Word Counter

Programació avançada i estructura de dades

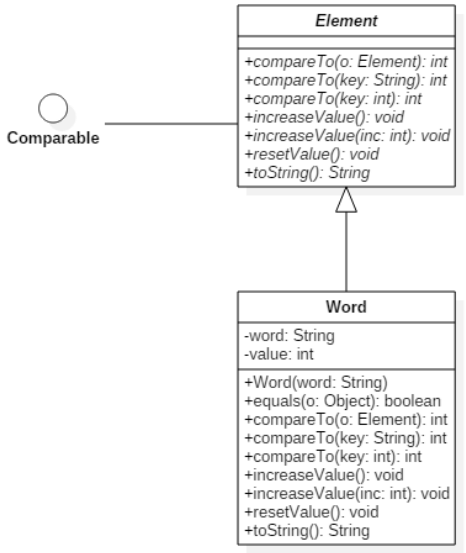
Joan Sanfeliu Vilarrassa - ls30498
Àlex Jordà Triginer - ls30687

Índex

1. Disseny dels modes de recompte

Al inici de la pràctica vam començar decidint les quatre estructures amb les que volíem experimentar i ens vam decantar per fer un arbre de cerca, un arbre AVL, una taula encadenada indirecta amb arbres AVL a mode de llistes de sinònims i, per tal de realment veure la diferència entre aquestes estructures i una estructura de dades lineal, una llista ordenada.

Abans de començar amb les diferents estructures caldria veure quin és l'element amb el que treballem:



L'objecte que estem llegint del fitxer i treballant amb ell a les diferents estructures serà de la classe "Word", que hereta d'Element. Està fet d'aquesta manera pensant en una previsió de futur en la que tornem a utilitzar les estructures, per tal de fer-ho genèric hem creat la classe abstracta Element de la qual hi haurem d'heretar o implementar per tal de poder utilitzar les estructures més endavant.

1. Arbre de cerca

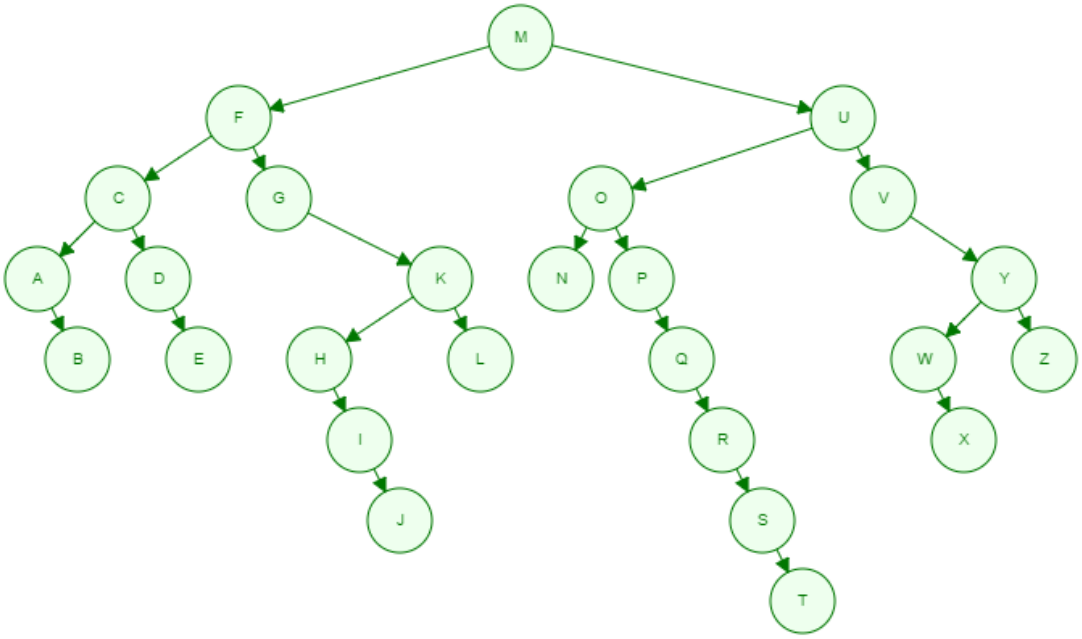
Un arbre de cerca parteix de la idea d'un arbre binari. A partir d'aquí, el que el diferència és el fet que està ordenat: el fill esquerre sempre serà més petit i el fill dret sempre serà més gran o igual (segons implementació). Aquest fet proporciona un seguit d'avantatges respecte un arbre binari i és que les busques són molt més ràpides donat que comparem l'element amb l'element del node actual i si no es tracta del mateix, ja sabem per quin dels dos fills hem de seguir.

En el nostre cas, si ens trobem que volem inserir un element que ja existeix, marquem aquesta addició al node apujant un comptador una unitat. Aquesta estructura ens serveix molt per fer les insercions i consultes per ordre alfabètic, doncs

aquest és el paràmetre que utilitzem per fer les comparacions; un cop ordenat d'aquesta manera, per tal de mostrar les paraules per nombre d'aparicions el que fem és traspassar tota aquesta informació a un arbre de cerca que utilitza l'atribut de l'element inserit que indica quantes repeticions hi ha hagut, per tal de fer les insercions a l'arbre. Però compte, això hi ha dues maneres de fer-ho: o mantenim l'estàndard i seguim posant els elements més repetits a la dreta cosa que implica haver de recórrer l'arbre amb un recorregut que anomenarem InOrdre invers, o bé trenquem la "normativa" i fiquem els elements més repetits a l'esquerra i llegir l'arbre amb un InOrdre normal. Aquest estàndard del que parlem és el fet de que a l'esquerra vagin els més petits i a la dreta els més grans.

Per tal de fer una petita demostració gràfica de com organitza aquesta estructura els elements, utilitzarem una seqüència que utilitzarem a altres punts donats de la memòria (a part de com a test per comprovar les correctes codificacions de les estructures, però això ja és part de la part pràctica del projecte):

M-F-C-U-V-O-Y-A-G-P-K-W-B-D-E-H-I-J-L-N-Z-Q-R-S-T-X

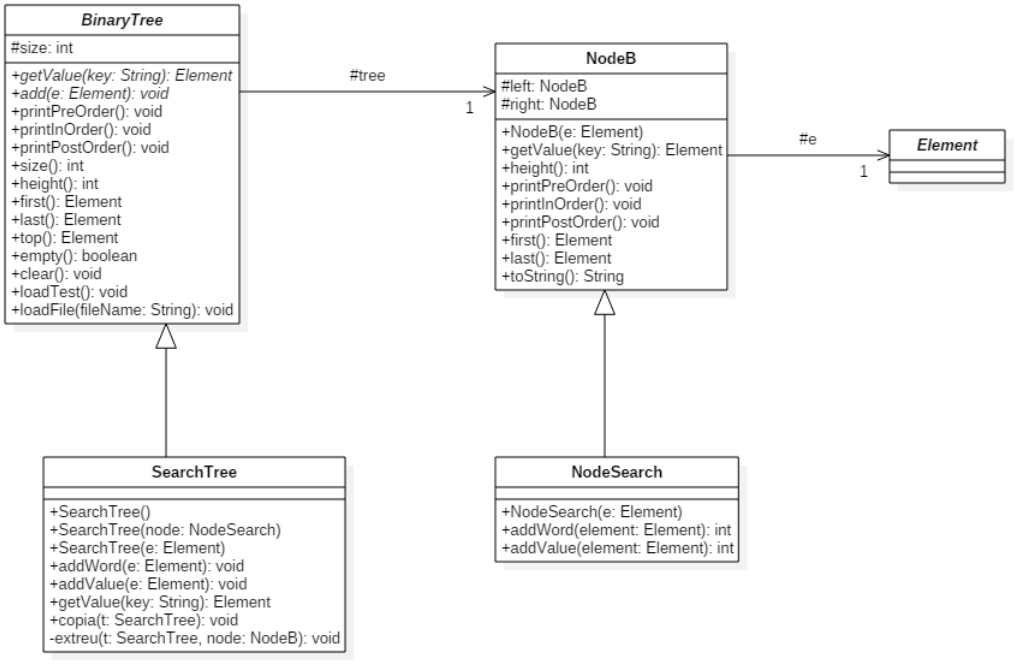


I aquesta és la representació gràfica de l'estat actual de l'arbre de cerca al inserir aquests elements. Com veiem, si fem un InOrdre, l'arbre manté l'ordenació alfabètica que hem comentat, així que a l'hora de buscar i inserir, sempre sabrà cap a quina direcció avançar.

Tot i això, si hem d'introduir grans quantitats d'elements, tenim un inconvenient i és que l'arbre no està balancejat i això implica que si nosaltres estiguéssim buscant l'element 'TA', aquest es trobaria a la dreta del node que conté 'T', així que l'arbre faria vuit comparacions abans de descobrir que aquest element no existeix. És en aquest punt en el que sorgeix una millora considerable: el balanceig. Aquesta millora s'explica en detall a la següent estructura de dades.

Diagrama de classes

A continuació el diagrama de classes de la nostra implementació. Si ens hi fixem bé, en aquesta implementació, per tal de poder-ho fer general i poder aprofitar classes en un futur, BinaryTree i Element són classes abstractes per tal de forçar a la utilització de certes característiques a qui vulgui fer un altre arbre binari o un element amb el que treballar. Si parem a pensar en aquest detall veurem que ens dona un avantatge i és que, per exemple, totes les classes del diagrama utilitzen la classe Element, tot i aquesta ser abstracte, és a dir, que no es pot instanciar; això vol dir que per tal de poder utilitzar els arbres haurem de crear una classe que hereti d'Element i, òbviament, implementar tots els mètodes que es requereixen. A més a més, podem treballar amb NodeSearch des de SearchTree donat que BinaryTree té un atribut que és NodeB, que és la classe de la que hereta NodeSearch.



Cal esmentar també la diferència entre addWord i addValue. La diferència simplement és el criteri d'inserció, és a dir: addWord l'usem per inserir utilitzant una comparació entre dos Strings (paraules) mentre que addValue ho fa comparant quantes vegades l'Element ha estat inserit. Òbviament utilitzem el primer mètode per inserir les paraules del text a l'arbre i el segon per abocar-ho tot del primer arbre a un segon arbre auxiliar per mostrar les paraules per nombre d'aparicions.

Costos

Pel que fa als cost de les operacions de l'arbre de cerca, només en podem obtenir el cost màxim, que seria:

Recompte

Com que per inserir, no sabem quants elements passarem abans de poder fer-ho, ens fiquem en el pitjor cas, que és haver de passar per tots els elements inserits.

O(h) , on h és l'alçada de l'arbre de cerca

Mostrar els nodes ordenats alfabèticament

Per mostrar tots els elements, hem de passar per tots ells un cop ja hem extret les dades del text. El cost de treure les dades està just a sobre, aquest és el cost d'examinar tot l'arbre.

O(n) , on n és el nombre de nodes a l'arbre de cerca

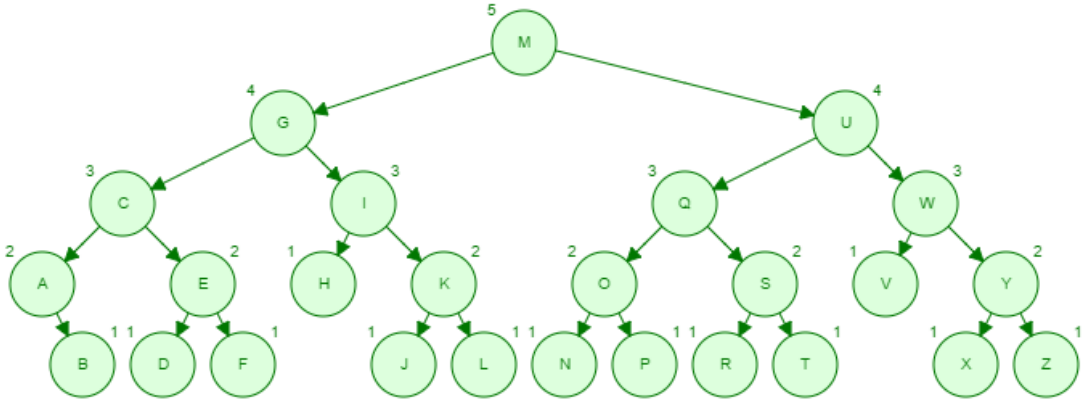
Mostrar els nodes ordenats per nombre d'aparicions

Degut a com hem decidit fer els càlculs, el cost de mostrar els elements ordenats per nombre d'aparicions és més elevat que simplement mostrar-ho alfabèticament ja que primer ho hem de passar a un arbre auxiliar amb un altre criteri de direcció. El cost tant per examinar és l'alçada de l'arbre i el cost d'inserir és n, per tant, fer les dues coses al mateix temps és el seu producte.

O(n * h) on n és el nombre d'elements i h l'alçada del segon arbre

2. Arbre AVL

Aquest arbre sorgeix de la necessitat de balancejar els arbres de cerca per tal d'evitar elevats recorreguts per simples consultes. El que fem és reorganitzar els nodes quan detectem un desnivell més gran d'1 i el que aconseguim d'aquesta manera és que mai s'iniciï un nou nivell si el que està dos per sobre encara no s'ha omplert del tot, d'aquesta manera ens assegurem d'anar omplint els nous nivells i no començar-ne de nous si encara podem omplir d'anteriors. La conseqüència d'això és que no hem d'anar tant avall a l'arbre per tal de fer consultes i per tant les farem més ràpid, però com que hem d'anar fent les diferents rotacions, a mesura que inserim, les insercions seran (normalment) més costoses.



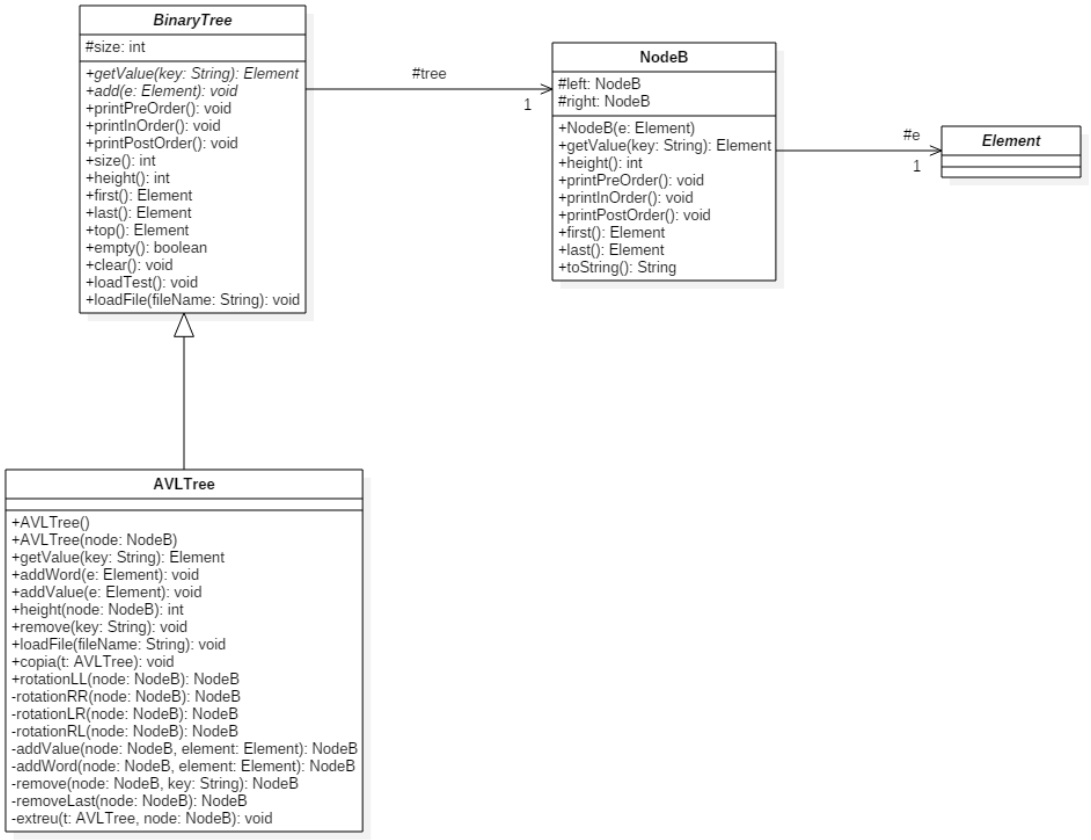
Aquesta és la representació gràfica de l'arbre AVL si inserim la mateixa seqüència de caràcters que a l'arbre de cerca que hem mostrat abans. Com es veu fàcilment, aquest arbre està molt més anivellat que l'anterior.

Seguint amb l'exemple anterior, veiem clarament la millora que això representa si ara volguéssim veure si existeix el node 'TA' donat que ara només faríem cinc comparacions davant de les vuit de l'arbre de cerca. La millora es veu encara més clara si escalem aquest simple exemple amb les lletres de l'abecedari a un altre on inserim milers de milions d'elements: la millora a les consultes és enorme.

Parlant ja del nostre programa, el nostre AVLTree ordena alfabèticament per defecte, que és l'única referència que tenim per ordenar si partim de zero. Per tant, a l'hora d'ordenar-ho per nombre d'aparicions, hem d'ordenar-ho inicialment de forma alfabètica i després passar tots els elements a un AVLTree que ordena per nombre d'aparicions.

Diagrama de classes

A continuació el diagrama de classes de la nostra implementació. Com hem explicat a l'apartat de l'arbre de cerca, al crear unes classes genèriques, només hem creat una nova classe que hereta de BinaryTree i reutilitzar totes les altres classes.



Aquesta estructura ha estat la més complicada de crear degut a les rotacions que implica i al càlcul de l'alçada.

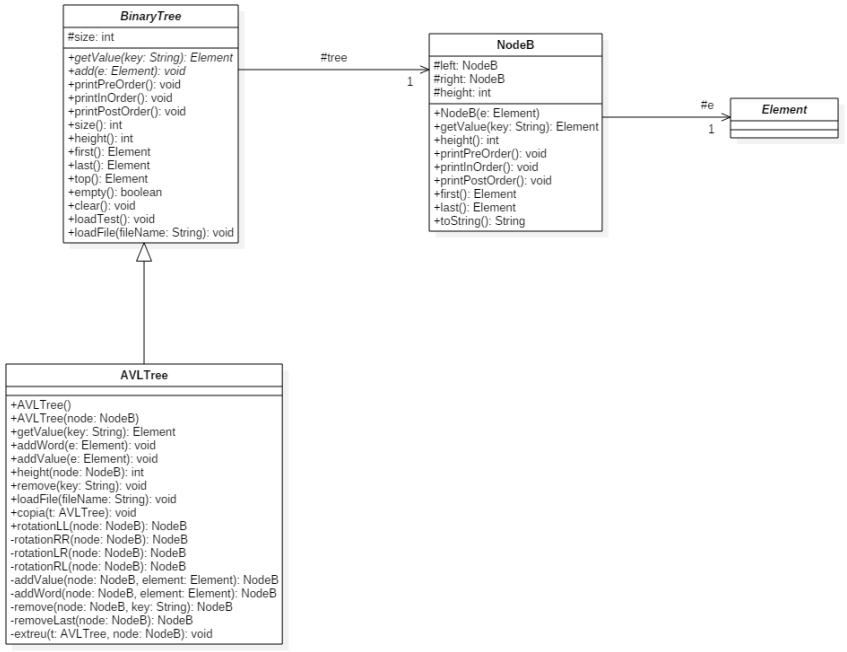
El problema amb les rotacions va ser un problema més que res de la implementació, de com fer-ho; però a base d'utilitzar una pàgina web per veure les simulacions, una pissarra i diversos intents, ho vam aconseguir.

El problema gran va ser el càlcul de l'alçada, que el fèiem de forma recursiva a cada inserció per les dues branques que té un node; és a dir, que per cada node que passàvem quan volíem inserir un element, teníem una crida recursiva de càlcul d'alçada per cadascun dels nodes que tenia, que a la seva vegada cadascun dels nodes tenia dues crides més. Això suposava un cost enorme per cada inserció, així que vam

millorar el codi al màxim per a que el impacte d'aquestes dues crides fos menor: originalment, amb un fitxer de prova, bolcar-lo trigava 4 minuts; al fixar-nos en la situació de les crides del codi, vam aconseguir reduir el temps a la meitat, 2 minuts i posteriorment a 100 segons.

Posteriorment vam veure que si gastàvem més memòria i ens guardàvem l'alçada actual del node ens estalviàvem la crida recursiva que feia que trigués tant. En principi fer això era relativament fàcil: quan creem un node la seva alçada és 1, i a mesura que anem desfent la cadena de crides per inserir actualitzem l'alçada del node pel que passem. És a dir, que un cop inserit anem pujant pel camí que hem seguit per baixar actualitzant l'alçada dels nodes pels que passem, que com que és només agafar el màxim de les alçades a esquerra i dreta més una unitat, no és complicat. El problema venia quan havíem de fer una rotació, que crèiem que canviava l'alçada de tots els nodes afectats per la branca rotada; però un cop fetes unes simulacions sobre paper, ens en vam adonar que només podien variar els nodes central, esquerre i dret del resultat i per tant, els únics dels que calia actualitzar l'alçada. Un cop feta aquesta millora i ja posada en marxar, vam veure una millora exageradament gran en els resultats: un fitxer de 3.679kB que originalment trigava gairebé vuit minuts, ara el feia en tres segons.

Nou diagrama de classes



Com veiem, la única cosa que canvia que es pot apreciar a un diagrama de classes és que NodeB té un atribut nou que es diu height.

Costos

Pel que fa als cost de les operacions de l'arbre AVL, només en podem obtenir el cost màxim, que seria:

Recompte

A diferència de l'arbre de cerca, la cota superior del cost no pot ser n donat a que mai permet que una branca tingui un desnivell a l'alçada molt gran. I gràcies a això podem extreure la següent fórmula:

$n = 2^h - 1$ on n és el nombre de nodes i h l'alçada de l'arbre

D'aquí podem extreure directament el nombre d'elements que podem tenir segons l'alçada de l'arbre. Però el que a nosaltres ens interessa és l'alçada que hem de recórrer per tal de donar el cost, així que jugant amb la fórmula obtenim:

$h = \log_2(n+1)$ on n és el nombre de nodes i h l'alçada de l'arbre

Així que el cost d'inserir un text a l'arbre és:

$O(\log_2(n+1))$

Mostrar els nodes ordenats alfabèticament

Per tal de mostrar els nodes ordenats alfabèticament és el cost de visitar tots els nodes, així que:

$O(n)$

Mostrar els nodes ordenats per nombre d'aparicions

Degut a com hem decidit fer els càlculs, el cost de mostrar els elements ordenats per nombre d'aparicions és més elevat que simplement mostrar-ho alfabèticament ja que primer ho hem de passar a un arbre AVL auxiliar amb un altre criteri de direcció. El cost per examinar un arbre AVL és n i el cost per inserir és $\log_2(n+1)$, per tant, fer les dues coses al mateix temps és el seu producte.

$O(n * \log_2(n + 1))$

3. Taula encadenada indirecta amb arbres AVL

Per la tercera estructura volíem utilitzar una taula de hash, així que ens vam decantar per la taula que ens oferia una senzillesa a l'hora de tractar els sinònims: una taula encadenada indirecta. La diferència entre la implementació vista a classe i la nostra és que la de classe utilitzava una estructura lineal per emmagatzemar els sinònims, mentre que nosaltres al veure que hauríem d'emmagatzemar moltes dades i que no volíem fer una estructura lineal per veure els sinònims, hem fet que cada casella de la taula sigui un arbre AVL.

Al fer-ho així aconseguim una encara millor distribució dels elements, és a dir, d'un arbre de cerca passem a un arbre AVL per tal d'optimitzar les cerques donat que no haurà de baixar tant a l'arbre per trobar els elements, però ara va encara més directe donat que ja no té que "viatjar" per totes les lletres de l'abecedari fins a arribar a la que li interessa (que segons els elements inserits fins el moment a l'arbre, varia el temps que perd) donat que ja hi va directe.

A l'hora de fer l'ordenació alfabètica no tenim cap problema donat que aquesta taula està clarament enfocada a aquest objectiu, el quid de la qüestió arriba quan s'ha d'ordenar per nombre d'aparicions. Com ja ve essent típic, el que fem és copiar-ho a un arbre AVL i llegir-ho des d'allà. Una alternativa a aquest mètode podria haver sigut

buscar a tots els arbres el node que tingui més nombre d'aparicions, però hagués tardat més a fer-ho i hauríem d'afegir un camp més al tipus dient si ja l'hem tret o no.

La representació gràfica de l'estructura dissenyada és:

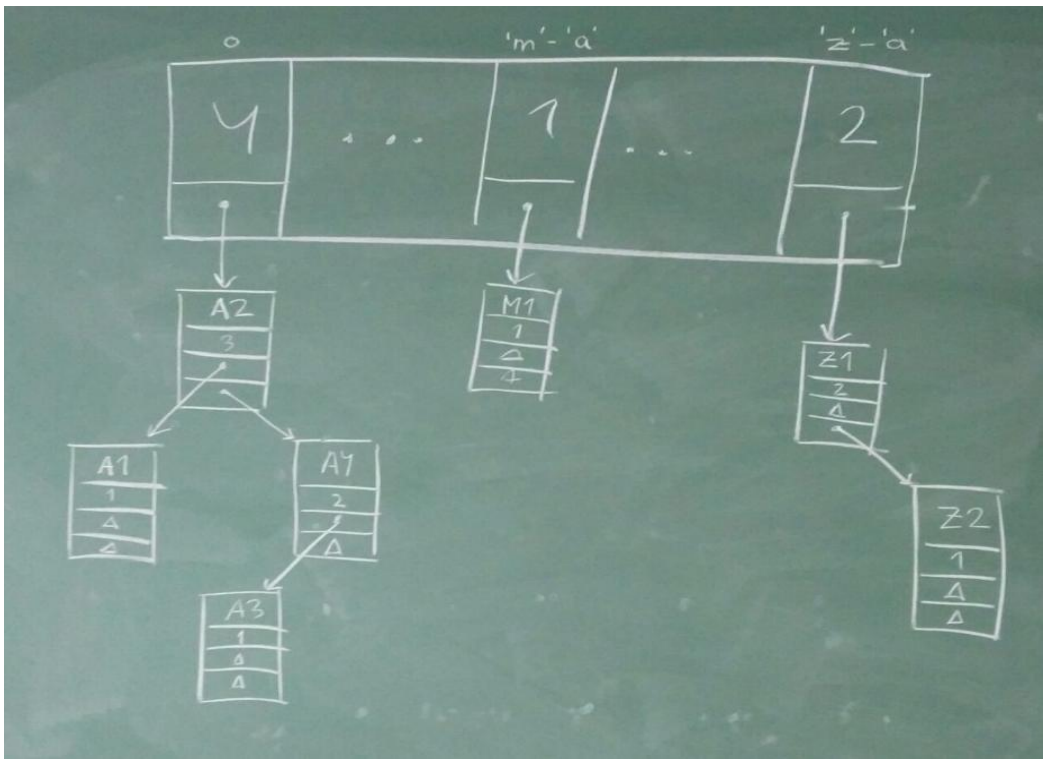
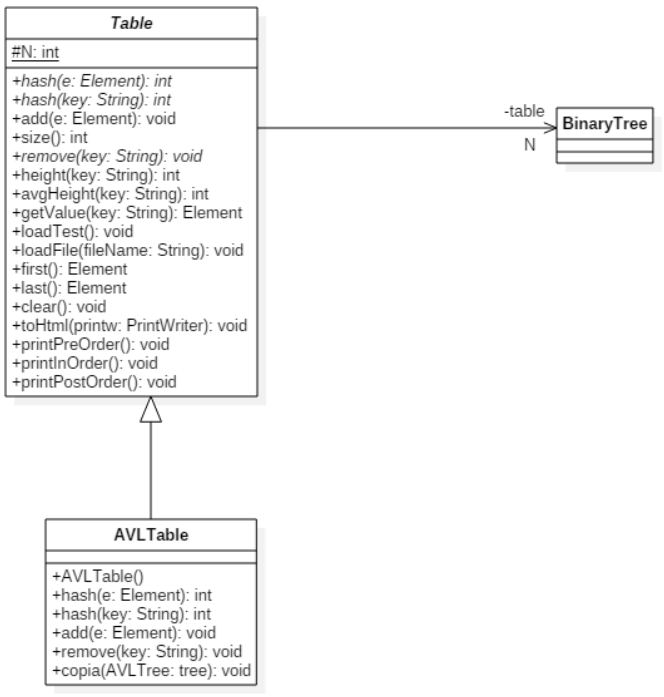


Diagrama de classes



Costos

Pel que fa als cost de les operacions de la taula AVL, només en podem obtenir el cost màxim, que seria:

Recompte

Paral·lelament a l'arbre AVL, el cost és normalment menor. Si que és cert que el que fem és inserir a un arbre AVL, però aquest arbre usualment serà de menor mida que el de l'estructura anterior, així que el cost d'inserció és:

$\log_2(q_i + 1)$ on $q \in [0, n]$,

$i \in [0, p]$

n és el nombre total de nodes a la taula

p és el nombre de caselles/arbres de la taula

El que volem dir és que el cost d'inserir és $\log_2(q_i + 1)$ on q_i és el nombre d'elements de l'arbre a la posició i de la taula. Cal a dir que q_i molt probablement sigui molt menor al nombre total de nodes de la taula, cosa que és l'avantatge d'aquesta estructura. Tot i això, no podem suposar una distribució equitativa, així que posant-nos en el pitjor cas de que totes les paraules del text comencin per la mateixa lletra, el cost serà:

$O(\log_2(n+1))$

Mostrar els nodes ordenats alfabèticament

Com fins ara, el cost de mostrar els nodes ordenats alfabèticament és el nombre de nodes que hi ha, per tant:

$O(n)$

Mostrar els nodes ordenats per nombre d'apariciones

El cost de mostrar els elements ordenats per nombre d'aparicions de la taula amb arbres AVL és en realitat el mateix que el d'un arbre AVL. Tot i que en aquesta estructura fem més coses: avançar per la taula, consultar tots els nodes i inserir-los a un arbre AVL. Així que el cost serà: el producte del nombre de caselles de la taula amb el nombre d'elements que hi hagi a l'arbre més ple i amb el cost d'inserir a l'arbre

auxiliar. Però com sempre, posant-nos en el pitjor dels casos, que és aquell en el que tots els elements del text comencen per la mateixa lletra, el cost és:

$O(p * n * \log_2(n+1))$

4. Llista ordenada

Si hem parat atenció, podrem veure que totes les estructures usades fins el moment són no lineals així que per tal de veure un contrast clar, entre aquelles i una estructura lineal, hem triat una llista ordenada amb PDI.

Aquesta estructura consta de dues referències a un node (no té perquè ser el mateix) i la mida de la llista. Cada node consta de l'element que guarda i d'una referència al següent node de la llista.

A diferència de les altres, una llista en aquest cas ens proporciona el desavantatge que té un accés seqüencial així que si tenim 10.000 elements i volem inserir un que vagi a la posició 10.001, la llista comparará un a un tots els elements fins a arribar al final. Cosa que es veu clara al punt sobre els costos de l'estructura.

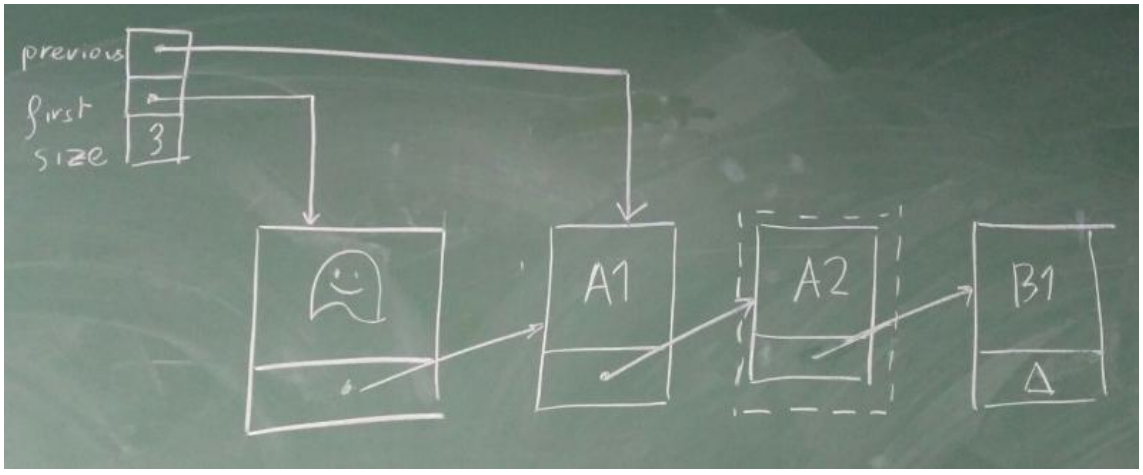
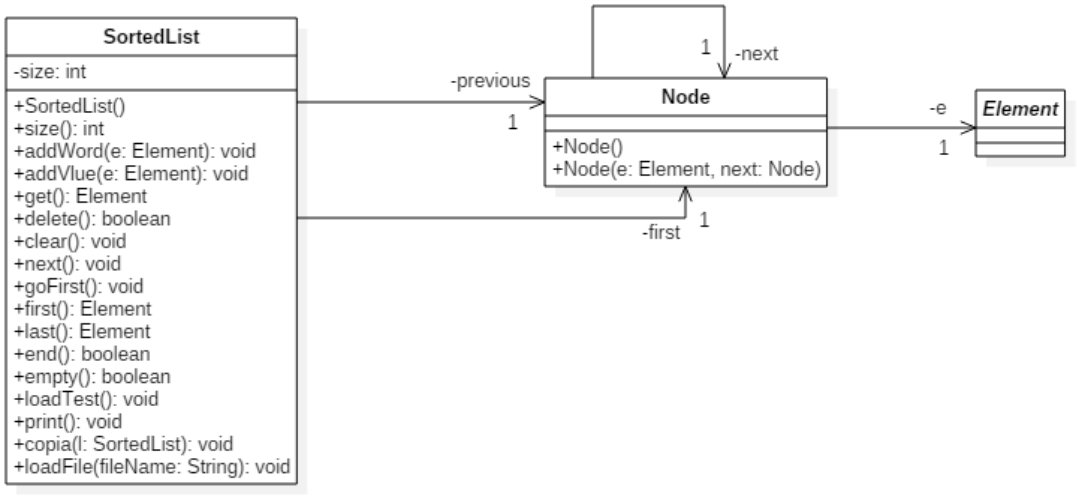


Diagrama de classes



Costos

Pel que fa als cost de les operacions de la llista ordenada, només en podem obtenir el cost màxim, que seria:

Recompte

Passem per tots els elements abans de poder inserir:

O(n)

Mostrar els nodes ordenats alfabèticament

Passem per tots els elements que hem de mostrar:

O(n)

Mostrar els nodes ordenats per nombre d'aparicions

En aquest cas hem de passar per tots els elements mentre els anem inserint a una llista auxiliar d'ordenació per nombre d'aparicions així que és el producte dels dos costos anteriors:

$O(n^2)$

2. Resultats

A continuació, els resultats obtinguts a través de l'anàlisi de les dades vistes a l'apartat anterior. Però abans de començar, cal assegurar que tenim uns conceptes clars per tenir-los en compte a l'hora de veure les dades que es mostren.

Primer: cal contemplar un cert temps d'error de l'ordinador a l'hora de calcular els temps d'execució.

Segon: No hi ha diferència de memòria utilitzada entre el primer i segon fitxer de les execucions per una simple raó: la memòria utilitzada la calculem utilitzant una fórmula matemàtica i el nombre de nodes de l'estructura, així doncs, com que el segon fitxer és el primer copiat tres vegades, té el mateix nombre d'elements (haurà de fer el triple de insercions igualment, però sobre elements repetits).

Dit això, ja podem començar a veure les dades obtingudes:

1. Arbre de cerca

Temps CPU	**1.227kB**	**3.679kB**	**11.271kB**
Ordre alfabètic	1.032ms	3.040ms	7.993ms
Ordre de repeticions	2.184ms	4.002ms	8.824ms

Memòria consumida	**1.227kB**	**3.679kB**	**11.271kB**
Ordre alfabètic	411.268B	411.268B	419.017B
Ordre de repeticions	822.536B	822.536B	838.034B

Pel que fa a l'arbre de cerca, veiem com la diferència de mode de recompte és clara, tarda un segon més en ordenar per repeticions donat que el que fa és agafar tots els elements que hi ha a l'arbre ordenat alfabèticament i els fica a un nou arbre de cerca auxiliar. A primera vista podria semblar que això és incorrecte donat que la primera vegada (alfabèticament) té molts més elements a inserir que quan ho fa a l'arbre auxiliar, però això té una explicació molt senzilla: l'arbre no està balancejat, el primer element que inserim de l'arbre original és l'element 'a', que es repeteix molt; aquestes dues coses són les principals causes donat que això fa que l'arbre auxiliar

fiqui una immensa majoria d'elements a la dreta, causant així un desequilibri molt fort. Com a aclariment, trigar un segon més, a la velocitat que fa l'ordinador els càlculs, és molt temps.

Referint-nos a la memòria que ocupa, el motiu que consumeixi el doble d'espai si ordenem per nombre de repeticions és degut a que aquesta ordenació requereix un arbre auxiliar amb el mateix nombre d'elements però diferent ordenació. És a dir, que per ordenar alfabèticament necessitem un sol arbre però per l'altre ordenació en necessitem dos igual de grans.

2. Arbre AVL

Temps CPU	**1.227kB**	**3.679kB**	**11.271kB**
Ordre alfabètic	1.030ms	3.047ms	8.904ms
Ordre de repeticions	1.046ms	3.044ms	8.898ms

Memòria consumida	**1.227kB**	**3.679kB**	**11.271kB**
Ordre alfabètic	411.268B	411.268B	419.017B
Ordre de repeticions	822.536B	822.536B	838.034B

Pel que fa a l'arbre AVL, cal fixar-nos que a diferència de l'arbre de cerca, a l'hora de fer l'ordenació per nombre de repeticions veiem que no és com l'arbre de cerca que tardava notòriament més sinó que tarda pràcticament igual. Això és degut a que no està desequilibrat com l'anterior així que tarda molt menys a fer les consultes.

Respecte a la memòria ocupada, veiem que segueix exactament el mateix creixement que l'arbre de cerca. Això es deu a que les dues estructures, tot i que el seu mode d'inserció sigui diferent, els seus elements ocupen el mateix espai en memòria.

Abans de continuar amb la següent estructura creiem que seria interessant veure la diferència de temps d'execució entre les dues implementacions de l'arbre AVL que hem acabat fent. Recordem que la primera implementació utilitzava un mètode recursiu per tal d'explorar tots els nodes dels que n'és pare i així calcular l'alçada d'un

node i la segona implementació ens guardem a cada node la seva alçada així que calcular l'alçada té un cost de O(1).

Arbre AVL	551kB	1.227kB	3.679kB
Primera solució	26.234ms	120.109ms	430.609ms
Segona solució	1.030ms	3.044ms	8.898ms

Com veiem hi ha una diferència exagerada de temps entre les dues implementacions i només hem afegit un atribut més al tipus node, però de tal manera que ens evitem passar per tots els nodes inferiors dels nodes per cada node, que és el que realment feia la primera implementació.

3. Taula amb arbres AVL

Temps CPU	**1.227kB**	**3.679kB**	**11.271kB**
Ordre alfabètic	995ms	2.930ms	8.582ms
Ordre de repeticions	999ms	2.927ms	8.591ms

Memòria consumida	**1.227kB**	**3.679kB**	**11.271kB**
Ordre alfabètic	411.372B	411.372B	419.121B
Ordre de repeticions	822.640B	822.640B	838.138B

Pel que fa a les taules de hash amb arbres AVL per guardar els sinònims, veiem que tarda més o menys el mateix que utilitzant un sol arbre AVL en les diferents ordenacions. Cosa que realment ens sorprèn, és a dir, els arbres que hi ha a la taula són clarament de menys alçada, per força, però tot i això no es diferència tant de tenir-ho tot al mateix arbre: un petit avantatge parlant del temps però també necessita una mica més d'espai a la memòria per funcionar.

Tot i així, cal dir que no sempre ha estat així. En la primera implementació de l'arbre AVL de la que ja hem parlat, hi havia una diferència exagerada de temps a l'hora de fer el parse del fitxer. A continuació la comparació dels temps d'ordenació

alfabètica (per ordenar la taula per nombre d'apariciones utilitzem un AVL, així que no es un punt de referència vàlid):

	551kB	1.227kB	3.679kB
Arbre AVL	26.234ms	120.687ms	430.609ms
Taula amb AVL	1.609ms	5.625ms	21.484ms

Com veiem la diferència és exageradament gran. Com que el punt dèbil de la nostra primera implementació de l'AVL era el càlcul de les alçades, degut a que aquest era recursiu, i el punt fort de la taula és que ens redueix molt el nombre d'elements per arbre (tot i que no té una distribució uniforme d'elements per arbre). Degut a aquest bon repartiment dels elements del text, arribàvem a tenir aquesta gran diferència de temps (20 vegades superior en el menor dels casos).

Respecte a la memòria ocupada en aquesta estructura en necessitem més ja que es tracta d'un vector d'arbres AVL, estructura en la que el node superior ocupa una mica més que els altres, així que necessitem més memòria per compensar aquest detall.

4. Llista ordenada

Temps CPU	**1.227kB**	**3.679kB**	**11.271kB**
Ordre alfabètic	106.413ms	221.530ms	1.031.755ms
Ordre de repeticions	110.743ms	291.777ms	1.027.824ms

Memòria consumida	**1.227kB**	**3.679kB**	**11.271kB**
Ordre alfabètic	254.604B	254.604B	259.401B
Ordre de repeticions	509.208B	509.208B	518.802B

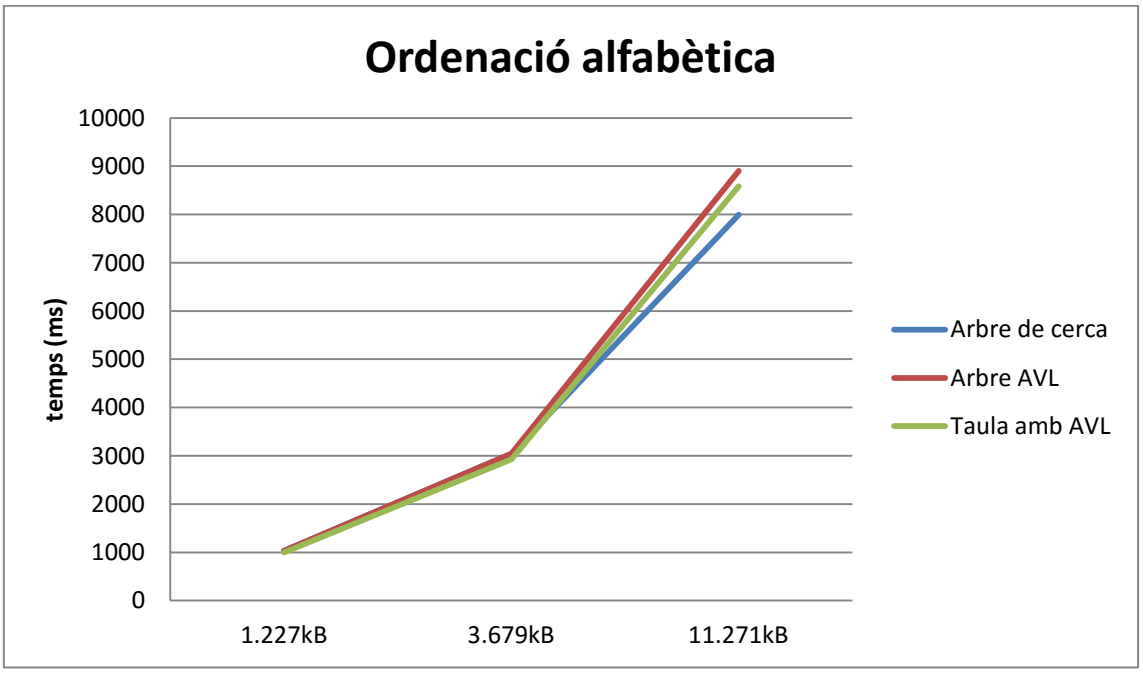
Pel que fa a la llista ordenada, veiem que tarda exageradament més que les altres estructures. Això és degut a que la llista té un accés seqüencial, per tant ha de consultar tots els nodes que troba fins a trobar el corresponent. A això se li suma el fet de que dintre dels elements amb mateix nombre d'apariciones els ordenem alfabèticament, així que si no són dels primers, encara hauran de recórrer elements

amb el mateix nombre d'insercions que ell fins a trobar el seu lloc. Cal dir que aquest és un truc que no es pot utilitzar amb els arbres donat que un igual sempre anirà a la dreta (i l'AVL encara es balancejarà després).

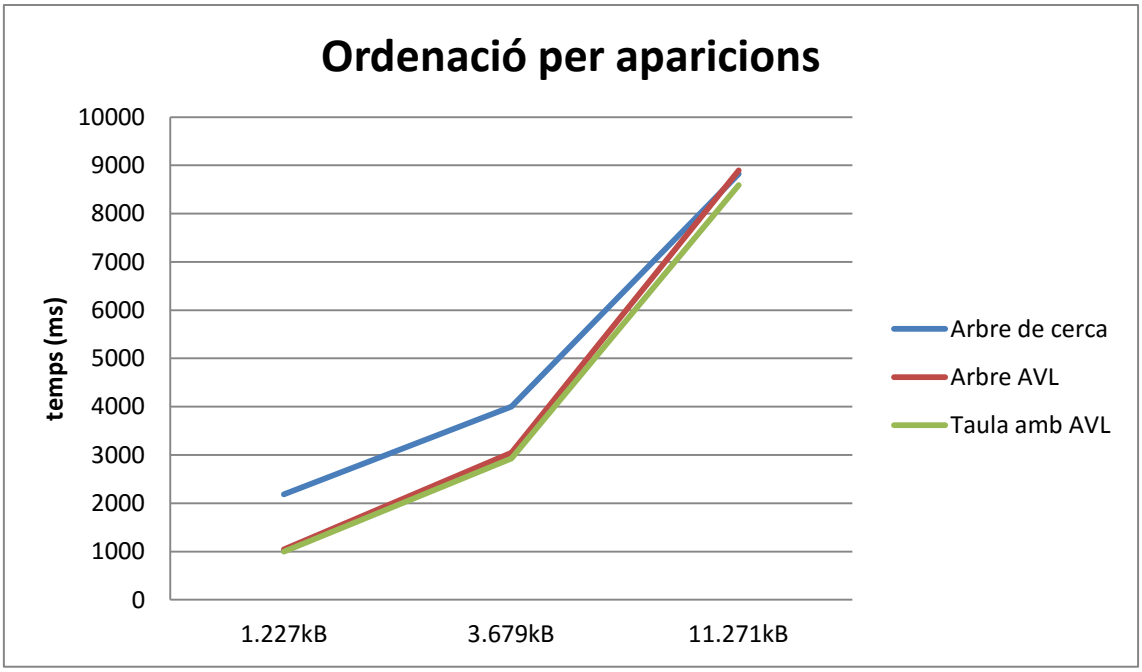
Tot i aquest gran increment en el temps de carregar tot el fitxer, cal veure que hi ha un remarcable decrement en la memòria consumida en extreure el fitxer. Aquest estalvi en memòria es deu a que cada element només guarda una única referència a un altre node enlloc de dues i, a més a més, els nodes de la llista no es guarden la seva posició dins l'estructura (referent a l'alçada que sí es guarden els nodes de l'arbre).

5. Globals

Per tal de veure els resultats globals, primer veurem el temps d'execució amb les tres primeres estructures, continuarem comparant-les a la llista ordenada i comentarem la relació temps memòria.



Com veiem per l'ordenació alfabètica qui, a més elements a inserir, fa millor temps és l'arbre de cerca, encara que si n'hi ha menys és la taula amb AVL. Això es deu a que a més nodes a inserir, més rotacions s'han de fer i això acaba castigant el temps que triga.

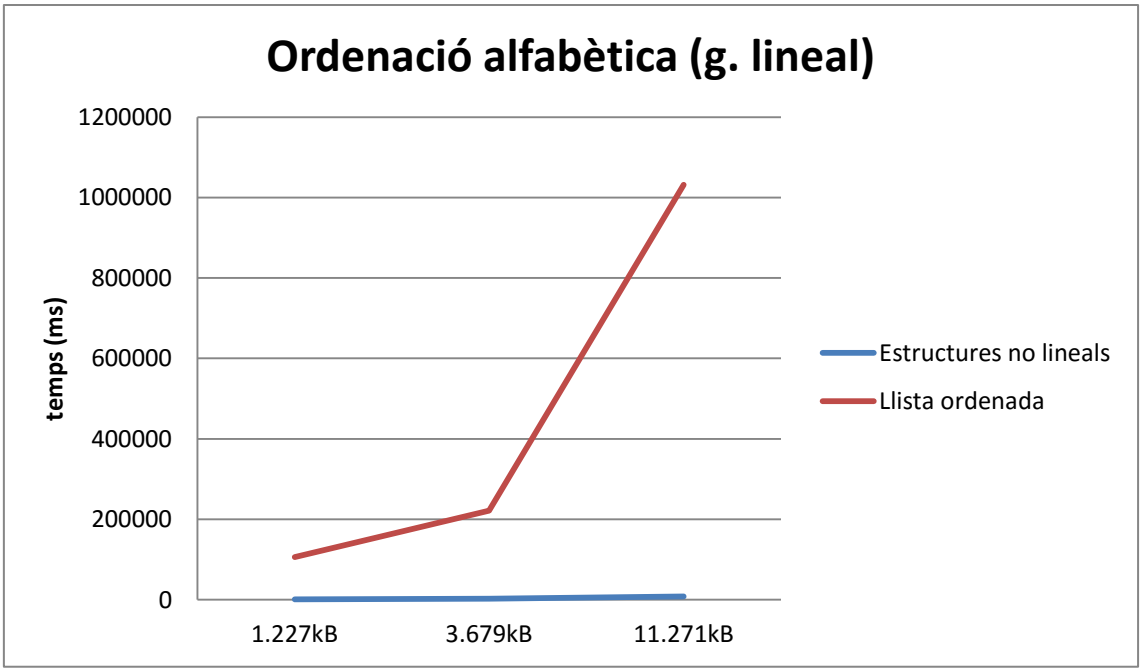


A l'ordenació per nombre d'aparicions, veiem que les dues estructures que fan un millor temps són les que utilitzen AVL. Però també veiem que a més gran el fitxer de text, s'acaben posicionant amb el mateix temps que el de cerca. Aquest augment progressiu es degut a la mateixa raó que a l'ordenació alfabètica: les insercions són més costoses.

Aleshores, tot i aquest cost més gran a l'hora d'inserir, perquè es triga menys amb els AVL que amb el de cerca, si hi ha el doble d'insercions que a la primera ordenació? Òbviament la resposta la dóna la diferència entre els dos tipus d'arbre: el balanceig. Per tant: sí que és cert que hi ha insercions més costoses, però a l'hora d'agafar els nodes per transferir-los al nou arbre ordenat triga molt menys que l'arbre de cerca, que té una selecció que pot arribar a ser exageradament costosa (dependrà de la primera aparició dels elements al text).

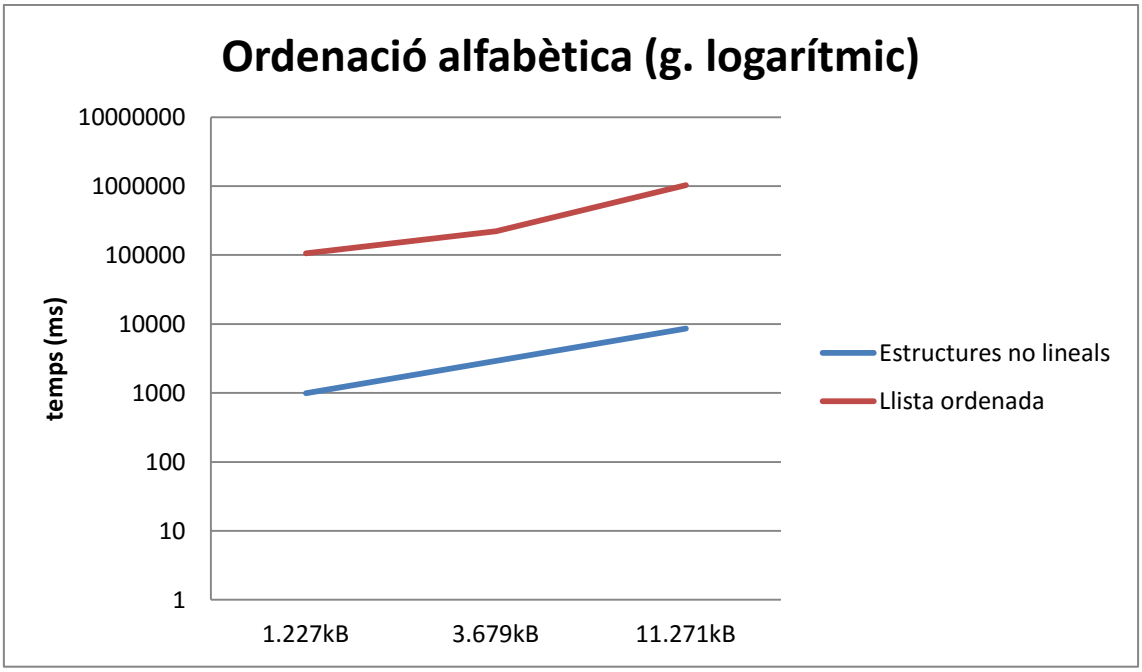
Tot i això, a més nombre de nodes acaben trigant el mateix ja que els AVL trigaran més a fer les insercions.

En els següents gràfics, hem resumit les tres línies formades per "Arbre de cerca", "Arbre AVL" i "Taula amb AVL" per "Estructures no lineals" degut a que es solapaven i quedaven unides com a una, així que no tenia sentit seguir-les tractant de manera diferent.

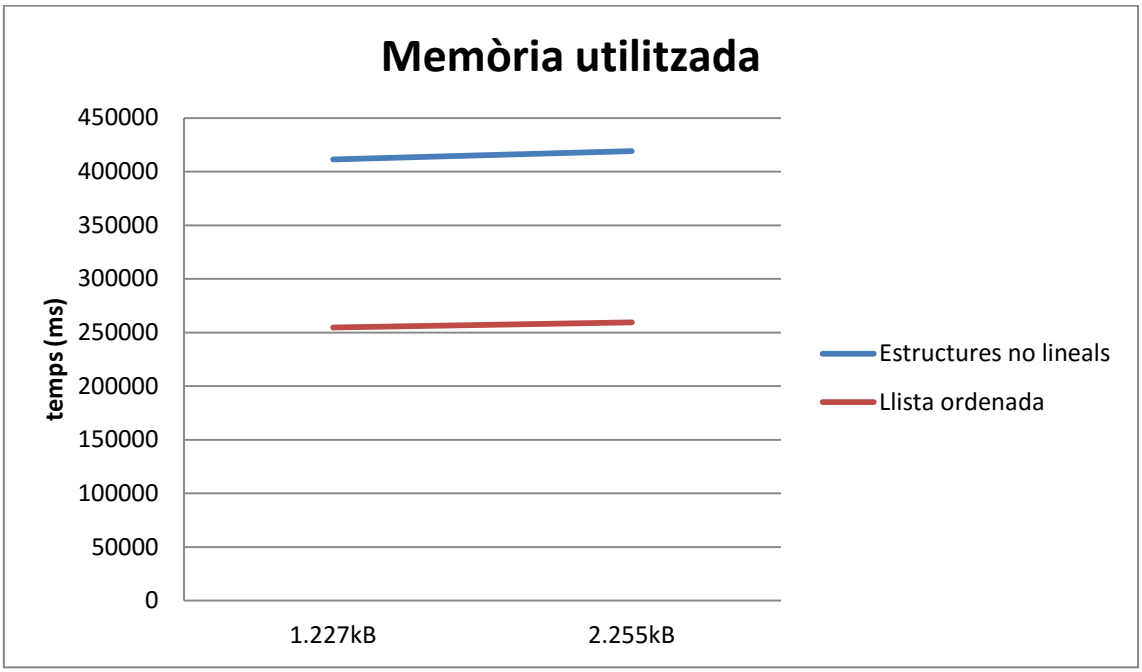


Més a mode d'anècdota que d'anàlisi, per tal de fer una comparació visual, aquest és el primer gràfic de l'apartat de resultats globals si hi afegim els resultats de temps de la llista ordenada.

Òbviament utilitzar un gràfic lineal no és la millor manera de veure això, així que utilitzem un gràfic logarítmic.



Tal com podem veure, hi ha tanta diferència de temps entre les estructures no lineals i la llista que necessitem utilitzar una escala diferent.



La mida dels fitxers per aquesta comparació no és la vista en comparacions anteriors. Això és degut a que el fitxer de 11.271kB és cinc vegades seguides la biografia de Benjamin Franklin. Per tant, el fitxer original (2.255kB) i l'usat en els experiments anteriors ocupen exactament la mateixa memòria a l'estructura, perquè hi ha el mateix nombre de nodes.

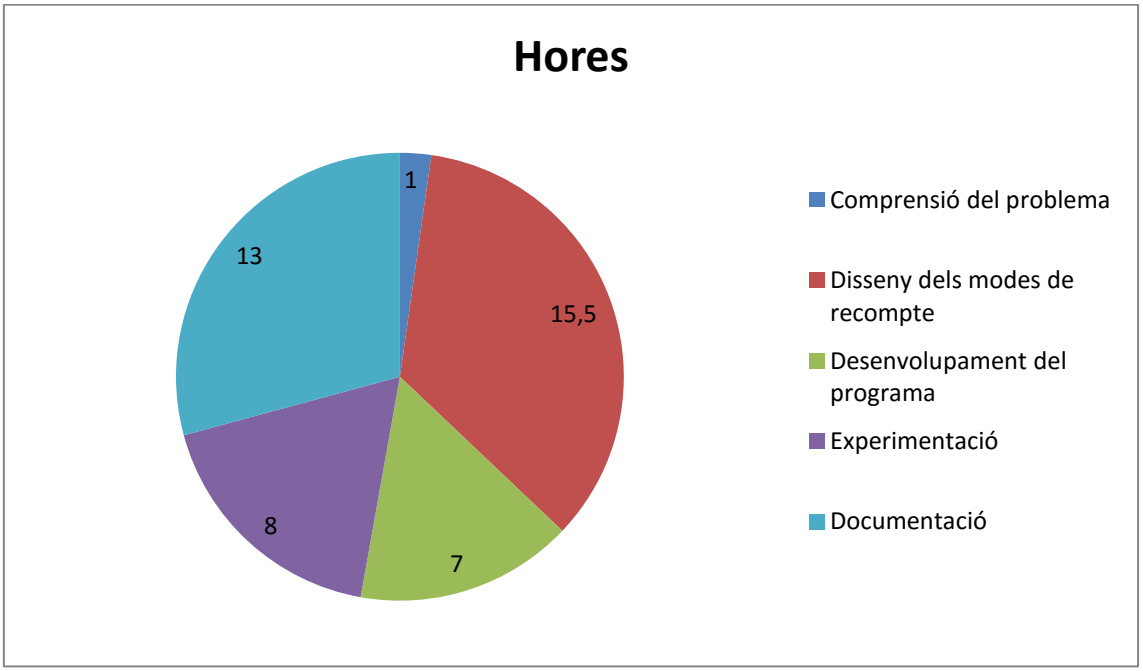
Un cop vist que referent al temps d'ordenació, l'arbre és molt superior, ara cal fixar-nos en la memòria ocupada i el resultat és clar: la llista ordenada guanya. Com ja hem explicat a l'anàlisi concret de la llista, això és degut a que els elements d'aquesta només guarden una referència al següent enlloc de dues com passa als arbres binaris.

3. Dedicació

A continuació mostrem un càlcul més o menys encertat del temps invertit per resoldre la pràctica. Cal dir que és una estimació doncs al final de cada "sessió" de treball afegíem les hores que havíem treballat, però hi ha dues coses a tenir en compte:

El 100% de les hores invertides no són eficaces al màxim: Distraccions per respondre el mòbil o berenar podrien desviar l'atenció del treball.

El 100% de les vegades que vam treballar no les podem apuntar aquí: Diferents moments quotidians del dia on no produïm res (al tren, a la dutxa, al llit) però pensem en possibles solucions i optimitzacions per muntar les estructures, l'organització del programa, previsió dels resultats obtinguts i possibles comentaris i conclusions per la memòria.



4. Conclusions

Per finalitzar, després de provar les quatre estructures amb diversos fitxers i diversos modes de recompte, hem arribat a una conclusió.

Primer de tot, hem vist com sí que és cert que com més ràpid volem que vagi una estructura, més memòria ocuparà i a l'inrevés, com menys volem que ocupi, més lent anirà. Això resulta obvi després de la nostra experimentació doncs perquè vagi més ràpid necessitem guardar més accessos, que ocupen memòria. Aquest punt l'hem vist clarament a l'apartat de "Resultats globals" on comentàvem la diferència entre les estructures lineals i les que no ho són.

Segon, també hem pogut veure com realment hi ha clares diferències dintre del món dels arbres: com que els arbres de cerca tenen (generalment) insercions més ràpides que els arbres AVL però l'arbre pot estar molt desequilibrat, cosa que castigarà força a l'hora de fer cerques i consultes, cosa que no passa amb els arbres AVL.

Tercer, hem pogut aprofundir encara més en un mateix arbre i trobar diferents maneres d'implementar-lo, i veure les diferències entre aquestes. Això s'ha vist en les dues diferents implementacions de l'arbre AVL que hem utilitzat on, totes dues funcionaven però n'hi havia una que tardava més que l'altre degut a que calculava l'alçada recursivament; i l'altra anava molt més ràpida però tenia l'inconvenient que gastava més memòria i requeria una lògica i planificació més complexa.

Quart, hem pogut veure que podem combinar dues estructures de dades diferents com són arbres i taules per crear una estructura millor encara. En el nostre cas això s'ha vist en la implementació de la tercera estructura, on guardàvem la informació en arbres AVL però utilitzàvem una taula d'accés directe per repartir encara més la distribució dels nodes i fer que els diferents arbres siguin més baixos enlloc d'un de més gran.

Per acabar ens agradaria dir que tot i que aquesta pràctica ens hagi portat alguns maldecaps, ha estat tot un repte i inclús divertit treballar i experimentar amb aquestes estructures vistes a classe, ja que allà fem estrictament teoria i, encara que se'ns digui, no podem veure nosaltres de primera mà les diferencies en temps i memòria de l'ús de les diferents estructures.

5. Bibliografia

Sobre la bibliografia utilitzada, cal dir que ha estat només en situació de consulta per no saber com es fa algun detall de la pràctica. Començarem per un llibre i continuarem per les pàgines utilitzades per ordre de rellevància.

Primer de tot el llibre, ha estat utilitzat per buscar informació sobre la implementació dels arbres AVL, però tot i que no va haver-hi sort, el vam consultar i creiem que ha d'estar aquí.

FCO. JAVIER CEBALLOS. *Java 2: Curso de programación*. 2a edició. Madrid: RA-MA, Maig 2002. ISBN 84-7897-521-7

La pàgina més important per poder fer la implementació dels arbres AVL ha estat sens dubte aquesta d'aquí, que et mostra una animació de com funciona un arbre AVL. A partir de mirar-la i fer-hi proves vam veure bé com funcionava i ens va inspirar a crear-lo.

AVL Tree Visualization. University of San Francisco. 2011 [consulta: 11 de maig]. Disponible a: http://cs.usfca.edu/~galles/visualization/AVLtree.html

Els dominis web més importants per tal de desenvolupar i fer els altres aspectes de la pràctica han estat stackoverflow.com i la documentació d'oracle. Per desgràcia hi ha hagut diverses consultes variades i no en podem especificar cap de concreta, però les utilitzàvem per resoldre dubtes de programació en si.

Per acabar, la pàgina d'on han sortit els diferents textos per fer les proves ha estat:

Project Gutenberg. Michael Hart. - [consulta: 11 de maig]. Disponible a: http://www.gutenberg.org/