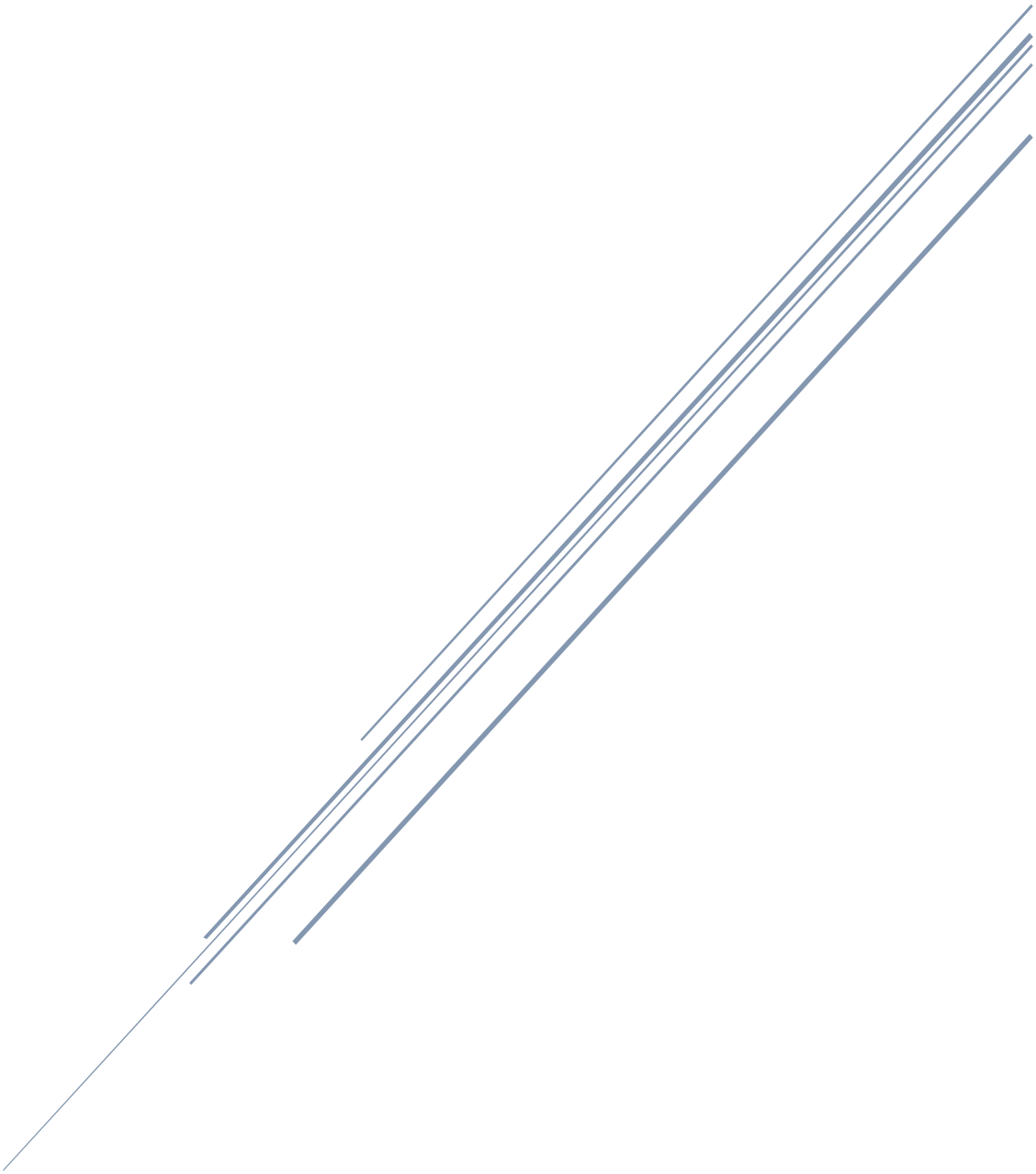
LABORATORI 3

Heaps i Cues amb Prioritat



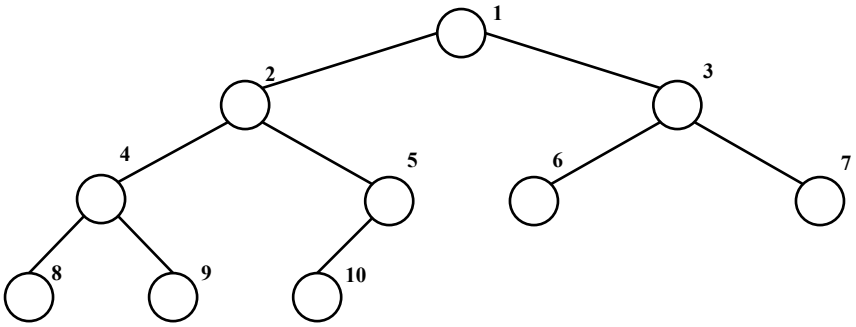
EPS UdL
Autors: Èric Bitriá i Adrià Fernàndez

Índex

1. Introducció

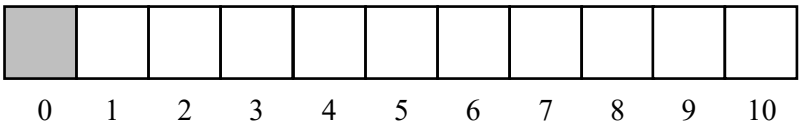
1.1 Heap

Una estructura de dades Heap es defineix com un arbre binari casi complet a la esquerra, de forma que tots els seus elements es van emplenant de esquerra a dreta prioritzant cada nivell:



[1] Heap

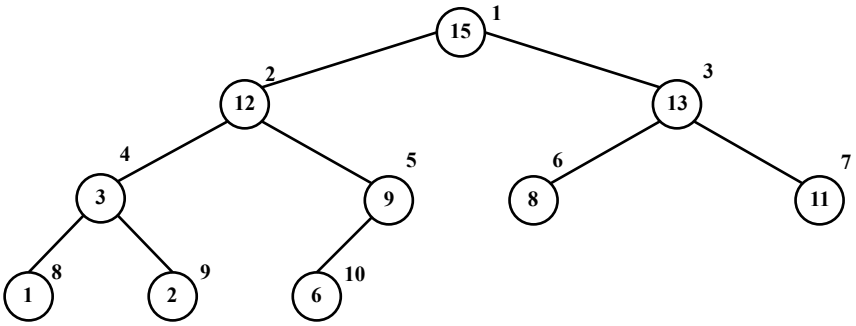
De forma interna emmagatzemarem les dades d'aquest arbre com si fos un array normal. Podem veure la posició que ocuparia cada node del heap [1] en el array [2], on cada número a fora del node serà la posició que ocupa en el array. Fixem-nos que no considerem la posició 0.



[2] Array del Heap

1.2 Max-Heap

Un max-heap segueix la mateixa estructura que un Heap, l'únic que canvia es la forma de col·locar els seus elements, on per cada node el valor que conté sempre és més gran que els dels seus nodes fills [3].



[3] Max-Heap

2. Triplets

A l'hora d'afegir elements en un Max-Heap haurem de tenir en compte tres propietats d'aquell element:

- **La prioritat**: Depenent de la prioritat que tingui aquell element podrà ocupar una posició major o menor dins del Heap.
- **El valor**: Serà la informació que contindrà aquell element.
- **El timestamp**: Es pot donar la situació en que dos elements tenen la mateixa prioritat, per desempatar aquests elements s'utilitzarà l'ordre d'arribada, fent que el element que hagi arribat primer tingui més prioritat.

D'aquesta manera podem definir els elements que contindrà la nostra estructura anomenats Triplets i contindran les propietats prèviament mencionades.

```
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;
    }
}
```

Per determinar la prioritat entre els elements haurem de definir un mètode per comparar els triplets basats en la seva prioritat i ordre d'arribada.

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

És el mètode més important de la classe doncs gran part del altres mètodes es basaran en aquest per manipular el heap i per tant l'haurem d'estructurar i comprovar correctament.

2.1 compareTo

Per començar a definir què ha de fer el mètode compareTo és bona pràctica llegir la interfície "Comparable" que l'implementa:

La interfície ens indica que ha de retornar un enter negatiu, zero o un enter positiu si aquest objecte és menor que, igual que o més gran que el objecte especificat. Recordem també que hem definit una prioritat *null* com la més baixa, per tant haurem de realitzar comprovacions de les dues prioritats:

- Si other es null llançarem NullPointerException
- Si les dues prioritats son *null*, són iguals per tant retornem 0.
- Si la prioritat de l'element que estem comparant amb un altre és *null* retornem -1.
- Si la prioritat de l'altre element a comparar és *null* retornem 1.
- En cas de no tenir null serà el cas per defecte.

I si convertim les definicions a codi:

```
if(other==null) throw new NullPointerExcpetion("...")
if (this.priority == null && other.priority == null) {
    priority = 0;
} else if (this.priority == null) {
    priority = -1;
} else if (other.priority == null) {
    priority = 1;
} else {...}
```

Com que treballem amb genèrics, haurem de fer una crida al mètode compareTo d'aquell element prioritari que estem comparant:

```
priority = this.priority.compareTo(other.priority);
```

Fixem-nos que el paràmetre de tipus P al definir la classe Triplet està definit com Comparable<? super P>, el que significa que els elements prioritaris han de ser instàncies de tipus que siguin comparables a ells mateixos o als seus supertipus. Per exemple no podrem comparar una prioritat de tipus *string* amb un *int*.

Ara ja hem cobert la majoria dels possibles casos, solament ens falta considerar quan dos prioritats son iguals desempatar-les en funció del seu ordre d'arribada.

Comprovarem quin dels dos Triplets té un temps d'arribada menor:

```
if(priority == 0){
    if(this.timeStamp < other.timeStamp){
        priority = 1;
    }else{
        priority = -1;
```

Fins ara creiem que tenim el compareTo esta complet i funciona tal com està definit, però per assegurar-nos realitzarem una sèrie de tests amb múltiples tipus de prioritats i timeStamps.

3. HeapQueue

Una vegada tenim els elements que formaran el max-heap podem començar a construir la classe HeapQueue. Aquesta emmagatzemarà els seus elements en un ArrayList de triplets.

Definirem una variable anomenada nextTimeStamp que serà un valor únic per cada element afegit i determinarà l'ordre d'arribada, així podrem comparar dos elements de mateixa prioritat com hem vist en el compareTo (2.1).

3.1 Constructor

Com hem vist en les definicions inicials del Heap en un array (1.1), el primer element del array sempre serà vuit i començarem a afegir elements a la segona posició. Per tant al inicialitzar el ArrayList no el farem vuit sinó que amb una capacitat inicial de 1.

```
triplets = new ArrayList<Triplet<P,V>>(1);
```

I seguidament inicialitzarem els valors del size i del nextTimeStamp a 0.

3.2 Funcions auxiliars

Per implementar el mètode add i remove implementarem un seguit de funcions auxiliars molt simples però que ens facilitaran la implementació d'altres més complexes.

3.2.1 isEmpty

Comprovarà si el heap no conté cap element, no es més que una comparació amb el size.

```
return size == 0;
```

3.2.2 parentIndex

El parentIndex retorna el índex del pare sobre el fill que estem buscant. Com que ens movem entre nivells simplement hem de dividir entre dos el índex del fill, ja que es tracta d'un arbre binari. Al tractar-se d'una divisió d'enters no hem de comprovar si el índex es enter o senar ja que sempre arrodoneix.

```
return i/2;
```

3.2.3 left and rightIndex

De la mateixa forma que podem buscar el índex del pare ho podem fer amb els fills, en aquest cas si que hem de diferenciar casos, hem de accedir al nivell inferior i quan es tracta del fill dret li afegirem un 1.

```
return i*2;       //Left
return i*2+1;     //Right
```

3.2.4 swap

Com que hem de realitzar modificacions dins del heap movent elements entre diferents nivells, un mètode molt útil serà el swap, que intercanviarà els triples dels índexs passats per paràmetre.

```
var temp = triplets.get(i);
triplets.set(i, triplets.get(j));
triplets.set(j, temp);
```

Primer guardem el primer triplet temporalment, seguidament utilitzem dos set per canviar l'element a cada posició.

3.3 add()

El mètode add té la funció d'afegir un nou element al heap mantenint el ordre d'aquest, per tant primer l'haurem d'afegir i seguidament situar-lo en la seva posició corresponent.

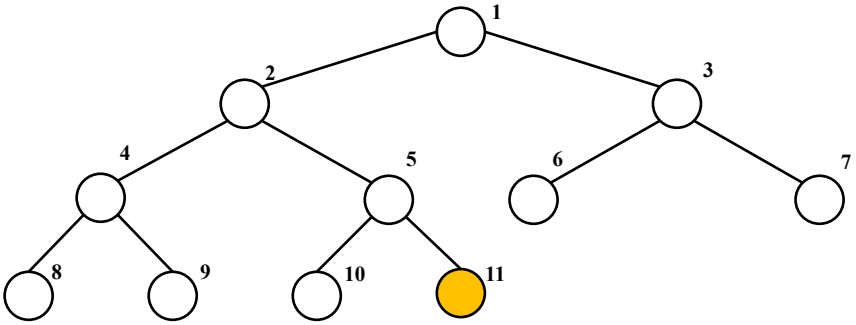
Ens passen per paràmetre la prioritat i el valor del nou element, així que primer haurem de construir un nou triplet. També incrementarem el valor d'arribada d'aquest element.

```
var triplet = new Triplet<>(priority, nextTimeStamp++, value)
```

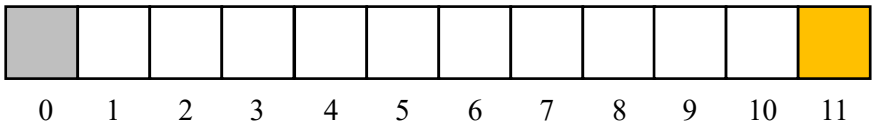
La forma més senzilla i eficient d'afegir el nou element és al final del arraylist a partir d'aquest punt haurem de fer comprovacions per situar-lo en la posició correcta. Primer l'afegim al final de la llista.

```
triplets.add(triplet);
```

Que si ho representem tant en el array com en el Heap ens queda:



[3] Max-Heap després d'afegir el nou element al final (marcat en groc)



[4] ArrayList després d'afegir el nou element al final de la llista (marcat en groc)

A partir d'aquí es poden donar dos casos, l'element nou té menor prioritat que el seu pare o el pare no existeix i per tant no haurem de fer res. I l'altre cas és quan l'element té major prioritat, en aquesta situació simplement haurem de intercanviar el triplet fill amb el pare fins que ens trobem el primer cas.

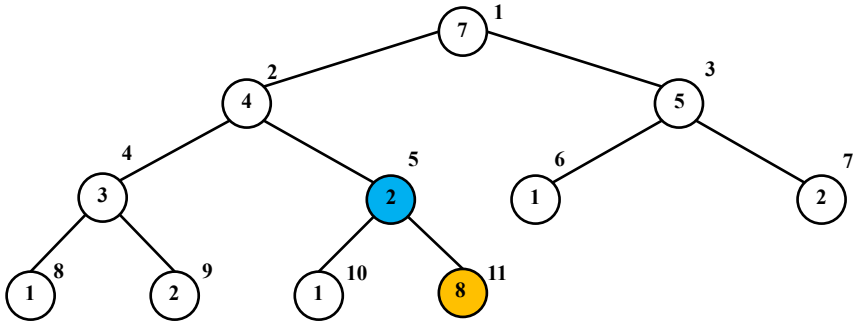
3.3.1 heapUp()

Cridarem a una funció auxiliar anomenada heapUp que ens realitzarà aquesta tasca. Primer haurem d'obtenir el índex del pare, i comprovar que és vàlid:

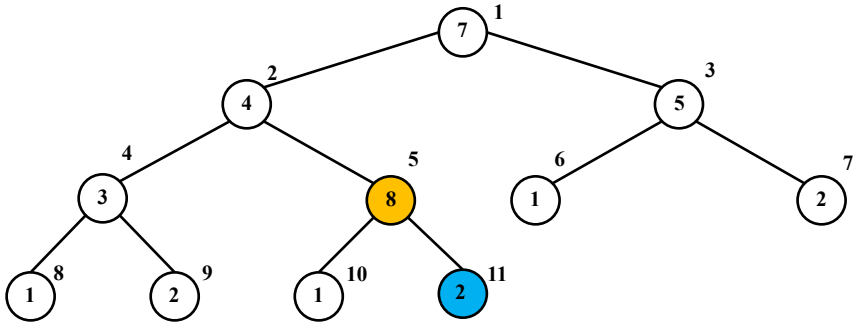
```
int parentIndex = parentIndex(i);
if(exists(parentIndex)){
```

Si no es compleix vol dir que no hi ha pare per tant no fem res.

A continuació representarem quan el Triplet fill serà intercanviat pel seu pare:

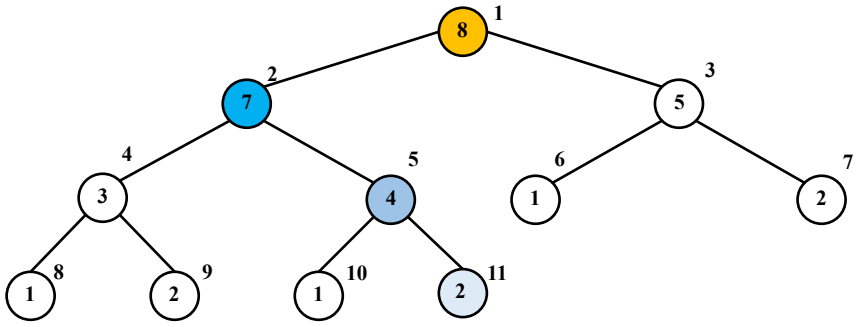


[5] Max-Heap comparant l'element 8 (groc) amb el seu pare (blau)



[6] Max-Heap intercanviat l'element 8 (groc) amb el seu pare (blau)

Seguint el mateix procediment el Max-Heap resultant tindria aquesta forma:



[7] Max-Heap resultant al afegir l'element amb prioritat 8 (groc)

Per cobrir el cas quan hem d'intercanviar pel pare primer hem d'obtenir els dos Triplets per comparar-los, com que ja tenim els indexes solament fem un get:

```
var son = triplets.get(i);
var father = triplets.get(parentIndex);
```

I la comprovació era si el fill tenia prioritat superior al pare:

```
if(son.compareTo(father) > 0){...}
```

Si el fill te major prioritat cridem al swap.

```
swap(i , parentIndex);
heapUp(parentIndex);
```

Després del swap farem una crida recursiva amb el índex del pare, ja que cada vegada pugem de nivell al comparar i al sortir incrementem el size.

```
size++;
```

3.4 remove()

El mètode remove s'encarregarà de retornar i eliminar el element amb major prioritat del Heap, sempre tindrem dos casos quan no hi ha elements i retornarem NSEE.

```
if(isEmpty()){
     throw new NoSuchElementException("Heap is empty.")
```

El segon cas requerirà retornar el primer element i reordenar el Heap per mantenir la seva estructura. Primer guardem l'element a retornar abans de modificar el Heap.

```
var removed = element();
```

La forma més eficient d'eliminar un element d'un arrayList es si aquest es troba en la última posició, per tant primer haurem de fer un swap entre el primer i últim element.

```
swap(1,size())
```

Ara podem eliminar l'últim Triplet.

```
triplets.removeLast();
```

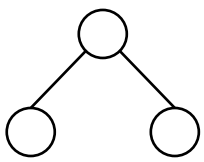
Per reordenar el Heap cridarem a una funció auxiliar, de manera similar que hem fet amb el add.

3.4.1 heapDown

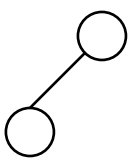
Com que tenim que reordenar comparant els fills amb el pare, obtindrem els índexs dels possibles fills de la dreta i esquerra amb les funcions auxiliars *Index*.

```
int rightIndex = rightIndex(i);
int leftIndex = leftIndex(i);
```

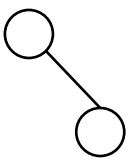
Ens podem trobar en quatre situacions :



[8] Dos fills



[9] Fill esquerra



[10] Fill dret

[11] Sense Fills

El fill que tingui major prioritat serà el que intercanviarem amb el Triplet que hem mogut, per evitar repeticions en el codi ens definirem una variable que contindrà el índex a intercanviar, ja que primer haurem de tractar en quin cas dels quatre anteriors ens situem:

```
int compareIndex;
```

Si el node té dos fills [8] primer comprovarem que aquests existeixen amb la funció exists, seguidament obtindrem els Triplets i haurem de comparar quin dels dos té major prioritat assignant aquest índex al compareIndex:

```
if(exists(rightIndex) && exists(leftIndex)){
    if(triplets.get(rI).compareTo(triplets.get(lI)) > 0){
        compareIndex = rightIndex;
    }else{
        compareIndex = leftIndex;
```

Si solament existeix el fill esquerre [9] o dret [10] el compareIndex serà el existent:

```
else if (exists(rightIndex)){
    compareIndex = rightIndex;
} else if (exists(leftIndex)){
    compareIndex = rightIndex;
```

Finalment en cas que no tingui fills ja haurem acabat i sortirem de la funció.

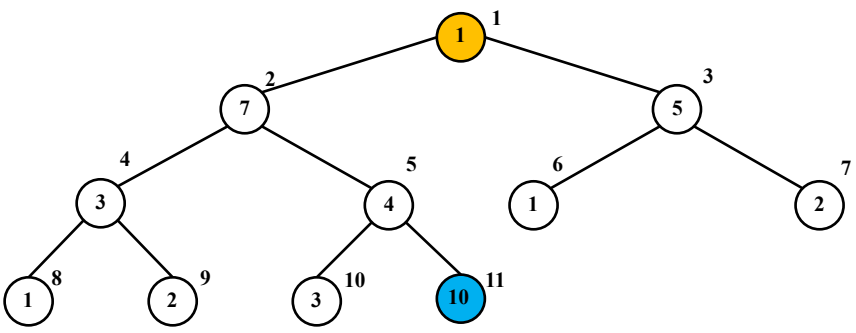
```
else {return;}
```

Ara solament ens queda realitzar el swap i tornar a cridar la funció heapDown per seguir ordenant el Heap. Compararem el node enviat per paràmetre i el resultant de les comprovacions anteriors, si el fill a compareIndex té més prioritat executarem el swap i continuarem la crida al índex intercanviat, sinó ja haurem acabat i no haurem de fer res.

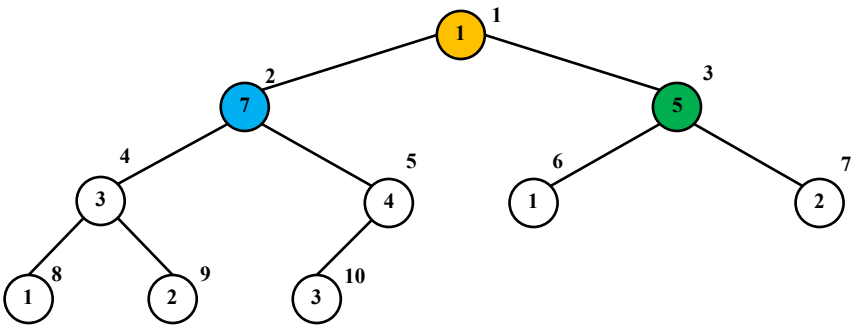
```
if (triplets.get(compareIndex).compareTo(triplets.get(i)>0{
    swap(compareIndex, i);
    heapDown(compareIndex);
```

Una vegada reordenat el Heap reduim el size i retornem l'element eliminat que teníem guardat a la variable removed.

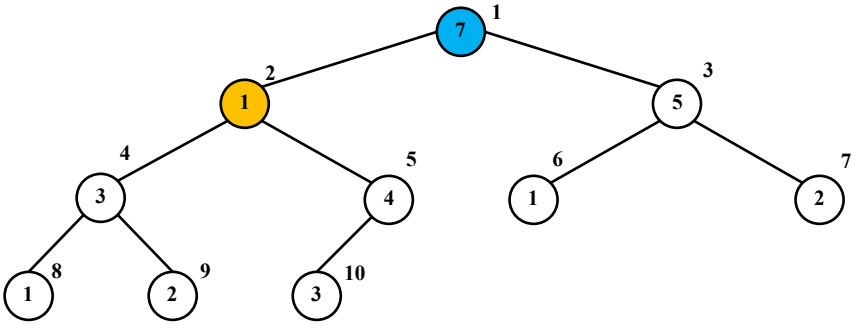
Podem veure com actua el remove amb el següent diagrama:



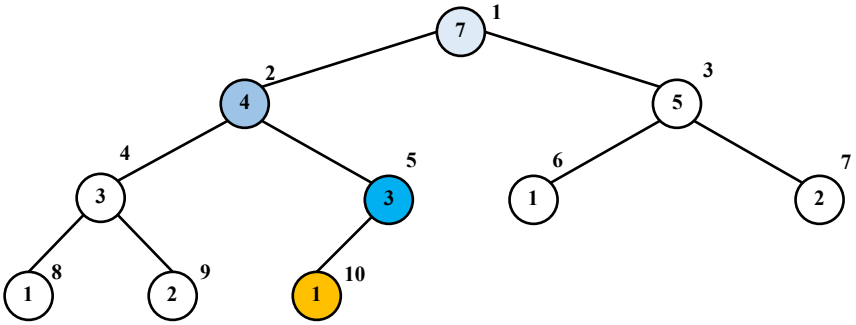
[12] Situació inicial on l'arrel (blau) es intercanviada amb l'últim element (groc) per posterior eliminació



[13] Comparació entre els fills de l'arrel actual (groc), on l'element amb més prioritat és el fill esquerre (blau)



[14] Heap després de realitzar el primer swap



[15] Resultat final del Heap després de realitzar un remove,
on els elements en blau representen aquells que s'han intercanviat.

3.5 element()

El mètode element no requereix gran explicació retorna el valor de l'arrel, i en cas de no existir llença NSEE.

```
return triplets.get(1).value;
```

3.6 size()

Retorna el nombre d'elements del heap, cal mencionar que el ArrayList sempre tindrà size+1 ja que en el heap la primera posició no la considerem però segueix estant allí.

```
return size;
```

4. Tests

Evidentment en tota implementació d'una classe amb diferents mètodes per manipular elements haurem de realitzar una sèrie de Tests per comprovar que obtenim els resultats esperats. Els elements que formaran els Triplets poden ser de diferents tipus i per tant no els podrem comprovar tots. Realitzarem una sèrie de tests amb objectes més senzills com poden ser Integers o Strings.