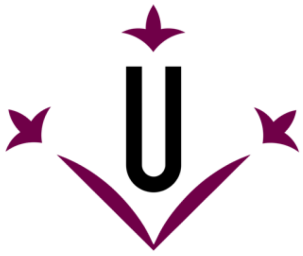

Universitat de Lleida

Heaps i cues amb prioritat

Boulhani Zanzan, Hamza

Serrano Ortega, Aniol

Data: 03/12/2023

Pràctica 3

Estructura de Dades

Escola Politècnica Superior

Índex

1. Introducció **1**

2. Definició i implementació **2**

2.1. Estructura del HeapQueue 3

2.2. Operacions del HeapQueue 4

2.3. Els mètodes privats *heapUp* i *heapDown* 5

2.4. Exemplificació visual de les operacions 6

3. Tests **12**

3.1. TripletTest 12

3.2. PriorityQueueTest 13

4. Conclusions **15**

Índex de Taules

Taula 1 Exemple de la llista del *HeapQueue* 4
Taula 2 Taula de la llista de la situació inicial de l'arbre exemple 6
Taula 3 Taula resultant de la llista després de la inserció 8
Taula 4 Taula resultant de la llista 11

Índex de Figures

Figura 1 Càlcul de l'índex dels fills 3
Figura 2 Exemple de la estructura del *HeapQueue* 3
Figura 3 Situació inicial de l'arbre exemple 6
Figura 4 Situació de l'arbre després d'afegir un node 7
Figura 5 Situació de l'arbre després de reordenar-lo amb el *heapUp* 7
Figura 6 Situació de l'arbre després d'eliminar el node més prioritari 9
Figura 7 Situació de l'arbre amb el node arrel substituït 9
Figura 8 Situació de l'arbre amb el node arrel substituït novament 10
Figura 9 Situació de l'arbre amb el node pare substituït 10
Figura 10 Situació final de l'arbre 11

1. Introducció

L'objectiu d'aquesta pràctica és el d'implementar una estructura de dades coneguda com a *munticle* o *heap* en anglès. Un *heap* és una estructura de dades tipus arbre que permet operacions eficients d'inserció, eliminació i cerca. En aquest cas, s'ha desenvolupat un tipus específic de *heap* anomenat *max-heap*.

A més, aquesta pràctica també es tracta el concepte d'una cua prioritària, on els elements amb més prioritat són els primers a ser eliminats (prioritat descendent). En cas d'empat en la prioritat, cada node té una marca de temps anomenada (*timeStamp*) que determina l'ordre d'arribada i resol el conflicte. Això permet gestionar de manera eficient situacions on diversos elements comparteixen la mateixa prioritat, assegurant que l'ordre d'extracció sigui coherent amb l'ordre d'inserció.

El principal repte resideix en el disseny i la implementació d'operacions bàsiques com la inserció (`void add()`) i l'eliminació (`V remove()`). A més, s'ha hagut de dissenyar una estructura del projecte adequada junt amb una sèrie de tests per a verificar la correcta implementació.

2. Definició i implementació

La implementació realitzada en aquesta pràctica es tracta d'una cua amb prioritat descendent, és a dir, el valor més gran és el més prioritari i, per tant, el primer a sortir de la cua. Cada element de la cua té associat un valor de prioritat que serveix per a ordenar els elements en la llista. A més, per a trencar el desempat en cas que la prioritat sigui la mateixa, es defineix una marca temporal (*timeStamp*) per a cada node de forma que dos nodes no poden tenir la mateixa marca temporal. D'aquesta manera, es defineix una tripleta de la següent forma:

```
Triplet(P priority, long timeStamp, V value)
```

Aquesta és la definició d'una tripleta de forma privada, ja que el *timeStamp* es calcula de forma privada en comptes de passar-li com a paràmetre. Per tant, per afegir elements només s'ha de fer passant-li la prioritat de tipus `P` l'element de tipus `V`.

D'altra banda, per a comparar tripletes s'ha definit un mètode `compareTo(Triplet<P, V> other)` que redefineix el de la interfície *Comparable*. Aquest mètode ha de retornar un enter i s'han de trencar els empats de la forma descrita anteriorment. Per tant, s'usa el mètode `compareTo(T o)` de *Comparable* i en cas que no hi hagi empat es retorna aquesta diferència entre les dues tripletes. En cas que hi hagi un empat, es trenca l'empat comparant els seus *timeStamp*. El fet que el *timeStamp* sigui menor significa que l'element és anterior i, per tant, més prioritari.

Les tripletes es mantenen en un *ArrayList*, denominat *triplets*, que constitueix la base de la implementació de *HeapQueue*. Aquesta cua de tipus *heap* es defineix com un arbre binari gairebé complet i balancejat cap a l'esquerra. En el cas d'un *max-heap*, es compleix que cada node és major o igual als seus nodes fills. Això garanteix que l'element amb la prioritat més alta, en aquest context, sempre estigui situat a l'arrel de l'arbre. A continuació es descriu l'estructura i funcionament d'aquest *HeapQueue*.

2.1. Estructura del HeapQueue

El *heapQueue* s'implementa utilitzant vectors mitjançant un *ArrayList*, on la relació entre els nodes pares i fills es pot determinar a través de les seves posicions en la llista. Per facilitar els càlculs, la primera posició no s'usa mai (`null`). Per un node en la posició i, els seus fills es troben a les següents posicions:

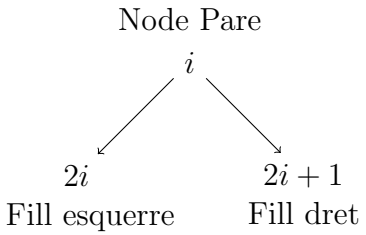

Figura 1: Càlcul de l'índex dels fills

Per exemplificar quina és l'estructura de la llista amb un arbre, se suposa el següent arbre on el contingut del node és la prioritat i el valor (`V`), mentre que l'exponent és la posició que ocupa en la llista. L'ordre d'inserció dels elements, és prioritzant l'esquerra:

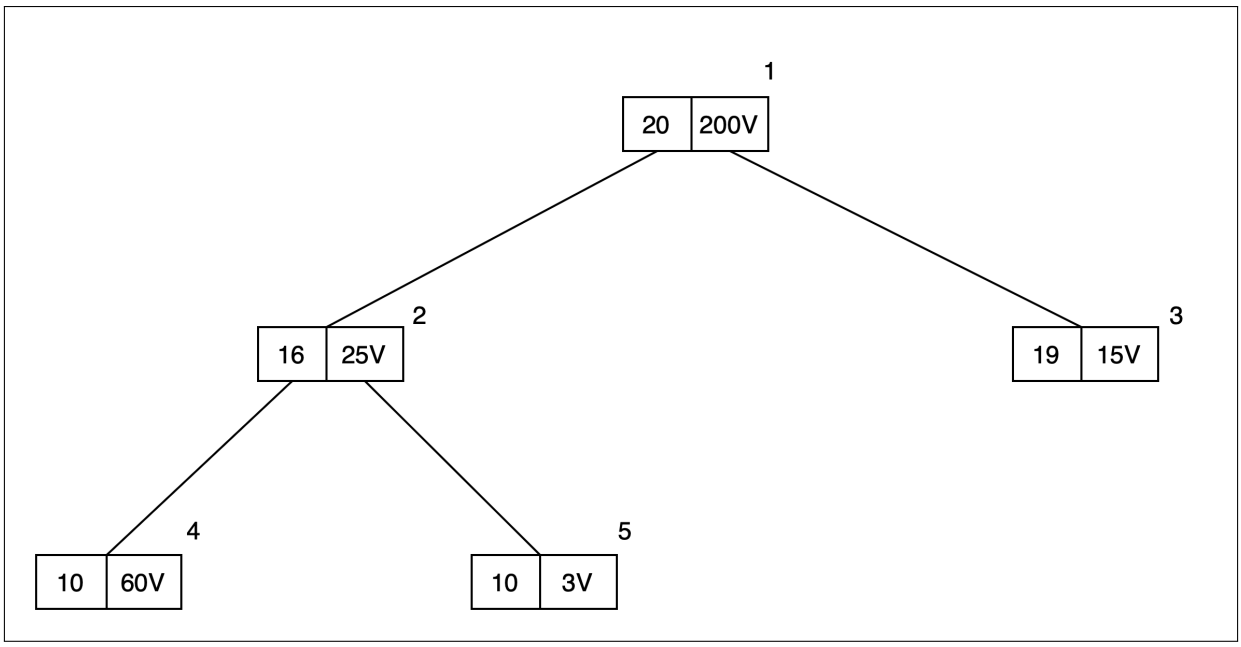

Figura 2: Exemple de la estructura del *HeapQueue*

D'aquesta manera, la llista del *HeapQueue* queda de la següent forma:

Posició	Prioritat	*TimeStamp*	Valor
0	`null`	-	`null`
1	20	0	200
2	16	1	25
3	19	5	15
4	10	2	60
5	10	4	3

Taula 1: Exemple de la llista del *HeapQueue*

Addicionalment, els valors representats són del tipus `Integer`, però aquests no tenen per què ser forçosament d'aquest tipus. Poden ser de qualsevol altre tipus de dades com `String`, `Boolean`, `Character`, `Floating-point`, etc.

2.2. Operacions del HeapQueue

Per a implementar el *HeapQueue* s'han implementat funcions per a poder manipular i consultar aquest *heap*. Aquestes operacions principalment són tres: inserció (*add*), eliminació (*remove*) i consulta (*element*). També, s'ha creat un mètode per a consultar la mida d'aquesta llista (*size*). Addicionalment, s'han creat operacions auxiliars per a implementar aquests mètodes.

- **Inserció (`void add(P priority, V value)`)**: Quan s'afegmeix un nou node, en cas que la prioritat o el valor sigui `null` es llança una `IllegalArgumentException`. En cas contrari, aquest s'insereix al final de la llista i s'incrementa la següent marca temporal pel següent element. Seguidament, es crida al mètode *heapUp* per reordenar el *heapQueue* en cas que sigui necessari. Aquest procés puja el node cap amunt de l'arbre fins que es compleixi la propietat de *max-heap*.

- **Consulta de l'element (`V element()`)**: Retorna l'element del tipus `V` més prioritari de la cua i amb *timeStamp* menor, és a dir, el node arrel. En cas que la llista estigui buida es llança excepció del tipus `NoSuchElementException`.

- **Eliminació (`V remove()`)**: Aquest mètode retorna l'element amb la màxima prioritat i l'elimina de la llista fent ús del mètode *element*. Per tant, primerament s'ha de guardar l'element de màxima prioritat per poder retornar-lo al final. A continuació, es substitueix el node arrel per l'últim node de la llista. Després, és crida al mètode *heapDown* per reorganitzar el *heap* i mantenir la propietat de *max-heap*. Finalment, es retorna el node eliminat.

- **Mida (`int size()`)**: Retorna la mida de la llista *triplets* menys 1, ja que al no usar-se la primera posició de la llista, s'ha de descomptar una posició. És a dir, en cas que la mida del *ArrayList* sigui igual a 1 ja es pot considerar buida.

Val a dir, que no és necessari comprovar si el node arrel és el que té la prioritat màxima amb el *timeStamp* menor en els mètodes *element* i *remove* perquè ja està garantit que està ben ordenat. Aquesta condició només es comprova quan es fa la inserció, ja que és l'única forma d'afegir nodes i en el mètode *heapUp* ja es tracta aquesta casuística fent ús del *compareTo* definit prèviament. De fet, els tests comproven que aquesta propietat és mantingui en tot moment.

2.3. Els mètodes privats *heapUp* i *heapDown*

Aquests mètodes implementen funcionalitats auxiliars que empren els mètodes *add* i *remove* respectivament. Aquests mètodes auxiliars mouen els nodes en l'arbre per a garantir que es compleix la propietat de *max-heap*, i en cas contrari, intercambiar nodes fins que així sigui.

- ***heapUp***: Aquest mètode s'utilitza quan s'afegeix un nou element a la cua. S'encarrega de reordenar el heap per assegurar-se que les propietats del max-heap es mantinguin, fent "pujar" l'element afegit cap a la seva posició correcta dins del heap.

- ***heapDown***: Després d'eliminar l'element de l'arrel (l'element amb la màxima prioritat), aquest mètode reordena la cua "baixant" l'element que substitueix a la cima cap a la seva nova posició correcta per mantenir les propietats del max-heap.

2.4. Exemplificació visual de les operacions

A continuació es proposa un exemple en el que se proposa una sèrie de operacions per veure com aquest arbre varia.

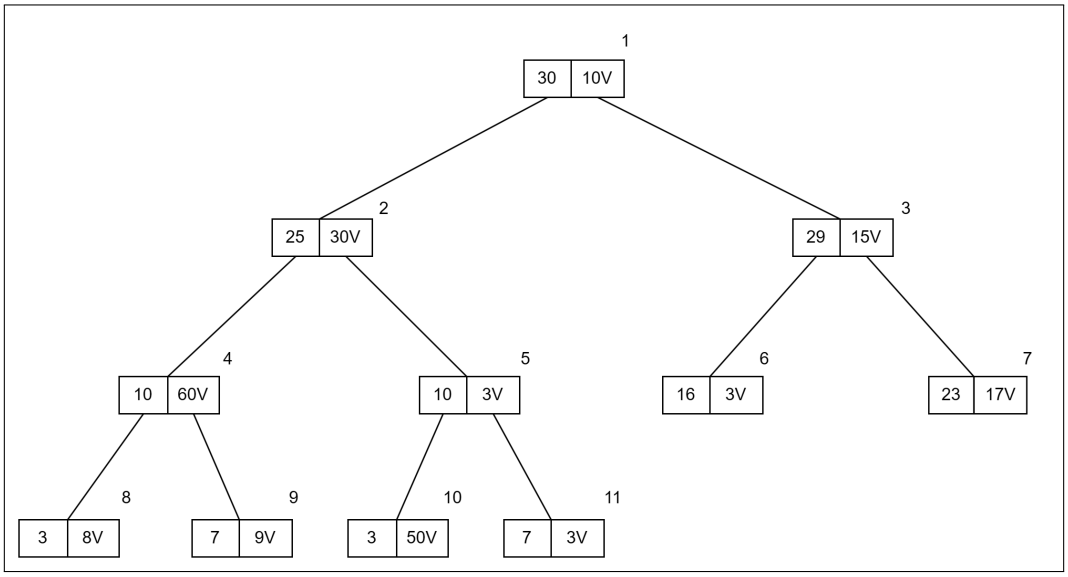

Figura 3: Situació inicial de l'arbre exemple

Índex	Prioritat	*TimeStamp*	Valor
0	`null`	-	`null`
1	30	5	10
2	25	2	30
3	29	4	15
4	10	0	60
5	10	8	3
6	16	7	3
7	23	3	17
8	3	10	8
9	7	6	9
10	3	1	50
11	7	9	3

Taula 2: Taula de la llista de la situació inicial de l'arbre exemple

A continuació, s'afegeix un node nou amb les següents instruccions:

```
1    HeapQueue<Integer, Integer> heapQueue = new HeapQueue<>();
2    heapQueue.add(29, 30);
```

Amb aquest nou node, l'arbre resulta així:

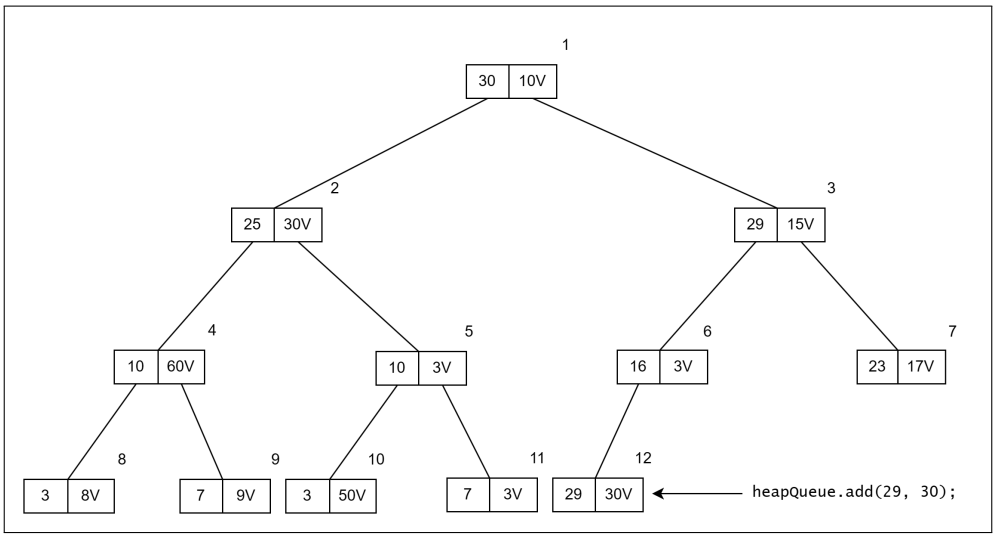

Figura 4: Situació de l'arbre després d'afegir un node

A continuació, es mostra com es crida al mètode *heapUp* per reordenar aquests nodes, ja que el pare té una prioritat inferior.

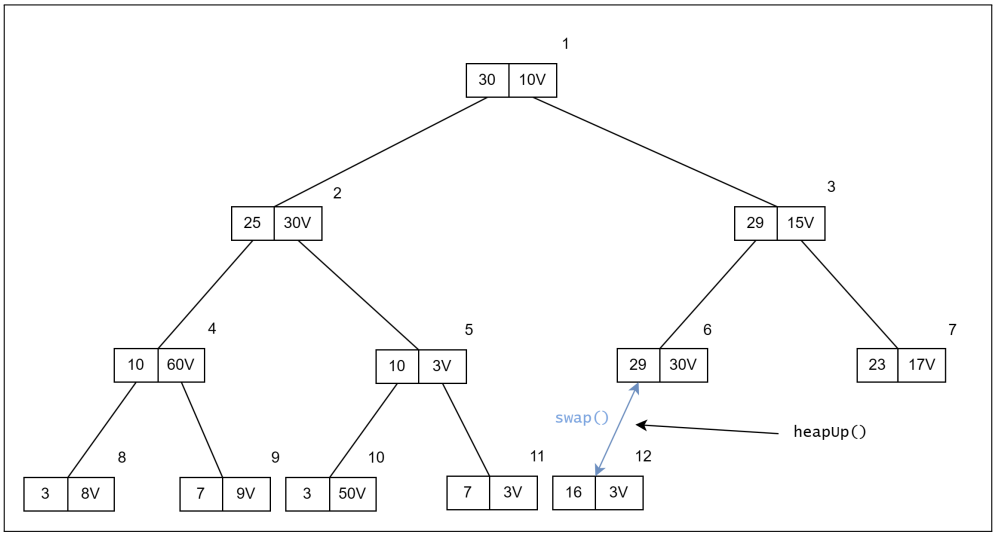

Figura 5: Situació de l'arbre després de reordenar-lo amb el *heapUp*

Tal com es mostra en la taula resultant, l'element ja està ben ordenat perquè malgrat empata en prioritat al seu node pare, el *timeStamp* del pare és menor ($4 < 11$) i, per tant, prioritari.

Índex	Prioritat	*TimeStamp*	Valor
0	`null`	-	`null`
1	30	5	10
2	25	2	30
3	29	4	15
4	10	0	60
5	10	8	3
6	29	11	30
7	23	3	17
8	3	10	8
9	7	6	9
10	3	1	50
11	7	9	3
12	16	7	3

Taula 3: Taula resultant de la llista després de la inserció

A continuació, a partir d'aquest arbre, es vol emprar el mètode *remove* i, per tant, primerament s'elimina el node arrel i es substitueix per l'ultim de la llista:

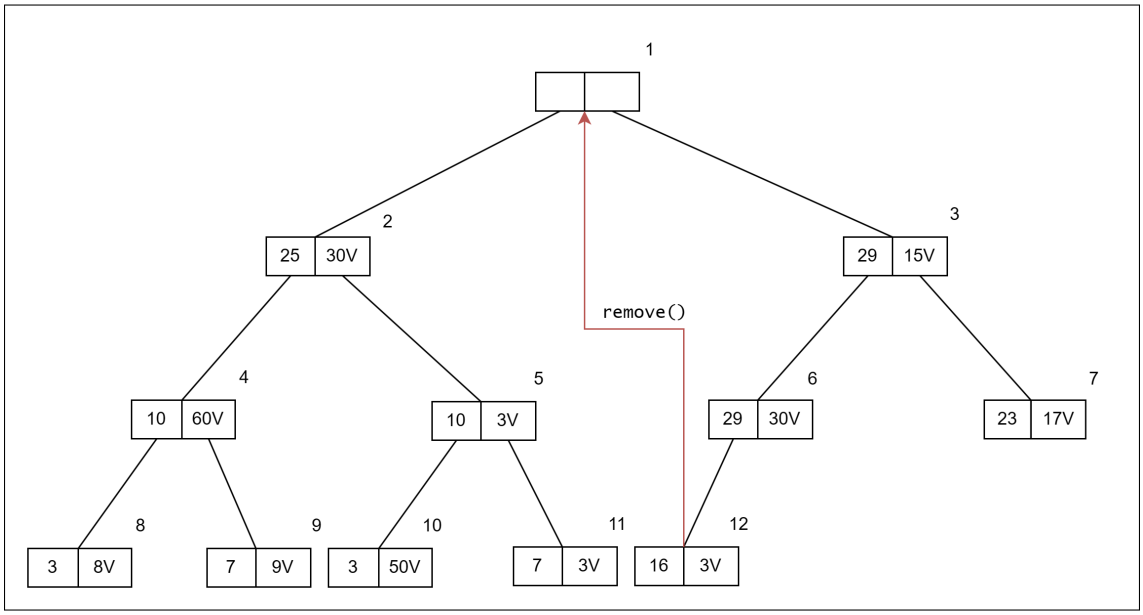

Figura 6: Situació de l'arbre després d'eliminar el node més prioritari

Un cop substituït el node arrel per l'ultim de la llista, s'ha de verificar si la prioritat del nou node arrel és la més alta, i en cas contrari, intercambiar nodes fins que es compleixi la propietat del *max-heap*:

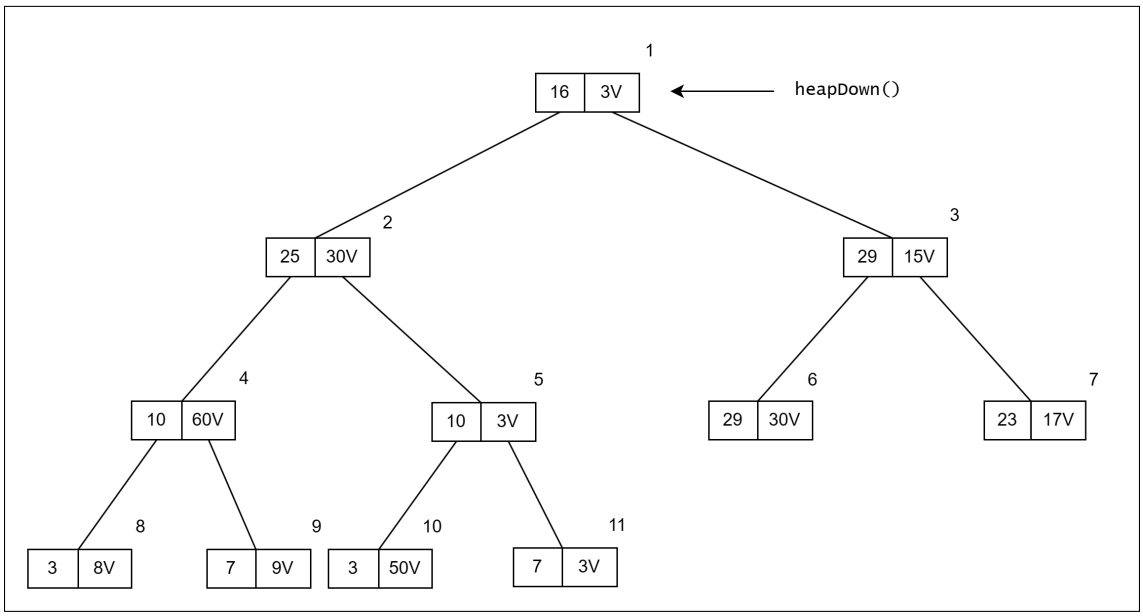

Figura 7: Situació de l'arbre amb el node arrel substituït

Seguidament, es crida al mètode auxiliar *getMaxPriorityChild* i es fa el *swap* amb el node més prioritari:

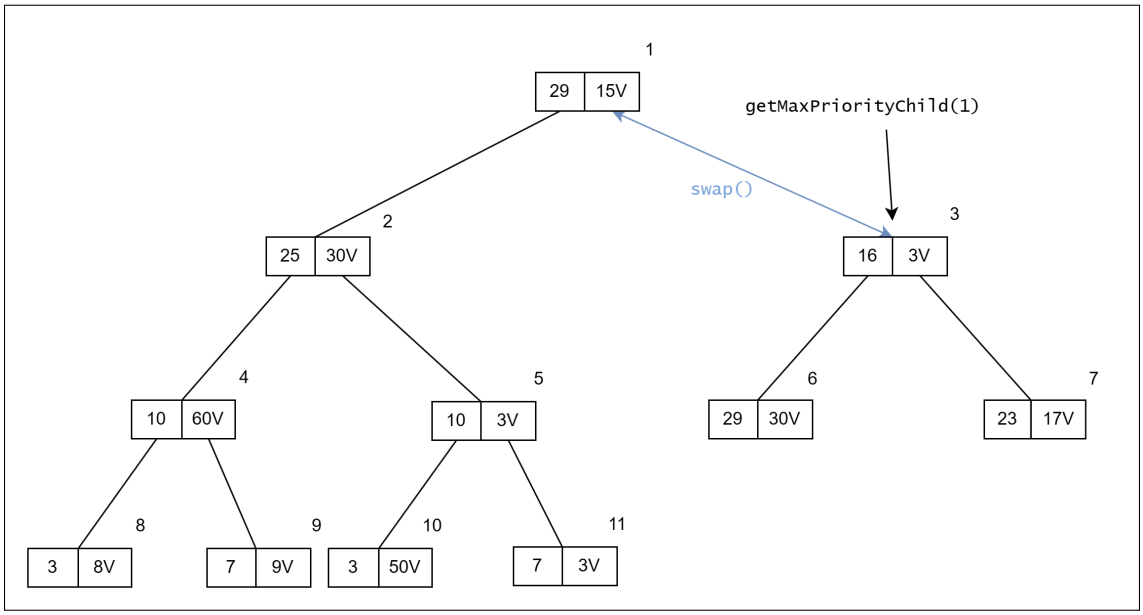

Figura 8: Situació de l'arbre amb el node arrel substituït novament

Novament, es comprova si el node pare és el que conté la prioritat més alta, i com que no és així, es torna a fer un *swap*:

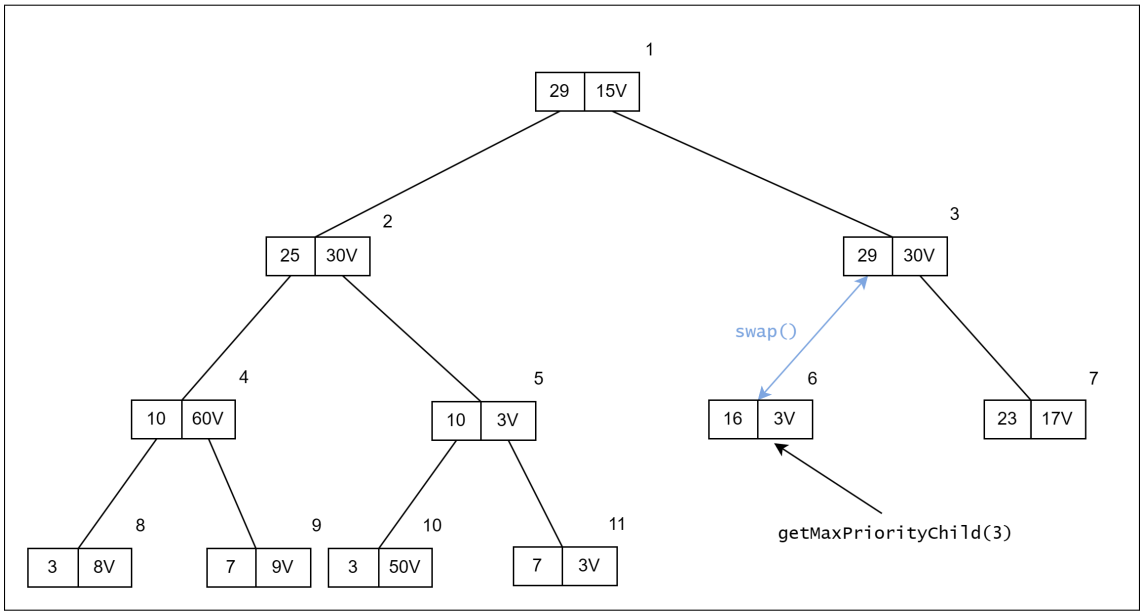

Figura 9: Situació de l'arbre amb el node pare substituït

D'aquesta manera després d'aplicar aquestes dues operacions l'arbre resulta així:

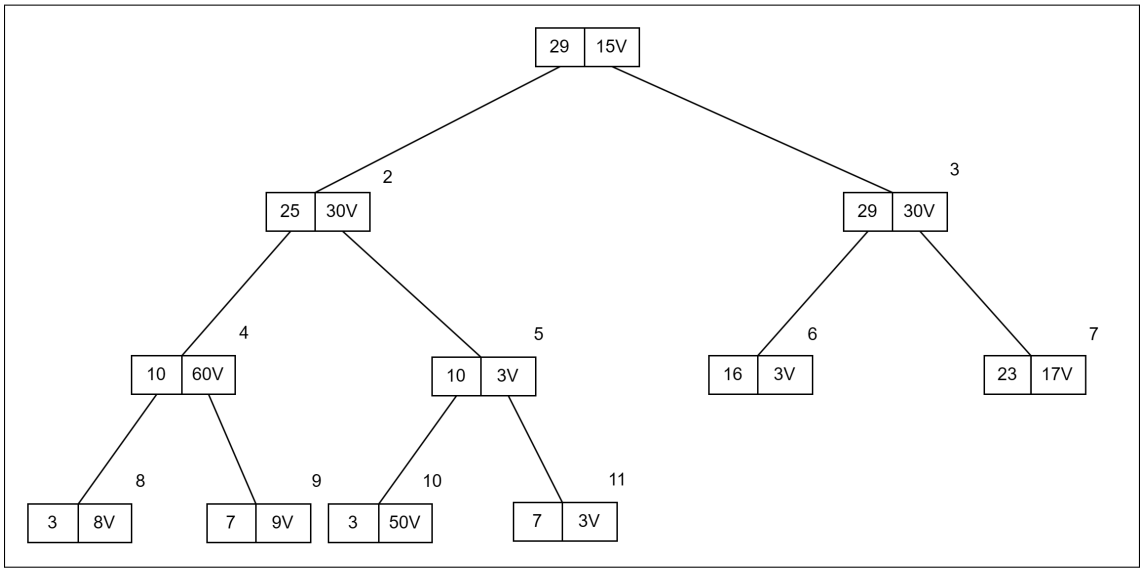

Figura 10: Situació final de l'arbre

Finalment, la taula resultant és aquesta:

Índex	Prioritat	*TimeStamp*	Valor
0	`null`	-	`null`
1	29	4	15
2	25	2	30
3	29	11	30
4	10	0	60
5	10	8	3
6	16	7	3
7	23	3	17
8	3	10	8
9	7	6	9
10	3	1	50
11	7	9	3

Taula 4: Taula resultant de la llista

3. Tests

Per a comprovar el correcte funcionament, és adient afegir una bateria de tests per a verificar que el comportament de la implementació és l'adequat. En aquest projecte s'han realitzat dues implementacions de tests diferents, el *TripletTest* i el *PriorityQueueTest*, el primer ja ha estat proporcionat en l'enunciat. Aquests tests s'ubiquen en els fitxers `TripletTest.java` i `PriorityQueueTest.java` respectivament.

3.1. TripletTest

En el cas del *TripletTest* serveix per a validar el correcte funcionament de la lògica de comparació en la classe `Triplet`, assegurant que els elements es prioritzen adequadament dins del *heap* segons la seva prioritat i el seu *timeStamp*. Aquesta validació és clau per a la integritat de l'estructura de dades de la cua de prioritat i per garantir que l'ordre d'extracció dels elements sigui consistent amb les expectatives i les especificacions de l'estructura *heap*.

A. Test de Comparació: Verifica que una Tripleta amb major prioritat es compara correctament respecte a una amb menor prioritat. Això és essencial per assegurar que la propietat de *max-heap* es mantingui, on un node sempre és major o igual als seus fills.

B. Test de Comparació de Prioritat: Similar al primer test, però amb enfocament directe en la prioritat. Aquest test assegura que la lògica de comparació de prioritats en `Triplet` funciona com s'espera.

C. Test de Comparació de *timeStamp*: Prova que quan dos tripletes tenen la mateixa prioritat, el *timeStamp* s'utilitza correctament per determinar l'ordre. Això és crucial per a trencar els desempats de prioritat.

D. Test d'Igualtat: Comprova que dos tripletes amb la mateixa prioritat i *timeStamp* es consideren iguals. Aquest cas és important per validar la consistència de la comparació.

3.2. PriorityQueueTest

Els tests en `PriorityQueueTest.java` estan dissenyats per validar el correcte funcionament de la cua de prioritat implementada amb el *heap*. Aquests tests inclouen la verificació de l'ordenació adequada dels elements en la cua, així com la gestió correcta dels casos límit, com ara la cua buida.

1. **Test d'Increment de Mida**: Comprova que la mida de la cua incrementa correctament quan s'afegeixen elements.

2. **Test d'Eliminació en Cua No Buida**: Verifica que l'element eliminat de la cua és el més prioritari segons l'ordre de prioritat del *max-heap*.

3. **Test d'Eliminació en Cua Buida**: Assegura que es llança una excepció quan s'intenta eliminar un element en una cua buida.

4. **Test de l'Element de la Cua No Buida**: Comprova que el mètode *element* retorna l'element correcte sense eliminar-lo de la cua.

5. **Test de l'Element en Cua Buida**: Verifica que es llança una excepció quan s'intenta accedir a l'element més prioritari d'una cua buida.

6. **Test de Mida de la Cua Buida**: Assegura que el mètode *size* retorna zero quan la cua està buida.

7. **Test de Mida Adequada**: Assegura que el mètode *size* retorna la mida actual de la cua.

8. **Test de Manteniment de la Propietat del Heap**: Comprova que després d'afegir elements, la cua manté la propietat del *max-heap* (veure exemple).

9. **Test per a Propietat del Heap amb Empat de Prioritat**: Comprova que després d'afegir elements amb prioritats iguals, la cua prioritza el primer element inserit fent ús del *timeStamp*.

10. **Test per a Propietat del Heap amb Empats de Prioritat Múltiples**: De forma similar al test anterior, en cas de múltiple empat de prioritat (10), s'ha de retornar l'element més prioritari que s'hagi inserit abans.

11. **Test de Comprovació de Inserció de Prioritat No Nula**: Comprova que si s'afegeix un node amb la prioritat nul·la es llança una excepció.

12. **Test de Comprovació de Inserció de Valor No Nul**: Verifica que en cas que la prioritat sigui `null` es llanci una excepció.

Aquests tests són fonamentals per assegurar que tots els mètodes públics del *HeapQueue* realitzen les operacions de la forma esperada.

4. Conclusions

En aquest laboratori s'ha explorat el concepte d'una cua prioritària i la relació que aquesta té amb un arbre. Es pot veure com definint una sèrie de regles senzilles permet implementar una cua prioritària amb operacions eficients. A més, també s'ha tractat com trencar desempats per a fer que la cua sigui consistent amb les operacions.

Tanmateix, s'ha vist la importància de realitzar tests i com aquests han de ser implementats prèviament a la implementació de la cua, ja que, en cas contrari resulta pràcticament impossible verificar si el comportament de la cua és l'adequat. Amb els tests es pot definir quin és el comportament esperat d'una forma senzilla, i d'aquesta manera facilitar la tasca d'implementació.

Un aspecte crític ha estat la definició del mètode *compareTo*, ja que aquest defineix com es trenquen els desempats, i en cas que estigui implementat de forma incorrecta pot propagar l'error fins a funcions més complexes. D'aquesta manera emprar els tests de les tripletes ha resultat ser molt útil.

Finalment, en aquest laboratori també s'ha hagut de crear el projecte des de zero, i per tant definir una estructura adequada. D'altra banda, en aquest projecte es pot veure la combinació dels conceptes teòrics vists a classe com els arbres i les cues es combinen per a realitzar una estructura de dades conjunta. D'aquesta manera, la comprensió d'aquests conceptes és clau per a realitzar una implementació correcta i consistent.

Acrònims

1. Test d'Increment de Mida *add_should_increase_size* 13

2. Test d'Eliminació en Cua No Buida

remove_on_non_empty_queue_should_return_correct_element 13

3. Test d'Eliminació en Cua Buida

remove_on_empty_queue_should_throw_nse_exception 13

4. Test de l'Element de la Cua No Buida

element_on_non_empty_queue_should_return_correct_element 13

5. Test de l'Element en Cua Buida

element_on_empty_queue_should_throw_nse_exception 13

6. Test de Mida de la Cua Buida *size_on_empty_queue_should_return_zero* 13

7. Test de Mida Adequada *size_should_return_correct_size* 13

8. Test de Manteniment de la Propietat del Heap

add_should_keep_heap_property 13

9. Test per a Propietat del Heap amb Empat de Prioritat

remove_should_prioritize_lower_timestamp_on_priority_tie 13

10. Test per a Propietat del Heap amb Empats de Prioritat Múltiples

remove_should_prioritize_lower_timestamp_on_multiple_priority_tie 14

11. Test de Comprovació de Inserció de Prioritat No Nula

add_null_priority_should_throw_ia_exception 14

12. Test de Comprovació de Inserció de Valor No Nul

add_null_value_should_throw_ia_exception 14

A. Test de Comparació *testSomeCompare* 12

B. Test de Comparació de Prioritat *testPriorityCompare* 12

C. Test de Comparació de *timeStamp* *testTimestampCompare* 12

D. Test d'Igualtat *testEquality* 12