Programació Avançada i Estructura de Dades

Projecte del Segon Semestre – Estructures No Lineals

Els Cavallers de la Taula de Hash

Departament d'Enginyeria
La Salle - Universitat Ramon Llull
13 febrer de 2023

	Nom i Cognoms
Alumnes	Alvaro Bello Guillem Alba Payà Isabel Rodriguez Nil Bagaria Nofre
Grup	G9

Índex

Llenguatge de Programació Escollit

Per a realitzar aquest projecte, s'ha escollit el llenguatge de programació Java.

Aquesta elecció s'ha basat en l'anàlisi dels diversos aspectes positius i negatius que comentarem a continuació.
Primer de tot, si ens basem en els seus avantatges, podem destacar la facilitat que ofereix Java a l'hora d'utilitzar certes funcions per recorrer i comparar les estructures de dades.
Seguidament, també podem tenir en compte el fet que és un llenguatge de programació molt utilitzat, això significa que ens permet, en moments de dubte, poder trobar informació al respecte de forma més senzilla.
Per últim, però no per això menys important, hem tingut en compte que és un llenguatge molt utilitzat i, en el nostre cas, el seguirem veient al llarg de la carrera en diferents matèries i, per tant, ens permet poder aprofundir més en el seu coneixement.

Per altra banda, si ens centrem ara en els aspectes negatius que pot tenir, hem valorat diversos factors.
Java, pot ser més lent que altres llenguatges de programació a l'hora d'executar certes tasques, això pot provocar que els algorismes siguin més lents en comparació a si s'implementen en altres llenguatges.
També hem tingut en compte que pot ser un llenguatge de programació més complex a l'hora d'implementar certes tècniques d'optimització, fet que pot afectar als termes temps i espai de l'algorisme.
A banda d'això, Java pot tenir una major sobrecàrrega de treball en termes de gestió de memòria i recursos, que pot afectar al rendiment de certs algorismes o funcions.

D'aquesta manera, observant els pros i contres de Java, hem decidit decantar-nos per Java com el llenguatge de programació utilitzat per a la implementació d'aquest projecte.

Finalment, també comentar que, pel que fa la plataforma de treball, hem utilitzat IntelliJ IDEA per crear i editar els nostres fitxers, ja que ens permet treballar de forma sincronitzada amb el git de Bitbucket, obtenint d'aquesta forma un mètode de treball en grup molt més eficaç.

Estructures de Dades del Projecte

Fase 1 - Sobre orenetes i cocos (Grafs)

DFS

Disseny de l'estructura

Per dissenyar i estructurar el Graf teníem dues opcions, la implementació mitjançant una llista d'adjacències o la matriu d'adjacències.
A causa de les característiques i la seva implementació més senzilla vam decidir utilitzar la llista d'adjacències, la qual sol ser més eficient a Grafs poc densos i amb pocs enllaços (més tard ens vam adonar que el graf tenia molts enllaços pel que considerem que hauria estat més eficient implementar la matriu d'adjacents).

Disposats a implementar la matriu d'adjacències, comencem creant les classes següents:

- LlocInteres: Aquesta classe serà el node del nostre Graf i representarà un lloc d'interès contenint informació com el seu clima i el regne al qual pertany.
- Trajecte: Aquesta classe serà l'enllaç entre els nodes del nostre Graf i contindrà una referència i un ID del node (LlocInteres) origen i destinació, així com la distància del trajecte, el temps que triga una sorreta Africana i el temps que triga una sorreta Europea a recórrer aquest trajecte.
- Graph: Aquesta classe representarà el nostre Graf i només contindrà una llista amb els nodes (LlocsInteres) del nostre Graf i les funcions que treballen amb el graf directament.
- Fase1: Aquesta classe contindrà funcions auxiliars (DFS, getLlocFromId, initGraph i Prim) que no facin directament una simple modificació o cerca en el Graf.

- Dijkstra: Hem decidit crear una clase només per aquest algoritme per la seva complexitat.

Després de crear les pertinents classes, amb l'objectiu d'enllaçar cadascun dels Llocs D'interès (nodes) del nostre Graf, creem a la classe LlocInteres una llista de trajectes que representarà la llista d'adjacents del node, indicant els Llocs D'interès (nodes) amb què està connectat.

Explicació dels algorismes implementats

initGraph

Aquesta funció s'encarrega d'inicialitzar el nostre Graf, llegint tota la informació dels fitxers proporcionats i plasmant-la sobre una llista amb els Llocs D'interès (la qual posteriorment se li passarà a l'objecte Graf).

containsByID

Aquesta funció del Graf comprova si hi ha algun node dins del Graf amb el mateix ID que el node introduït (no ho comprova per referència).

searchForID

Aquesta funció del Graf cerca el primer node dins del Graf amb el mateix ID que l'introduït (no el cerca per referència).

trajecteMesCurt

Aquesta funció del Graf (la qual només és utilitzada per l'algorisme Prim) rep la referència d'un altre Graf per recórrer tots els nodes que es trobin tant al Graf actual com al proporcionat, i després recorre tots els trajectes d'aquests nodes sempre i quan portin a un node no visitat al Graf proporcionat; de tots els trajectes que compleixin aquests requisits retornem el que menys distància tingui i marquem el LlocInteres al que ens porti com a visitat.

Prim

Aquesta funció de la Fase1 rep un Graf per formar el seu MST de la següent manera:

1. Crea un nou Graf (el qual serà el MST).
2. Afegeix un node qualsevol a aquest nou Graf per tenir un node pel qual començar el MST.

3. Va buscant i afegint el trajecte més curt a l'MST (obtingut mitjançant la funció anterior explicada) amb un bucle fins que s'hagin visitat tots els nodes del Graf original.

DFSRecursiu

Aquesta funció cerca dins del Graf proporcionat, els nodes que es troben al mateix regne que el node proporcionat, i és inicialment cridada des de la funció DFS la qual li indica el graf on ha de cercar, el node pel qual ha de començar a cercar (el qual hem assignat per defecte com el mateix node proporcionat), el regne on ha de buscar i l'id del node proporcionat.

El funcionament de la funció consisteix a marcar el node actual com a visitat, comprovar si aquest node pertany al regne que busquem i no és el node que inicialment ens van proporcionar (en aquest cas ho mostrem per pantalla).
I finalment trucar recursivament a la funció passant com a nou node actual els nodes adjacents a l'actual que no hagin estat visitats.

Cal esmentar que abans de trucar a aquesta funció s'ha d'executar la funció cleanVisitats() per marcar tots els nodes com a no visitats abans de començar amb l'exploració.

Dijkstra

Aquesta classe conté la funció dijkstraMillorTrajecte() la qual rep: el graf on s'ha de buscar, l'id del node origen, l'id del node destí i un booleà indicant si l'oreneta carregarà un coco en el trajecte (en aquest cas l'oreneta no pot fer trajectes de més de 50 km).
Aquesta funció executa dijkstraEuropeas() i dijkstraAfricanes() les quals calculen el millor trajecte per a una oreneta Europea i una oreneta Africana i guarda la que arribi a la destinació en menys temps.

L'algorisme Dijkstra emprat per calcular el millor trajecte de l'oreneta d'un tipus funciona de la manera següent:

En primer lloc, es realitzen les inicialitzacions de variables, on es crea un vector de distàncies i un vector de temps amb una longitud igual al nombre de nodes del graf. També es crea una llista de rutes per a cada node, inicialment buides.

A continuació, es realitzen les inicialitzacions de distàncies i temps. Totes les distàncies i temps s'estableixen en un valor infinit (representat per Float.MAX_VALUE), excepte per al node d'origen, que s'estableix en 0. Es determina l'índex del node d'origen i del node de destí dins de la cua de nodes (graph.getQueue()) i s'assigna el node d'origen com a node actual.

Després s'inicia el bucle principal, el qual s'executa fins que s'arribi al node de destí o s'hagin visitat tots els nodes que no són del clima que no pot recorre l'oreneta.
Dins d'aquest bucle es recorren tots els nodes adjacents al node actual, es verifica si el node adjacent compleix certes condicions: si el node adjacent no té el clima que no pot recorre l'oreneta i si el node adjacent no ha estat visitat.

Si es compleixen les condicions anteriors, es calcula la distància i el temps per anar des del node d'origen fins al node adjacent passant pel node actual, i en cas de que la oreneta porti coco es comprova si aquesta distancia es inferior a 50 km (50000 m).

Si el temps per arribar al node adjacent des del node d'origen és millor (més petit) que el temps emmagatzemat anteriorment, s'actualitzen les distàncies, els temps i la ruta per arribar a aquest node adjacent.

A continuació, s'indica que el node actual s'ha visitat establint la propietat "visitat" com a true i es busca el proper node no visitat amb la distància més petita des del node d'origen. Si es troba un node amb una distància no infinita (no visitat), s'estableix com a nou node actual.

Aquest procés es repeteix fins que s'arriba al node de destí o s'han visitat tots els nodes que no són del clima que no pot recorre l'oreneta. Un cop es compleix aquesta condició, es verifica si el temps per arribar al node de destí des del node d'origen és millor (més petit) que el millor temps emmagatzemat fins al moment. En cas afirmatiu, s'actualitza el millor temps, la millor distància, la millor ruta i el tipus d'oreneta que ha fet el trajecte.

Anàlisi de rendiment i resultats

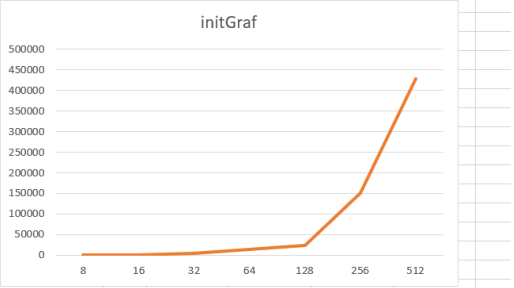
Per a cada algorisme hem fet entre 3 i 8 proves d'execució per cada dataset proporcionat, calculant la mitjana del temps que ha trigat a executar-se per cada Dataset i guardant els resultats en una gràfica [dades(x), temps d'execució(y)] per verificar-ne el cost d'aquests algorismes.

initGraph

La funció initGraph() té un cost O(N * M), on N és el nombre de llocsInteres i M és el nombre de trajectes, ja que té dos bucles on es recorren aquestes dades; per la qual cosa és una funció exponencial.

Això ho podem comprovar observant l'anàlisi dels resultats:

	Quantitat Dades		initGraf	DFS	Prim	Dijkstra
XXS	8		987	578,8	205	1231
XS	16		1574	1532	787	1900
S	32		4267	7976	2875	1294
M	64		14697	23137	12835	12287
L	128		24625	71678	37989	25678
XL	256		150449	140186	127202	11465
XXL	512		428832	794460	686492	1025943



DFS

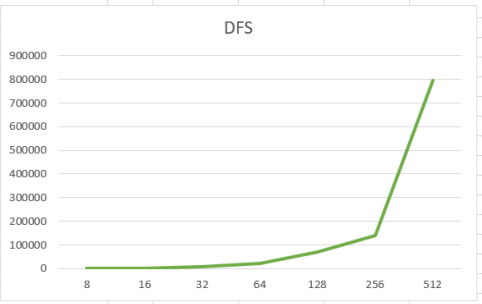
Considero que aquest algorisme té un cost lineal O(N) on N és la quantitat de LlocsInteres, ja que va fent trucades recursives en bucle per cada LlocInteres no visitat.

No obstant això, l'anàlisi del temps d'execució de la funció DFS indica un cost exponencial, considero que es perquè la funció DFSRecursiu es truca recursivament per visitar tots els llocs d'interès i a cada trucada recursiva, s'exploren els llocs d'interès

adjacents al node actual, de manera que si el nombre de llocs d'interès i trajectes al graf és gran, el nombre de trucades recursives augmentarà significativament.

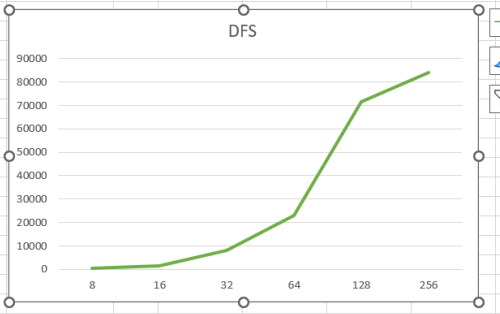
Aquestes trucades generen un cost addicional a l'algorisme i considero que el fa exponencial quan el nombre de dades N es excesivament gran.

	Quantitat Dades		initGraf	DFS	Prim	Dijkstra
XXS	8		987	578,8	205	1231
XS	16		1574	1532	787	1900
S	32		4267	7976	2875	1294
M	64		14697	23137	12835	12287
L	128		24625	71678	37989	25678
XL	256		150449	140186	127202	11465
XXL	512		428832	794460	686492	1025943



Si descartem el valor obtingut amb el dataset XXL obtenim un gràfic amb un comportament més lineal, cosa que em fa deduir que la meva suposició anterior sobre el cost agregat per les trucades recursives és correcte.

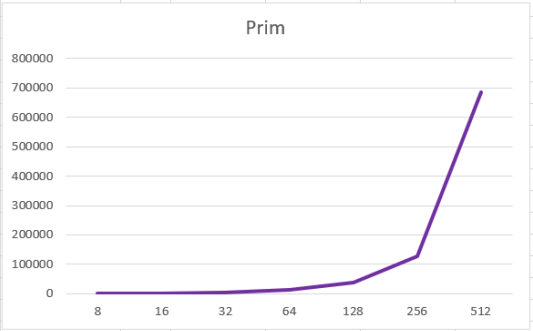
	Quantitat Dades		initGraf	DFS	Prim	Dijkstra
XXS	8		987	578,8	205	1231
XS	16		1574	1532	787	1430
S	32		4267	7976	2875	1928
M	64		14697	23137	12835	12287
L	128		24625	71678	37989	18678
XL	256		150449	83953	127202	20465
XXL	512		428832	794460	686492	44772



Prim

Aquest algorisme te cost O(N * N) (on N es el nombre de nodes) perquè l'algorisme fa un bucle que recorre els nodes fins que tots hagin estat visitats cost O(N), i dins aquest bucle executa la funció trajecteMesCurt() (el qual té cost O(N) ja que també recorre tots el nodes) a cada iteració. Per tant, és una funció exponencial i podem comprovar aquest comportament al següent anàlisi de resultats:

	Quantitat Dades		initGraf	DFS	Prim	Dijkstra
XXS	8		987	578,8	205	1231
XS	16		1574	1532	787	1900
S	32		4267	7976	2875	1294
M	64		14697	23137	12835	12287
L	128		24625	71678	37989	25678
XL	256		150449	140186	127202	11465
XXL	512		428832	794460	686492	1025943



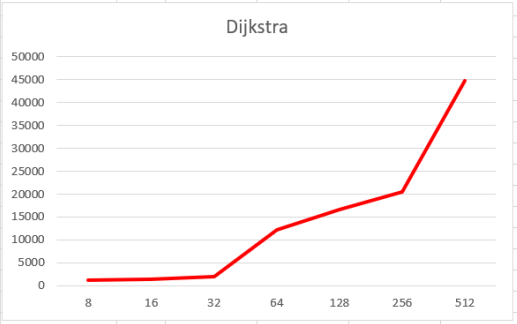
Dijkstra

Considero que el cost de la meva implementació de l'algorisme Dijkstra és O(N * log(N)) (on N es el nombre de nodes) pel fet que l'única concatenació de bucles dins del meu algorisme és: dins del bucle que recorre tots els nodes (el qual té cost O(N))), el bucle que recorre tots els nodes adjacents a l'actual (el qual en no recórrer tots els nodes del graf, sinó únicament un grup limitat considero que té cost log(N)).

Aquest cost és més òptim que l'exponencial O(N*N), però menys eficient que el lineal O(N).

Podem verificar el cost de l'algorisme en la següent anàlisi de resultats, ja que té un comportament prou semblant al cost esmentat:

	Quantitat Dades		initGraf	DFS	Prim	Dijkstra
XXS	8		987	578,8	205	1231
XS	16		1574	1532	787	1430
S	32		4267	7976	2875	1928
M	64		14697	23137	12835	12287
L	128		24625	71678	37989	16678
XL	256		150449	83953	127202	20465
XXL	512		428832	794460	686492	44772



Hem de tenir en compte que hi ha molts altres factors que afecten l'anàlisi de resultats, com ara el sistema operatiu o l'ús de memòria de l'ordinador en el moment de l'execució, per això és normal que no sempre ens surtin els gràfics perfectes i que moltes vegades tinguin desviacions com aquest.

Explicació del mètode de proves utilitzat

Principalment, hem utilitzat els datasets graphsXXS i graphsXS per testejar els algorismes, ja que al tindre poques dades són més fàcils de debugar. Però al començament, com no ens acabava de quedar molt clar el funcionament del graf ens vam crear un dataset propi que havíem dibuixat a paper per tindre una referència més visual. Un cop comprovats els algorismes amb aquests datasets vam passar a testejar-los amb datasets més grans.

Problemes observats

Stack Overflow

En algunes parts de l'algorisme, com en l'algorisme Dijkstra, en l'accés als índexs dels vectors distàncies i temps, és important garantir que els índexs estiguin dins dels límits vàlids, ja que sinó ens podem trobar errors d'índex fora de rang.

Ens vam trobar aquest tipus d'error moltes vegades i per solucionar-lo ajustàvem millor els marges dels bucles que accedien a aquests índexs.

Punters modificant el mateix objecte

Com Java treballa amb referències, va haver-hi vegades que teníem dos punters apuntant al mateix objecte i sense adonar-nos-en, quan treballàvem amb un punter modificàvem els valors per l'altre punter també.

Per solucionar això vam emprar la funcionalitat clone() de Java, la qual ens permet crear un nou objecte idèntic al proporcionat, però amb una direcció de memòria diferent.

Cicles o bucles infinits

En alguns bucles vam ajustar de forma incorrecta el moment en què havien de parar, lo qual causava que no finalitzessin mai.

Un exemple d'això va ser en l'algorisme Dijkstra, on anteriorment utilitzàvem una funció anomenada totsNodesVisitats() per aturar el bucle principal, el problema radica en que les orenetes sempre tenen nodes d'un clima pels quals no poden passar, per la qual cosa mai no visitaran tots els nodes i això causava que en alguns casos no sortíssim del bucle.

Per solucionar-ho vam crear les funcions totsNodesNoTropicalsVisitats() i totsNodesNoPolarsVisitats() les quals només comproven els nodes pels quals les orenetes poden passar.

Fase 2 - Caça de bruixes (Arbres binaris de cerca)

BTREE

Disseny de l'estructura

Per realitzar i representar l'estructura del arbres binaris vam crear diferents classes per tal de gestionar les dades de la millor manera possible.

En primer lloc, la classe Persona representa els nodes de l'arbre on guardem les dades necessàries per cada habitant (id, nom, pes, regne). A més a més a cada node guardem informació sobre altres nodes de l'arbre per tal de construir l'estructura:

- personaL: fa referència al fill esquerre respecte l'actual.
- personaR fa referència al fill dreta respecte l'actual
- personaAnterior: fa referència al node pare respecte l'actual
- profunditat: longitud des de l'arrel fins al node actual.

En segon lloc, tenim la classe BTree on representem l'arbre binari de cerca. A BTree es gestiona l'estructura i les funcionalitats. La classe BTree té "personaArrel" que consisteix en el node arrel de l'arbre que quan està buit es null. L'estructura està pensada per que sigui des del node arrel de d'on podem accedir a tota la resta.

A BTree és on es troben totes les funcionalitats per gestionar l'arbre binari, aquestes funcionalitats son: afegir node, eliminar node, recorre l'arbre, cercar un node, calcular l'alçada, balancejar.

Explicació dels algorismes implementats

Afegir habitant

La funció implementada per afegir una nova persona (node) a l'arbre binari l'hem anomenat "addPersonaRecursiva" que te com paràmetres: persona actual de l'arbre, persona que es vol afegir y la profunditat actual de l'arbre.

Sempre que es vol afegir una nova persona la primera persona que se li pasa es la l'arrel, és a dir l'algoritme sempre comença a partir de la persona arrel. Si no hi havia persona arrel perquè l'arbre estava buit s'afegeix directament la persona que es volia afegir a la posició de l'arrel. Contràriament, si l'arbre no estava buit es compara el pes de la persona que es vol afegir amb la persona actual en la que estem de l'arbre.

Si el pes de la persona és més petit que el de l'actual comprovem si existeix un fill esquerre. Si no existeix es posa la persona que es volia afegir com nou fill esquerre, s'assigna la persona anterior a la persona actual i se li assigna a la nova persona afegida la profunditat. Si existeix un fill esquerre es crida a la funció recursiva de d'addPersonaRecursiva pero passant com persona actual al fill esquerre i sumant 1 a la profunditat.

Si el pes de la persona que es vol afegir és més gran o igual a la persona actual en la que estem de l'arbre es fa el mateix que abans però pel fill dret.

Un cop s'ha afegit la nova persona a l'arbre es crida a la funció de balanceig per tal de mantenir un arbre equilibrat.

Balanceig

Per tal de balancejar el'arbre primer calculem el factor de balanceig restan-li a l'alçada de l'arbre dret l'alçada de l'arbre esquerre. Si el factor es menor que -1 vol dir que està desequilibrat i s'han de fer rotacions.

Si el factor del subarbre esquerre és més petit o igual a 0 hem de fer una rotació a la dreta respecte el node que hem afegit, però si és més gran que 0 fem una rotació a

l'esquerra en el fill esquerre del node que hem d'afegir i després fes una rotació a la dreta en la persona afegida.

Si el factor és més gran que 1 vol dir que el subarbre dret està desequilibrat i l'hem de balancejar. Si el factor de balanceig en la dreta dona més gran o igual a 0 fem una rotació a l'esquerra en el node afegit. En cas contrari es fa una rotació a la dreta en el fill dret del afegit i després una rotació a l'esquerra en el node afegit.

D'aquesta manera aquest algoritme ens permet afegir nous nodes a l'arbre de manera recursiva i despres balancejar l'arbre

Eliminar habitant

L'algoritme per eliminar un persona (node) el primer que fa és comprovar si el node te fills. En cas de que no tingui fills asigna als fills del seu pare com nulls.

Si la persona (node) només té un fill esquerre el procés és similar ja que s'assigna al pare del que es col eliminar el fill d'aquest mateix. Seguidament es crida la funció de balanceig explicada a l'apartat anterior. En el cas de que només tingui un fill dret es fa exactament el mateix que si només tenia un fill esquerre però aquest cop pel fill dret.

Finalment en cas de que el node que es vol eliminar tingui 2 fills, busquem el més gran del subarbre esquerre per substituir-lo, després modifiquem la relació del node pare amb el node substitut. Finalment fem el balanceig de l'arbre.

Representació visual (recorre)

La representació de l'arbre binari consisteix en recórrer l'arbre mostrant de forma ordenada les dades de cada node. En el funció recursiva de representació ens passem per paràmetre: la persona actual en la que estem de l'arbre que a l'inici sempre és l'arrel, un limit string definit quin tipus de fill sigui, un llistat de límits anteriores i un valor per saber si es un fill dret=1 o esquerre=-1.

La manera que hem utilitzat per recórrer l'arbre binari es inordre. En cada crida recursiva s'afegeix al llistat de límits el límit que toca, seguidament determina els límits

a la dreta i esquerra depenent de quin fill sigui. Es fan crides recursives per mostrar els fills de la dreta i els de l'esquerra per cada node actual.

Identificació de bruixes (Cerca)

Per realitzar aquest apartat hem realitzat 3 cerques diferent. En la primera, recorrem l'arbre fins trobar una persona amb el pes que estem buscant. L'algorisme realitza una cerca en profunditat (DFS) en l'arbre que consisteix en fer crides recursives amb els fills drets i després amb els fills esquerres fins que troba una persona amb el pes que estem buscant que aleshores para la crida recursiva. Si no es troba cap coincidencia es retorna null.

En la segona cerca recorrem el l'arbre sencer i retornem la persona (node) amb el pes més semblant però més petit que el que estem buscant. L'algorisme sempre comença a partir del node arrel i a partir d'aquest recorrer la resta. A la variable es guarde el node que mos compleix amb les condicions. Si el node actual es una fulla retorna encontrado. Si el node actual compleix millor les condicions que el que tenim guardat fins ara aleshores el substitueix. Es fa la crida recursiva per tots el fills drets i després per tots els fills de l'esquerra.

La tercera cerca és igual que l'anterior però buscant el pes més semblant però més gran.

Batuda

La batuda consisteix en recorre tots el nodes els qual han estat identificats anteriorment com bruixa i mostrar-los.

Anàlisi de rendiment i resultats

Afegir persona

L'algorisme de d'afegir una persona (node) te un cost de O(log n) de promig perquè l'arbre està equilibrat. En el pitjor dels casos tindria cos O(n), aquests casos seran si tots els nodes que s'introdueixen estan en ordre creixent o decreixent.

- Dataset treeXXS

```
Quina funcionalitat vol executar? A

Identificador de l'habitant: 099
Nom del habitant: 40
Pes del habitant: 33
Regne de l'habitant: LOL
Tiempo de ejecución: 31000 nanosegondos

40 ens acompanyarà a partir d'ara.
```

- Dataset treeXS

```
Quina funcionalitat vol executar? a

Identificador de l'habitant: 022
Nom del habitant: j
Pes del habitant: 54
Regne de l'habitant: lol
Tiempo de ejecución: 63600 nanosegondos

j ens acompanyarà a partir d'ara.
```

- Dataset treeS

```
Quina funcionalitat vol executar? A

Identificador de l'habitant: 055
Nom del habitant: G
Pes del habitant: 200
Regne de l'habitant: LOL
Tiempo de ejecución: 107400 nanosegondos
Duración en milisegundos: 0.1074 ms

G ens acompanyarà a partir d'ara.
```

- Dataset treeM

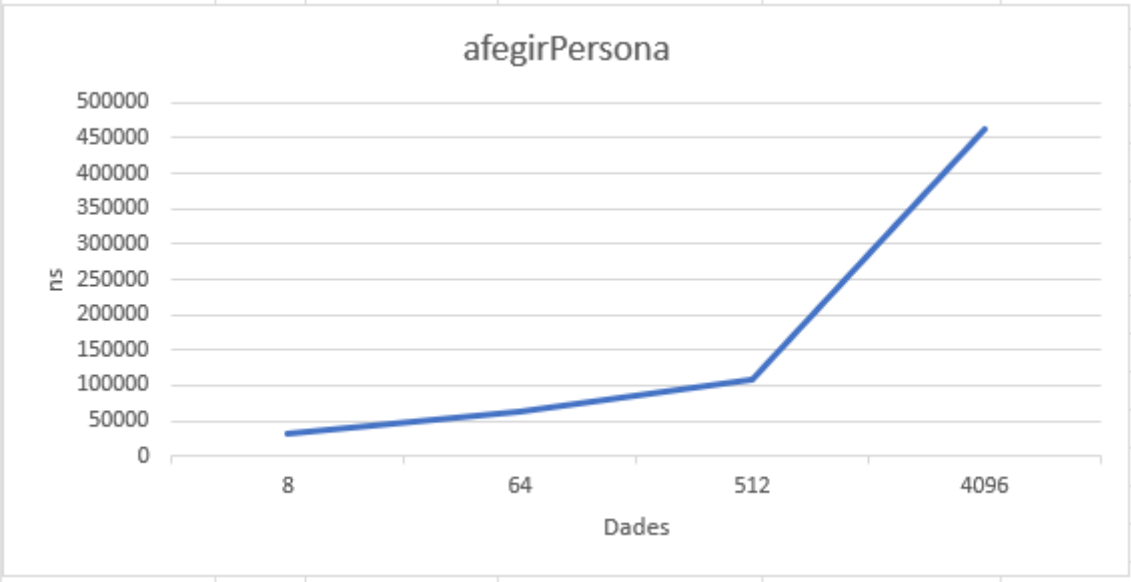
```
Quina funcionalitat vol executar? A

Identificador de l'habitant: 022
Nom del habitant: J
Pes del habitant: 200
Regne de l'habitant: LOL
Tiempo de ejecución: 463500 nanosegondos
Duración en milisegundos: 0.4635 ms

J ens acompanyarà a partir d'ara.
```

- Dataset treeL / treeXL/treeXXL

Tots aquests datasets son tan grans que surt un error de StackOverflowError això és degut a que no hi ha memoria suficient per aquests datasets.

-

- GRÀFIC: A partir del gràfic podem observar com a mesura que augmenta les persones (nodes) de l'arbre el cost en temps també augmenta.



Eliminar persona

Al igual que afegir una persona eliminarla també té un cost logaritmic de promig O(log(n)).

- Dataset treeXXS

```
Quina funcionalitat vol executar? B

Identificador de l'habitant: 099
Tiempo de ejecución: 21300 nanosegondos
40 ha estat transformat en un grill.

A. Afegir habitant
```

- Dataset treeXS

```
Quina funcionalitat vol executar? b

Identificador de l'habitant: 022
Tiempo de ejecución: 46300 nanosegondos
j ha estat transformat en un grill.
```

- Dataset treeS

```
Quina funcionalitat vol executar? B

Identificador de l'habitant: 44
Tiempo de ejecución: 277200 nanosegondos
Duración en milisegundos: 0.2772 ms
Sir Narks Hayne V the Giant ha estat transformat en un grill.
```

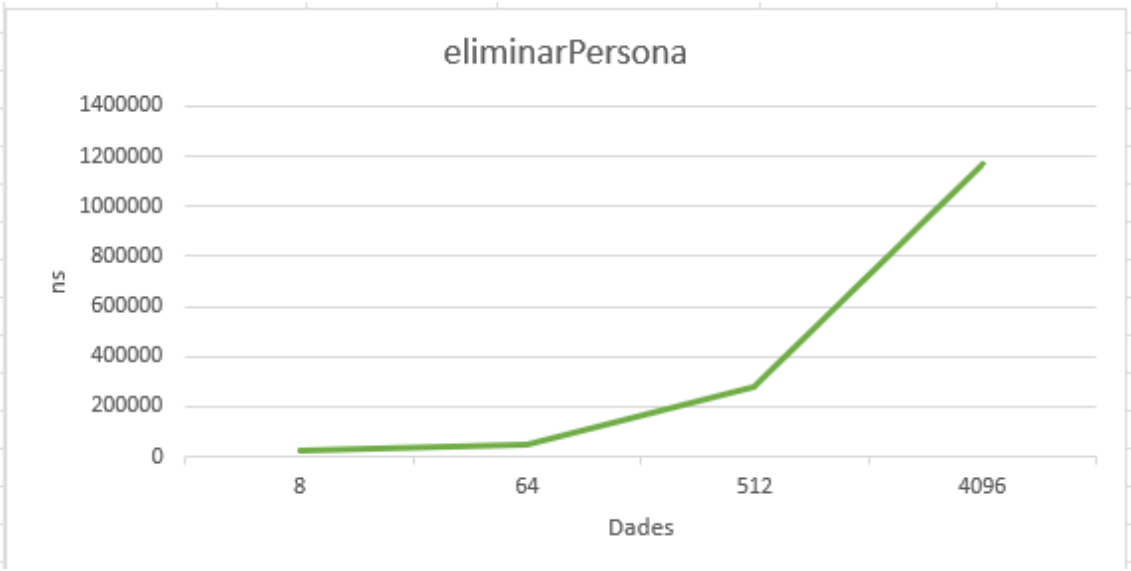
- Dataset treeM

```
Quina funcionalitat vol executar? B

Identificador de l'habitant: 4805
Tiempo de ejecución: 1168300 nanosegondos
Duración en milisegundos: 1.1683 ms
Brother Colan Brlda VII ha estat transformat en un grill.
```

- Dataset treeL / treeXL/treeXXL

Tots aquests datasets son tan grans que surt un error de StackOverflowError això és degut a que no hi ha memoria suficient per aquests datasets.

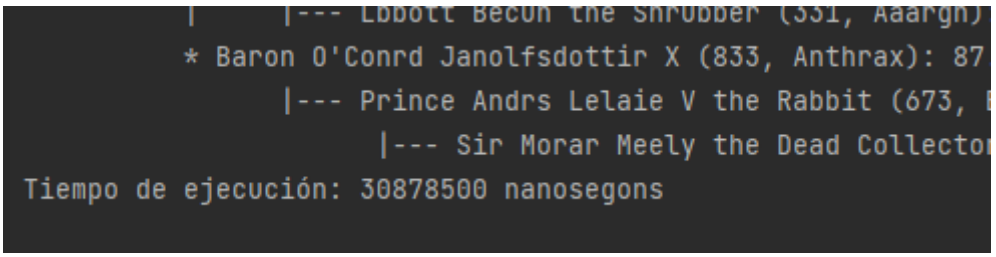
- GRÀFIC: A partir del gràfic podem observar com a mesura que augmenta les persones (nodes) de l'arbre el cost en temps també augmenta.



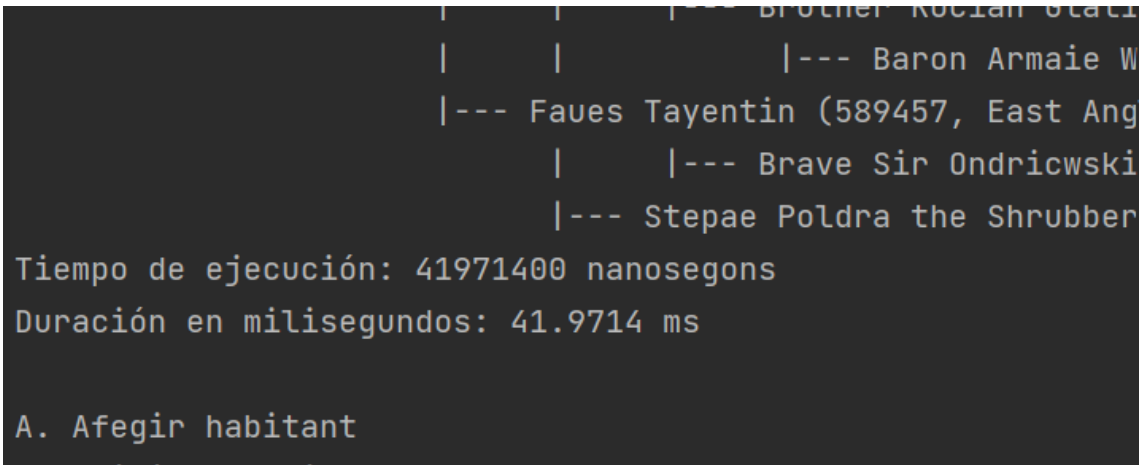
Representació (recorrer)

L'algorisme per representar visualment l'arbre té un cost O(n) ja que ha de recórrer tot l'arbre, per la qual cosa el seu cost és (hauria) proporcional a la mida de l'arbre.

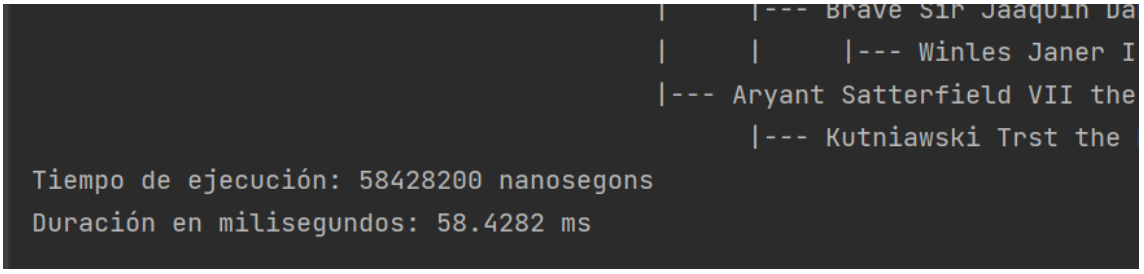
- Dataset treeXXs



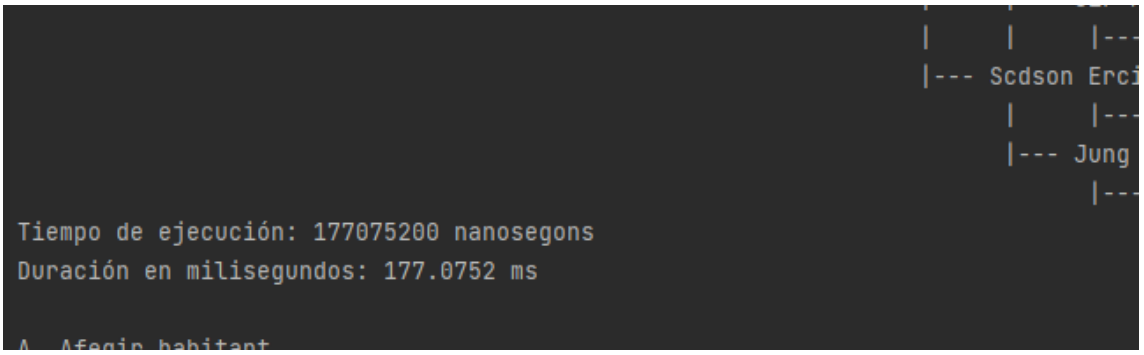
- Dataset treeXS



- Dataset treeS



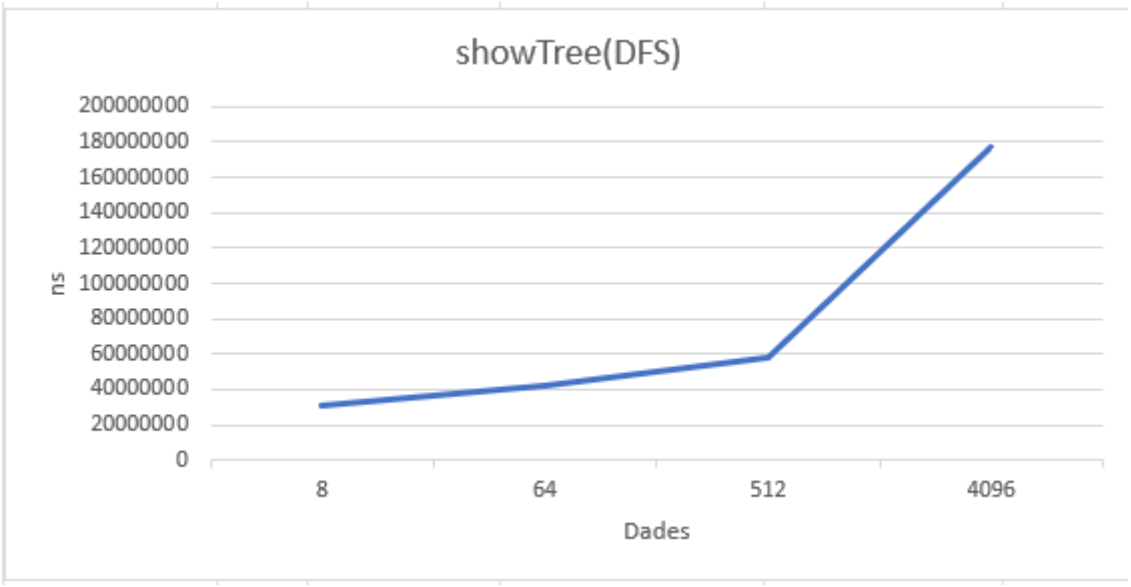
- Dataset treeM



- Dataset treeL / treeXL/treeXXL

Tots aquests datasets son tan grans que surt un error de StackOverflowError això és degut a que no hi ha memoria suficient per aquests datasets.

- GRÀFIC: A partir del gràfic podem observar com a mesura que augmenta les persones (nodes) de l'arbre el cost en temps també augmenta.



Identificació (cerca)

En el pitjor dels casos tindran un cost de O(n) ja que realitzen una cerca en profunditat (DFS) i hauran de recórrer tots els nodes de l'arbre

- Datasets treXXS

```
Quina funcionalitat vol executar? D

Nom de l'objecte: d
Pes de l'objecte: 40
Tipus de l'objecte: WOOD


Tiempo de ejecución: 39900 nanosegons
S'ha descobert 1 bruixa!
* Stiara Wrooks the Inquisitor (607, Sussex): 104.51kg
```

- Dataset treeXS

```
Quina funcionalitat vol executar? D

Nom de l'objecte: j
Pes de l'objecte: 37
Tipus de l'objecte: WOOD


Tiempo de ejecución: 45700 nanosegons
S'ha descobert 1 bruixa!
* Gayler Karton the Servant (49302400, Ni): 122.219kg
```

- Dataset treeS

```
Quina funcionalitat vol executar? D

Nom de l'objecte: j
Pes de l'objecte: 37
Tipus de l'objecte: WOOD


Tiempo de ejecución: 258500 nanosegons
Duración en milisegundos: 0.2585 ms
S'ha descobert 1 bruixa!
* Coltt Adrinon (611359843, Kent): 36.996kg
```

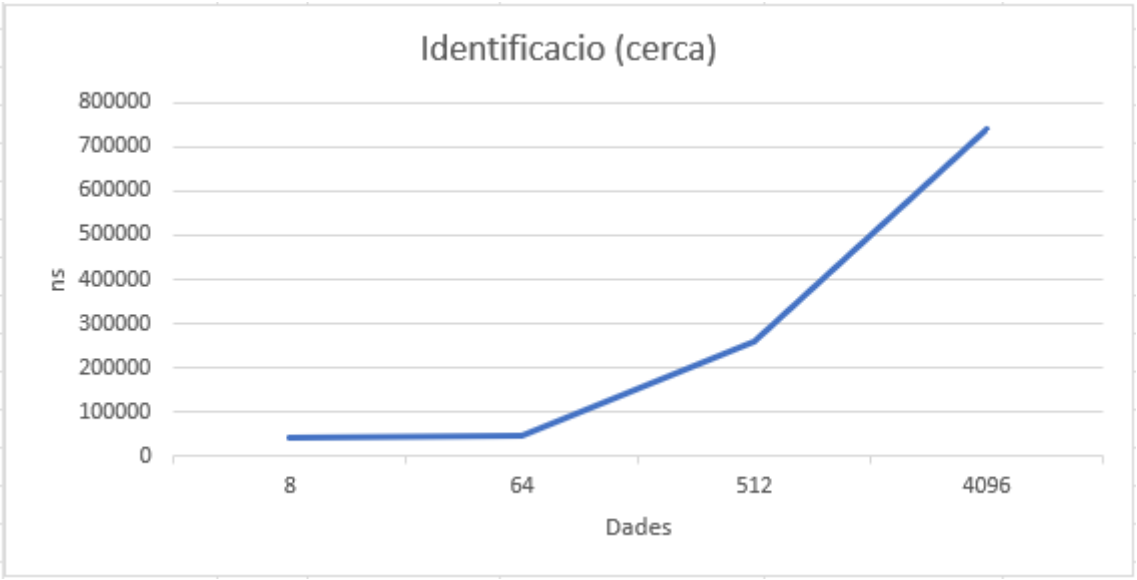
- Dataset treeM

```
iempo de ejecución: 738100 nanosegons
uración en milisegundos: 0.7381 ms
'ha descobert 1 bruixa!
 King Hily Adosie VII the Black Knight (8009408, Anthrax): 36.989kg
```

- Dataset treeL / treeXL/treeXXL

Tots aquests datasets son tan grans que surt un error de StackOverflowError això és degut a que no hi ha memoria suficient per aquests datasets.

GRÀFIC: A partir del gràfic podem observar com a mesura que augmenta les persones (nodes) de l'arbre el cost en temps també augmenta.



Explicació del mètode de proves utilitzat

Per comprovar que teniem bé els algoritmes i testajar-los van crear els nostres propis datasets en el quals sabiem com havia de quedar l'arbre ja creat. També vam utilitzar principalment el dataset treeXXS sobretot inicialment. Un com vam aconseguir que l'algoritme fes la creació de l'arbre i els altres algoritmes també funcionesin correctament vam comprovar els algoritmes als altres datasets.

Problemes observats

Un dels problemes més complicat que vam trobar va ser a l'hora de fer la visualització de l'arbre. La complicació no esta en mostrar l'arbre en ordre sino que la dificultat es trobava en escriure correctament totes les línies de profunditat dels nodes i les verticals. Aquest problema el vam poder resoldre un cop balencejat l'arbre ja que amb l'arbre desequilibrat era molt complicat.

Una altra dificultat va re fer l'algorime de eliminar nodes ja que hi ha diferents tipus de casos d'eliminar depenent que com sigui el node. Ja que si el node es fulla només

s'havia de fer modificacions simples en els pares però en cas de que no fossin fulles s'havien de fer diverses rotacions.

Fase 3 - Tanques de bardissa (Arbres R)

R-Tree

Disseny de l'estructura

Per a dissenyar i representar aquesta estructura s'ha creat diferents classes per poder gestionar la informació amb més claredat i eficiencia.

- RTree: Hem creat aquesta classe principal que contindra dos atributs, un per al màxim de fills que pot tenir cada node i un altre que sera el rectangle arrel que definim com a "root node".
- Punt: Aquesta classe únicament serveix per definir un punt que contindra unes coordenades (x, y).
- Figure: Aquesta classe es una superclase de Rectangle i Bardissa, tindrà un atribut que indicará el rectangle anterior o "pare" (És de tipus rectangle ja que sabem que tota figura que tingui pare ha de ser un rectangle). També implementara els mètodes que calgui per gestionar les figures.
- Rectangle: Aquesta classe tindrà dos Punts com a atributs que indiquen la posició on es troben dos extrems del rectangle. També tindrà un llistat de figures (filles) les quals poden ser tant rectangles com bardisses. També implementara diferents mètodes necessaris per gestionar tots els rectangles.
- Bardissa: Aquesta classe tindrà atributs com ara tipus, mida, punt i color, que defineixen una bardissa (herencia d'una figura).
- Fase3: Aquesta classe en un principi contindria funcions auxiliars (getLongestFigures, getDist) que no facin directament una simple modificació de l'arbre, però després ens vam donar compte que la podríem haver eliminat i moure els mètodes a la classe RTree.

Explicació dels algorismes implementats

agregarPuntoEnNodo

Aquest mètode té la finalitat d'afegir una nova bardissa a l'arbre R, per això, rep com a paràmetre un objecte Bardissa i crida al mètode privat "agregarPuntEnNodoRecursivo", passant com a arguments la Bardissa i el Rectangle arrel de l'arbre.

El mètode privat "agregarPuntEnNodoRecursivo" és el nucli de l'algoritme d'inserció. Comença verificant si el rectangleActual (el segon paràmetre que se li passa al mètode) és una fulla, és a dir, si no conté figures o si la primera figura que conté és una Bardissa. En aquest cas, simplement s'afegeix la Bardissa a la llista de figures del rectangleActual cridant al mètode "addFigure" de l'objecte rectangleActual.

Si el rectangleActual no és una fulla, significa que és un node intern de l'arbre. En aquest cas, l'algoritme busca el fill que millor s'ajusti al punt que es vol afegir. Recorre totes les figures contingudes en rectangleActual i calcula la distància entre la Bardissa i cada una d'aquestes figures utilitzant el mètode "getDist". L'objectiu és trobar el fill (Rectangle) que estigui més a prop de la Bardissa.

Un cop identificat el fill més proper, l'algoritme es crida a si mateix de forma recursiva, passant com a arguments la Bardissa i el Rectangle seleccionat com el millor fill. Això significa que l'algoritme continuarà descendent per l'arbre, buscant el següent nivell adequat per afegir la Bardissa.

En conclusió, l'algorisme recorre l'arbre R recursivament buscant el millor lloc per afegir la nova Bardissa. Si el node actual es una fulla, afegim directament la Bardissa al llistat de figures d'aquell node. Si no es fulla, es busca el fill més proper a la bardissa i

continuem amb el procés d'inserció en aquest fill fins trobar la fulla adient per a la nova Bardissa.

eliminarBardissa

Aquest mètode rep com a parametres les coordenades (x, y) de la bardissa que volem eliminar de l'arbre. Primerament crida al mètode "trobarBardissa" (que explicarem seguidament) per trobar la bardissa corresponent a les coordenades donades. Si no es troba cap bardissa es retorna false indicant que no ha sigut possible eliminar la bardissa, pero en cas de trobar-la, s'accedeix al rectangle anterior de la bardissa i eliminem aquesta de la llista de figures del rectangle. Seguidament actualitzem els valors maxims i minims del rectangle anterior recorrent totes les figures del rectangle.
Finalment es retorna true indicant que s'ha eliminat la bardissa amb èxit.

El mètode "trobarBardissa" s'utilitza per buscar una Bardissa a partir de dues coordenades. Rep com a paràmetres les coordenades x i y i crea una còpia d'una Bardissa amb aquestes coordenades anomenada "tarjetCopy". Després, es crida al mètode privat "trobarBardissaRecursiu" per realitzar la cerca recursiva.

El mètode "trobarBardissaRecursiu" és el nucli de l'algoritme de cerca. Si el Rectangle actual no conté cap figura, es retorna null indicant que no s'ha trobat la Bardissa.
Si el Rectangle conté figures i la primera figura és una Bardissa, es recorren totes les figures del Rectangle per comprovar si alguna d'elles coincideix amb les coordenades de la figura que estem buscant. Si es troba una coincidència, es retorna la Bardissa que hem trobat.

En cas que el rectangle contingui figures que no son bardisses, es crida recursivament el metode "trobarBardissaRecursiu" en els rectangles fills i es comprova si algun

d'aquests conté la bardissa que estem buscant amb el metode "containsFull", que retorna true o false indicant si la té o no. En cas que la tingui, es realitza una crida recursiva al mètode amb aquest Rectangle fill. Si es troba la Bardissa en alguna crida recursiva, es retorna aquella Bardissa.

Si no es troba la Bardissa en cap de les iteracions anteriors, es retorna null indicant que no s'ha trobat la Bardissa.

printTree

Aquest algoritme té com a objectiu fer la visualització de l'arbre per consola. Per fer-ho crida al mètode privat "printTreeRecursive" passant com a arguments el Rectangle arrel de l'arbre, un prefix buit inicial i un booleà que indica si és l'últim element o no.

Aquest mètode recursiu serà l'encarregat de navegar per tota l'estructura dibuixant l'arbre. Primer mostra la informació del Rectangle actual utilitzant el mètode "mostraRectangle" (que detallarem més endavant). Seguidament, comprova si el Rectangle conté figures filles. De ser així, recorre totes les figures i, per a cada una, actualitza el prefix que es mostrarà a la següent iteració. Si és l'últim fill, s'afegeix cuatre espais " " al prefix; si no és l'últim fill, s'afegeix la branca i tres espais "| ". A més, es determina si la figura actual és un Rectangle o una Bardissa.

Si la figura és un Rectangle, es crida de manera recursiva al mètode "printTreeRecursive" per mostrar la informació d'aquest Rectangle i es passa el nou prefix i el booleà corresponent.

Si la figura és una Bardissa, es mostra la informació de la Bardissa utilitzant el mètode "mostraBardissa" i es para la crida recursiva, ja que una Bardissa no té fills.

El mètode "mostraRectangle" es fa servir per mostrar a consola la informació d'un rectangle. Es mostra el prefix que s'ha anat creant recursivament seguit d'un indicador com ara " └──" o " ├──" que indica si és l'última figura del nivell o no, seguidament es mostra el nombre de figures que conté aquest rectangle i les seves coordenades.

El mètode "mostraRectangle" es fa servir per mostrar a consola la informació d'una bardissa. Es mostra el prefix que s'ha anat creant recursivament seguit d'un indicador com ara " └──" o " ├──" que indica si és l'última bardissa del nivell o no, i seguidament es mostren les seves coordenades.

cercaPerArea

Aquest mètode te com a objectiu trobar totes les bardisses que es troben dins d'unes coordenades o rectangle escollit.
Aquest rep les coordenades puntMax i puntMin que defineixen el rectangle d'àrea de cerca. En primer lloc, es crea un Rectangle "rectangleArea" utilitzant aquestes coordenades. A continuació, es crea una llista buida "bardisses" per emmagatzemar les bardisses trobades. Aleshores, es crida al mètode recursiu "cercaPerAreaRecursiu" passant el rectangleArea, la llista de bardisses i el rectangleArrel com a paràmetres.

El mètode "cercaPerAreaRecursiu" serà l'encarregat de navegar per tota l'estructura buscant les bardisses que es troben dins de l'àrea. Primer verifica si el rectangleActual no conté cap figura. En aquest cas, no fa res i retorna. Si la primera figura del rectangleActual és una Bardissa, itera sobre totes les figures d'aquest rectangle i comprova si cada figura està totalment continguda dins del rectangleArea. En cas afirmatiu, afegeix la bardissa a la llista "bardisses". Si el rectangleActual conté rectangles fills, itera sobre totes les figures i verifica si alguna d'elles està continguda en el rectangleArea. Si es compleix aquesta condició, crida recursivament el mètode

"cercaPerAreaRecursiu" amb el rectangleArea, la llista de bardisses i el rectangle fill corresponent com a paràmetres. Finalment, el mètode "cercaPerArea" retorna la llista de bardisses trobades.

OptimitzacioEstetica

Aquest algorisme d'optimització estètica té com a objectiu buscar les K bardisses més properes a un punt donat:

El mètode "OptimitzacioEstetica" rep un punt i un enter K com a paràmetres. En primer lloc, es crea una llista buida "closestBardisses" per emmagatzemar les K bardisses més properes. A continuació, es crea una altra llista buida "allBardisses" per emmagatzemar totes les bardisses de l'arbre R. Per fer això, es crida al mètode "GetBardissesFromAllRectangles" (que més endavant detallarem) passant el rectangleArrel i la llista "allBardisses" com a arguments.

Un cop obtingudes totes les bardisses, s'ordenen en funció de la seva distància al punt donat. Es fa servir la classe "Collections.sort" per ordenar la llista "allBardisses". Es crea un comparador personalitzat que compara dues bardisses en funció de la distància del seu punt màxim al punt donat. Això s'aconsegueix mitjançant el mètode "getDistanciaPunts", que calcula la distància entre dos punts.

Seguidament s'afegeixen les K bardisses més properes a la llista "closestBardisses". Es fa servir un comptador "count" per controlar el nombre de bardisses afegides. Es recorre la llista ordenada "allBardisses" i es van afegint les bardisses a "closestBardisses". Es verifica si s'ha arribat al límit de K bardisses (count >= k), i si és així, es surt de la iteració. Finalment, es retorna la llista "closestBardisses" que conté les K bardisses més properes al punt donat.

El mètode “GetBardissesFromAllRectangles” s’encarrega de recórrer tots els rectangles de l’arbre R i afegir les bardisses a la llista “bardisses”. Comença verificant si el rectangle és nul, en aquest cas, retorna. Després, es recorre la llista de figures del rectangle i si una figura és una Bardissa, s’afegeix a la llista “bardisses”. A continuació, es recorren els rectangles fills del rectangle actual de manera recursiva, cridant el mateix mètode amb el rectangle fill com a argument.

Anàlisi de rendiment i resultats

Abans de començar a analitzar el rendiment i els resultats cal remarcar que les proves s’han realitzat amb els datasets per defecte: rtreeXXS, rtreeXS, rtreeS, rtreeM, rtreeL, rtreeXL, i rtreeXXL. També comentar que en les imatges dels resultats apareixerà el temps en nanosegons a l’eix Y, i la mida del dataset a l’eix X.

agregarPuntoEnNodo

Aquest algoritme no s’ha pogut comprovar amb la inserció d’una bardissa en especific, degut a un error que comentarem en els problemes observats, és per això que he decidit fer l'anàlisi en la creació i inserció de totes les dades del fitxer.

El cost d'aquest algoritme depèn principalment de la quantitat de figures en el node actual i de l'estructura de l'arbre. En la majoria dels casos, es pot considerar que el cost de inserció és eficient, amb una complexitat propera a O(log N) en termes de l'altura de l'arbre. No obstant això, el cost pot augmentar en situacions on l'arbre estigui desbalancejat o si la quantitat de figures en els nodes és molt gran, la qual cosa podria afectar el temps d'execució de l'algoritme.

Com podem observar en la següent imatge, es pot apreciar una tendencia logarítmica, sobretot si no es té en compte el dataset més gran, que fa generar una mica de dubtes.

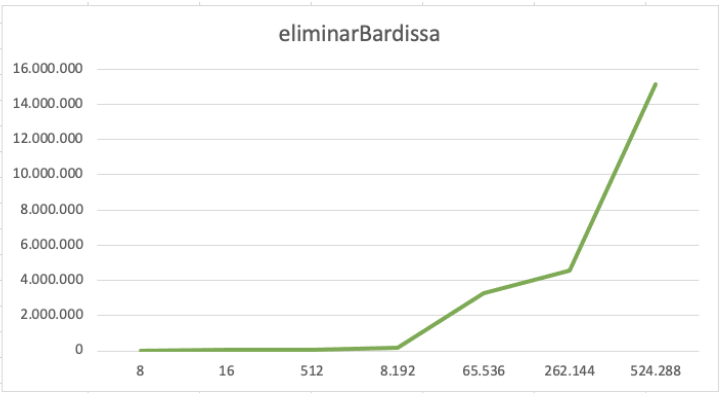
Quantitat Dades	agregarPuntoEnNodo
8	661.551
16	1.191.132
512	12.185.888
8.192	56.508.016
65.536	372.543.876
262.144	577.706.213
524.288	1.062.499.519

eliminarBardissa

El cost de l'algorisme està dominat per la cerca de la bardissa i l'actualització del rectangle anterior. Si l'arbre està balancejat i la distribució de les figures és uniforme, com es el nostre cas, el cost hauria de ser eficient, es a dir, O(log N), on N és el nombre total d'elements a l'arbre. No obstant això, si l'arbre està desbalancejat o el rectangle anterior conté un nombre significatiu de figures, el cost pot augmentar i afectar el temps d'execució de l'algorisme.

Com podem observar en la següent imatge, es pot apreciar una tendencia logarítmica, sobretot si no es té en compte el dataset més gran, que fa generar una mica de dubtes.

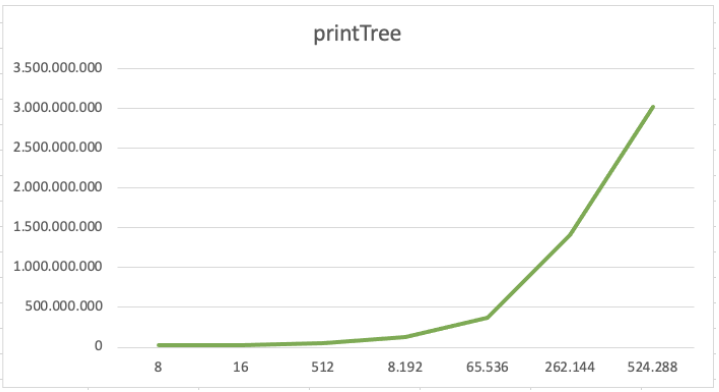
Quantitat Dades	eliminarBardissa
8	27.843
16	64.208
512	85.042
8.192	193.319
65.536	3.297.377
262.144	4.534.929
524.288	15.170.823



printTree

Aquest algoritme té un cost lineal en relació al nombre total d'elements a l'arbre. En cada crida recursiva, es processa una figura (ja sigui un rectangle o una bardissa) i es mostra la seva informació per pantalla. A més, es realitzen crides recursives per a cada fill del rectangle actual. Per tant, el temps d'execució d'aquest algoritme dependrà directament del nombre total d'elements a l'arbre. Tindrà un cost de complexitat O(N). En la següent imatge es pot observar com els resultats obtinguts augmenten al igual que augmenta la quantitat de dades.

Quantitat Dades	printTree
8	27.114.350
16	29.344.855
512	51.230.487
8.192	125.119.186
65.536	375.282.402
262.144	1.409.175.749
524.288	3.016.061.703



cercaPerArea

L'algorisme té un cost el qual depèn de diversos factors, incloent la profunditat i l'estructura de l'arbre, així com la quantitat i la distribució de les bardisses dins dels rectangles.

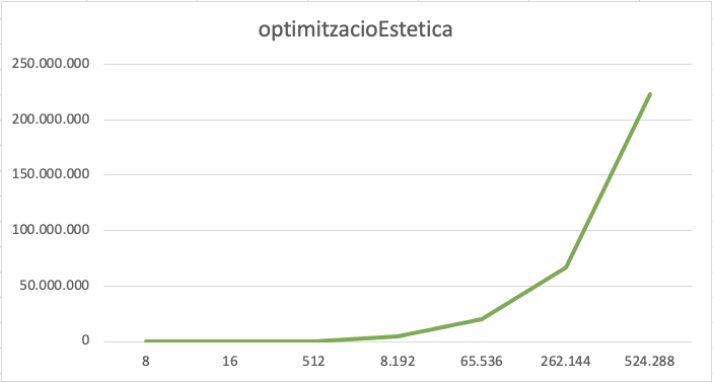
En el millor cas, amb un arbre balancejat i les bardisses distribuïdes uniformement, el cost d'execució hauria de ser proporcional a l'alçada de l'arbre, és a dir, O(log N), on N és el nombre total d'elements a l'arbre.

OptimitzacioEstetica

El cost total d'aquest algoritme depèn de la quantitat de bardisses a l'arbre (N) i del valor de k. Si l'estructura de l'arbre R està balancejada, la major part del cost recaurà en la obtenció de totes les bardisses de l'arrel i en l'ordenació de les bardisses. El cost total hauria de tenir una complexitat aproximada de O(N log N) + O(k).

Com es pot observar en la següent imatge, es pot apreciar un creixement del cost semblant a una funció exponencial. Això pot ser atribuït al fet que el cost de l'algoritme multiplica el nombre d'elements N per el logaritme de N, el qual cosa fa que el cost no sigui purament logarítmic.

Quantitat Dades	optimitzacioEstetica
8	50.719
16	76.693
512	331.537
8.192	4.750.588
65.536	20.155.847
262.144	66.842.968
524.288	222.871.959



Explicació del mètode de proves utilitzat

Per comprovar que teniem bé els algoritmes i testajar-los hem anat creant els nostres propis datasets personalitzats amb dades més visuals o senzilles a l'hora de comprovar que l'algoritme està funcionant correctament. D'aquesta forma podem forçar l'output de l'algoritme segons les dades que afegim als fitxers.

Per mesurar el rendiment de cada algorisme, hem utilitzat funcions com 'nanoTime()' de Java, que retorna el temps actual en nanosegons. Aquesta variable l'hem instanciada abans i just després de cridar al mètode que volem analitzar el seu rendiment, i finalment hem restat els dos temps, obtenint així el temps d'execució de l'algorisme.

Problemes observats

Un dels problemes que vam enfrontar va ser implementar la visualització de l'arbre. Ens va costar comprendre com guardar la informació que s'havia de mostrar a cada línia, com espais o branques.

Durant les proves, vam descobrir un error en el mètode d'afegir una nova bardissa (opció A). Després d'afegir-la, no s'incorporava correctament ja que passàvem "null" com a argument del rectangle anterior, provocant un comportament inadequat. Malgrat això, funciona correctament quan creem tota l'estructura des de zero.

En l'execució del mètode d'eliminar bardissa, hi ha un error en el controlador. Sempre mostra el missatge de que s'ha eliminat la bardissa, fins i tot si aquesta no s'elimina amb èxit. A més, si eliminem una bardissa, una de les coordenades del rectangle no s'actualitza correctament, la qual cosa provoca un mal funcionament quan intentem eliminar una bardissa del mateix rectangle posteriorment.

En l'execució del mètode de cerca per àrea ens hem donat compte que no acaba de funcionar correctament amb els datasets per defecte, hem comprovat que amb els nostres si que funciona correctament, el problema pot ser en els decimals de les coordenades.

Finalment, alguns algoritmes i mètodes no estan prou comentats, dificultant la comprensió del codi. És important proporcionar comentaris clars i descripcions adequades per facilitar la comprensió dels components implementats.

Fase 4 - D'heretges i blasfems (Taules)

Taula de Hash

Per a realizar la Fase 4 del projecte, s'ha creat un nou paquet anomenat 'Taules' on, a dins, s'hi han creat dues noves classes anomenades 'Acusat.java' i 'HashTable.java' .

La clase 'Acusat' representa un acusat amb la seva informació corresponent. L'utilitzarem en conjunt amb la clase 'HashTable' per tal d'emmagatzemar i manipular els objectes 'Acusat' en l'estructura de taula de hash.
Seguidament, la clase 'HashTable' implementa una estructura de dades en forma de taula de hash per emmagatzema i manipular elements mitjançant la clau de cerca (key). Cada element s'emmagatzema en la taula en funció d'aquesta clau, fet que permet un ràpid accés a cadascun d'aquests elements.

Disseny de l'estructura

HashTable

La classe 'HashTable' utilitza una estructura de dades basada en una llista de llistes, és a dir, un 'ArrayList<Arraylist<Acusat>>' anomenada 'hashTable', per emmagatzemar els objectes 'Acusat' en una taula de hash.
La taula de hash s'inicialitza amb un tamany determinat, i cada posició de la taula conté una llista on es poden emmagatzemar múltiples objectes 'Acusat'.

El constructor de la classe 'HashTable' rep un paràmetre 'size' que representa el tamany inicial per a la taula de hash. Dins del constructor, s'inicialitza la llista 'hashTable' i es calcula tant el tamany real de la taula de hash, com el tamany

proporcionat. Finalment, s'itera des de zero fins al tamany real de la taula i s'afegeix una llista buida en cada posició de la taula de hash.

Acusat

La classe 'Acusat' representa a una persona acusada dins el context de la fase 4.

Aquesta classe, conté els següents atributs:

- nom (String) -> Representa el nom de la persona acusada (de l'objecte 'Acusat' en concret).
- conills (int) -> Representa el nombre de conills vistos per l'acusat.
- professio (String) -> Representa la professió de la persona acusada.
- heretge (boolea) -> Indica si la persona acusada és considerada heretge o no.

La classe conté un constructor públic 'Acusat' que rep tres paràmetres d'entrada (nom, conills, professio). El constructor, inicialitza els atributs de la classe amb els valors proporcionats i estableix l'atribut herege com a fals.

Els mètodes que conté aquesta classe, són els següents:

- getNom -> Retorna el nom de l'objecte 'Acuat'.
- getConills -> Retorna el nombre de conills vistos de l'objecte 'Acusat'.
- getProfessio -> Retorna la professió de l'objecte 'Acusat'.
- isHeretge -> Retorna un booleà que indica si l'objecte 'Acusat' és considerat heretge o no.
- setHeretge -> Estableix l'atribut heretge a l'objecte 'Acusat' amb el valor indicat.

Explicació dels algorismes implementats

hashFunction

La funció 'hashFunction' pren una cadena de text anomenada 'key' com entrada i retorna un valor enter 'hash' que representa aquesta clau. Aquest valor s'utilitza posteriorment per poder accedir a la clau associada de manera més eficient.

Utilitza una suma i un mòdul per a calcular el valor de hash a partir d'una cadena de text introduïda com a paràmetre d'entrada.

Itera per cada caràcter en la cadena de text per cada clau proporcionada. Llavors, per a cadascun d'aquests caràcters es calcula la potència de 2 elevada al valor enter del caràcter. Seguidament, es sumen aquests valors a la variable hash i, aquesta, es redueix utilitzant l'operació mòdul (%) per el tamany especificat, en aquest cas, size.

insert

La funció rep dos paràmetres d'entrada. Primer de tot, 'key', que representa la clau de l'objecte 'Acusat' que volem insertar. I, seguidament, 'value' que representa l'objecte 'Acusat' que s'afegirà a la taula de hash.

A continuació, s'utilitza la funció 'hashFunction' per tal d'obtenir l'índex corresponent a la llista 'hashTable' i, llavors, s'accedeix a la llista en la posició de l'índex obtingut i s'afegeix l'objecte, que li hem passat per la variable 'value', a la llista.

get

La funció 'get' busca i retorna un objecte 'Acusat' d'una taula de hash, en cas de que la clau específica introduïda com a paràmetre coincideixi amb la de l'objecte.

La funció, rep un paràmetre 'key' que representa la clau de l'objecte 'Acusat' que es vol obtenir de la taula de hash. Seguidament, s'utilitza la funció 'hashFunction' amb el paràmetre 'key' per tal d'obtenir l'índex corresponent en la llista 'hashTable'.

Seguidament, s'itera sobre cada objecte 'Acusat' que es troba en la llista corresponent a la posició de l'índex obtingut. Per a cadascun d'aquests objectes 'Acusat', es verifica si el valor de la clau es igual a la clau 'key' proporcionada. En cas de coincidir es retorna l'objecte 'Acusat'.

delete

La funció 'delete' elimina un objecte 'Acusat' d'una taula de hash en funció de la clau específica introduïda com a paràmetre.

La funció, rep un paràmetre 'key' que representa la clau de l'objecte 'Acusat' que es desitja eliminar de la taula de hash. S'utilitza la funció 'hashFunction(key)' per obtenir l'índex corresponent a la llista 'hashTable'.

Seguidament, s'itera sobre cada objecte 'Acusat' en la llista que es troba en l'índex obtingut. Per a cadascun d'aquests objectes 'Acusat' es verifica si el valor de la clau és igual a la clau 'key' proporcionada.

Si es troba un objecte 'Acusat' que coincideix amb la mateixa clau, aquest s'elimina de la llista 'hashTable'.

marcarHeretge

La funció 'marcarHeretge' realitza una operació per marcar/definir un 'Acusat' com heretge o no, segons certes condicions relacionades amb la seva professió.

La funció, pren una clau 'key'i un boleà 'heretge' com a paràmetres.

Primer de tot, s'obté una instància de la clase 'Acusat' utilitzant la clau proporcionada a través del mètode 'get(key)'. Seguidament, es verifica si la professió de l'instància 'Acusat' no és igual a "KING", "QUEEN" o "CLERGYMAN" utilitzant el mètode 'getProfessio()'. Si la professió no coincideix amb cap d'aquestes tres opcions,

s'actualitza el valor de la propietat 'heretge' de l'instància 'Acusat' amb el valor proporcionat 'heretge' utilitzant el mètode 'setHeretge(heretge)'.

judiciFinal

La funció 'judiciFinal' realitza un judici per dictar sentència sobre un 'Acusat' en funció dels conills vistos i de les condicions de la seva professió.

La funció, pren dos paràmetres 'minConills' i 'maxConills', que representen els límits inferior i superior del nombre de conills que s'està avaluant.

Seguidament, s'itera sobre la llista 'acusats' en l'estructura 'hashTable' i, per cada objecte 'Acusat', es verifica si el número de conills (mitjançant el mètode 'acusat.getConills()') està dins del rang especificat per 'minConills' i 'maxConills'.

Si el nombre de conills compleix la condició, s'imprimeix la informació de l'acusat.

showHistogram

La funció 'showHistogram' mostra un histograma d'informació per a cada objecte 'Acusat' emmagatzemat dins l'estructura de dades 'hashTable'.

La funció itera sobre cada llista 'Acusat' en l'estructura de dades 'hashTable'. Seguidament, dins de cada llista, itera sobre cada objecte 'Acusat' i, per a cadascun d'ells, imprimeix la seva informació.

Anàlisi de rendiment i resultats

insert

Per tal d'analitzar el rendiment de la funció insert, executarem la funció a partir de dos Datasets diferents amb una forta diferència entre el seu nombre de dades.

Seguidament podem observar l'anàlisi del temps d'execució en funció del Dataset d'entrada:

Dataset TablesM.paed

```
Nom de l'acusat: Joan
Nombre de conills vistos: 14
Professio: KING
S'ha enregistrat un nou possible heretge.
Tiempo de ejecución: 167100 ns
Tiempo de ejecución en milisegundos: 0.1671 ms
```

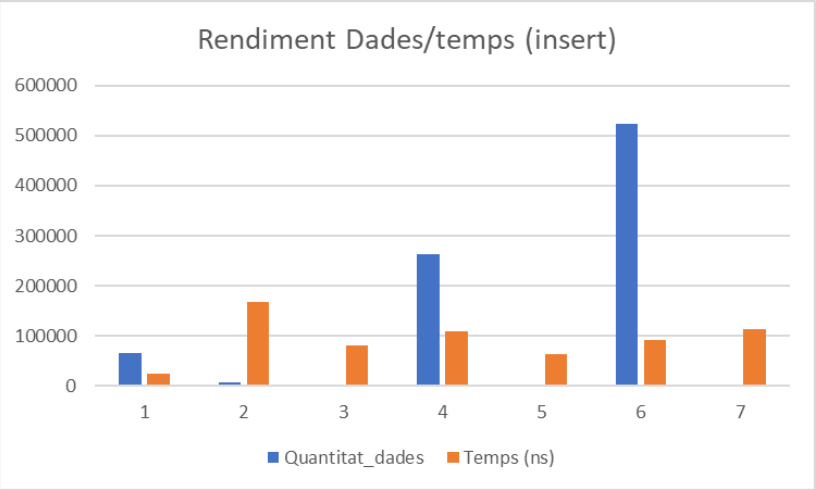
Dataset TablesXXL.paed

```
Quina funcionalitat vol executar? A

Nom de l'acusat: Miquelin
Nombre de conills vistos: 4
Professio: KING
S'ha enregistrat un nou possible heretge.
Tiempo de ejecución: 92800 ns
Tiempo de ejecución en milisegundos: 0.0928 ms
```

A simple vista, sembla ser que la funció no depèn de la quantitat de dades del fitxer, així doncs, per tal de comprovar-ho millor, realitzarem un anàlisi amb tots els casos dels fitxers:

Quantitat_dades	Temps (ns)
65536	24.356
8192	167.100
512	80.800
262144	108.600
16	63.500
524288	92800
8	114.600

Seguidament, podem observar la gràfica (Quantitat_dades, temps) per visualitzar millor el seu rendiment:



Al observar les dades del gràfic obtingudes, podem notar que el temps d'execució no segueix un patró simple i no sembla estar directament relacionat amb el tamany de les dades.

En general, podem dir que el temps d'execució no sembla seguir una relació lineal directa, fet que suggereix que la funció pot tenir un cost millor que lineal.

Donat que la funció 'insert' simplement realitza una operació d'inserció a la taula de hash, i no implica cerques o eliminacions complexes, podem estimar que tindrà un cost promig prop de O(1) o constant. Això, significa que el temps d'execució no augmenta proporcionalment amb el tamany del nombre de dades, sino que es manté relativament constant.

delete

El rendiment d'aquesta funció depèn de l'estructura de dades 'hashTable' i, també, de l'eficiència de la funció 'hashFunction'.

Per altra banda, la funció té un temps d'execució proprcional al tamany de la llista que es troba en la posició de l'índex obtingut. En el pitjor dels casos, si s'han de recórrer tots els elements de la llista per trobar l'objecte 'Acusat' amb la clau que coincideixi, pot tenir un temps lineal, és a dir, amb un cost O(n) o 'n' equival al nombre d'objectes 'Acusat' que conté la llista.

Per tal d'analitzar el rendiment de la funció insert, executarem la funció a partir de dos Datasets diferents amb una forta diferència entre el seu nombre de dades.

Seguidament podem observar l'anàlisi del temps d'execució en funció del Dataset d'entrada:

Dataset TablesL.paed

```
Quina funcionalitat vol executar? B

Nom de l'acusat: Rorenzo
L'execució pública de Rorenzo ha estat un èxit.
Tiempo de ejecución: 23408400 ns
Tiempo de ejecución en milisegundos: 23.4084 ms
```

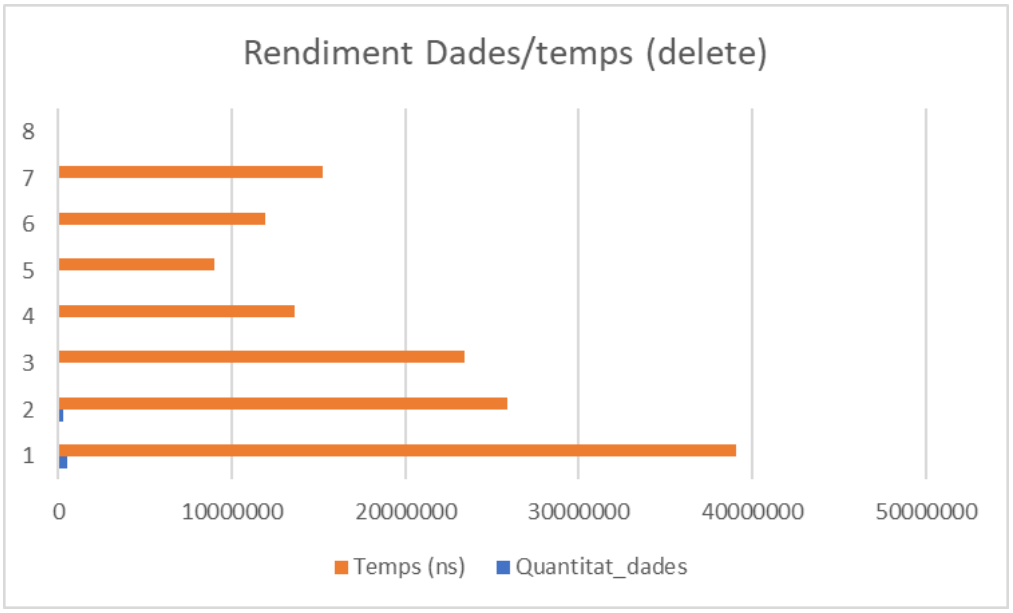
Dataset TablesXXS.paed

```
Quina funcionalitat vol executar? B

Nom de l'acusat: Jirlon
L'execució pública de Jirlon ha estat un èxit.
Tiempo de ejecución: 15253500 ns
Tiempo de ejecución en milisegundos: 15.2535 ms
```

A simple vista, sembla ser que la funció depèn de la quantitat de dades del fitxer, tal com haviem comentat al principi, així doncs, per tal de comprovar-ho millor, realitzarem un anàlisi amb tots els casos dels fitxers:

Quantitat_dades	Temps (ns)
524288	39.054.500
262144	25.903.800
65536	23.408.400
8192	13.638.500
512	8.978.000
16	11.917.300
8	15.253.500

Seguidament, podem observar la gràfica (Quantitat_dades, temps) per visualitzar millor el seu rendiment:



En conclusió, podem dir que el temps d'execució de la funció 'delete' és sensible al tamany de dades, però no de forma lineal.
Donat que la funció implica buscar un element a la taula de hash, el seu rendiment està influenciat per la distribució d'elements en la taula i l'eficiència de la funció hash.
Tot i així, si que podem confirmar que en el pitjor dels casos (on s'hagués de recorrer tots els elements 'Acusats' emmagatzemats en la cerca) ens estarem posicionant en un cost de O(n) amb n com a nombre total d'elements emmagatzemats.

marcarHeretge

Per tal d'analitzar el rendiment de la funció insert, executarem la funció a partir de dos Datasets diferents amb una forta diferència entre el seu nombre de dades.
Seguidament podem observar l'anàlisi del temps d'execució en funció del Dataset d'entrada:

Dataset TablesXXL.paed

```
Quina funcionalitat vol executar? C

Nom de l'acusat: Rewana Kearee I
Marcar com a heretge (Y/N)?Y
La Inquisició Espanyola ha conclòs que Rewana Kearee I és un heretge.
Tiempo de ejecución: 25503300 ns
Tiempo de ejecución en milisegundos: 25.5033 ms
```

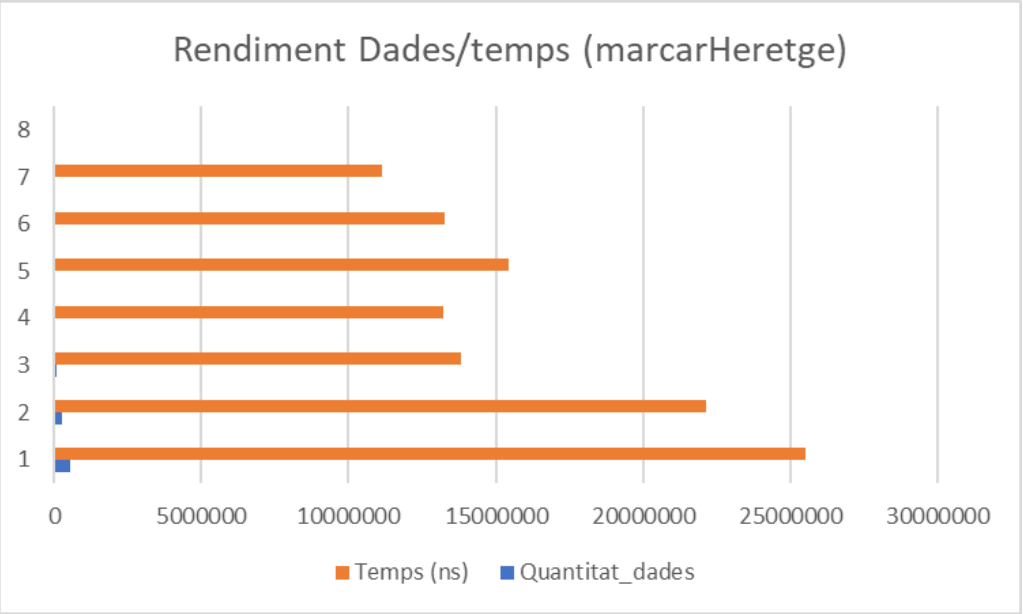
Dataset TablesXXS.paed

```
Quina funcionalitat vol executar? C

Nom de l'acusat: Millr Elake
Marcar com a heretge (Y/N)? Y
La Inquisició Espanyola ha conclòs que Millr Elake és un heretge.
Tiempo de ejecución: 11126600 ns
Tiempo de ejecución en milisegundos: 11.1266 ms
```

A simple vista, sembla ser que la funció depèn de la quantitat de dades del fitxer, tot i que la diferència és pràcticament mínima. Així doncs, per tal de comprovar-ho millor, realitzarem un anàlisi amb tots els casos dels fitxers:

Quantitat_dades	Temps (ns)
524288	25.503.300
262144	22.155.600
65536	13.839.300
8192	13.237.900
512	15.444.400
16	13.248.400
8	11.126.600

Seguidament, podem observar la gràfica (Quantitat_dades, temps) per visualitzar millor el seu rendiment:



Si observem les dades, podem notar que el temps d'execució varia de manera irregular a mesura que canvia el tamany de les dade. No sembla seguir un patró clar degut al valor obtingut al fitxer amb 512 dades, que té un temps superior al fitxer de 8192 dades. Tot i així, si que s'observa que, quan augmenta de forma elevada el valor de les dades, el temps d'execució també s'incrementa degut a que la cerca de l'Acusat ha d'iterar una quantitat superior d'objectes.

Dins de la funció 'marcarHeretge' estem realitzant una crida al mètode 'get' per buscar l'element dins la taula de hash, per tant, el rendiment d'aquesta funció 'marcarHeretge' dependrà en gran mesura del rendiment del mètode 'get' també.

Si assumim que la funció 'get' té un rendiment promig proper a O(1), constant, llavors podriem estimar que la funció marcarHeretge també té un cost promig proper a ser constant, O(1).

judiciFinal

Per tal d'analitzar el rendiment de la funció insert, executarem la funció a partir de dos Datasets diferents amb una forta diferència entre el seu nombre de dades.

Seguidament podem observar l'anàlisi del temps d'execució en funció del Dataset d'entrada:

Dataset TableL.paed

```
Quina funcionalitat vol executar? D

Nom de l'acusat: Klocine Koen III
Registre per "Klocine Koen III":
    * Nombre de conills vistos: 377
    * Professio: SHRUBBER
    * Heretge? false
Tiempo de ejecución: 10974200 ns
Tiempo de ejecución en milisegundos: 10.9742 ms
```

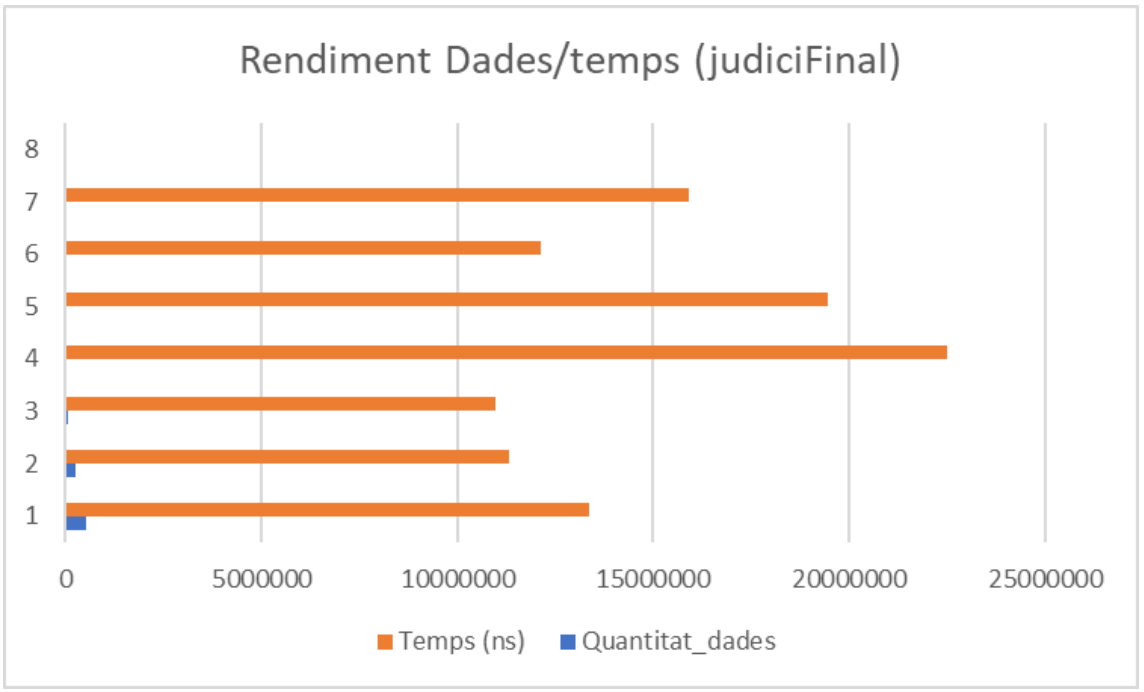
Dataset TableXXS.paed

```
Nom de l'acusat: Jirlon Welclis VIII
Registre per "Jirlon Welclis VIII":
    * Nombre de conills vistos: 2411
    * Professio: QUEEN
    * Heretge? false
Tiempo de ejecución: 15923900 ns
Tiempo de ejecución en milisegundos: 15.9239 ms
```

Observem que a simple vista, el temps d'execució no augmenta en funció de la quantitat de dades. Tot i així, realitzarem un estudi de cada cas (Dataset) per veure millors resultats:

Quantitat_dades	Temps (ns)
524288	13.380.000
262144	11.314.400
65536	10.974.200
8192	22.517.000
512	19.472.400
16	12.139.400
8	15.923.900

Seguidament, podem observar la gràfica (Quantitat_dades, temps) per visualitzar millor el seu rendiment:



Observem que mitjaçant només aquestes dades no podem extreure un resultat gaire precís del que està passant. Així doncs, realitzarem un altre anàlisi, en aquest però mitjançant la funció 'judiFinal' dins d'un rang determinat.

Si ens fixem ara en l'execució d'aquesta funcionalitat, podem observar el següent:

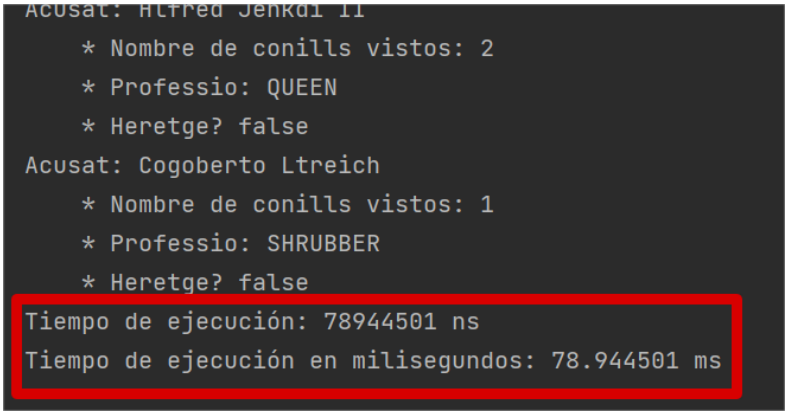
Dataset TablesM.paed

```
Quina funcionalitat vol executar? E

Nombre mínim de conills: 1
Nombre màxim de conills: 3
Acusat: Corar Marmiston III
    * Nombre de conills vistos: 2
    * Professio: KNIGHT
    * Heretge? false
Acusat: Murrae Willdio X
    * Nombre de conills vistos: 2
    * Professio: PEASANT
    * Heretge? false
Tiempo de ejecución: 3662100 ns
Tiempo de ejecución en milisegundos: 3.6621 ms
```

Dataset TablesXXL.paed

```
Quina funcionalitat vol executar? E

Nombre mínim de conills: 1
Nombre màxim de conills: 3
Acusat: Moscly Armstne
    * Nombre de conills vistos: 3
    * Professio: KING
    * Heretge? false
Acusat: Nion Mcyle IV
    * Nombre de conills vistos: 1
    * Professio: SHRUBBER
    * Heretge? false
Acusat: Brave Sir Glb Mraham
    * Nombre de conills vistos: 2
```

(...)



Amb aquests dos outputs, podem veure clarament que si utiltzem la funció judiciFinal dins un rang, la quantitat de dades té una relació directa amb el temps resultant d'execució.

Si observem totes les dades obtingudes, podem veure que el temps d'execució del primer anàlisi, no segueix una relació lineal directa amb el tamany de dades, però, en canvi, en el segon anàlisi (el del judicFinal dintre d'un rang) si que s'observa que hi ha una relació directa amb el tamany de dades.

Sabem que en la funció 'judiciFinal' realitzem dos bucles per recórrer la taula de hash i els elements dins de cada llista. Teòricament, el rendiment de la funció hauria de dependre del nombre total d'elements dins la taula de hash i la quantitat d'elements que compleixin les condicions de dintre els bucles.

Si asumim que el mètode 'getConills' té un rendiment constant, el rendiment de la funció 'judiciFinal' hauria d'estar dominat pel nombre d'elements dins la taula de hash i el nombre d'elements que compleixen les condicions. En aquest escenri, podriem estimar que el cost de la funció 'judiciFinal' sería aproximadament proporcional al nombre d'elements, m, que compleixen les condicions, en comptes del nombre total d'elements que hi ha a la taula. Per tant, podem dir que seria un cost de O(m), on m són els valors que compleixen les condicions.

showHistogram

Per tal d'analitzar el rendiment de la funció insert, executarem la funció a partir de dos Datasets diferents amb una forta diferència entre el seu nombre de dades.

Seguidament podem observar l'anàlisi del temps d'execució en funció del Dataset d'entrada:

Dataset TablesXXL.paed

```
    * Nombre de conills vistos: 4397
    * Professio: KING
    * Heretge? false
Acusat: Lvana Bertraski X
    * Nombre de conills vistos: 1378
    * Professio: MINSTREL
    * Heretge? false
Acusat: MarRueden Ankg III
    * Nombre de conills vistos: 1552
    * Professio: QUEEN
    * Heretge? false
Tiempo de ejecución: 6442065200 ns
Tiempo de ejecución en milisegundos: 6442.0652 ms
```

Dataset TablesXXS.paed

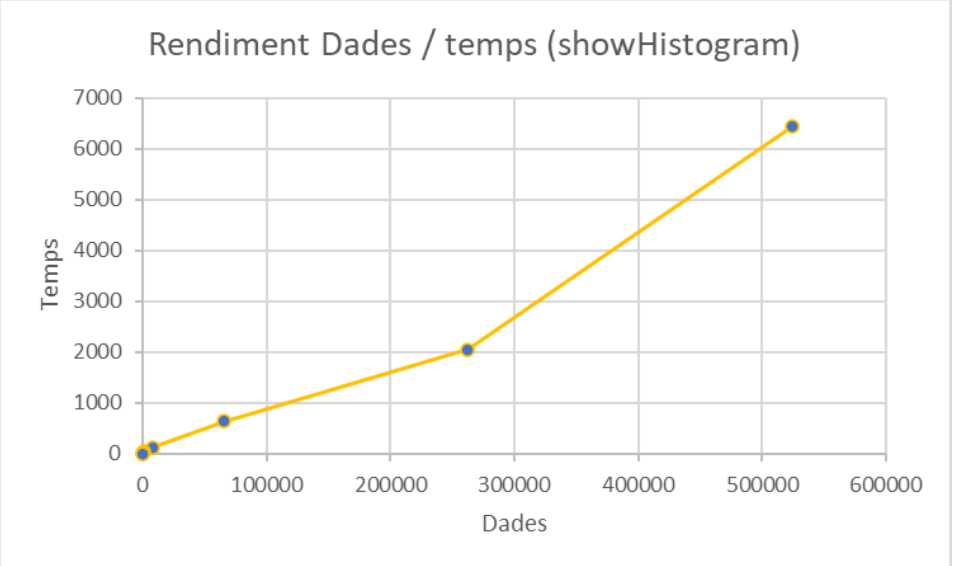
```
Acusat: Sir Upyon Cassn
    * Nombre de conills vistos: 3071
    * Professio: KING
    * Heretge? false
Acusat: Vee Jacobson VI
    * Nombre de conills vistos: 3332
    * Professio: MINSTREL
    * Heretge? false
Acusat: Jirlon Welclis VIII
    * Nombre de conills vistos: 2411
    * Professio: QUEEN
    * Heretge? false
Tiempo de ejecución: 8297600 ns
Tiempo de ejecución en milisegundos: 8.2976 ms
```

Observem que a simple vista, el temps d'execució augmenta considerablement en funció de la quantitat de dades. Tot i així, realitzarem un estudi de cada cas (Dataset) per veure millors resultats:

Quantitat_dades	Temps (ns)	Temps (ms)
524288	6.442.065.200	6442
262144	2.058.982.600	2058
65536	636.549.500	636
8192	127.539.100	127
512	40.260.800	40
16	11.561.500	11
8	8.297.600	8

Seguidament, podem observar la gràfica (Quantitat_dades, temps) per visualitzar millor el seu rendiment:

(En aquest cas, utilitzem la dada temps en (ms) per tal de veure millor el gràfic).



En aquest cas, veiem clarament que el rendiment de la funció 'showHistogram' està influenciat pel nombre total d'elements dins la taula de hash. La majoria dels elements han de ser impresos, per tant, el temps d'execució de la funció està directament relacionat amb la quantitat d'aquests elements que hi ha.
Donat que la funció recorre tots els elements de la taula de hash i realitza operacions d'impressió per a cadascun d'ells, podem estimar que el seu cost és proporcional al nombre total d'elements, n. En aquest cas, podem dir que el cost serà de O(n), on n sigui el número total d'elements a la taula de hash.

Explicació del mètode de proves utilitzat

Per tal de realitzar totes les proves de cadascuna de les funcions, el que hem fet es inicialitzar una variable amb el temps prèviament a l'execució de la funció i, un cop s'ha cridat la funció i ha realitzar el seu procés, realitzem un càlcul del temps final menys el temps inicial per obtenir el transcurs de temps durant l'execució de la funció.
Seguidament, hem anat realitzant les execucions de cada funció amb tots els diferents Datasets que tenim, tenint en compte el valor de dades que conté cadascun.

A més, hem tingut en compte que quan s'havia d'introduir els noms dels Acusats, hem introduït sempre l'últim nom del fitxer. D'aquesta forma, obliguem al mètode a recórrer tot el fitxer de dades per tal de trobar l'objecte que estem apuntant.

Finalment, a partir d'aquestes dades, hem realitzat els diferents gràfics per tal de poder analitzar els resultats d'una forma clara i coherent.

Conclusions

Durant el desenvolupament del projecte, hem tingut l'oportunitat de treballar en quatre fases diferents, cadascuna d'elles centrada en el tractament d'estructures de dades de forma diferenciada: Grafs, Arbres binaris (B-Tree), R-Tree i Taules de Hash.
Al llarg d'aquest procés, hem implementat les funcionalitats requerides i hem analitzat el rendiment i el cost dels diferents algoritmes aplicats a cadascuna d'aquestes estructures.

Pel que fa a nivell tècnic, aquest projecte ens ha brindat experiència en el camp de la programació i la manipulació de les diferents estructures de dades. Hem pogut posar en pràctica els coneixements adquirits al llarg de l'assignatura sobre cadascuna de les estructures de dades mencionades, comprenen les seves característiques, avantatges, inconvenients, així com la seva aplicació en diferents situacions.

Un dels aspectes més destacats del projecte ha sigut l'anàlisi exhaustiu del rendiment i el cost de tots els algorismes implementats. Hem avaluat la complexitat temporal (O) i els temps d'execució dels algorismes de cadascuna de les fases. Aquests anàlisis ens han permès entendre en profunditat l'impacte de les nostres decisions de disseny i algorismes en el rendiment global del projecte. A més, ens ha ofert una visió clara sobre com millorar l'eficàcia dels nostres programes i optimitzar els recursos que utilitzem en cada moment.

En quant a l'aspecte més personal, hem desenvolupat habilitats de resolució de problemes al llarg del projecte. Cada fase ens ha presentat desafiaments específics relacionats amb la manipulació de les estructures de dades corresponents, fet que ens ha exigit aplicar els nostres coneixements i enfocar la nostra manera de pensar des de diferents punts de vista per tal de trobar solucions el màxim d'eficients possibles per a

cada problema. Aquesta experiència ens ha permès millorar la nostra capacitat d'anàlisi i disseny a l'hora de programar.

Per últim, el treball en equip i la col·laboració entre els diferents membres del grup ha sigut fonamental per tal d'assolir els objectius del projecte. Hem treballat de forma estreta, compartint idees i punts de vista, debatin diferents solucions i distribuint-nos les diferents tasques a realitzar, però sempre mantenint-nos al dia amb la feina que duia a terme cadascun dels membres per tal de poder reutilitzar els seus coneixements i la seva feina. Aquesta experiència ens ha demostrat la importància de la comunicació efectiva entre els diferents membres i la sinergia en un entorn de treball col·laboratiu, el qual segur que ens serà de gran utilitzat en futurs projectes.

En resum, el projecte 'Els Cavallers de la Taula de Hash' ens ha permès consolidar els coneixements apresos al llarg de l'assignatura 'Programació Avançada i Estructura de Dades' i ens ha permès millorar les nostres habilitats de programació en Java i ampliar la nostra comprensió envers les diferents estructures de dades treballades. A més, també s'han desenvolupat habilitats personals com el correcte treball en equip, per tal de facilitar la feina als companys amb qui s'ha treballat.

Bibliografia

Bibliografia referenciada segons la norma ISO 690:2010 per a pàgines web.

Tutoriales.edu. *Estructura de datos y algoritmos: Tabla hash* [Data de consulta: Maig de 2023].
Disponible a:
https://tutoriales.edu.lat/pub/data-structures-algorithms/hash-data-structure/estructura-de-datos-y-algoritmos-tabla-hash

LaSalle URL. *Programació Avançada i Estructura de Dades*. *Inserció Arbres R.pdf* [Data de consulta: De Abril a Maig de 2023].
Disponible a:
https://estudy2223.salle.url.edu/course/view.php?id=1324&sectionid=1456

GeeksForGeeks. *Introduction to R-Tree* [Data de consulta: De Abril a Maig de 2023].
Disponible a: https://www.geeksforgeeks.org/introduction-to-r-tree/

DelftStack. *Implementar árbol en Java* [Data de consulta: Abril de 2023].
Disponible a: https://www.delftstack.com/es/howto/java/java-tree/

LaSalle URL. *Programació Avançada i Estructura de Dades*. *Apunts Semestre 2.pdf* [Data de consulta: De Març a Maig de 2023].
Disponible a:
https://estudy2223.salle.url.edu/course/view.php?id=1324&sectionid=1456

Baeldung. *B-Tree Data Structure.* [Data de consulta: De Març a Abril de 2023].
Disponible a: https://www.baeldung.com/cs/b-tree-data-structure

LaSalle URL. *Programació Avançada i Estructura de Dades*. *Apunts Extra TRIES.pdf* [Data de consulta: Maig de 2023].
Disponible a:
https://estudy2223.salle.url.edu/course/view.php?id=1324&sectionid=1456

Baeldung. *Graphs in Java.* [Data de consulta: De Març a Maig de 2023].
Disponible a: https://www.baeldung.com/java-graphs

Google Sites. *Estructuras de datos en java Rotaciones simples.* [Data de consulta: De Març a Abril de 2023].

Disponible a:
https://sites.google.com/a/espe.edu.ec/programacion-ii/home/arboles/arboles-avl/rotaciones-simples