UNIVERSITAT ROVIRA i VIRGILI

Estructura de Dades

PRÀCTICA 3: Indexador de textos

Cristina Izquierdo Lozano i Aleix Mariné Tena

Doble Grau Biotecnologia + Enginyeria Informàtica

17/6/17

Contenido

Disseny

Breu explicació dels TADs escollits i les implementacions triades. Justificació de cada decisió.

Hem dividit el nostre programa en diversos paquets per tal de separar les diferents funcions de cadascun, i programar de manera clara i ordenada. Aquests són els següents:

- **Paquet Principal**: En aquest paquet trobem una classe pel programa principal:
    - **Main:** Aquest fa la crida dels diferents mètodes i s'encarrega del tractament de dades. Hem utilitzat una estructura *top-to-down*, de manera que hem programat diferents funcions que s'encarreguen d'una determinada tasca específica. Aquestes funcions són cridades al main. La descripció d'aquestes funcions i dels seus arguments així com dels seus retorns es pot trobar a la capçalera. Dins el programa principal al destacar la funció llegir fitxer que s'encarrega de llegir el fitxer i afegir totes les dades al nostre TAD. A l'hora de llegir les paraula hem tingut en compte que aquesta podia començar amb apòstrofs, comes y punts. Podríem haver tingut en compte altres grafies, però amb aquests ja comptava gairebé totes les paraules.
- **Paquet TAD**: En aquest paquet trobem les diferents classes creades per a cada tipus d'estructura, les quals implementaran els diferents mètodes definits a la interfície.
    - **MeuIterator** - Accedeix seqüencialment als elements d'una col·lecció. Aquest conté el constructor per a diverses EDs, com les dues llistes (tant ordenada com no ordenada).
    - **LlistaGenericaNoOrd**- Una llista que afegeix elements de manera seqüencial. Ens feia falta per a construir l'*iterator* en l'arbre AVL.
    - **LlistaGenericaOrd** - Una llista que afegeix elements de manera ordenada. Ens feia falta per fer un Índex ordenat alfabèticament en la classe JavaUtil i en la TaulaHashEncadenadaIndirecta.
    - **JavaUtil** - Classe opcional, implementa la classe *Hashtable*, definida al java.util. La classe que hem creat nosaltres no és més que un *wrapper* d'una instància d'aquesta classe. D'aquesta manera podem fer que aquesta classe s'adapti a la interfície proposada. Conté els métodes propis d'una taula de Hash, més alguns extres. Hem escollit aquest JavaUtil perquè una taula de Hash és una de les formes més ràpida d'accedir a les dades, tot i que a costa de gastar força memòria.
    - **AVLdinamic** - Aquesta classe conté el codi que vam fer al laboratori de l'arbre ABC força modificat i amb diversos mètodes extra per a reequilibrar l'arbre cada cop que s'afegeix o s'extreu un element. Hem escollit l'arbre AVL perquè té cost logarítmic per a fer les principals operacions (afegir, esborrar i consultar). Contràriament al que li pasa a l'arbre ABC, l'arbre AVL no es desequilibra, pel que el cost logarítmic és estrictament logarítmic independentment de com es manipuli l'arbre. El fet que l'arbre ABC estigui molt desequilibrat comporta que la ED acabi tenint forma de llista dinàmica, és a dir

cost lineal. Pel problema proposat llavors, l'arbre AVL és el més adequat. Una altra opció amb menys rendiment però que hagués estat adequada hagués sigut l'arbre TRIE, que se sol utilitzar per a diccionaris.

    - **NodeHash**: Conté una referència a un altre NodeHash, una clau i un valor. S'utilitzen en la TaulaDeHashEncadenadaIndirecta.
    - **TaulaHashEncadenadaIndirecta**: Hem escollit aquesta implementació perquè a més de ser la més senzilla i intuïtiva, és la que desaprofita menys espai (el principal inconvenient de les taules de Hash), ja que s'utilitza memòria dinàmica per a emmagatzemar les col·lisions. D'igual manera, pel problema que teníem era difícil arribar a tenir una col·lisió amb una taula no massa gran, de manera que el rendiment seguia sent constant i no es troba alterat perquè hi hagin col·lisions. Hem utilitzat la versió que vam fer en una de les classes del laboratori, modificada per a que sigui encadenada indirecta. De manera que té com a extra un anàlisi sobre les col·lisions que hem mantingut. No ho tindrem en compte a l'hora de comentar els costos perquè no s'utilitza per al funcionament de la taula

- **Paquet Tipus:** Paquet que conté els diferents tipus d'objecte que trobem al programa. Amb els constructors i els getters i setters corresponents.
    - **Aparició:** Conté dos enters que indiquen línia i plana d'aparició.
    - **Índex:** Conté una taula d'aparicions, així com un enter que indica la última posició afegida.
- **Paquet Interface**: Paquet que conté les diferents interfícies del programa.
    - **TADIndex:** Ens serveix de guia per a implementar els mètodes de cada classe que hauria de tenir una classe Índex. També és abstracta i de la qual hereten les altres 2 interfícies de manera que ens permet utilitzar una sola variable per a referenciar qualsevol dels tipus de tipus de llista que l'usuari vulgui utilitzar. Així només hem d'implementar les funcions del main un sol cop sobre una variable d'aquest tipus.
    - **TADTaulaHashGenerica:** Interfície abstracta que conté els métodes que cal implementar en una taula de Hash.
    - **TADAVL**: Interfície abstracta que conté els métodes que cal implementar en un arbre AVL.

TADs

L'especificació dels TADs

TADAVL

```java
package Interface;

import TAD.AVLdinamic;

/**
 * Interface per a definir l'arbre binari AVL per a la pràctica 3
 *
 * @author Aleix Mariné
 *
 **/
public abstract interface TADAVL<K extends Comparable<K>, V> extends TADIndex<K, V>{

    /**
     * Afegeix un nou element a l'arbre
     */
    public boolean afegir( K k, V v);

    /**
     * Esborra un element de l'arbre utilitzant la seva clau.
     */
    public boolean esborrar( K k);
    /**
     * Comprova si exiteix un element cercant-lo amb la seva clau. retorna cert si hi és, retorna fals si no
     */
    public boolean existeix( K k);
    /**
     * Retorna les dades d'un element a partir de la clau
     */
    public V consultar(K k);
    /**
     * Calcula el nombre de nodes i el retorna
     * @return retorna nombre de nodes
     */
```

```java
public int numNodes();
/**
 * Retorna l'altura d'un arbre rebut per paràmetre
 * @param k arbre del que es vol calcular l'altura
 * @return altura máxima
 */
public int height(AVLdinamic<K, V> k);
    /**
     * Retorna l'element més gran de tot l'arbre
     * @return clau de l'element
     */
public K maxim();
/**
 * Retorna l'element més petit de tot l'arbre
 * @return clau de l'element
 */
public K minim();
/**
 * retorna cert si l'arbre està buit, sino retorna fals
 * @return
 */
public boolean esBuit();

/**
 * Retorna una llista amb el recorregut preordre
 * @return llista
 */
public LlistaGenericaNoOrd<K> preordre();
```

```java
/**
 * Retorna una llista amb el recorregut inordre
 * @return llista
 */
public LlistaGenericaNoOrd<K> inordre();
/**
 * Retorna una llista amb el recorregut postordre
 * @return llista
 */
public LlistaGenericaNoOrd<K> postordre();
/**
 * retorna una instància de l'arbre clonada
 * @return isntància de l'arbre
 */
public TADAVL<K,V> clone();
```

TADIndex

```java
package Interface;

public abstract interface TADIndex <K extends Comparable<K>, V>{
    /**
     * Afegeix un element del que es fa seguiment a l'index
     * @param k
     */
    public boolean afegir(K k, V v);    // aquest ha de ser k i v
    /**
     * Afegim un nou element k
     * @param k
     */
    public boolean afegirAparicio(K k, int plana, int linia);

    /**
     * Esborra un element de l'index (deixa de fer el seguiment)
     */
    public boolean esborrar(K k);

    /**
     * Consulta un element (paraula) de l'índex
     */
    public V consultar(K k);
    /**
     * Mostra per pantalla tot l'index
     */
    public String mostrarIndex();
```

```java
/**
 * Mira si existeix l'element amb la clau k
 * @param k
 * @return
 */
boolean existeix(K k);
```

TADTaulaHashGenerica

```java
package Interface;


/**
 * Interface per a definir una taula de hash generica.
 *
 * @author Professors de l'assignatura 16-17
 *
 */

public abstract interface TADTaulaHashGenerica<K extends Comparable<K>,V> extends TADIndex <K, V>{

    /**
     * Afegeix un element a la taula de hash
     * @param k - clau de l'element a afegir
     * @param v - element a afegir
     */
    public boolean afegir(K k, V v);

    /**
     * Afegeix una aparicio d'un element a la taula de hash
     * @param k - clau de l'element on hem d'afegir l'aparicio
     * @param plana - plana on es troba l'element
     * @param linia - linia on es troba l'element
     */
    public boolean afegirAparicio(K k, int plana, int linia);

    /**
     * Esborra un element a la taula de hash
     * @param k - clau de l'element a esborrar
     */
    public boolean esborrar(K k);

/**
 * Consulta un element a la taula de hash
 * @param k - clau de l'element a consultar
 */
public V consultar(K k);

/**
 * Retorna el factor de carrega actual de la taula de hash
 */
public float getFactorDeCarrega();
/**
 * Mostra per pantalla tot l'index
 */
public String toString();
```

Anàlisi del cost de les operacions

Anàlisi de cost de les operacions en les diferents implementacions

Només es descriuran els costos de les operacions principals de cada estructura de dades (afegir, eliminar, consultar i alguns altres mètodes rellevants), ja que hi ha molts mètodes escrits a les implementacions que no s'han fet servir o bé simplement serveixen per estalviar un parell de línies de codi. A més, la majoria d'aquests mètodes tenen cost constant. Consideraré el cost espacial igual a la mida dels elements que tingui l'estructura de dades, és a dir, no tindré del tot en compte si el llenguatge de programació treballa amb punters o no, o com organitza de manera interna.
Cost de les operacions:

1. **Taula de Hash encadenada indirecta**: La taula de hash encadenada indirecta consisteix en una taula de mida n que conté en cada posició un element HashNode. Aquest element conté dos elements genèrics k (clau) i v (valor), i una referència al següent nodeHash. Degut a que és una taula d'objectes gastem com a mínim n*mida(NodeHash) per a reservar memòria. Això es força memòria desaprofitada, tenint en compte que la taula de Hash més òptima és aquella que no està massa plena per a no tenir col·lisions. Tot i així, les col·lisions es guarden encadenades a cada NodeHash, de tal manera que les col·lisions ocupen memòria de manera dinàmica, així no estem reservant espai per a col·lisions, sinó utilitzant memòria per a col·lisions exclusivament quan es necessari. A més, per al nostre problema i recursos dels que disposem, la taula pot ser força gran sense tenir problemes per falta de memòria, no hi hauran gairebé col·lisions, pel que accedir a un element depèn d'un cost constant, no de recórrer totes les col·lisions.

   Les taules de Hash ocupen força memòria però permeten un accés constant a les dades. Per tant hem escollit aquesta implementació perquè ens sembla que és el que més compensa aquest malbaratament de memòria sense afectar massa al rendiment.
   a. **Constructor**: O(k). Reservem espai per a totes les estructures i variables.
   b. **Afegir:** O(numcol·lisions). Com ja sabem els costos sempre s'avaluen en el pitjor dels casos. De tal manera que el pitjor cas en la nostra taula de Hash és que haguem d'accedir a les dades O(k) i hi hagin col·lisions que calgui recórrer per a trobar la última posició O(numcol·lisions). Per la regla de la suma s'arriba al resultat.
   c. **Eliminar:** O(numcol·lisions). Semblant al cas anterior, hem d'accedir a la posició de la taula O(k) i després recorre'ns les col·lisions O(numcol·lisions)fins a trobar l'element a esborrar, que pot no ser-hi. Si hi és només cal recolocar els punters amb cost O(k). Per la regla de la suma arribem al resultat.
   d. **Afegir Aparició**: O(numcol·lisions). Aquest és un mètode no genèric ja que tracta l'element V com un índex ja que ha d'afegir una posició. Accedim a les dades segons la clau O(k), recorrem les col·lisions O(numcol·lisions) i finalment (si trobem l'element) li afegim una aparició, aquesta operació té cost constant ja que guardem en quina posició es troba l'última inserció. S'ha escollit implementar-ho com un mètode no genèric, però som conscient que es podria haver fet amb un consultar en aquests i totes les EDs.
   e. **Consultar:** O(numcol·lisions). El mateix que afegiraparició, però aquí ens estalviem afegir aparició i retornem l'Objecte V directament.
   f. **Mostrar Índex**: Primer recorrem tota la taula de Hash O(n) i a cada posició recorrem les possibles col·lisions O(numcol·lisions). Per regla de la multiplicació tenim que O(numcol·lisions*n). Cada posició no nul·la s'afegeix a una taula ordenada, cada afegiment dins d'aquesta taula té un cost de

O(elements), ja que la taula es recorre buscant la posició pel nou element. Pel que per regla de la multiplicació de moment tenim cost O(n*numcolisions*elements). Es fa un Iterator d'aquesta taula amb cost O(elements), que per regla de la suma no cal tenir en compte. Finalment s'imprimeixen els elements que surten de l'iterator, per tant aquest últim recorregut té cost O(elements), que tampoc cal tenir en compte per regla de la suma.

2. **Arbre AVL:** Els arbres AVL consisteixen en arbres que s'autoequilibren cada cop que sofreixen modificacions. Com ja sabem el cost de fer una cerca es de O(log n) igual que en un arbre ABC, mentre que reequilibrar tot l'arbre en el pitjor dels casos també seria O(log n), pel que tant les insercions, com les delecions, com les consultes (cerques) tenen cost O(log n). Pel que fa al ús de memòria, l'arbre AVL en una implementació dinàmica només gasta memòria pels Nodes que estan plens. Aquests Nodes, són una classe interna de l'arbre anomenada node ABC (ja que també funcionen com un node per a un arbre ABC). En el cas dels arbres AVL, si volem que el cost de les operacions ja comentades no es dispari hem d'utilitzar un punter per anar a l'arrel de l'arbre, així com reservar dos enters per a l'altura i el factor d'equilibri (per a no haver d'estar cercant l'arrel contínuament ni re calculant aquests termes). Podem observar com el Node ABC ocupa més espai que no el Node Hash, però només utilitzem memòria per als nodes utilitzats contràriament al que passava amb l'altra implementació.

   a. **Constructor:** O(k). Depenent del constructor farem una cosa o una altra, però el cost sempre és constant.
   b. **Afegir:** O(log n). Fem una cerca en l'arbre de la posició on afegim el nou element O(log n). Quan el trobem connectem els punters O(k) i recorrem l'arbre fins a l'arrel reequilibrant-lo (re calculant factors d'equilibri, altura i fent rotacions) el que té un cost O(log n). Per la regla de la suma arribem al resultat.
   c. **Eliminar:** O(log n). Fem una cerca en l'arbre de la posició on esborrem l'element O(log n). Depenent del nombre de fills del node l'algoritme és pot complicar una mica, però la idea es que com a màxim l'hem de recórrer fins a baix per a poder re col·locar els elements per a poder fer l'eliminació. El que té cost O(log n). Finalment, pugem fins a l'arrel reequilibrant l'arbre O(log n). Per regla de la suma arribem al resultat.
   d. **Consultar:** O(log n). Es tracta de fer una cerca.
   e. **Mostrar Index**: O(n). Fem el recorregut inordre de l'arbre pel que visitem tots els elements O(n). Cada element és afegit a una llista no ordenada amb cost O(k). Es crea un Iterator d'aquesta llista amb cost O(n). Recorrem tot l'iterator imprimint en un String O(n). Aplicant a regla de la suma a tots els apartats obtenim el resultat.

Anàlisi temps d'execució del joc de proves

Anàlisi dels temps d'execució dels diferents jocs de proves comparant les diferents implementacions.

Hem suposat que el contingut del fitxer sempre és correcte i generarà cap error. Per tant, els possibles errors que ens podem trobar en aquesta pràctica són deguts al usuari, per exemple:

1. Escollir una implementació que no existeix.
2. Escollir una opció del menú que no existeix.
3. No introduir bé el nom del fitxer.

1.

```
Quina versio vols utilitzar?
1.- Estructura en Taula de Hash.
2.- Estructura en arbre AVL.
3.- Estructura de les java.util.collections.
4

Aquesta opcio no esta a la llista.

Quina versio vols utilitzar?
1.- Estructura en Taula de Hash.
2.- Estructura en arbre AVL.
3.- Estructura de les java.util.collections.
```

2.

```
Ha tardat 0.101129279 segons
Quina operacio vols fer?(Les aparicions de cada paraula es mostren com 'plana:linia')
1.- Mostrar index.
2.- Consultar paraula a l'estructura.
3.- Esborrar paraula.
4.- Reinicia el programa.
5.- Sortir del programa.
6

Aquesta opcio no esta a la llista...
```

3.

```
Quin es el nom del fitxer de dades?
sdf
ERROR: No existeix aquest fitxer!
```

Val a dir, que no hem vist necessari l'ús de cap classe Exception feta per nosaltres ja que hem controlat tots els errors que hi pugui haver mitjançant condicionals. Les excepcions que estan controlades (try/catch) son aquelles que es generen en temps d'execució per errors E/S, i la majoria estan al main.

Les diferents operacions de consulta són:

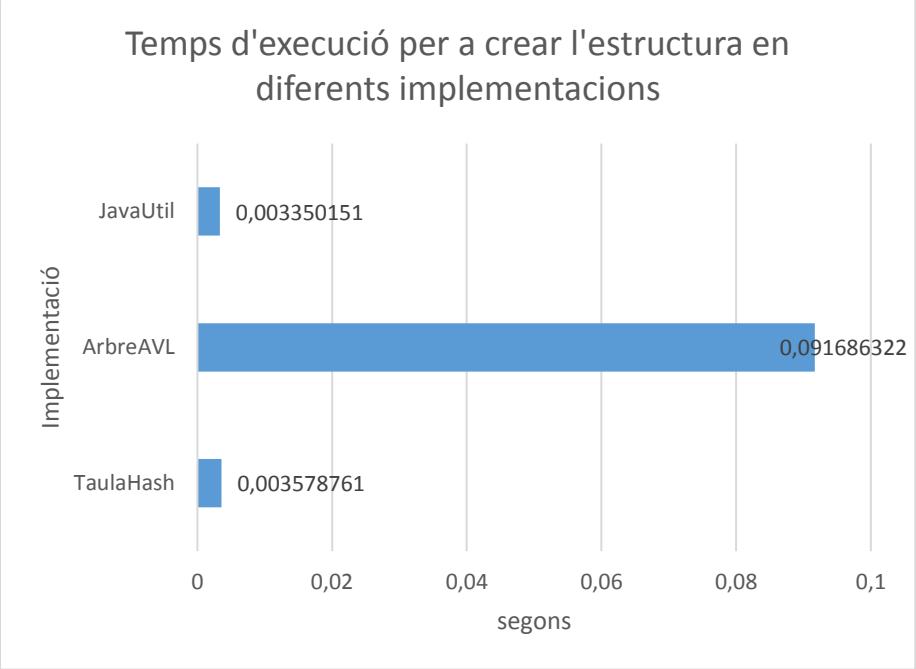
1. Mostrar índex
2. Mostrar Aparicions d'una paraula
3. Borrar una paraula

Hem utilitzat diferents texts els quals hem destacat diferents paraules. Hem calculat el temps per a cada consulta utilitzant diferents paraules i hem fet la mitjana d'aquest temps.

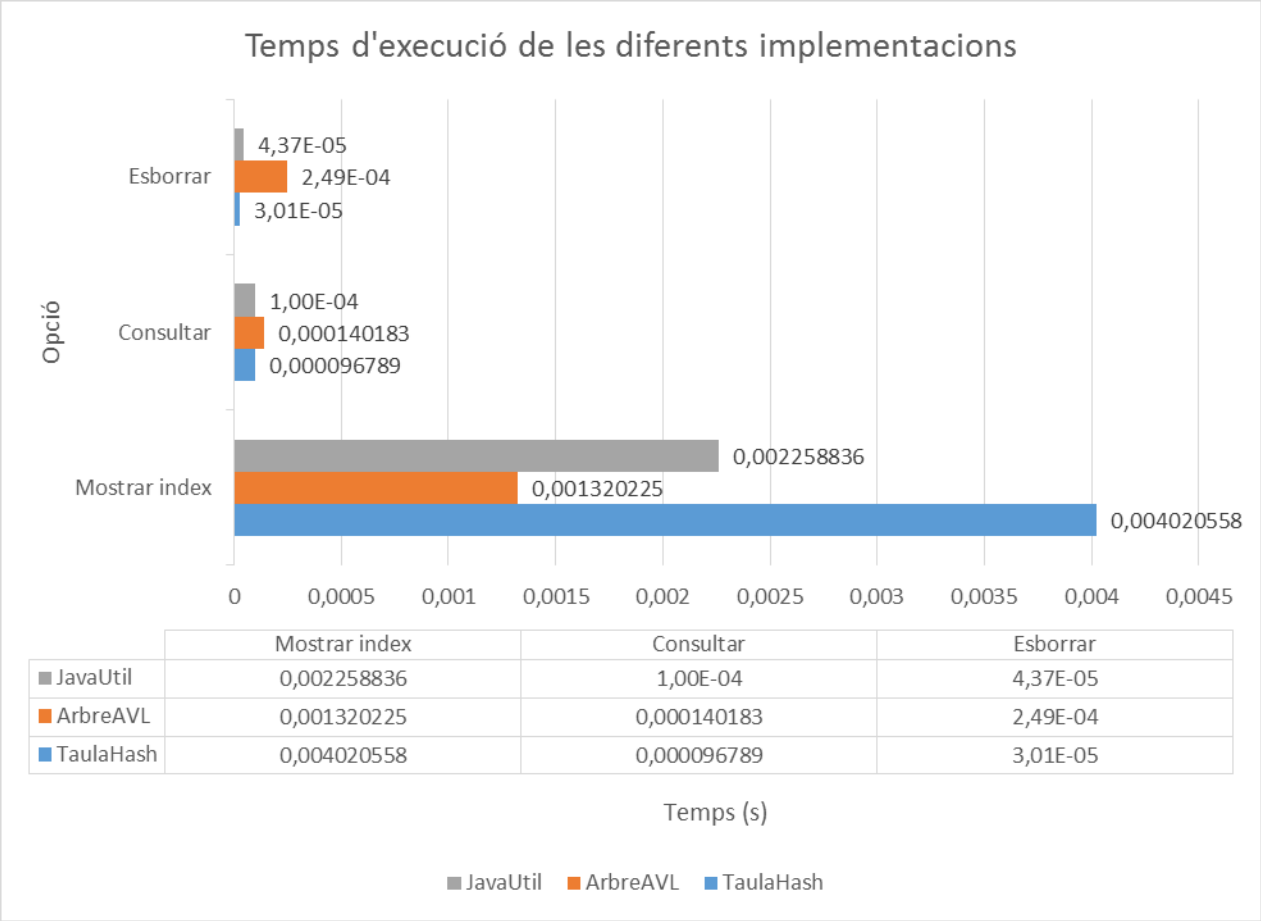
Un dels texts que hem utilitzat és aquest, que hem extret de wikipedia.

```
<Plana numero=1>
In computer science, an $AVL $tree is a self-balancing $binary
search tree. It was the first such $data structure
to be invented. In an AVL tree, the heights of the two child
subtrees of any node differ by at most one; if at any
time they differ by more than one, rebalancing is done to restore
this property. Lookup, insertion, and deletion all
 take $O(log n) time in both the average and worst cases, where n
is the number of nodes in the tree prior to the
 operation. Insertions and deletions may require the tree to be
rebalanced by one or more tree rotations.

<Plana numero=2>
The AVL tree is named after its two Soviet inventors, Georgy
Adelson-Velsky and Evgenii Landis, who published it in
their 1962 paper "An algorithm for the organization of
information".

<Plana numero=3>
AVL trees are often compared with red–black trees because both
support the same set of operations and take O(log n)
time for the basic operations. For lookup-intensive applications,
AVL trees are faster than red–black trees because
they are more strictly balanced.[4] Similar to red–black trees,
AVL trees are height-balanced. Both are, in general,
```
neither weight-balanced nor μ-balanced for any μ≤1⁄2; that is, sibling nodes can have hugely differing numbers of descendants.

A continuació es mostra un gràfic amb els resultats.



Aquest gràfic correspon al temps per a introduir totes les dades, el que es correspon a llegir el fitxer, obtenir les línies, tokenitzar-lo i anar afegint les paraules amb les seves aparicions.
Podem observar com el JavaUtil i la TaulaHash gasten poc temps en aquesta operació, degut a que tenen accés amb cost constant als elements. L'arbre en canvi, tarda 30 cops més que les altres estructures de dades, ja que ha de buscar la posició on cercar l'element, així com reequilibrar l'arbre gairebé cada cop que afegeix un element. El gràfic per tant, es correspon al calculat a l'apartat de costos.



	Mostrar index	Consultar	Esborrar
JavaUtil	0,002258836	1,00E-04	4,37E-05
ArbreAVL	0,001320225	0,000140183	2,49E-04
TaulaHash	0,004020558	0,000096789	3,01E-05

En el següent gràfic veiem el cost que tenen les diferents operacions. Tant la ED JavaUtil com la taula Hash presenten el mateix cost aproximadament en consultar y esborrar. Això es degut a que les dues tenen cost constant per accedir a les dades. L'arbre en canvi, presenta pitjor rendiment per a aquestes dues operacions, degut al cost que implica esborrar una dada i consultar-la. També té sentit que el cost de consultar una paraula sigui menor que el d'esborrar en l'arbre AVL degut als reequilibraments que cal fer.

Per últim l'operació mostrar Índex és tot el contrari, l'arbre AVL presenta millor rendiment que les altres dues EDs. Això coincideix amb el calculat a l'apartat de costos. Podem adonar-nos de un dels inconvenients de les taules Hash per a aquest problema. Les taules de Hash no són ordenades, i a més, tenen un cost lineal quan es volen mostrar totes les dades en la ED.

Degut a això com a conclusió hem pensat que la millor ED per a resoldre aquest problema és l'arbre AVL, ja que en un índex és necessari optimitzar sobretot l'operació de mostrar Índex i de consulta. Les altres operacions poden ser lentes ja que pocs cops es faran servir.