



Pràctica S2 - Pirates del mediterrani

Grup 2

Pol Piñol Castuera (pol.pinol)
Inés Graells Sanz (ines.graells)
Marta Hernández Jimenez (marta.hernandez)
Roger Casas Aisa (roger.casas)

Índex

1	Introducció	1
2	Grafs	2
2.1	Disseny	2
2.2	Algorismes	2
2.2.1	Funcionalitats bàsiques	2
2.2.2	Cerca DFS	2
2.2.3	Cerca BFS	2
2.2.4	Càlcul del MST	3
2.2.5	Cerca del camí més curt (Dijkstra)	3
2.3	Resultats	4
2.3.1	Resultats del DFS	4
2.3.2	Resultats del BFS	5
2.3.3	Resultats del MST	5
2.3.4	Resultats del Dijkstra	6
2.4	Proves	7
2.5	Problemes	7
3	Arbres AVL	8
3.1	Disseny	8
3.2	Algorismes	8
3.2.1	Funcionalitats bàsiques	8
3.2.2	Recorreguts	9
3.2.3	Cerca per valor exacte	9
3.2.4	Cerca per rang	9
3.3	Resultats	9
3.3.1	Resultats d'inserir	9
3.3.2	Resultats d'eliminar	10
3.3.3	Resultats de recorregut per preordre	10
3.3.4	Resultats de recorregut per postordre	11
3.3.5	Resultats de recorregut per inordre	12
3.3.6	Resultats de recorregut per nivells	12
3.3.7	Resultats de cerca per valor exacte	13
3.3.8	Resultats de cerca per rang	14
3.4	Proves	14
3.5	Problemes	14
4	Arbres R	15
4.1	Disseny	15
4.2	Algorismes	15
4.2.1	Funcionalitats bàsiques	15
4.2.2	Visualització	16
4.2.3	Cerca per àrea	16
4.2.4	Cerca per proximitat	16
4.3	Resultats	17
4.3.1	Resultats d'inserir	17
4.3.2	Resultats d'eliminar	17
4.3.3	Resultats de la cerca per àrea	18
4.3.4	Resultats de la cerca per proximitat	19
4.4	Proves	19
4.5	Problemes	20
5	Taules	21
5.1	Disseny	21

5.2	Algorismes	21
5.2.1	Funcionalitats bàsiques	21
5.2.2	Cerca i histograma	21
5.3	Resultats	22
5.4	Proves	23
5.5	Problemes	24
6	Estructures de dades utilitzades	25
6.1	Llistes (ArrayList)	25
6.2	Cues (Queue)	25
6.3	HashMap	25
7	Conclusions	26
8	Referències	27

1 Introducció

El llenguatge escollit ha estat Java (versió 15.0.0) en la plataforma IntelliJ de JetBrains. La raó per la qual hem escollit aquest llenguatge i plataforma és que es tracta del mateix en què hem realitzat la pràctica i el projecte del primer semestre, així com les activitats d'avaluació contínua. Com ja ens trobem familiaritzats amb la plataforma i les possibilitats que ens ofereix programar en Java, i fins ara ens hem trobat molt còmodes implementant les esmentades pràctiques i projectes així, ens hem decantat per la comoditat de repetir llenguatge.

Per destacar l'elecció del llenguatge Java respecte al llenguatge de programació C, més habitual al primer any del grau, és bàsicament per la seva gestió automàtica de memòria. Java utilitza els anomenats *recol·lector de brossa*, que reciclen aquells objectes que no són assolibles des de cap variable del programa. Per tant nosaltres com a usuaris ens alliberem de fer totes aquelles tasques tan habituals en C com gestionar, reservar i eliminar la memòria. En resum, en Java tenim una gestió automàtica de la memòria i en C tenim una gestió manual.

Java ens ofereix una modularitat molt ben definida que ens permet treballar de forma clara en entorns com ara Github, on cada integrant pot tenir els seus mòduls, els seus blocs en branques diferents i evitar fàcilment qualsevol conflicte. També la facilitat de la plataforma IntelliJ a l'hora de detectar errors de codi i tant solucionar com visualitzar els possibles errors i/o resultats. Tot i que algunes consideracions i avantatges no són únics del llenguatge de programació Java, sumant-hi els gustos personals, ens fa més fàcil l'elecció final del llenguatge el qual utilitzarem per al desenvolupament d'aquesta pràctica.

2 Grafs

2.1 Disseny

Creiem necessari comentar que no hem seguit l'ordre d'estructures de dades que proposava la pràctica ja que, de cara a la implementació de la llista d'adjacències, hem fet servir un Hashmap, on les keys són els nodes fixats i els values són els nodes adjacents continguts en un ArrayList. Aquesta decisió s'ha fet perquè ja existeix una classe de Java (HashMap) que ens facilita la feina considerablement.

De cara a la implementació d'aquesta estructura de dades, vam considerar els pros i contres de generar una llista d'adjacència respecte generar una matriu d'adjacència. La nostra conclusió va ser que una matriu d'adjacència Figura 1, tot i ser més fàcil d'entendre a nivell visual, conté molts zeros que podrien fàcilment ser ignorats i per tant la converteix en una estructura molt poc eficient (de cara a reservar memòria), mentre que en la llista d'adjacències simplement guardem els valors que són rellevants i per tant reduïm molt el cost de memòria. En la pràctica vam implementar les dues formes per representar un graf, tot i que finalment ens vàrem decantar per la llista d'adjacència per les seves avantatges ja comentades.

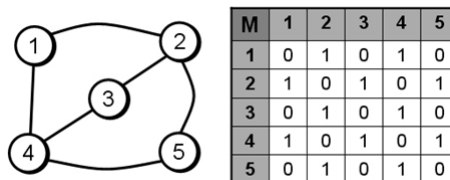


Figura 1: Exemple de la representació d'un graf com a matriu d'adjacència

2.2 Algorismes

2.2.1 Funcionalitats bàsiques

Es demana que el codi defineixi i implementi la representació d'un graf per a desar les dades dels datasets corresponents de forma eficient.

Per crear inicialment un graf, començarem amb un graf buit, és a dir, crearem un HashMap (explicat als annexs) que guardarà tots els nodes en claus per després guardar als valors de cada clau els nodes adjacents amb un ArrayList. Quan s'han creat totes les claus, anirem llegint les diferents arestes del dataset, i anirem accedint a les claus i canviant / actualitzant la llista que conté els nodes adjacents. En el cas que es vulgui eliminar un node, s'haurà d'eliminar la seva clau corresponent i buscar en totes les seves adjacències i eliminar-lo també.

2.2.2 Cerca DFS

Es demana oferir una opció per llistar llocs d'interès mitjançant l'algorisme Depth-First Search.

En aquest algorisme hem expandit i guardat tots els nodes de forma recursiva a partir d'un node origen, anant accedint tota l'estona als següents nodes adjacents abans de recular, tenint en compte sempre els nodes visitats, marcant-los en una llista. En el fons, és molt semblant al disseny del primer semestre del backtracking. Cal destacar que guardarem aquests nodes sempre i quant siguin llocs d'interès tal i com demana l'enunciat.

2.2.3 Cerca BFS

Es demana oferir una opció per llistar llocs de perill mitjançant l'algorisme Depth-First Search.

En aquest algorisme hem expandit el graf per nivells, és a dir, hem creat una cua per anar guardant els nodes adjacents recorrent sempre els més pròxims al node origen i deixant pels últims els nodes adjacents que anem trobant. En general, visitem els adjacents pròxims al node origen abans que els seus vèrtexs adjacents. Cal destacar que guardarem aquests nodes sempre i quant siguin llocs de perill tal i com demana l'enunciat.

2.2.4 Càlcul del MST

Es demana afegir la generació d'una carta nàutica universal, és a dir, el mínim graf que contingui tots els nodes del graf original.

Per fer aquest algorisme, hem pensat en l'algorisme Greedy ja presentat al primer semestre. Més en concret, ens hem basat en l'algorisme Kruskal. Crearem un graf nou completament buit, i ordenarem les arestes segons la seva distància. Seguidament anirem afegint les arestes al graf buit amb les condicions que els nodes no estiguin ja continguts en el graf i que no generi un graf desconnectat. Anirem repetint aquest procés fins que obtinguem un graf connectat i amb tots els nodes que el graf original.



Figura 2: Resultats obtinguts pel nostre algorisme i per graphonline.ru respectivament

2.2.5 Cerca del camí més curt (Dijkstra)

Es demana un algorisme que trobi la ruta més òptima en dos llocs d'interès. És a dir, es demana trobar el camí més curt que eviti llocs perillosos sempre que sigui possible.

Per desenvolupar aquest algorisme, ens hem basat l'algorisme Dijkstra vist a classe. Donat un node inicial i un node final, l'algorisme trobarà el camí més curt per arribar de l'inici al final. Això ho farà anotant el camí que es va recorrent i la seva distància. Per fer-ho, anirem explorant els camins més curts des del node origen fins a tots els altres nodes del graf. Quan s'obté el camí més curt des del node origen fins als altres nodes del graf, l'algorisme implementat s'atura i es calcula finalment el camí més curt del node inicial al node final. Per evitar els llocs perillosos hem puntuat amb una distància molt quan el node per on es passava era perillós.

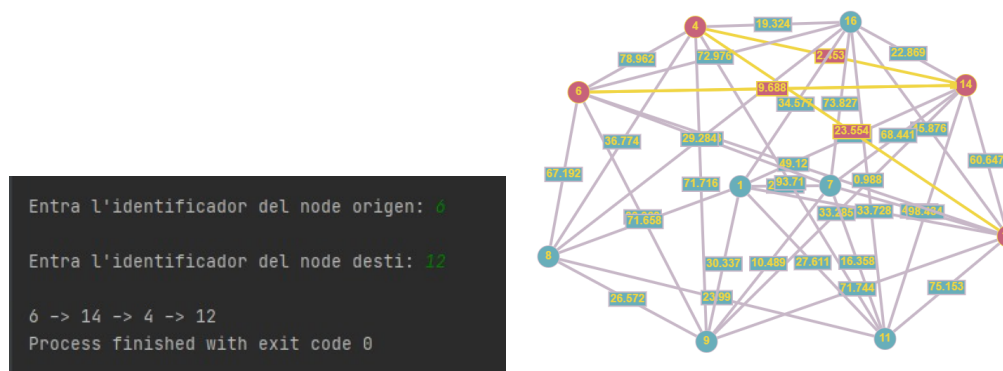


Figura 3: Resultats obtinguts pel nostre algorisme i per graphonline.ru respectivament

2.3 Resultats

Un dels pocs anàlisis de rendiment ha estat el de mesurar el temps que transcorre en completar-se l'algorisme, ja que és una bona manera de comprovar com el número de dades afecta de forma exponencial, lineal, o d'altres formes al temps d'execució dels nostres algorismes, i ens dona una vaga idea de si el codi està ben optimitzat o no. Recordem que els temps d'execució és aquell temps l'algorisme tarda en trobar la solució i no el temps per imprimir-la. Els eix X serà la quantitat de nodes i el eix Y els segons.

2.3.1 Resultats del DFS

Observem que el cost de l'algorisme implementat és aproximadament $O(n)$, segons les gràfiques obtingudes (Figura 5) i el detall de la taula (Figura 4). Això és pel fet que, com ja hem explicat, l'algorisme Depth First Search es va movent entre nodes adjacents fins a arribar a l'últim nivell, que després retornarà amb un backtracking per agafar el possible següent node. Més en detall, aquest algorisme en grafs no només depèn de la quantitat de nodes que existeixen, sinó de les seves relacions. Per tant, podríem afirmar que realment el cost seria $O(n + a)$, on n seria el nombre de nodes i a el nombre d'arestes. Cal destacar que per implementar aquest algorisme hem fet ús de la classe *ArrayList* i del seu mètode *add* que té un cost $O(1)$.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	5	0.0
XS	10	0.0
S	50	0.0.0009686
M	100	0.0030201
L	500	0.0657583
XL	1000	0.1306773
XXL	5000	10.019692

Figura 4: Resultats obtinguts per l'algorisme implementat

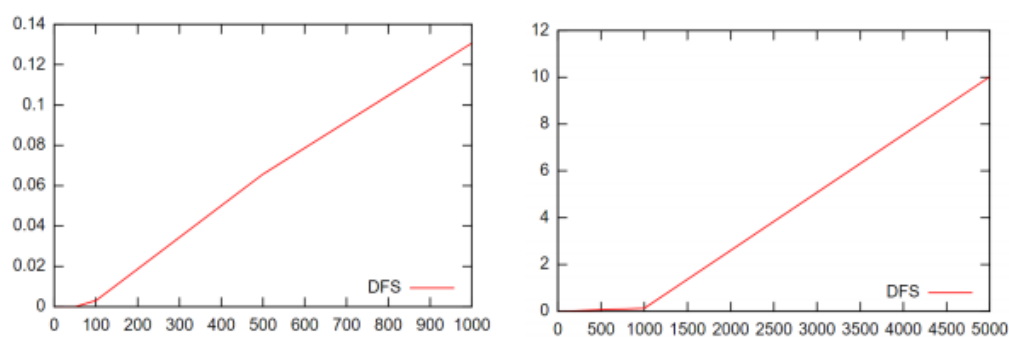


Figura 5: Gràfiques dels resultats utilitzant gnuplot.respawned.com

2.3.2 Resultats del BFS

Observem que el cost de l'algorisme implementat és aproximadament $O(n^2)$, segons les gràfiques obtingudes (Figura 7) i el detall de la taula (Figura 6). Si analitzem l'algorisme de Breadth First Search ens adonem que, el seu cost ideal no hauria de ser en cap cas $O(n^2)$, al no tenir cap sentit de recórrer un graf de forma tan poc bona. El cost teòric que ens hauria d'haver sortit hauria d'haver sigut el mateix que el del DFS, és a dir, $O(n + a)$ on n seria el nombre de nodes i a el nombre d'arestes. Això és degut a un error en la implementació de l'algorisme, al no utilitzar l'estructura de dades corresponent (Classe *Queue* de Java) i forçar una cua a través de la classe *ArrayList*. Observem que en la implementació fem servir el *remove* de la classe *ArrayList*, que té un cost de $O(n)$ a l'haver de reordenar tots els elements.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	5	0.0
XS	10	0.0
S	50	0.0
M	100	0.000966
L	500	0.0209796
XL	1000	0.1048263
XXL	5000	11.8335301

Figura 6: Resultats obtinguts per l'algorisme implementat

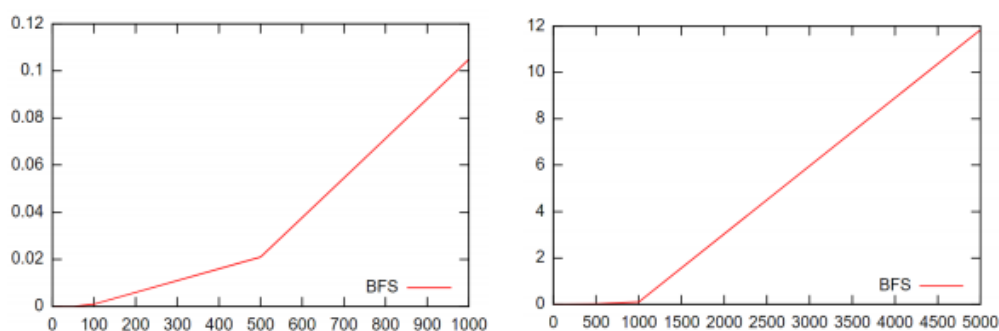


Figura 7: Gràfiques dels resultats utilitzant gnuplot.respawned.com

2.3.3 Resultats del MST

Observem que el cost de l'algorisme implementat és aproximadament $O(n^2)$, segons les gràfiques obtingudes (Figura 9) i el detall de la taula (Figura 8). Analitzant el algorisme Minimum Spanning Tree

Kruskal utilitzat, ens adonem que idealment el seu cost hauria de ser $O(a + n \log a)$ on n seria el nombre de nodes i a el nombre d'arestes. Aquesta diferència entre el cost del algorisme implementat i el cost teòric és degut a que hem dissenyat el graf utilitzant un *HashMap*. Com ja hem explicat, el algorisme MST Kruskal parteix d'un graf buit d'arestes per anar omplint les arestes amb menys pes. És aquí on s'incrementa el cost del nostre algorisme a mesura que existeixen més nodes i arestes. Creiem que al anar accedint tota l'estona al *HashMap* i fent mètodes com add, remove, contains i put fa que el cost, quan hi han molts nodes i arestes, en el pitjors dels casos sigui $O(n)$, i al global del algorisme $O(n^2)$.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	5	0.0019963
XS	10	0.0029917
S	50	0.0029929
M	100	0.002991
L	500	64.0872
XL	1000	+600.0 (*)
XXL	5000	+600.0 (*)

Figura 8: Resultats obtinguts per l'algorisme implementat

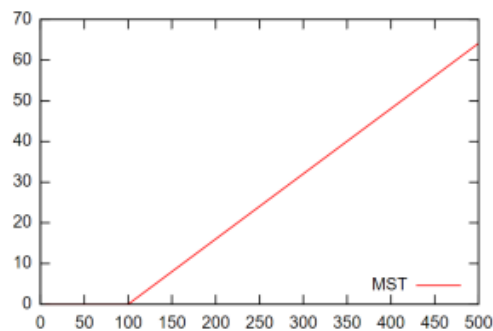


Figura 9: Gràfica dels resultats utilitzant gnuplot.respawned.com

2.3.4 Resultats del Dijkstra

Observem que el cost de l'algorisme implementat és aproximadament $O(n^2)$, segons les gràfiques obtingudes (Figura 11) i el detall de la taula (Figura 10). El cost obtingut de l'algorisme Dijkstra era l'esperat, al ser un algorisme que ha de realitzar $O(n^2)$ operacions per determinar la longitud del camí més curt entre dos nodes. Indicar que hem obtingut més taules i gràfiques per diferents camins entre diferents nodes, però en general, donaven els mateixos resultats i costos.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	5	0.00026
XS	10	0.0009692
S	50	0.0289226
M	100	0.7025391
L	500	+600.0 (*)
XL	1000	+600.0 (*)
XXL	5000	+600.0 (*)

Figura 10: Resultats obtinguts per l'algorisme implementat

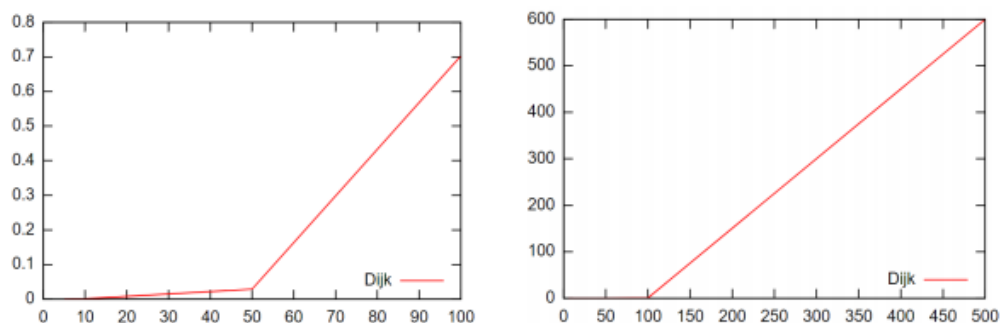


Figura 11: Gràfiques dels resultats utilitzant gnuplot.respawned.com

Cal indicar finalment que fins al fitxer XL el temps de creació dels Grafs era gairebé negligible, sent el temps de creació per XL 2.20783 segons i pel XXL 97.34008 segons. A més, el asterisc (*) en les taules obtingudes indica que el temps d'execució del algorisme ha superat els 10 minuts.

2.4 Proves

De cara a comprovar el correcte funcionament dels grafs implementats, hem fet ús de la pàgina web proporcionada pel professor de la assignatura (graphonline.ru), ja que aquesta ens permetia visualitzar no només els nostres mapes de nodes, sinó també les rutes generades per diferents algorismes, i així poder comparar amb els resultats obtinguts pels nostres algorismes.



Figura 12: Resultats del algorisme Dijkstra del dataset XS obtinguts per graphonline.ru

Per altra banda, hem fet ús de datasets personalitzats amb imatges d'aquests mostrades per consola, amb tal de poder tenir resultats dels quals en sabem les solucions, i així facilitar la cerca i solució d'errors.

2.5 Problemes

Un dels problemes que més temps ens va costar de solucionar va ser de que a vegades, el nostre algorisme havia "recorregut" tots els nodes d'una ruta, però en veritat havia activat els booleans de cada node respecte que els havia visitat sense mirar si estaven realment connectats, i això ens generava que enlloc de tenir una ruta sovint ens quedàvem amb dues rutes que no s'acabaven de connectar per una aresta. Aquest cas es donava perquè no comprovàvem el número d'arestes i que els nodes, tot i haver estat visitats, estiguessin realment connectats i generant una única ruta contínua.

L'algorisme Dijkstra també ens ha donat bastantes complicacions, al haver de calcular sempre per quina ruta seria la més òptima i saber quan tornar enrere i anar per una altra adjacència va suposar-nos un repte a l'hora d'implementar-lo.

3 Arbres AVL

3.1 Disseny

De cara a la implementació d'aquesta estructura de dades, hem considerat que els arbres que anem a dissenyar es basaran en un únic node principal que ens farà d'arrel de l'arbre. A partir d'un únic node podem definir un arbre sencer. Estem en un cas de dissenyar un arbre binari (nodes amb com a molt grau 2) auto-balancejat.

Perquè aquest raonament funcioni, hem creat una classe Node que guardarà sempre a dos fills en un array bidimensional: el fill esquerre i el fill dret (existeix la possibilitat que apunti a null). A més, es guardarà el node pare per facilitar certs algorismes dissenyats. També cada node obtindrà un factor de balanceig que inicialment serà 0.

Cal destacar que, encara que sempre ens guardem dos fills en cada node, calcularem en tots els casos que aquests fills existeixin abans de fer qualsevol operació. És a dir, dins de l'estructura de dades, tractem diferent els nodes depenent si són nodes fulla (els dos fills són null) o bé si tenen grau 1 o grau 2 (algun fill o bé els dos existeixen).

3.2 Algorismes

3.2.1 Funcionalitats bàsiques

Es demana que existeixi una opció per poder afegir i eliminar tresors en l'estructura de dades dissenyada.

La idea per crear un algorisme per inserir un node es basa en anar comparant de mica en mica el node a inserir amb el arbre. Per anar accedint a l'arbre, hem de fer-ho tal que anirem al fill esquerra si el valor és més petit i anirem al fill dret si el valor és més gran. Llavors, recursivament apliquem aquest algorisme fins a trobar un node fill null on guardarem el node a inserir. A més, com estem tractant d'arbres binaris auto-balancejats, haurem de balancejar els pares (i els pares dels pares) del node inserit fins que arribem a l'arrel o fins que el sub-arbre estigui balancejat.

L'algorisme per eliminar un node és més complicat, ja que no és tan senzill com anar inserint. Primer de tot hem de localitzar el node a eliminar, per tant, haurem de recórrer l'arbre binari fins a trobar aquell node. Un cop localitzat haurem de plantejar diferents situacions.

Si el node a eliminar és un node fulla, només l'haurem de borrar i balancejar els cops que facin falta als seus pares.

Si el node a eliminar és un node amb els dos fills, pujarem el primer node dret sense fill esquerra al node eliminat, per així agafar el node esquerra i afegir-lo al fill esquerra buit mencionat. Després sempre haurem de mirar si hem produït un desbalanceig i arreglar-ho recursivament els cops que facin falta.

Per últim, en el cas que s'elimini un node amb un únic fill, només caldrà pujar el fill al node eliminat i mirar si s'ha de balancejar el seu pare.

Indicar que quan parlem de balancejar, ho fem seguint les indicacions donades a classe i als apunts. És a dir, hem construït un factor de balanceig per cada node que s'anirà actualitzant sempre que s'inserixi o s'elimini un node. Igualment cal destacar que no el calculem a través de les alçades dels dos sub-arbres, sinó que sempre que inserim un node, aquest parteix de factor de balanceig 0 i farà canviar recursivament els factors de balanceig del seu pare i pares recursius. Per balancejar, hem dissenyat els algorismes-rotacions vistes a classe: Left-Left, Right-Right, Left-Right, Right-Left.

3.2.2 Recorreguts

Es demana que existeixi diferents formes de llistar l'inventari dels pirates. Aquestes formes de llistar s'anomenen recorreguts i es demanen 4 tipus: recorregut en preordre, postordre, inordre i per nivells.

- Preordre: Hem dissenyat l'algorisme d'acord que es mostra un node, i a continuació, es mostra el prordre dels seus fills. Apliquem aquest algorisme recursivament des de l'arrel.
- Postordre: Hem dissenyat l'algorisme igual que l'anterior però el node mostrarà primer el postordre dels seus fills.
- Inordre: Aquest recorregut l'hem dissenyat tal que nostra el node al mig dels seus dos fills, recursivament.
- Nivells: Finalment aquest recorregut es basa en mostrar el nodes d'acord al mateix nivell de profunditat recursivament, semblant al BFS dels Grafs.

3.2.3 Cerca per valor exacte

Es demana que es dissenyi un algorisme per cercar un tresor en concret de forma eficient. Donat un valor, hauríem d'haver dissenyat un algorisme tal que començant per l'arrel es va recorrent l'arbre segons si el valor és més gran (accedim node dret) que al node que estem comparant o més petit (accedim fill esquerre). Aplicaríem aquest algorisme recursivament fins a trobar exactament el valor demanat. En canvi, hem agafat l'algorisme de recorregut per nivells i hem comparat si anàvem trobant el valor mentre feïem el recorregut (forma no òptima al no aprofitar l'estructura de dades creada).

3.2.4 Cerca per rang

Es demana que es dissenyi un algorisme per cerca tresors en un rang de valors en concret de forma eficient. Igual que l'algorisme anterior, el dissenyat no correspon amb l'algorisme òptim aprofitant els avantatges d'un arbre binari auto-balancejat. Hem dissenyat un algorisme que al recórrer l'arbre per nivells, comprova a cada node si es troba en el rang de valors donats i l'afegeix a una llista per mostrar-lo en finalitzar el recorregut.

3.3 Resultats

3.3.1 Resultats d'inserir

Observem que el cost de l'algorisme implementat és aproximadament $O(\log n)$, segons les gràfiques obtingudes (Figura 14) i el detall de la taula (Figura 13). És el resultat esperat, i en termes de cost, és un resultat òptim. Això és degut a que estem utilitzant una estructura de dades optimitzada per ordenar informació; al inserir, partim des de l'arrel i anirem descartant sub-arbres depenent si el valor a inserir és més gran o més petit al node comparat, fet que fa que l'algorisme tingui un cost $O(\log n)$. A més, com estem parlant d'arbres auto-balancejats, no trobarem mai el possible pitjor cas que ens trobem en una llista amb els valors ordenats que fa que el cost sigui $O(n)$.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.00008
XS	64	0.00012
S	512	0.00018
M	4096	0.00026
L	32768	0.00367
XL	262144	0.06098
XXL	2097152	0.20984

Figura 13: Resultats obtinguts per l'algorisme implementat

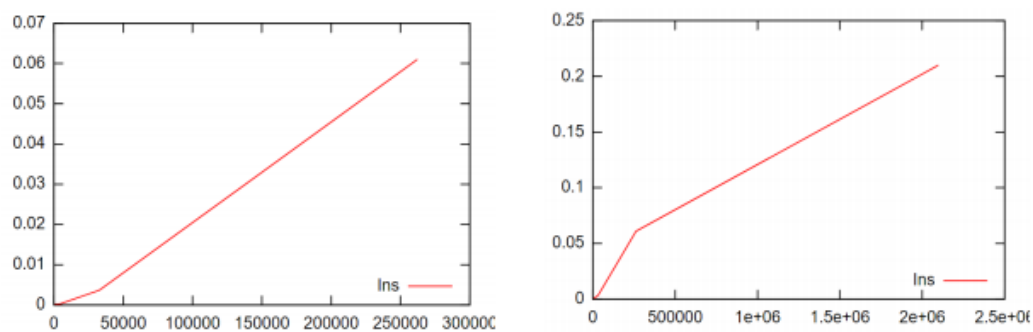


Figura 14: Gràfiques dels resultats utilitzant gnuplot.respawned.com

3.3.2 Resultats d'eliminar

Observem que el cost de l'algorisme implementat és aproximadament $O(n)$, segons les gràfiques obtingudes (Figura 16) i el detall de la taula (Figura 15). Això és degut al fet que s'ha de recórrer tot l'arbre buscant el nom a eliminar i després eliminar aquell node balancejant l'arbre restant. En el cas que es volgués eliminar pel valor, i no pel nom, seria el mateix que l'algorisme d'inserció i es podria aconseguir un cost òptim de $O(\log n)$.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.00011
XS	64	0.00014
S	512	0.00020
M	4096	0.00027
L	32768	0.00425
XL	262144	0.08237
XXL	2097152	0.36721

Figura 15: Resultats obtinguts per l'algorisme implementat

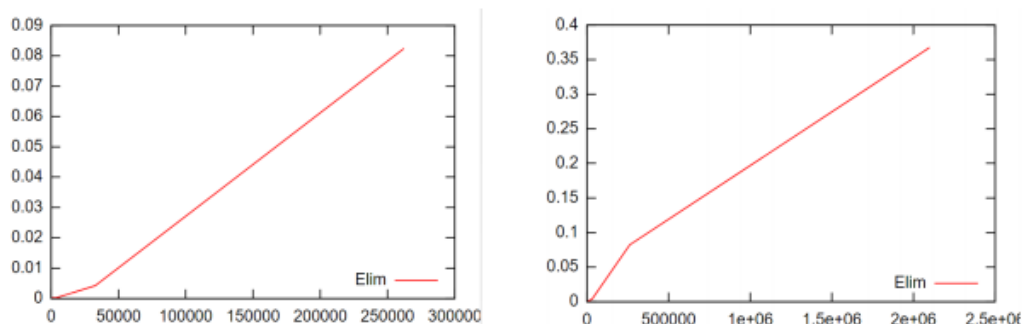


Figura 16: Gràfiques dels resultats utilitzant gnuplot.respawned.com

3.3.3 Resultats de recorregut per preordre

Observem que el cost de l'algorisme implementat és aproximadament $O(n)$, segons les gràfiques obtingudes (Figura 18) i el detall de la taula (Figura 17). Aquest és el cost estàndard per recórrer tota l'estructura de dades.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.0040708
XS	64	0.0079782
S	512	0.0219405
M	4096	0.0439098
L	32768	0.1476043
XL	262144	0.4285282
XXL	2097152	3.0827819

Figura 17: Resultats obtinguts per l'algorisme implementat

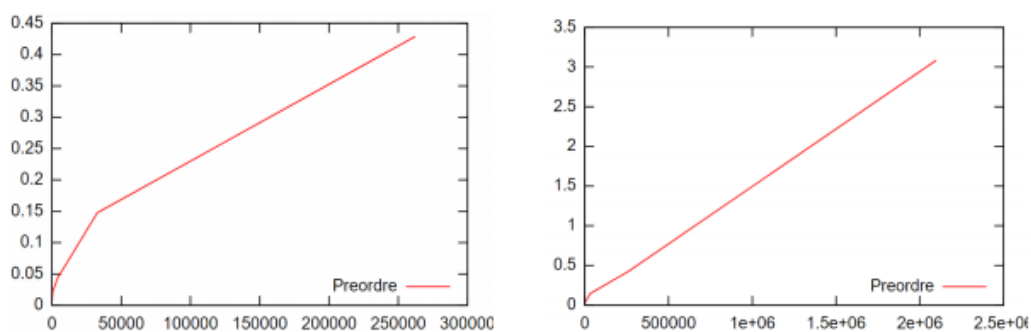


Figura 18: Gràfiques dels resultats utilitzant gnuplot.respawned.com

3.3.4 Resultats de recorregut per postordre

Observem que el cost de l'algorisme implementat és aproximadament $O(n)$, segons les gràfiques obtingudes (Figura 20) i el detall de la taula (Figura 19). Aquest és el cost estàndard per recórrer tota l'estructura de dades.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.0035231
XS	64	0.0052016
S	512	0.0079776
M	4096	0.0219717
L	32768	0.1755295
XL	262144	0.3739641
XXL	2097152	2.9337804

Figura 19: Resultats obtinguts per l'algorisme implementat

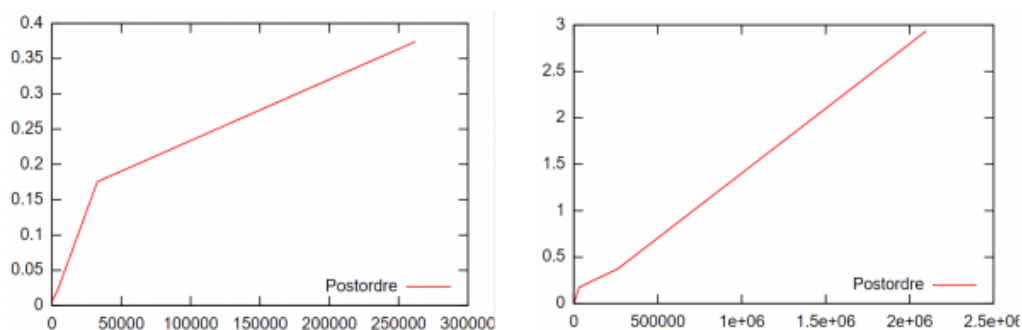


Figura 20: Gràfiques dels resultats utilitzant gnuplot.respawned.com

3.3.5 Resultats de recorregut per inordre

Observem que el cost de l'algorisme implementat és aproximadament $O(n)$, segons les gràfiques obtingudes (Figura 22) i el detall de la taula (Figura 21). Aquest és el cost estàndard per recórrer tota l'estructura de dades.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.0043891
XS	64	0.0076325
S	512	0.020534
M	4096	0.0188356
L	32768	0.0628111
XL	262144	0.324221
XXL	2097152	2.9822112

Figura 21: Resultats obtinguts per l'algorisme implementat

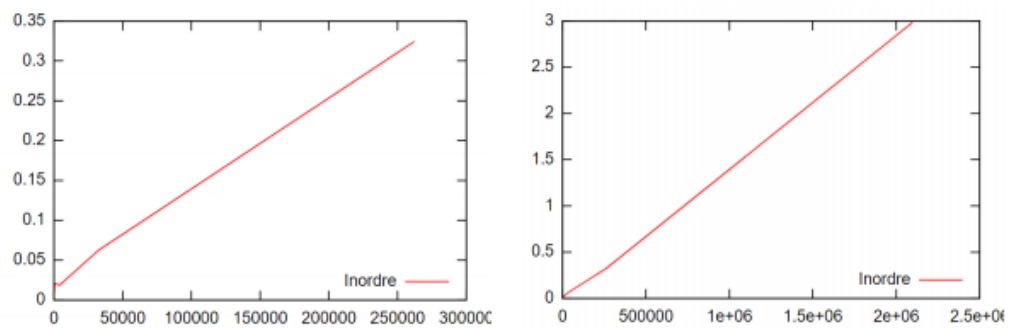


Figura 22: Gràfiques dels resultats utilitzant gnuplot.respawned.com

3.3.6 Resultats de recorregut per nivells

Observem que el cost de l'algorisme implementat és aproximadament $O(n^2)$, segons les gràfiques obtingudes (Figura 24) i el detall de la taula (Figura 23). Observem que no assoleix el cost estàndard per recórrer tota l'estructura de dades, no és òptim i és pitjor comparat amb la resta dels altres recorreguts. Això és degut al no utilitzar l'estructura de dades corresponent (Classe *Queue* de Java) i forçar una cua a través de la classe *ArrayList*. Observem que en la implementació fem servir el *remove* de la classe *ArrayList*, que té un cost de $O(n)$ a l'haver de reordenar tots els elements un cop esborrat l'element indicat.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.0039885
XS	64	0.0070067
S	512	0.0069861
M	4096	0.0089417
L	32768	0.0568745
XL	262144	0.8067461
XXL	2097152	41.7698905

Figura 23: Resultats obtinguts per l'algorisme implementat

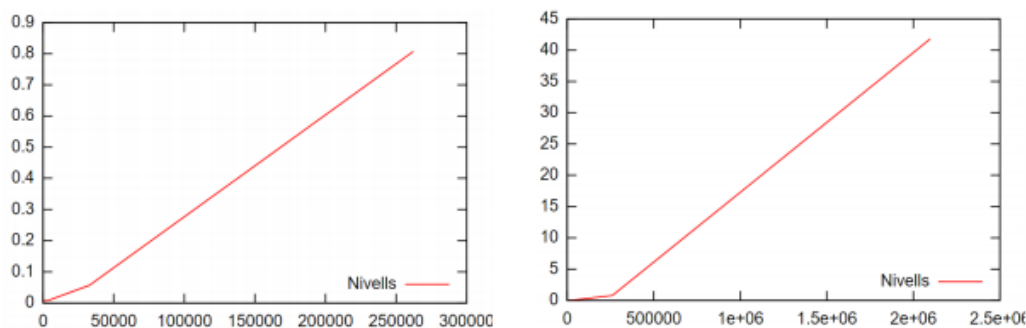


Figura 24: Gràfiques dels resultats utilitzant gnuplot.respawned.com

3.3.7 Resultats de cerca per valor exacte

Observem que el cost de l'algorisme implementat és aproximadament $O(n^2)$, segons les gràfiques obtingudes (Figura 24) i el detall de la taula (Figura 23). Analitzant l'algorisme de cerca per valor exacte per un arbre binari auto-balancejat, hauria de ser un cost òptim de $O(\log n)$ i en cap cas $O(n^2)$. Indicar que hem obtingut més taules i gràfiques per diferents valors exactes buscats, però en general, donaven els mateixos resultats i costos.

Com ja hem comentat, existeix un error molt important a l'hora d'implementar l'algorisme a l'aplicar un recorregut per nivells i comparar si es trobava el valor mentre es feia aquest recorregut. A més, agreujat que, com ja hem vist a l'apartat anterior, el recorregut per nivells no estava implementat d'una forma òptima. Per tant, ajuntant aquests dos errors, fa que la cerca per rang implementada tingui un cost exponencial $O(n^2)$. S'hauria d'haver implementat un algorisme tal que començant per l'arrel es va recórrer (per exemple amb postordre) l'arbre segons si el valor és més gran (accedim node dret) que al node que estem comparant o més petit (accedim fill esquerre). Aplicaríem aquest algorisme recursivament fins a trobar exactament el valor demanat.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.0
XS	64	0.0
S	512	0.0
M	4096	0.0010161
L	32768	0.0119554
XL	262144	0.2922201
XXL	2097152	15.2273055

Figura 25: Resultats obtinguts per l'algorisme implementat

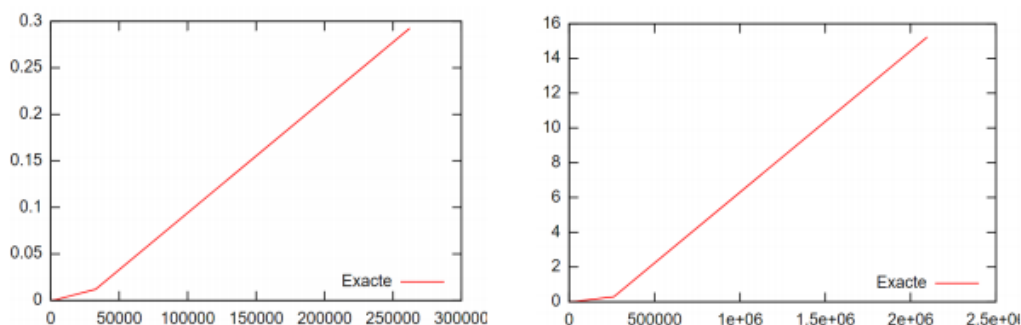


Figura 26: Gràfiques dels resultats utilitzant gnuplot.respawned.com

3.3.8 Resultats de cerca per rang

Pels resultats de cerca per rang, es veuen igualment afectats com l'algorisme anterior de cerca exacte. L'algorisme implementat no correspon amb l'algorisme òptim aprofitant els avantatges d'un arbre binari autobalancejat. Hem dissenyat un algorisme que en recórrer l'arbre per nivells (demostrar que en el nostre cas no és òptim), comprova a cada node si es troba en el rang de valors donats i l'afegeix a una llista per mostrar-lo en finalitzar el recorregut. A més, es torna a utilitzar l'estructura de dades *ArrayList* per fer-la servir com a la classe *Queue* de Java (fent servir el *remove* de la classe *ArrayList*). Els resultats obtinguts, depenent del dataset i dels valors buscats dins l'arbre, variaven bastant, però existia el pitjor dels casos que sigués de cost aproximadament $O(n^2)$.

3.4 Proves

En aquest cas, el nostre mètode de proves ha estat implementar una funció la qual ens retorna una impressió per consola dels arbres per a poder comprovar visualment el seu funcionament i si les insercions/eliminacions i balanceig es realitzaven de forma correcta (Figura 27).

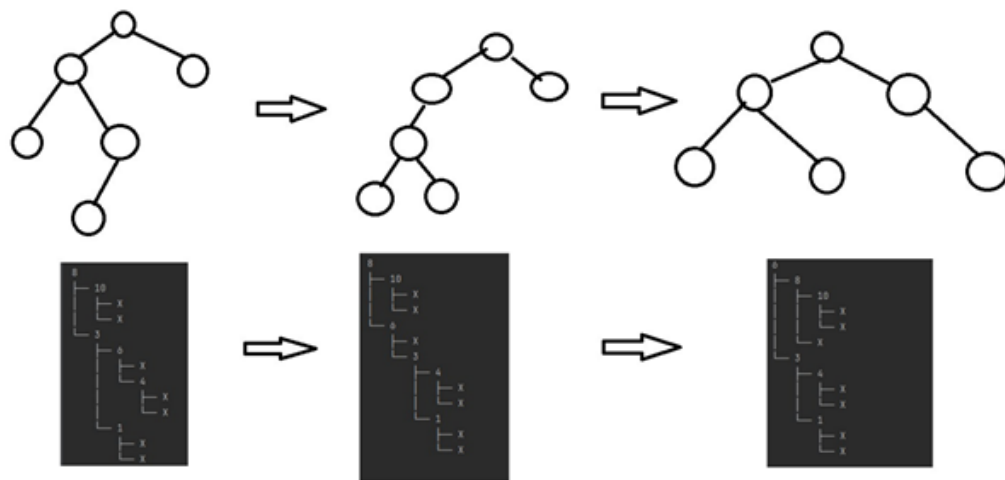


Figura 27: Eliminació d'un node i posterior balanceig mostrat per consola amb els nostres algorismes

Respecte als arbres binaris, un cost $O(n)$ i $O(n^2)$ no són òptims, i intentem trobar l'ideal cost $O(\log n)$. Ja que el *ArrayList* té un cost $O(n)$, totes les estructures que hem fet intenten aconseguir el cost logarítmic, ja que són millors que l'*ArrayList*. Si veiem que el cost és més elevat o el no esperat, significa que l'estructura està mal dissenyada.

3.5 Problemes

Un dels principals problemes observats ha estat quan s'eliminen nodes específics, com ara l'arrel, els nostres algorismes no poden fer el balanceig de l'arbre. Per a la resta de casos ens funciona de forma correcta.

4 Arbres R

4.1 Disseny

De cara a la implementació d'aquesta estructura de dades, ens hem inspirat bastant en la idea utilitzada anteriorment en els arbres binaris autobalancejats. La nostra classe *Arbre* serà únicament un node (o rectangle) arrel que dins d'aquest tindrà més nodes.

Per dissenyar els arbres R, sempre partirem que l'arbre és el rectangle més gran possible que conté tots els altres rectangles o punts de l'arbre. A partir d'aquesta idea i de les especificacions de l'enunciat hem creat la classe *Node* amb les següents condicions.

La classe *Node* (classe principal de l'estructura de dades) bàsicament serà definida pels seus atributs de punts: *p1*, *p2*, *punt intermig* i *punt que fa overflow*. Aquesta classe tindrà dos comportaments diferents depenent si és un node fulla o no.

- En el cas que *Node* sigui un node fulla, vol dir que no tindrà fills rectangles. Aquest node contindrà com a molt 3 punts i un quart que activarà el mecanisme d'overflow.
- En el cas que *Node* no sigui un node fulla, vol dir que tindrà fills rectangles i no punts com a tal. Per tant, els punts-atributs canviaran el seu significat: *p1* i *p2* seran dues cantonades imaginàries que definiran el rectangle (que poden coincidir amb un punt real amb nom o no). A més, el node tindrà una llista (*ArrayList*) de tots els rectangles que té continguts.

Indicar que s'ha creat a més una classe *Punt* on es guardaran les posicions exactes del punt (coordenades X/Y) i el nom del tresor trobat en aquest punt. Si aquesta classe no té nom, s'utilitza com a punt imaginari que defineix rectangles.

Cal destacar que de cara a la qualitat de codi i a la qualitat de l'estructura de dades, s'hauria d'haver fet que la classe *node* extengués d'una classe geometria on aquesta definís totes les operacions bàsiques i necessàries que es fan servir en els diferents algorismes implementats.

4.2 Algorismes

4.2.1 Funcionalitats bàsiques

Es demana que es realitzi una bona implementació de les opcions tant d'afegir tresors com eliminar-los.

Pel que fa a inserir un element dins al arbre R, aquest s'inserirà començant des de l'arrel, entrant de mica en mica de rectangle a rectangle (nodes no fulles), fins a arribar a un node fulla i guardar-se en un dels 4 punts mencionats al disseny de l'algorisme. Per calcular a quins rectangles es prioritza accedir, escollim el rectangle que incloent aquell punt creixi menys.

Anem a explicar ara el possible overflow. Existeixen dos tipus d'overflow que s'han implementat de formes diferents.

- En el primer cas és quan existeix overflow al ocupar la posició del *punt overflow*. Per tant, apareix un rectangle que conté 4 punts. En aquest cas, crearem dos rectangles nous, calculant totes les possibilitats d'àrea que facin que la suma de les dues àrees sigui la més petita possible amb els punts anteriors (que després farem la reinserció corresponent). Ara només ens faltaria revisar de forma recursiva que no es torni a produir overflow.
- En el segon cas és quan un node que no és fulla, té més de 3 rectangles com a fills al *ArrayList*. Llavors la idea és aplicar el mateix que el primer cas, però pensant en rectangles i punts que defineixen els rectangles i no com a punts de tresors.

Pel que fa a eliminar un tresor és bastant més senzill. Només s'ha de recórrer l'arbre R fins a trobar el punt que estem buscant i l'eliminem del seu pare. Si el pare del node eliminat té algun fill més ja hem acabat l'algorisme. Però si aquest pare es queda sense punts, l'hem d'eliminar i tornar a reinserir. Després hauríem de tornar a comprovar-ho per l'avi recursivament.

4.2.2 Visualització

Es demana llistar tots els rectangles de l'arbre i el seu contingut (punts).

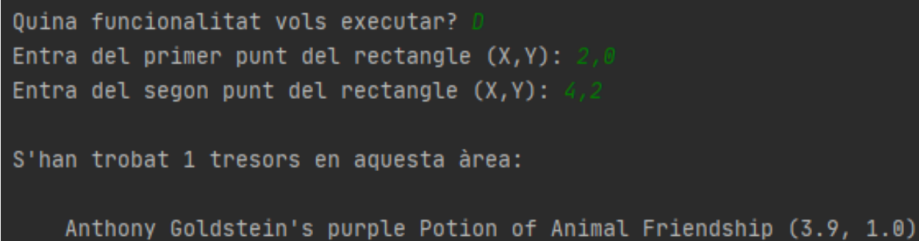
Per visualitzar d'alguna forma els rectangles i els seus punts amb tresors, hem hagut de recórrer l'arbre de forma recursiva. Ens hem ajudat de les estructures de dades ja creades per recórrer l'arbre per nivells i així anar guardant la informació en Strings per després visualitzar-la per pantalla.

Per visualitzar la informació per pantalla, amb l'ajuda de les llibreries swing i awt vam crear un JFrame, a on creàvem els rectangles i els punts extrets en recórrer l'arbre R per nivells. A més, hem creat un JSlide per fer zoom a la pantalla i apreciar amb més detall el mapa obtingut.

4.2.3 Cerca per àrea

Es demana oferir una forma per cerca per àrea, trobant tot el que hi hagi en un rectangle introduït per l'usuari.

Per implementar aquest algorisme s'ha anat accedint als rectangles que tallaven al rectangle introduït per l'usuari (i descartant tots els altres casos). Una vegada arribat a un node fulla, es comprovava que aquells punts estiguessin dins el rectangle introduït i formessin part de la solució.



```
Quina funcionalitat vols executar? 0
Entra del primer punt del rectangle (X,Y): 2,0
Entra del segon punt del rectangle (X,Y): 4,2

S'han trobat 1 tresors en aquesta àrea:

Anthony Goldstein's purple Potion of Animal Friendship (3.9, 1.0)
```

Figura 28: Exemple de cerca per àrea

4.2.4 Cerca per proximitat

Es demana oferir una forma de cerca per proximitat, trobant la quantitat de tresors que hagi introduït l'usuari en unes coordenades concretes.

Per implementar aquest algorisme, hem anat recorrent l'arbre R accedint des dels rectangles més grans fins als nodes fulla. Es prioritza els rectangles que estaven més a prop. Una vegada que ja estan guardats els possibles punts, s'han de comprovar que realment aquests punts siguin els que estan més a prop. Per tant, se segueix recorrent l'arbre descartant tots aquells rectangles que les quatre cantonades (el mínim de les 4) del rectangle estiguin més llunyanes que tots els punts ja guardats. en rectangle, i si un rectangle

```

Quina funcionalitat vols executar? E

Entra el nombre de tresors a trobar: 2
Entra el punt a on cercar (X,Y): 4,2

Els 2 tresors més propers a aquest punt són:

Anthony Goldstein's purple Potion of Animal Friendship (3.9, 1.0)
Xenophilus Lovegood's cyan Decanter of Endless Water (5.1, 2.7)

```

Figura 29: Resultats obtinguts per l'algorisme implementat

4.3 Resultats

4.3.1 Resultats d'inserir

Observem que el cost de l'algorisme implementat és aproximadament $O(\log n)$, segons les gràfiques obtingudes (Figura 31) i el detall de la taula (Figura 30). És el resultat esperat, i en termes de cost, és un resultat òptim. Això és degut a que estem utilitzant una estructura de dades optimitzada per ordenar informació en forma de coordenades. Cal destacar que podria ser que en arbres R molts grans i per alguns nodes inserits, produïssin un overflow a molts nodes i el cost no sigués exactament igual al vist a les nostres comprovacions i resultats.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.00009
XS	64	0.0009967
S	512	0.0009682
M	4096	0.0010781
L	32768	0.0037062
XL	262144	0.0040829
XXL	2097152	0.0079712

Figura 30: Resultats obtinguts per l'algorisme implementat

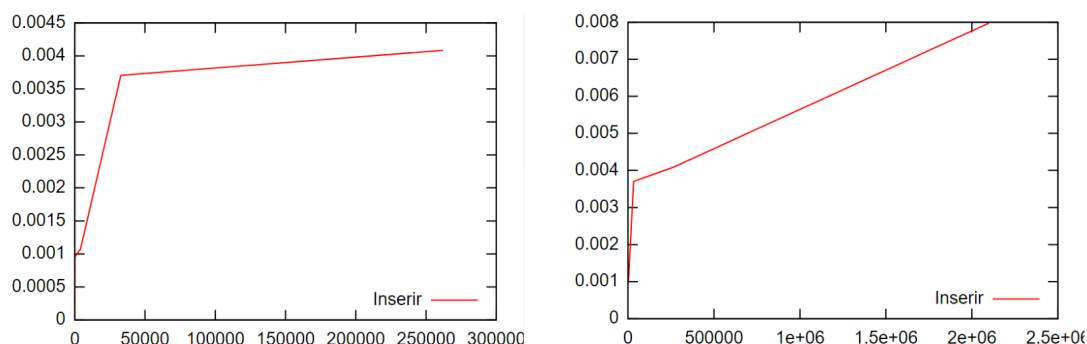


Figura 31: Gràfiques dels resultats utilitzant gnuplot.respawned.com

4.3.2 Resultats d'eliminar

Destacar que és un resultat similar al d'eliminar als arbres AVL. S'observa que el cost de l'algorisme implementat és aproximadament $O(n)$, segons les gràfiques obtingudes (Figura 33) i el detall de la taula

(Figura 32). Això és degut al fet que s'ha de recórrer tot l'arbre R buscant el nom a eliminar per després eliminar aquell node i mirar si apareix underflow.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.00009
XS	64	0.00018
S	512	0.001025
M	4096	0.0019666
L	32768	0.0090028
XL	262144	0.4709011
XXL	2097152	57.9071055

Figura 32: Resultats obtinguts per l'algorisme implementat

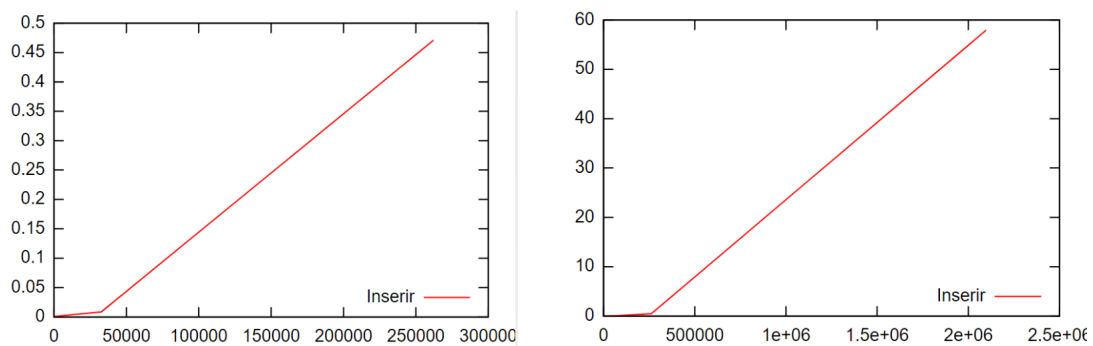


Figura 33: Gràfiques dels resultats utilitzant gnuplot.respawned.com

4.3.3 Resultats de la cerca per àrea

Observem que el cost de l'algorisme implementat és aproximadament $O(\log n)$, segons les gràfiques obtingudes (Figura 35) i el detall de la taula (Figura 34). És el resultat esperat, i en termes de cost, és un resultat òptim i estàndard en una cerca en una estructura de dades destinada a emmagatzemar coordenades. La lògica d'haver aconseguit el cost òptim $O(\log n)$ està explicada a l'apartat d'algorismes.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.00010
XS	64	0.00018
S	512	0.00028
M	4096	0.00058
L	32768	0.0009971
XL	262144	0.0012826
XXL	2097152	0.001994

Figura 34: Resultats obtinguts per l'algorisme implementat

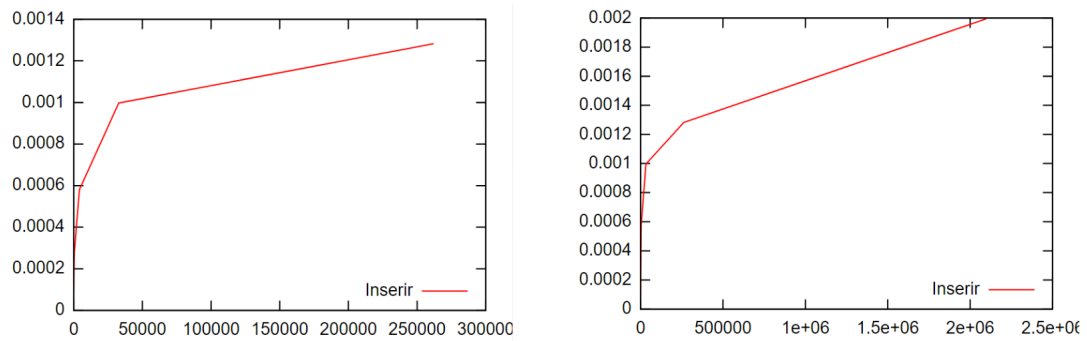


Figura 35: Gràfiques dels resultats utilitzant gnuplot.respawned.com

4.3.4 Resultats de la cerca per proximitat

Com també la cerca anterior, observem que el cost de l'algorisme implementat és aproximadament $O(\log n)$, segons les gràfiques obtingudes (Figura 37) i el detall de la taula (Figura 36). És el resultat esperat, i en termes de cost, és un resultat òptim i estàndard en una cerca en una estructura de dades destinada a emmagatzemar coordenades. La lògica d'haver aconseguit el cost òptim $O(\log n)$ també està explicada a l'apartat d'algorismes.

Fitxer	Número Nodes	Temps Execució (segons)
XXS	8	0.00010
XS	64	0.00016
S	512	0.00032
M	4096	0.0009687
L	32768	0.0019018
XL	262144	0.0030194
XXL	2097152	0.0060119

Figura 36: Resultats obtinguts per l'algorisme implementat

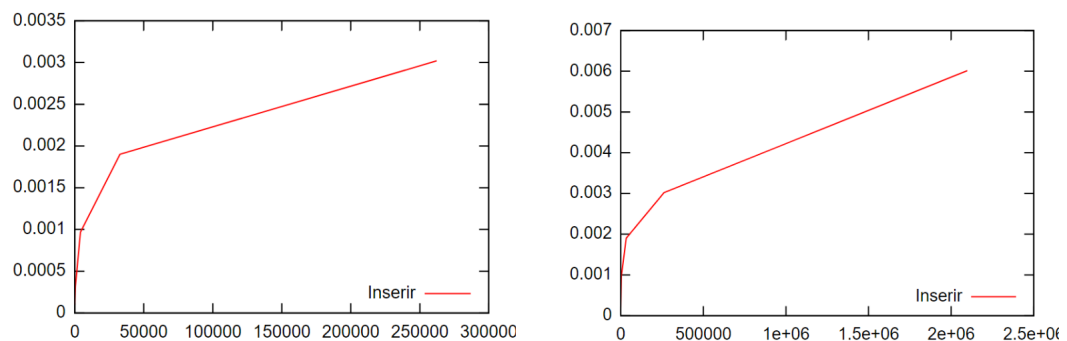


Figura 37: Gràfiques dels resultats utilitzant gnuplot.respawned.com

4.4 Proves

De cara al mètode de proves utilitzat ens hem decidit per fer ús de les capacitats de Java i hem implementat una petita interfície gràfica usant les llibreries Swing/AWT per a poder visualitzar com ens quedaria el R-Arbre (Figura 27).

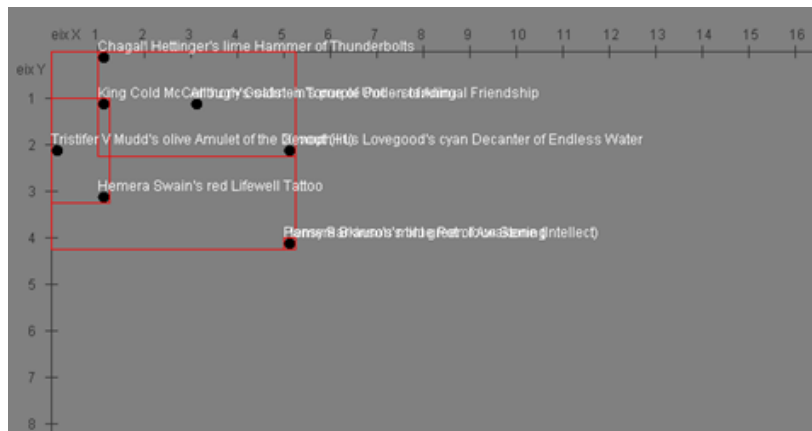


Figura 38: Exemple simplificat de la interfície gràfica que ens permet veure els rectangles creats i la posició dels punts relativa als altres

4.5 Problemes

Un dels principals problemes que hem tingut amb els R-Tree ha estat la implementació pràctica, ja que a nivell teòric teníem assimilat el concepte però al no haver molta documentació al respecte i que no sabíem com transformar la teoria en pràctica, el disseny inicial ens ha dut bastant temps. Un altre aspecte que ens ha donat bastants maldecaps ha estat l'eliminació de nodes, que ens ha costat en general molts més esforços que no pas la inserció d'aquests. Com amb gairebé tot en els R-Trees, enteníem la teoria però no trobàvem la manera òptima d'implementar-ho.

Un tercer problema seria el número de funcions i procediments que hem hagut de crear per a situacions molt similars però amb paràmetres/condicions diferents dins la classe Node del R-Arbre. Això més aviat ha resultat ser un problema de temps de codificació i comprovacions del correcte funcionament de cada un dels procediments.

5 Taules

5.1 Disseny

De cara a la implementació d'aquesta estructura de dades, hem decidit que el més adient era crear un HashMap que sigues de tipus `String[][]` degut a que en la primera casella posicionarem l'id, o key, a on guardarem la informació, mentre a la segona guardarem la informació adient. Una altra idea que teníem era de fer un `ArrayList<String[]>`, al final hauríem arribat al mateix resultat, el posicionament el podríem haver fet amb `Get` o un `Add` (amb index), i el resultat hagués sigut el mateix, però per raons més visuals i de quantitat d'informació ens vam decantar per fer un `String[][]`.

A l'hora d'afegir informació vam implementar un rehash del tipus Quadratic probing, tot i que primerament havíem fet un linear probing, en cas de tenir moltes dades guardades, amb el linear probing es tenien que fer moltes iteracions i en alguns casos petava el programa a causa d'això, degut a que anava recorrent tot l'array de casella en casella, per altra banda, amb el Quadratic probing multipliquem l'índex de les caselles per si mateixos, fent que no les recorrem totes i trobem abans una posició lliure per guardar la informació.

A l'hora de generar les claus per guardar la informació vam sumar els valors numèrics dels noms dels pirates, cosa que en cas de que diversos pirates tinguin les mateixes lletres en el seu nom, es generin col·lisions.

Per decidir el tamany que tindria l'estructura vam decidir multiplicar el nombre més gran possible segons els valors de la taula ASCII per el total de pirates, tot i així amb el quadratic probing ens podia passar que la clau sigues més gran que el total de caselles disponibles, en aquest cas el que fèiem era generar un nou hashmap amb el tamany màxim que ens ha sortit.

5.2 Algorismes

5.2.1 Funcionalitats bàsiques

Com hem mencionat a l'apartat de disseny, a l'hora d'afegir hem creat un quadratic probing, per prevenir problemes en cas que hi hagi una col·lisió, i així en el millor cas tingui un cost $O(1)$, per tant el que fem es sumar-li a la clau generada la posició de l'array a on la guardarem, multiplicada per si mateixa, en cas de que l'espai estigui ocupat es buscarà un altre espai recursiva ment fins a trobar un lloc lliure.

En el cas que volem esborrar informació recorrem tota l'estructura guardant tota la informació en un altre array, a excepció de la informació que volem esborrar.

5.2.2 Cerca i histograma

A l'hora de buscar un pirata tornem a sumar les lletres del seu nom i fer un rehash comparant si és l'indicat.

En el cas de l'histograma per edats vam crear un array amb 101 caselles, considerant que les edats anirien de 0 a 101, i inicialitzant totes les caselles a 0, a continuació anàvem recorrent tots els pirates i sumant 1 a la casella que corresponguis a l'edat del pirata en qüestió, per exemple, si el pirata té 50 anys a la casella 50 de l'array li sumarem 1, i així repetidament.

Finalment amb l'ajuda de les llibreries swing i awt vam crear un JFrame, a on representàvem per mitjà d'un gràfic de barres totes les edats, a l'esquerra afegint un JPanel amb el llistat de les edats en forma de columna, i a la resta de la pantalla tenim les barres, el que anem fent és fer-les més grans o més petites segons el nombre de pirates que tenim d'aquesta edat sumant-li 40, perquè cada pirata tingui un espai de 40 píxels a la pantalla.

5.3 Resultats

En aquest últim apartat hem pogut analitzar, que en la teoria, tant l'algorisme implementat per afegir, com eliminar i com cercar, haurien de tenir tots el mateix cost, sent aquest en el millor cas $O(1)$ i en el pitjor cas $O(n)$ (com més dades apareixen més col·lisions, i en efecte, augmenta el cost a $O(n)$).

En les nostres implementacions, observem que la teoria l'hem aplicat de forma eficient pel que fa als algorismes d'inserir i cercar. Això és pel fet que afegir i cercar utilitzen el mateix mètode, quadràtic probing. Per tant en afegir un pirata en qüestió i en cercar-lo acabem tenint el mateix cost, amb l'única diferència sent la solució, perquè en afegir busquem un espai buit i en cercar anem comprovant que aquell espai contingui la informació que volem.

En el cas de l'algorisme implementat per eliminar hauríem de tenir el mateix cost que en la cerca i la inserció, però pel fet que hem de passar totes les dades excepte una, en aquell moment vam pensar que era una bona idea utilitzar un bucle que recorregués tot el *HashMap* dada a dada, fent així que ens queda un cost constant $O(n)$ i no òptim.

Finalment, en el cas de l'algorisme implementat per visualitzar l'histograma, tenim que la generació de les dades té sempre un cost lineal $O(n)$, ja que, per mostrar tota la informació guardada en l'estructura de dades s'ha de recórrer tots els nodes amb un bucle, justificant el cost obtingut.

Dataset	Nodes	Inserir	Eliminar	Cerca	Histograma
XXS	8	0.0129647	0.00006	0.0158212	0.00005
XS	16	0.2134675	0.00012	0.1302554	0.00019
S	512	0.2114041	0.0046152	0.1354514	0.0025845
M	8192	0.2214583	0.0578556	0.1458524	0.0458742
L	65536	0.2084115	0.4927384	0.1455547	0.5893123
XL	262144	0.2245223	1.8324306	0.1524562	1.6884126
XXL	524288	0.2299498	4.5508009	0.2124546	3.5678989

Figura 39: Resultats obtinguts pels diferents algorismes implementats

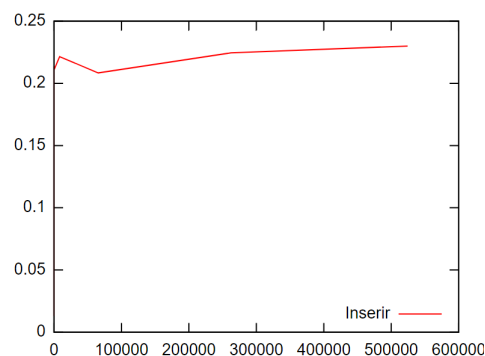


Figura 40: Gràfica dels resultats de l'algorisme d'inserir utilitzant gnuplot.respawned.com

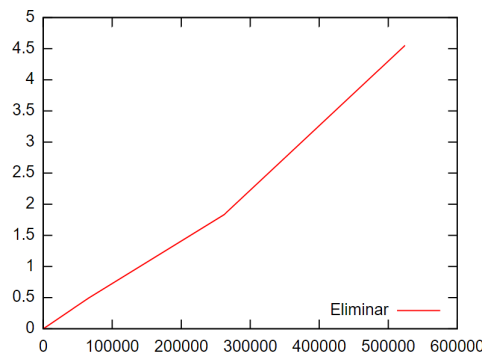


Figura 41: Gràfica dels resultats de l'algorisme d'eliminar utilitzant gnuplot.respawned.com

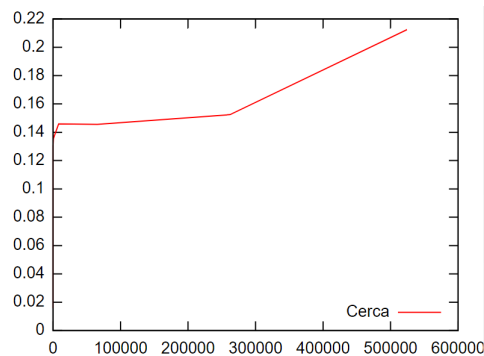


Figura 42: Gràfica dels resultats de l'algorisme de cerca utilitzant gnuplot.respawned.com

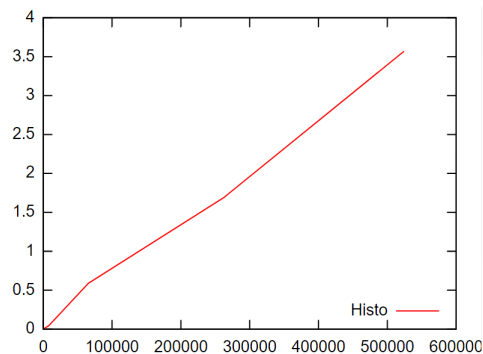


Figura 43: Gràfica dels resultats de l'algorisme de l'histograma utilitzant gnuplot.respawned.com

**En la primera entrega de la memòria hi havien errors importants en les gràfiques i en les dades de les taules.

5.4 Proves

A l'hora de fer les comprovacions vam utilitzar l'eina debugging de IntelliJ. Primerament per fer la comprovació de manera més senzilla vam utilitzar l'arxiu més petit de pirates que teníem i vam calcular quina posició tindria cada pirata, a continuació ho vam comprovar amb el debugging entrant a l'array on guardem la informació i mirant que està el pirata corresponent, en cas de rehashing també vam tenir en compte la següent posició a on aniria a parar per mitja de la utilització de la formula corresponent ja mencionada segons el tipus de rehashing que vam fer, tot això en el cas d'afegir un nou pirata.

En el cas d'esborrar un pirata simplement vam esborrar-lo i vam utilitzar la funció Get per a comprovar que ja estava efectivament esborrat. En el cas de la cerca vam fer les mateixes comprovacions que en las d'afegir un pirata, debugging a certes funcions i comparar que quadren els resultats i que les caselles ocupades tenen el mateix valor i no ens estem equivocant.

5.5 Problemes

El principal problema que ens va sorgir implementat aquest apartat va ser a l'hora d'inserir elements, ja que al realitzar linear probing, i que aquest avançava d'un en un, si el dataset era de dimensions considerables li resultava molt difícil trobar el lloc. Al ser recursiu, el programa detectava que s'havia executat masses cops i interrompia l'execució donant error. Aquest problema es va aconseguir canviant al quadratic probing.

Important: Un problema que ens vam trobar ja havent entregat la pràctica va ser que tant les funcions per buscar i eliminar pirates com els comentaris dels grafs se'ns van eliminar de la versió guardada a git, i a dia d'avui encara no en sabem la raó. Ho vam comentar a la entrevista amb el professor i ens va dir que deixéssim constància per escrit en la memòria. Això causa que un cop eliminat un pirata i després buscat, doni error d'execució.

6 Estructures de dades utilitzades

6.1 Llistes (ArrayList)

Aquesta estructura conté 5 funcions, cadascuna amb un cost propi, la primera que explicarem és l'*add*, aquesta funció té cost $O(1)$, perquè s'afegeix un cop al final de totes les dades i no necessita recorre-les per guardar la informació. Per altra banda, en el cas que s'hagués de copiar tota la informació guardada, el cost pujaria a $O(n)$.

Tenim també una segona funció d'*add*, la diferencia en aquesta està en el fet que li diem la posició en la qual volem guardar un element, això fa que aquesta tingui un cost mitjà de $O(n)$.

En el cas de la funció *get* tenim que té un cost $O(1)$, ja que només ha d'anar a buscar la informació directa de la posició que li demanem.

També tenim la funció *remove*, en aquest cas porta un cost mitjà $O(n)$ pel fet que ha de recórrer tot l'array per trobar l'element que vol esborrar i reordenar els altres elements; passa el mateix per a les funcions *indexOf* i *contains*.

6.2 Cues (Queue)

En aquest cas tenim una estructura cua, aquesta té les funcions *add*, *remove*, *getFirst*, principalment, depenent del llenguatge de programació i si és dinàmica o estàtica en pot contenir més. En tots els casos aquestes funcions tenen cost $O(1)$ perquè en aquesta estructura, afegim automàticament la informació al final de la cua, i per esborrar o extreure informació sempre ho fem extraient el primer element que tenim al principi d'aquesta, per tant no ha de recórrer tota l'estructura i sempre té cost $O(1)$.

En el nostre cas, hem utilitzat en la majoria de casos la definició de cua, però no la classe. Hem fet servir la classe pròpia de Java *ArrayList* per fer-la servir com una cua, fent en molts algorismes créixer el seu cost.

6.3 HashMap

En aquest cas tenim que la complexitat a l'hora d'afegir informació a l'estructura és de $O(1)$ en el millor cas i $O(n)$ en el pitjor dels casos, tot i que això dependrà del nombre d'iteracions que necessiti per guardar la informació en cas de tenir moltes caselles ocupades. En aquesta estructura es treballa amb hashing, fent que assignem una clau a cada element i aquesta no es repeteix mai, en cas que una casella estigués buida en cercar-la simplement trauríem resultat null.

Altres funcions com el cas de *clear*, que esborren tota la informació que conté el *HashMap* té un cost $O(n)$ perquè ha de recórrer-l tot casella per casella.

El mètode *remove* crida a la funció *RemoveEntryForKey(key)*, mètode intern que calcula la posició final de la clau de l'objecte en qüestió, i a continuació utilitza aquest valor a la funció *indexOf* per a trobar la informació que busquem. El mètode *remove* té un cost $O(1)$ en la majoria de casos.

Finalment existeixen els mètodes *size* amb cost $O(1)$ i el mètode *values* que té cost $O(n)$, a l'haver de recórrer tot el *HashMap*.

7 Conclusions

En general, un dels aspectes més complicats, però que alhora ens ha resultat bastant interessant ha estat el fet d'haver de crear nosaltres mateixos les estructures de dades perquè s'adaptin als nostres requeriments, i decidir el funcionament d'aquestes. El fet que aquestes estructures fossin bastant més complexes que les que vam implementar en el primer semestre ha requerit d'un major temps de planificació i disseny per poder assegurar un funcionament correcte.

No només crear les estructures, sinó fer-les òptimes i optimitzar el codi per a mantenir sempre un cost proper a $O(\log n)$ ha estat el principal objectiu d'aquesta pràctica. Si bé en la del primer semestre ens en podíem eludir en certa manera si implementàvem un codi que no estava optimitzat al 100%, en aquesta pràctica ens penalitzava d'una forma molt més accentuada. El fet de dissenyar un codi que no només compili, sinó que ho faci de forma òptima i sense generar un alt cos computacional gràcies també al disseny de les estructures de dades ha estat un punt d'enfocament bastant prioritari i que ens ha aportat molta experiència al respecte.

Referències

- [1] En.wikipedia.org. 2021. R-tree - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/R-tree>> [Accessed 29 May 2021].
- [2] Mathcs.emory.edu. 2021. [online] Available at: <http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/3-index/R-tree2.html>> [Accessed 29 May 2021].
- [3] Trees in Java — How to Implement a Binary Tree?. (2021). Retrieved 29 May 2021, from <https://medium.com/edureka/java-binary-tree-caede8dfada5>.
- [4] Gonzalez, M. (2021). Implementing a Binary Tree in Java | Baeldung. Retrieved 29 May 2021, from <https://www.baeldung.com/java-binary-tree>.
- [5] Graph and its representations - GeeksforGeeks. (2021). Retrieved 29 May 2021, from <https://www.geeksforgeeks.org/graph-and-its-representations/>.
- [6] Chandrakant, K. (2021). Graphs in Java | Baeldung. Retrieved 29 May 2021, from <https://www.baeldung.com/java-graphs>.
- [7] What is Depth First Search?. (2021). Retrieved 11 July 2021, from <https://www.educative.io/edpresso/what-is-depth-first-search>.
- [8] Bibing us (2021). Retrieved 11 July 2021, from <http://bibing.us.es/proyectos/abreproy/11320/fichero/Capitulos%252F10.pdf>.