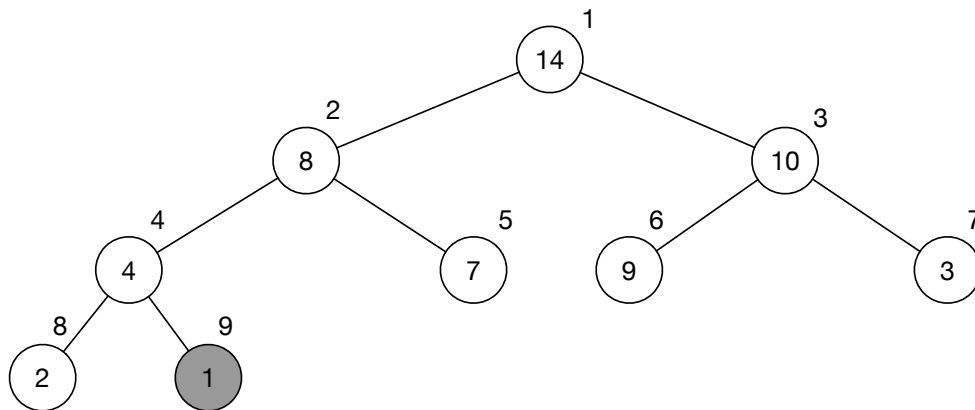
El siguiente procedimiento consistirá en ir bajando por el árbol arreglando las tripletas incorrectas. Por tanto, partiremos del nodo raíz, y lo comparamos con sus hijos existentes y buscamos el que sea mayor de todos. En nuestro caso el mayor entre 1, 14 y 10, es 14. Intercambiamos el nodo raíz por 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.

NOTA IMPORTANTE: cuando manipuléis los elementos del heap, es decir, del `ArrayList`, por ejemplo, para intercambiarlos, la única operación que os garantiza tiempo constante y, por tanto, logarítmico en la operación de inserción o eliminación, es `set`.

`getElement()`

Esta operación, además de comprobar que la cola tiene elementos y lanzar `NoSuchElementException` en caso de que no los tenga, devuelve el elemento de mayor prioridad que se haya añadido en primer lugar.

Int size()

Esta operación devuelve el número de elementos que hay en el heap (que se corresponde con el **número de tripletas reales** del ArrayList).

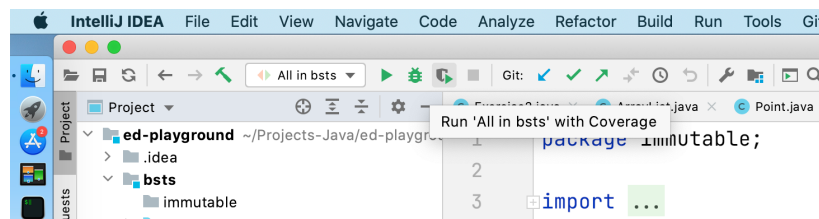
Consideraciones sobre el testing

Los test han de comprobar que los métodos devuelvan los resultados esperados según su especificación, no han de comprobar cómo están las clases implementadas por dentro.

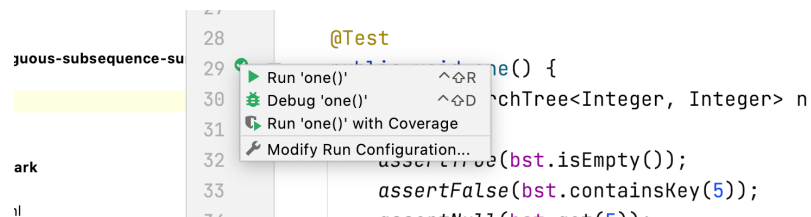
Por ejemplo, si añadís una serie de elementos a la cola con prioridad, sabéis exactamente en qué orden los iréis obteniendo hasta vaciar la cola y justamente eso es lo que debéis testear.

Por ello, para ver si aún quedan casos por comprobar, en la elaboración de las pruebas de esta tarea, se recomienda que, al ejecutar el código de prueba, se realice con la **opción de cobertura de código**² 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:

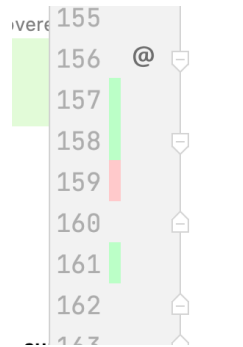


Una vez ha sido ejecutado de esta manera, el entorno IntelliJ nos informará de qué líneas de código han sido ejecutadas con la prueba.

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

² <https://www.jetbrains.com/help/idea/code-coverage.html>

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

A modo de resumen

En resumen, la práctica consiste en definir la interfaz `PriorityQueue` y su implementación en la clase `HeapQueue` (ambas en el paquete `heaps`) junto con una serie de test que comprueben su buen funcionamiento.

No olvidéis compilar usando la opción `-Xlint:unchecked`, y eliminar la opción de compilación al guardar, para que el compilador os avise de errores en el uso de genéricos.

En el informe, además de comentar la implementación realizada, deberéis comentar cómo habéis diseñado los juegos de prueba para conseguir el máximo de cobertura.

Entrega

El resultado obtenido debe ser entregado por medio de un fichero comprimido en **formato ZIP** con el nombre “Lab3_NombreIntegrantes.zip” que contenga:

- proyecto IntelliJ con vuestro código.
- un documento de texto en **formato PDF**, en el cual se explique el funcionamiento de la implementación realizada.

Solamente es necesario que entregue uno de los miembros del grupo.

Consideraciones de Evaluación

Al evaluar el laboratorio se tendrán en cuenta los siguientes aspectos:

- El código ofrece una solución al problema planteado.
- **Explicación adecuada del trabajo realizado, valiéndose de diagramas, extractos del código (que os recuerdo, es texto y no imagen)**
- Realización de tests con JUnit 5.
- Calidad y limpieza del código.