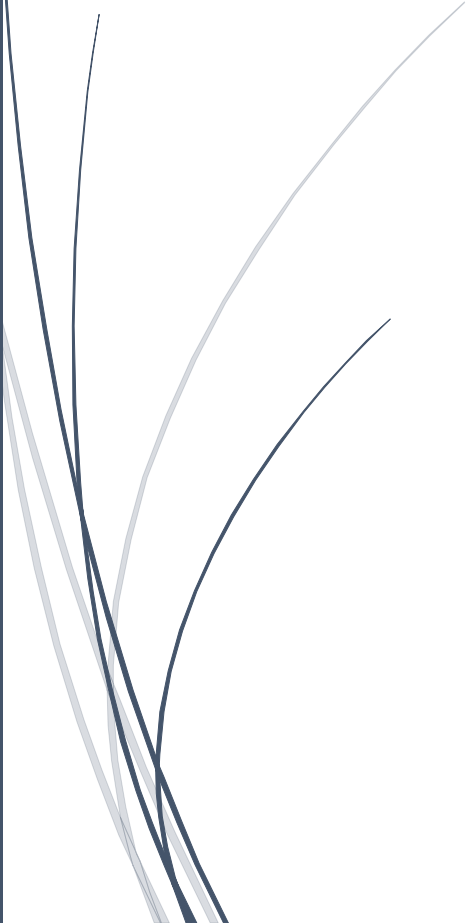


A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

22-8-2016

# WordCount

Pràctica Segon Semestre

Several thin, curved lines in dark blue and light grey originate from the bottom left and sweep upwards and to the right.

PROGRAMACIÓ AVANÇADA I ESTRUCTURA DE DADES  
CURS 2015/16  
LS31289 JORDI RUBIÓ JORNET  
LS30652 GABRIEL CAMMANY

## Índex

Disseny dels modes de recompte .....	2
Array.....	2
Arbre Binari .....	3
Arbre AVL .....	4
Taula HASH.....	4
Taula hash amb AVL .....	6
Taula hash amb Binari .....	7
Resultats.....	8
Dedicació.....	10
Conclusions .....	11
Bibliografia .....	12

## Disseny dels modes de recompte

Exposeu el disseny de cadascun dels 4 modes de recompte que heu implementat. Concretament, dibuixeu les estructures de dades dissenyades i doneu el seu diagrama de classes. Expliqueu també el funcionament i doneu el cost computacional del procés de recompte i obtenció de les 2 modalitats de resultats per a cadascun dels 4 modes de recompte

Durant els últims mesos de classe, hem pogut conèixer diferents estructures que tenen característiques pròpies fent-les apropiades per un cas o altre. Així doncs, per poder dur a terme aquesta practica, hem realitzat un estudi de les diferents tipus de estructures i amb els seus pros i contres alhora de realitzar la tasca de comptar paraules en un text.

Tot i que en el enunciat ens demanen nomes 4 estructures, hem fet la recerca amb un parell mes per tenir una visió mes amplia de quina estructura es mes adient depenent dels casos.

### Array

L'array o també conegut com vector d'accés directe, no es res nou, des de l'any passat ja la coneixíem. Tot i així hem volgut fer la prova i veure quins son les seves avantatges o desavantatges respecte els altres mètodes.

Segurament es l'estructura mes senzilla de totes. Com totes les altres, conte nodes que cada node conte la informació. La diferencia però, consisteix en la manera que cada node esta vinculat al següent i la manera d'organitzar-se.

La particularitat de l'array es que te una mida estàtica, en el cas de l'array estàtic, i per tant un cop que s'han omplert totes les caselles ja no es poden afegir nous nodes.

Com es pot veure a la figura, cada node te un índex que l'identifica i per tant es pot accedir directament a cada node sabent el índex d'aquest.

[0]	[1]	[2]	[3]	[4]
2	5	1	3	4

Per tant, un cop coneguda l'estructura podem arribar a la conclusió que:

- Omplir l'array tindrà cost  $O(n)$ , es a dir, el cost serà el numero d'elements a inserir.
- Inserir un nou element si l'array ja conté elements també tindrà cost  $O(n)$  ja que si no tenim l'índex de l'últim element inserit haurem de moure'ns de l'índex 0 fins a N.
- El cost de cerca també serà  $O(n)$ , ja que en el pitjor dels casos haurem de moure'ns per tota l'array per trobar el node.
- Si tenim el índex del node, el cost de cerca serà  $O(1)$  ja que es podrà accedir directament al element sense tenir que moure'ns per tota l'array.

La memòria que utilitza l'array estàtic sempre serà  $O(n)$ , ja que haurà de tenir tants espais com nodes hi hagin.

En el nostre cas, aquesta estructura, te cost  $O(n^n)$  tant per obtenir les paraules per ordre alfabètic com per número de repeticions.


Aquestes dos segueixen uns principis comuns

- ## Arbre Binari

L'estructura en forma d'arbre no ha de estar balancejada, es a dir, en una mateixa branca pots tenir una diferencia de nivell major a 1 respecte l'altre branca.

- [illegible]

- En el millor dels casos l'altura del arbre seguirà la següent expressió  $\text{altura} = \log(n)$  sent  $n$  el número de elements del arbre.
- En el pitjor dels casos, l'altura serà igual a  $N$ , ja que tots els elements estaran situats en una mateixa branca. Com en la figura següent.



- En quant a la inserció d'un nou node, el cost es similar al de cerca, ja que es tracta de buscar la posició que li correspongui al node i per tant com a màxim tindrà un cost  $O(n)$  en casos que l'altura del arbre sigui igual a  $N$  i en el millors dels casos serà  $O(\log n)$ .

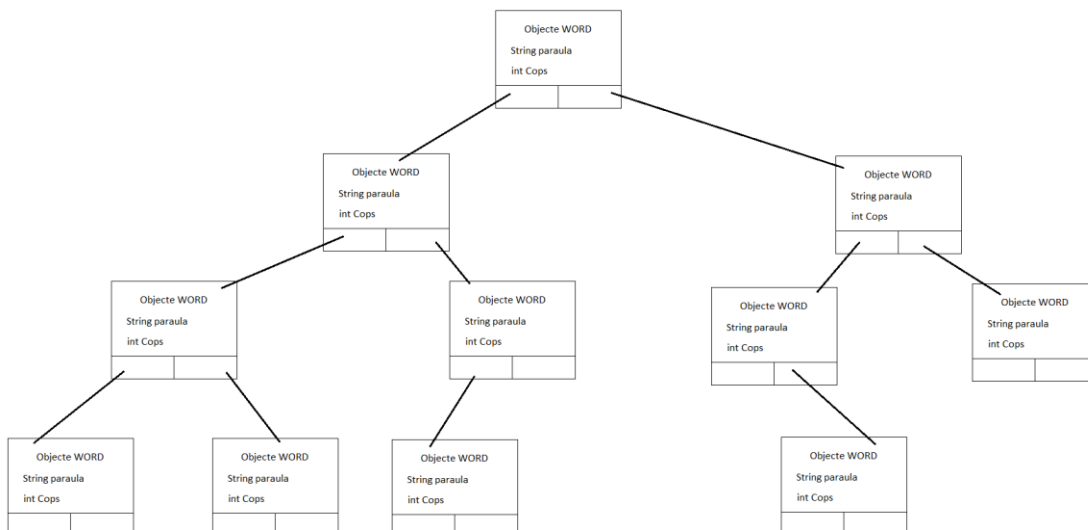
## Arbre AVL

Un cop coneixem el arbre binari, que podem dir que es l'estructura en forma d'arbre mes senzilla, el AVL es un arbre que te la particularitat que es balanceja automàticament per tal de tenir una diferencia entre els dos fills del node pare menor o igual a 1.

Com ja sabem, en el arbre binari la majoria d'operacions com venen a ser, cerca, inserció, etc. Tenen un cost similar a  $O(h)$  (sent  $h$  l'altura del arbre) o en els pitjors dels casos  $O(n)$  (Quant l'altura del arbre es igual a  $N$ ). Els AVL però, com que ens assegurem que l'altura del arbre sempre te una altura  $O(\log n)$  gracies al auto balanceig quan inserim o eliminem un element, el cost de totes les operacions sempre acaben sent  $O(\log n)$  en el millor o pitjor cas.

En la figura següent podem comprovar que el arbre te una estructura que s'adequa a les característiques del AVL. Tots els seus nodes tenen una diferencia menor a 1 seguint la següent operació:

$$\text{diferència} = \text{altura}(\text{BrancaDreta}) - \text{altura}(\text{BrancaEsquerra})$$



Tal com hem comentat, per poder dur a terme el balanceig i per tant tenir un cost de  $O(\log n)$ , es fan les conegudes rotacions que hem donat a classe.

- Rotació dreta (R)
- Rotació esquerra (L)
- Rotació dreta i esquerra (RL)
- Rotació esquerra i dreta (LR)

Per obtenir les paraules per ordre alfabètic, el cost tant el de AVL i el de arbre binari correspon a  $O(n)$  on  $n$  es el número de nodes ja que es visita nomes un cop cada paraula. Això fa que sigui molt mes eficient que altres estructures de dades com venen a ser arrays o taules hash que el seu cost es  $O(k^k)$  on  $k$  es el número de caselles.

Per mostrar per número de aparicions continua sent mes rapida l'estructura de AVL i binari respecte hash i array, ja que una te cost  $O(n^n)$  on  $n$  es el número de nodes i l'altre té  $O(k^k)$  on  $k$  es la mida de la taula o array (teòricament).

## Taula HASH

La taula HASH consisteix en una estructura igual o molt similar a una Array, ja que cada node te un índex que correspon a la posició en l'array i el seu contingut en la casella corresponent. La diferència amb l'array es la manera d'accedir a aquests nodes i l'estructura d'aquests. A més a més, s'utilitza una taula hash quan es vol cercar dintre l'estructura per valors alfanumèrics, es a dir, quan volem organitzar la informació de manera que a partir de per exemple un nom de una persona obtenir el seu índex dintre la taula.

Cada node en una taula Hash te la següent estructura que relaciona una clau amb un valor:

- Clau, valor únic alfanumèric que identifica el valor del node
- Valor, contingut que s'ha volgut afegir a cada node.

Juntament amb això existeix la funció de hash, que tracta sobre una operació que es realitza a partir de un valor alfanumèric, es a dir, la clau, que dona com a resulta la posició dintre la taula.

De manera que una taula de hash ideal, la funció de hash assignaria a cada clau una posició única, sense produir-se les conegudes col·lisions (Diferents claus que se li assigna una mateixa posició).

En les taules de hash també s'ha de tenir en compte la dimensió de la taula per tenir un equilibri en el número de col·lisions i memòria usada i el conegut factor de càrrega.

El factor de càrrega consisteix en la següent operació:

$$factor\ de\ càrrega = \frac{n}{k}$$

- N el número d'elements.
- K mida de la taula.

A mesura que el factor de càrrega augmenta, el rendiment de la taula de hash disminueix i si es un valor molt proper a 0 no interfereix en el temps de cerca sinó que te com a resultat un us ineficient de la memòria, ja que molta d'aquesta no es dona cap ús. Així doncs, s'ha de trobar un valor entremig per no tenir problemes amb el rendiment de la taula i a la vegada fer un us eficient de la memòria.

Es per això, que hi han molts tipus de taules de hash, ja que la manera d'organitzar les col·lisions es fa de manera diferent entre una i altre.

En el nostre cas, hem utilitzat la taula d'adreçament obert que consisteix en l'organització de les col·lisions de la següent manera:

1. Es detecta una col·lisió, es a dir, trobem que hi ha un altre clau en la mateixa posició que retorna la funció de hash.
2. Incrementem linealment el valor hash que hem trobat i realitzem la funció rehash.
3. Quan trobem que la posició retornada no hi ha cap node, podem assegurar que la clau no existeix i per tant l'afegirem a l'estructura.

El cost computacional a l'hora de fer una cerca es el següent:

- En el millor dels casos el cost serà  $O(1)$ , ja que la posició que retorna la funció de hash equival a la posició de la clau.
- En els pitjors dels casos el cost serà  $O(n)$ . Aquest cas es dona quan totes les claus donen la mateixa posició i per tant es passarà com a màxim per tots els elements.

De la mateixa manera, quan inserim o esborrem, el cost mínim pot ser de  $O(1)$  i el màxim  $O(n)$ . Això fa que les taules de hash, si la funció de hash es bona i s'ha tingut en compte un bon factor de càrrega, la cerca d'elements sigui majoritàriament més ràpida que la majoria d'estructures, com venen a ser arbres, vector d'accés directe, etc.

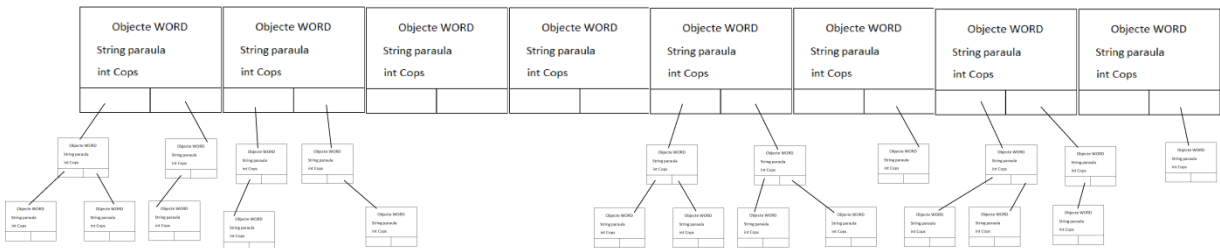
En el nostre cas, donat que es tracta de veure el número de repeticions de paraules en texts, el factor de càrrega si fem que la taula sigui igual al número de paraules serà bastant proper a 0. A la vegada que la cerca tindrà un cost molt similar a  $O(1)$  ja que el número de paraules úniques del total de paraules acostuma a ser menor al 20% i per tant el número de col·lisions hauria de ser bastant baix.

Cada element tindrà com a clau la paraula i el valor serà un objecte Word. Aquest conte el la paraula i el número de repeticions d'aquesta.

### Taula hash amb AVL

Un cop vistes les estructures anteriors, hem decidit fer una estructura més complexa ajuntant dos de les estructures vistes esperant un augment del rendiment.

L'estructura que hem decidit ha sigut la unió de una taula de hash amb un arbre AVL per cada casella.



Cada casella correspon a una lletra del alfabetari, per tant, en la casella 0, només hi hauran les paraules que comencin amb la lletra A, en la segona només les paraules amb la lletra B, etc.

I dintre de cada casella hi haurà un arbre AVL que tindrà les paraules organitzades en forma d'arbre. Hem decidit aquesta estructura per el seu equilibri, es a dir:

- Trobar les paraules de manera ordenada és molt senzill, ja que només hem d'anar casella per casella i obtenir el InOrder del arbre, per lo tant a diferència de una taula de hash normal, la velocitat és molt major (cost  $O(n)$ ). Utilitzant de una taula normal de hash seria un cost  $O(n^n)$ .
- El cost d'inserció i cerca és sempre  $O(\log n)$  on  $n$  és el número de paraules que conte el arbre de la casella corresponent. Potser no és la manera més ràpida, però tenint en compte que amb la taula de hash podem tenir casos on el cost s'aproxima a  $O(n)$  ens assegurem un cost estàtic sempre.

- El factor de carrega no ens importarà ja que per molts elements que tinguem la restricció vindrà donada per el temps que tarda el AVL en fer la cerca o inserció no en la cerca mitjançant la funció de hash o altres.
- I com hem vist anteriorment, l'estructura AVL es més ràpida que una estructura hash a l'hora d'obtenir-ho per ordre d'aparicions. Per tant, tenint en cada posició un AVL fa que el cost torni a ser  $O(n^n)$  on  $n$  es el número de nodes a diferencia de  $O(k^k)$  on  $k$  es el número de caselles.

### Taula hash amb Binari

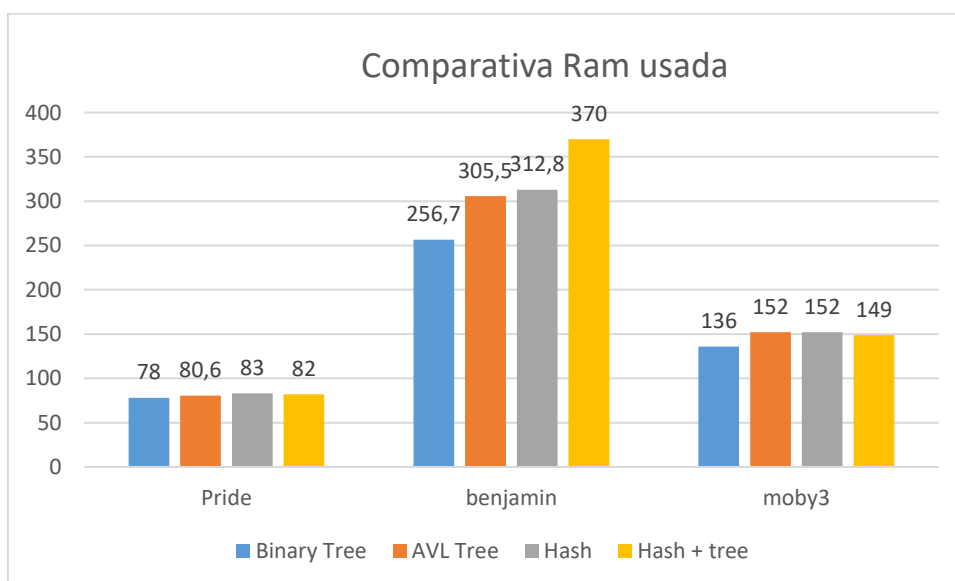
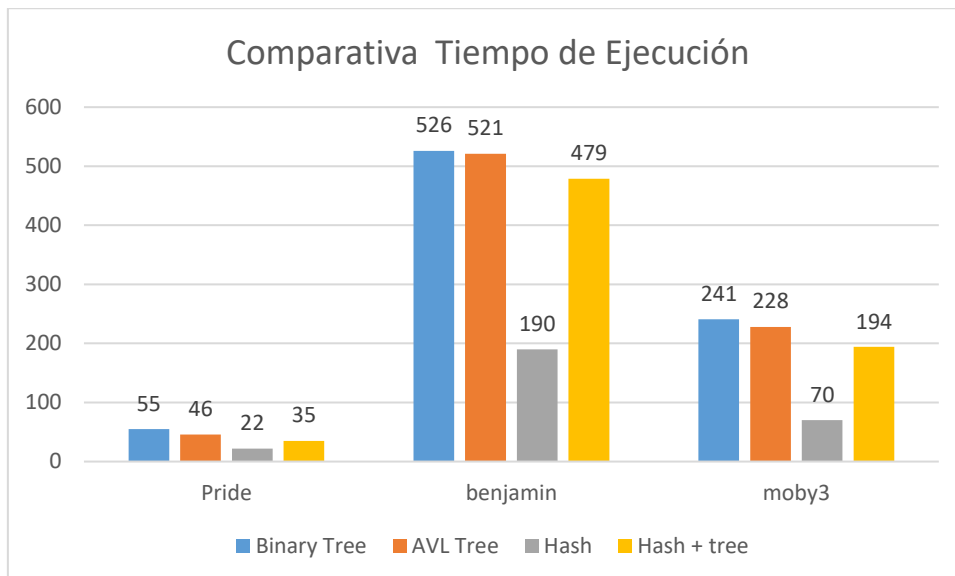
Un cop hem combinat la taula de hash amb el AVL, també hem provat combinar-ho amb el binari però com ja hem esmentat abans, la característica que ens aporta el AVL amb el binari es que el cost de les operacions sempre son  $O(\log n)$  que a diferencia del binari, que nomes en el millor dels casos el cost es  $O(\log n)$ .

Per tant, si hem d'escollir entre les dues estructures, la combinació Hash mes AVL dona millor resultat i per tant una millor elecció.

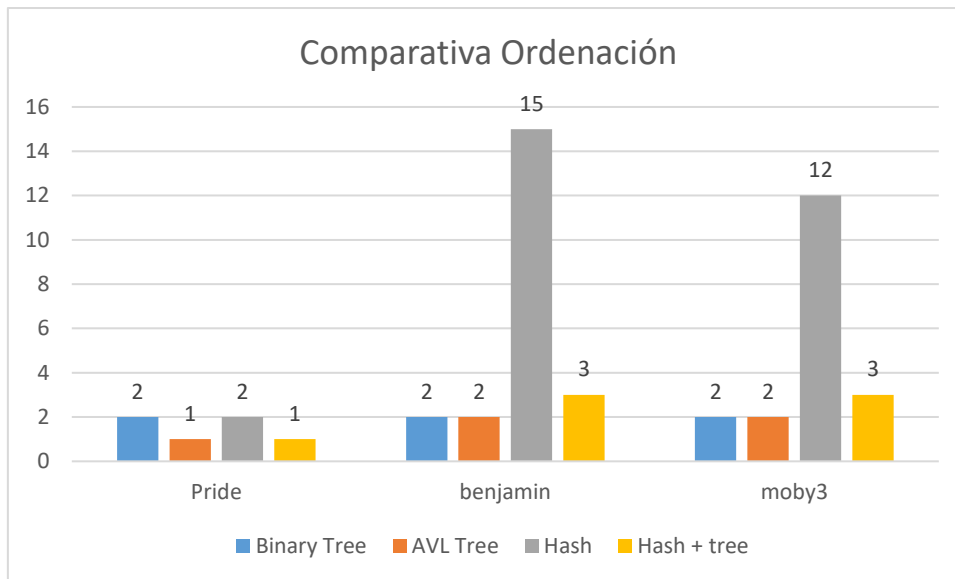


## Resultats

Durant l'execució del programa en diferents arxius (701kb, 11271kb, 3679kb) vam obtenir els següents resultats:

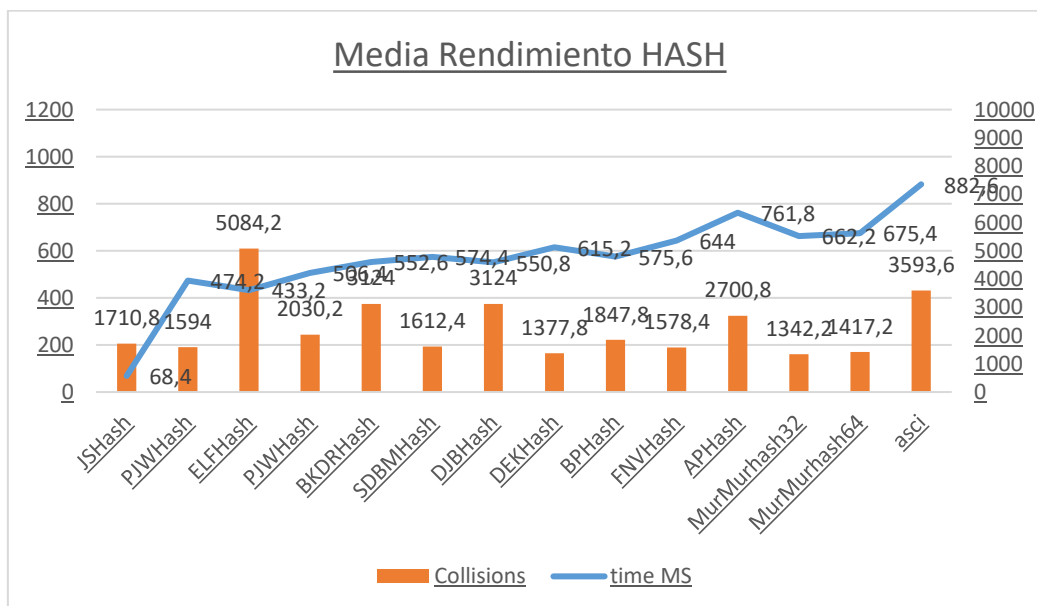


Com podem veure, el més ràpid es el Hash, això es degut a que tenim una array amb el nombre de paraules total, això ens fa gastar moltíssima ram, però tenim una busqueda molt rapida. En canvi amb la unió amb els arbres no fa que disminueixi el temps, ja que hi han poques col·lisions (JShash).



Una cosa que ens vam donar compte, que la funció hash tenia que ser de la major qualitat possible, si fem un promig entre els 3 documents que hem analitzat (de diferents tamany), ens trobem que amb una funció bàsica hem tingut un promig de 1166460670 col·lisions. Això significa que hem tingut que fer 1166460670 la funció rehash, cosa que fa que incrementi el temps d'execució fins a 51,6 segons de promig. Si fem una funció senzilla, però més lògica, com mirar el valor ascii, la cosa canvia molt. Les col·lisions baixen fins a 3593 (324557 vegades menys) y el temps es redueix a menys d'un segon.

Un cop analitzat aquest problema, vam cercar molts algorismes hash orientats a cadenes de caràcter (y no a la seguretat) y vam trobar una llista de algorismes, a continuació deixem la gràfica del anàlisi de les mateixes:



Com podem veure, la funció JSHash es la que millor ens funciona de manera absoluta amb uns temps de 68,4 ms y 1710 col·lisions de promig.

Adjuntarem els fitxers CSV y Excel que em obtingut en el desenvolupament de les proves.

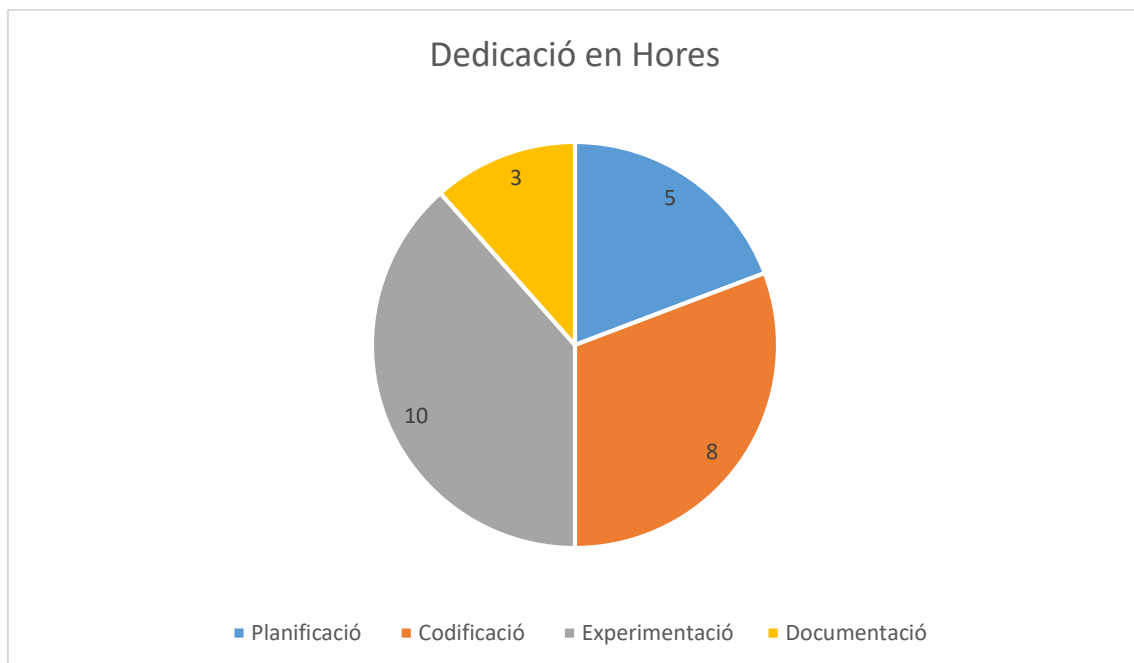
## Dedicació

Planificació i disseny: Crear/replantejar les diferents estructures de dades que ens semblàvem més correctes per el desenvolupament de la practica. Hi vam dedicar unes 5 hores.

Codificació de les estructures: Un cop teníem ja la idea clara, ens vam posar a picar el codi, dels quals podríem dir que vam tardar 8 hores, ja que vam haver de implementar tot el que havíem pensat, i després comprovar que fos correcte.

Experimentació: Aquesta part ha sigut la que mes ens ha costat a nivell de horari, ja que em experimentat amb varies funcions hash y en molts documents, així com combinació d'estructures. Hi vam dedicar unes 10 hores

Documentació: Finalment la creació d'aquest document i dels comentaris, ha sigut de unes 3h.



## Conclusions

Durant la execució d'aquesta practica em assolit tots els coneixents teòrics d'aquest últim semestre. També ens ha servir per donar-nos compte, que la programació vista fins ara, pot no ser eficient en un mon real.

Per això em donat la possibilitat de executar els fitxers amb el mode default, com hem fem sempre, una simple llista, apart de les 4 estructures que teníem que realitzar.

Evidentment el mode default era el pitjor, però són interessants alguns detalls que em observat, com per exemple, la importància de la funció hash en una taula de hash.

Després de fer aquesta practica, ens ha quedat clar on i perquè implementar cada tipus de estructura, per exemple la taula hash, es molt bona per accés directe, però no per recorre-la sencera (te molts forats) també te un gran impacte en la memòria del equip.

Una de les estructures que ens han sorprès a sigut el arbre avl, ja que es senzill de implementar però dona un bona relació rendiment/consum en la majoria de situacions.

En el nostre intent de combinar Hash amb avl, em tingut un pitjor rendiment, suposem que gracies a una funció hash com la JShash, no tenim tantes col·lisions i els arbres llavors, no eren tant necessaris com amb una funció mes bàsica.

## Bibliografía

[En línea] <http://griho2.udl.es/josepma/eines/hash/hash1.html>.

**Sole, Xavi.** Apunts. *Estudy*.