Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili

Tercera pràctica d'Estructures de dades

Índex alfabètic d'un text

Integrants del Grup:
Bernat Bosca Candel
Albert Cañellas Solé

Grau:
Doble grau Biotecnologia e Enginyeria Informàtica

Professor de Pràctiques:
Maria Ferré Bergadà

Índex

Disseny de la solució del problema proposat

La solució plantejada consta d'un programa que deixa triar al usuari el nom del fitxer a utilitzar, i en el cas de que no existeixi en demana un de nou fins tindre un que si.
I també cal triar quin tipus d'estructura de dades es farà servir si un arbre o una taula de hash. A continuació es tractaran les dades, guardant a l'estructura de dades escollida les paraules repetitives marcades amb el signe '$', i després a partir d'aquesta ED es crea un fitxer índex amb les paraules ordenades alfabèticament, i mostrant en quines planes i línies d'aquesta apareixen al fitxer escollit.

El programa dissenyat consta de 2 paquets, però que els diferenciarem com a 4:

- Aplicació: conte el main i les funcions principals.
    - Aplicació: classe que implementa el menú, tots els mètodes per recollir la informació que es vol del usuari i els mètodes per llegir i guardar en fitxers.
- TADs: realment conte la resta de classes del programa, però sols comentarem les que hi ha dintre de la carpeta TADs, i despès comentarem les carpetes que hi ha a dintre d'aquesta.
    - TAD(interficie): classe que implementa tots els mètodes que han de tenir ambdues estructures de dades.
    - Valors: classe que implementa tots els mètodes necessaris per a emmagatzemar i operar els valors de pàgines i línies on apareix una paraula.
- Arbre(TADs.Arbre): Conté totes les classes necessàries per construir i operar l'estructura de dades Arbre.
    - Arbre: conté els mètodes necessaris per construir i operar l'ED arbre.
    - LlistaGenericaNoOrd: conté els mètodes necessaris per construir i operar una llista que es requereix en algunes operacions de l'ED arbre.
    - MeuIterator: és el iterator de la classe LlistaGenericaNoOrd.
    - NodeArbre: conté els mètodes necessàries per manipular un node que forma el Arbre i els valors que conté. El conjunt de nodes d'aquest tipus construeix un Arbre.
- TaulaHash(TADs.TaulaHash): Conté totes les classes necessàries per construir i operar l'ED taula de hash
    - TaulaHash: conté els mètodes necessaris per construir i operar l'ED taula de hash.
    - IteratorHash: és el iterator de la classe TaulaHash.
    - NodeHash: conté els mètodes necessàries per manipular un node que forma la taula de hash i els valors que conté. El conjunt de nodes d'aquest tipus construeix una taula de hash.

Justificació estructures escollides i adequació a la pràctica

Les estructures escollides són:

- Com a arbre: Hem fet un arbre binari de cerca que ens classifica una paraula com a fill esquerre o fill dret segons si alfabèticament va abans de la paraula que actua com a arrel, aleshores es classificarà com a fill esquerre de l'arrel, o per contra va després, i per tant, es classificarà com a fill dret de l'arrel.
    - Pensarem en fer aquest tipus d'arbre a més a més equilibrat, però no ens ha donat temps a implementar-ho i a més en adonarem que utilitzant un iterator de l'arbre(o un clon) i agafant cada cop la primera fulla que no tingues cap fill esquerre cada cop (després la deuríem borrar al iterator) podíem aconseguir fer l'índex correctament.
    - Iterator: en aquesta ED no l'hem implementat ja que amb el clone ens era suficient per obtenir una copia de la ED i poder manipular-la sens por a perdre realment les dades.
    - Implementació: hem realitzat aquest tipus d'implementació explicada, ja que era un tipus d'implementació amb unes característiques que s'adequaven a les que necessitàvem per a realitzar aquest programa amb un arbre.

- Com a taula de hash: Hem fet una taula de hash que distribueix els objectes a la taula a partir del seu hash code (línia comentada en //), però en tractar-se de Strings que s'havien de classificar alfabèticament hem dissenyat la nostra pròpia funció per a que ens generes un codi específic per cada paraula i distribuir-la en la taula, de manera que aquesta quedarà ordenada alfabèticament amb les paraules que comparteixen un mateix lexema encadenades.
    - Codi hash: el codi que generem es realitza a partir de les lletres que formen la paraula, i la dimensió de la pròpia taula. De manera que quan major sigui la taula millor podrem diferenciar les paraules, en el nostre cap hem utilitzar una dimensió de 676(26x26) ja que disposem de 26 lletres de la 'a' a la 'z', i així podem diferenciar amb posicions diferents a la taula totes les paraules que es diferencien en les dos primeres lletres (1ª posició per una paraula que comença per aa, 2ª per ab... fins a la última zz.
    - Les paraules que queden encadenades no tenen perquè estar ordenades alfabèticament ja que sols podem garantitzar que tenen les 2 primeres lletres iguals, de manera que aquest fet el tenim en compte al hora de tractar les dades al main, per generar el fitxer amb l'índex de les paraules.
    - Iterator: en aquest cas hem implementat un iterator per aquesta classe per poder manipular al main les dades sense por a fer alterar les dades originalment classificades a l'ED.
    - Implementació: hem realitzat aquest tipus d'implementació explicada, ja que un tipus d'implementació que amb el codi generat ens permetia després generar l'índex amb facilitat, sense comptar amb els cassos concrets en els que tenim les paraules encadenades.

Especificació TADs pre/post

TADs (comú a les dos ED)

Afegeix un element k de tipus K i amb els valors v de tipus V a l'ED
@pre cert
@post S'afegeix un element nou a l'ED
void afegir (K k, V v);

Esborra un element k de tipus K, i retorna els seus valors v de tipus V. En el cas de que no es trobi el element k, es retorna null
@pre cert
@post L'element queda eliminat de l'ED
V esborrar(K k);

Consulta si un element k de tipus K es troba l'ED, i retorna els seus valors v de tipus V. En el cas de que no es trobi el element k, es retorna null
@pre cert
@post L'ED es manté inalterada
V esborrar(K k);

Ara sols comentarem alguns mètodes de gran importància a cada ED en particular:

Arbre

Construeix un nou arbre amb l'arrel en un element k de tipus K i amb els valors v de tipus V a l'ED
@pre cert
@post Es construeix l'ED
Arbre (K k, V v);

Construeix un nou arbre sense l'arrel. El primer element afegit serà l'arrel.
@pre cert
@post Es construeix l'ED
Arbre ();

Retorna el node que correspon al primer element de tipus K que no te cap fill esquerre
@pre cert
@post retorna el node
NodeArbre<K,V> mesEsquerre();

TaulaHash

Construeix una nova taula de hash amb la dimensió rebuda com a capacitat
@pre cert
@post Es construeix l'ED
TaulaHash (int capacitat);

Col·loca un node rebut com a paràmetre a la posició indicada, aquest mètode s'ha implementat per facilitar la construcció d'un iterator d'aquesta classe
@pre cert
@post Col·loca el node rebut a la posició rebuda a l'ED
Void afegir2(NodeHash<K, V> node, int posicio);

Genera un codi específicament per a una paraula
@pre cert
@post retorna el codi generat
int hashCode(K k);

Retorna un node en la posició demanada
@pre cert
@post retorna un punter al node situat en posició i
NodeHash<K, V> consultarIessim(int i);

Anàlisi de cost de les operacions

Arbre

- Crear: $\theta(1)$
- Afegir: $\theta(\log_2 n)$
- Esborrar: $\theta(\log_2 n)$
- Consultar: $\theta(\log_2 n)$
- Arrel: $\theta(1)$
- MesDret: $\theta(\log_2 n)$
- MesEsquerre: $\theta(\log_2 n)$
- Clone: $\theta(n)$

El cost espacial que té aquesta ED serà tan gran com nodes tinguem. Sols ens ocupa espai un node.

TaulaHash

- Crear: $\theta(1)$
- Afegir: $\theta(1)$ i si sempre s'encadena a la mateixa casella $\theta(n)$
- Afegir2: $\theta(1)$
- Esborrar: $\theta(1)$ i si sempre s'encadena a la mateixa casella $\theta(n$
- Consultar: $\theta(1)$ i si sempre s'encadena a la mateixa casella $\theta(n$
- HashCode: $\theta(1)$
- ConsultarIessim: $\theta(1)$
- Iterator2: $\theta(n)$

El cost espacial que té la taula de hash es el que hem reservat per aquesta (la seva dimenció) més els nodes que queden encadenats a fora de la taula.

Per tant, el cost espacial de un arbre podem considerar que serà en general menor que el de una taula hash. O serà igual en el cas de que la taula hash quedi plena al complet de manera que el arbre tindrà tants nodes com posicions te la taula hash.

Anàlisi del temps d'execució en base al nombre de paraules en general

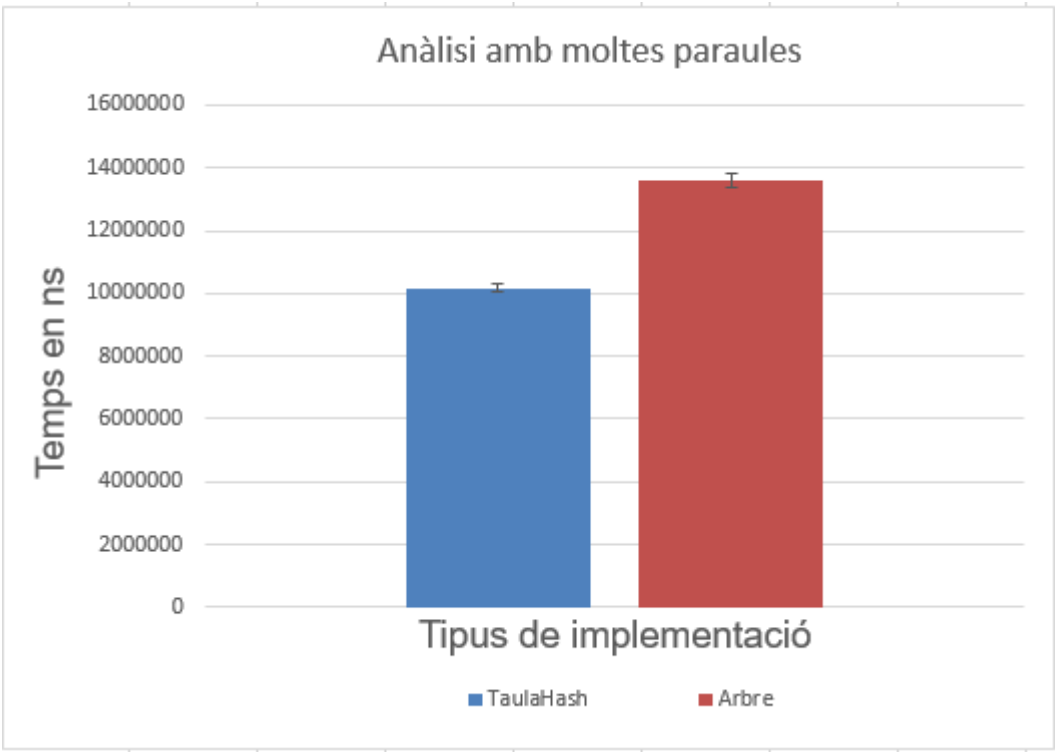
S'ha utilitzat un fitxer anomenat TextLlarg.txt per analitzar els diferents temps d'execució amb a les dues EDs. Hem fet l'anàlisi del cost per triplicat i la mitja d'aquest per obtindré valors més reals.

En aquest anàlisi el que es pretén es veure el temps d'execució de les diferents EDs en un cas real.

Hem obtingut les següents dades:

nº intent	TaulaHash	Arbre
1	10085320	13839323
2	10329026	13419303
3	10123229	13526979
Mitja	10179192	13595202
Desviació	131137	218163

I hem realitzat el següent gràfic comparatiu amb aquestes:



Com a conclusions d’aquesta comparació podem dir que:

-El cost per a construir l’estructura de dades i operar amb ella és menor a la taula de hash que no pas al arbre.

Això es deu a que al tenir un codi hash bastant òptim en aquest cas podem accedir directament a la posició que necessitem en cada paraula i ens permet operar amb un menor temps que no pas en l’arbre.

Anàlisi del temps d'execució en base al nombre de paraules amb un mateix lexema

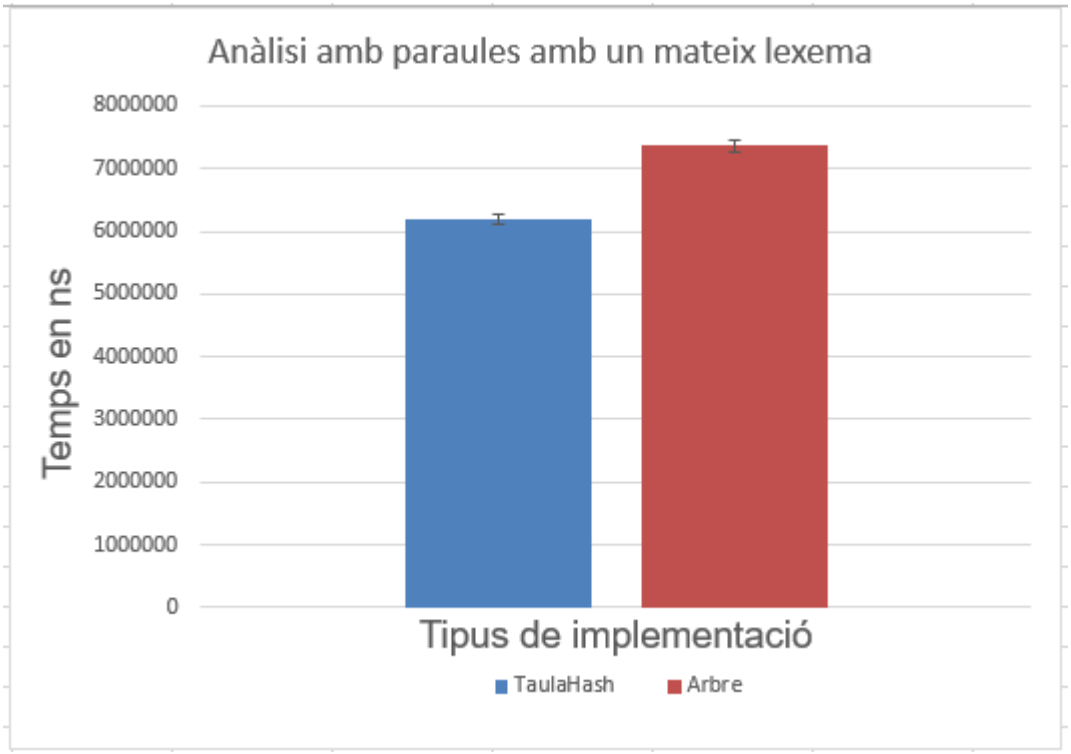
S'ha utilitzat un fitxer anomenat TextLexema.txt per analitzar els diferents temps d'execució amb a les dues EDs. Hem fet l'anàlisi del cost per triplicat i la mitja d'aquest per obtindré valors més reals.

En aquest anàlisi el que es pretén es veure el funcionament de les EDs en el cas de tractar-se de paraules que comparteixen lexema. De manera que a l'ED Arbre no es veurà afectada, però la taulaHash si ja que no distribuirà correctament les diferents paraules.

Hem obtingut les següents dades:

nº intent	TaulaHash	Arbre
1	6296894	7258285
2	6184946	7472357
3	6129399	7356988
Mitja	6203746	7362543
Desviació	85315	107144

I hem realitzat el següent gràfic comparatiu amb aquestes:



Com a conclusions d’aquesta comparació podem dir que:

- El cost per a construir l’estructura de dades i operar amb ella és menor a la taula de hash que no pas al arbre.
-La diferència de cost entre ambdues EDs es menor que en el cas anterior.

Contràriament, a lo esperat hem obtingut de nou una taula de hash més ràpida que l’arbre. Això podem pensar que sigui degut a que tot i hi haure paraules encadenades, el fet de operar amb aquestes si resulti més costos a la taula de hash, però com que hi ha diversos grups de paraules igualment, l’avantatge obtingut al tenir cada grup a una posició de la taula faci que en el còmput global de temps sigui més ràpida la taula de hash que l’arbre. Aquesta hipòtesi explicaria perquè la taula de hash es més ràpida i perquè ha hi ha menor diferencia de temps entre un i l’altra respecte el cas anterior.

Joc de proves

Hem realitzat les següents proves:

Prova	Resultat
Paraules amb caràcters especials(ç, á, ì, etc)	Aquests caràcters són transformats en caràcters similars que es troben a dintre de l'abecedari de 26 lletres de la a-z que es troben de forma consecutiva al codi ascii. Per tant, es tracten correctament.
Paraules en majúscula	Es passen a paraules en minúscula per evitar problemes en el tractament d'aquestes. Per tant, es tracten correctament.
Paraules amb guions, apòstrof, etc	S'eliminen els caràcters que hi ha abans o després del caràcter conflictiu(-, ' , etc) per evitar problemes així ens quedem amb la paraula realment. Sempre ens quedem amb la part de la cadena de caràcters més gran de manera que ens acabem quedant la paraula. Per tant, es tracten correctament.
Paraules que ja havien aparegut prèviament amb el caràcter '$'	Les conta com si no el tinguessin, ja que ja tenim aquesta paraula guardada a la ED, sols hem de anotar la plana i la línia noves.
Paraules que ja havien aparegut al text prèviament	Sols les conta a partir de que hagin sigut marcades com a repetitives amb '$'
Si no hi ha cap paraula marcada com a repetitiva?	Simplement es genera un fitxer índex en blanc
Si apareix una paraula repetitiva dues vegades en la mateixa línia?	Sols considerem la primera aparició d'aquesta a l'ED, de manera que sols anotem la plana i la línia una vegada.
Amb una taula de hash es crea l'índex correctament?	Si
Amb un arbre es crea l'índex correctament?	Si
L'ED queda alterada al utilitzar el iterator de la taula hash?	No, ja que es crea una copia de l'ED i és aquesta copia la que pateix les modificacions.
L'ED queda alterada al utilitzar el clone de l'arbre?	No, ja que es crea una copia de l'ED i és aquesta copia la que pateix les modificacions. Té la mateixa funció que el iterator de la taula hash.
Si introduïm algun caràcter al escollir opció?	Demana una nova opció

Es crea un fitxer Índex correctament?	Si, es crea un fitxer amb les paraules ordenades alfabèticament de la mateixa manera que es mostra al exemple del guió de la pràctica.
Warnings al JavaDoc	Tots solucionats

Per a realitzar aquestes proves concretes, s'han utilitzar els fitxers Text 1,2,3 i 4. I s'han anat manipulant, el text 2 i 3 són similars principalment s'han utilitzat per veure com actuava la taula de hash en encadenar paraules similars. Mentre, que el fitxer 4 es una mescla dels fitxers 1 i 3, per tal de fer una prova definitiva.

D'altra banda, s'han utilitzar altres dos fitxers per fer el anàlisi del temps de execució que ja han sigut comentats abans.