BEN EL HADJ Ali
20901154

Devoir n°1

I. Introduction :

J'ai choisie d'implémenter les structures suivantes :

-Tableau de mots non trié car j'ai trouvé intéressant de comparer les autres structures que j'ai implémenté avec une structure naïf.
-Arbre dictionnaire.
-ABR.
-AVL.

Comme vous pouvez le constater, les trois dernières structures sont des arbres, on a jugé plus cohérent et logique de comparer des structures qui ont le même type (arbre) mais qui utilisent des méthodes de stockage et de recherche différentes,
Le langage choisie est JAVA (c'est un choix sans raison !!).

L'implémentation du Tableau et du ABR était sans difficultés. Mais pour AVL, j'ai pas réussi à trouver la méthode d'équilibrage, donc je me suis servi d'internet pour avoir l'idée.
L'implémentation Arbre dictionnaire était la plus difficile.

Pour avoir le temps d'exécution, on a utilisé la méthode suivant :

```
long start = System.nanoTime();
//instructions
long duree = System.nanoTime() - start;;
System.out.println(duree/1000000000.0 +" secondes");
```

Les programmes sont lancés dans un terminal (java programme fichier) …
Type processeur de la machine utilisée pour les tests : Mobile DualCore Intel Pentium, 2200MHz.
Logiciel de traitement des données : LibreOffice calc.

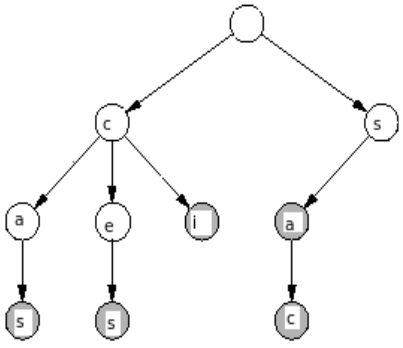
II. Description brève des implémentations :

1.Tableau :

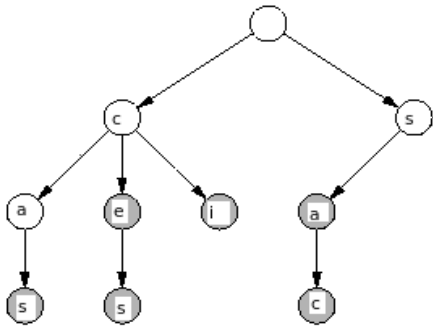
Pour insérer une chaîne dans le tableau, on alloue un nouveau tableau plus grand que le tableau initial d'une case et qui contiendra tout le tableau initial plus la chaîne à rajouter dans la dernière case.
Pour chercher une chaîne dans le tableau, on parcoure le tableau du début a la fin, si la chaîne est trouvée, la requête envoie une réponse et ne parcoure pas le reste du tableau.

2.Arbre dictionnaire :

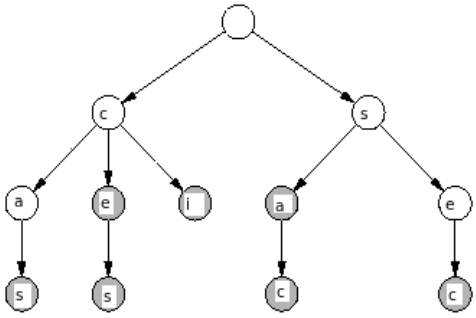
La définition d'Arbre dictionnaire a été donnée dans le devoir.
Pour l'insertion, voici un exemple :
Ici l'arbre initiale contient les mots 'cas', 'ces', 'ci', 'sa' et 'sac" (les nœuds gris signifie que le mot est present) :



On insère le mot 'ce' :



Puis j'insère le mot 'sec' :



Pour la recherche d'une chaîne, partant de la racine, on regarde si la première lettre de la chaîne est un des fils directe de la racine, si oui on continue récursivement jusqu'à arriver à la dernière lettre de la chaîne, une fois arrivé a la dernière lettre de la chaîne trouvé dans l'arbre, on vérifie si elle est coloriée en gris, si oui on retourne 'true', sinon 'false', sinon 'false'.

3.ABR :

Voir cours algorithmique L3.

4.AVL :

Voir chapitre 13 dans le livre Algorithmique de Cormen.

III. Occupation mémoire :

1.Tableau :

Pas de meilleur ni pire cas pour cette structure. On suppose que l'espace moyen occupé par une ligne (une case du tableau) est égale a L, donc la complexité en espace est égale à [somme de (L x i)] pour 1=<i<=N] et avec N le nombre de ligne du fichier.

2.Arbre dictionnaire :

```
class ArbreDictionnaire{
        public ArbreDictionnaire[] noeuds;
        public boolean bool;
}
```

-Cas meilleur : est quand tout les mots (ou la plupart) de l'arbre sont des préfixes d'autres mots dans l'arbre et que tout les nœuds sont coloriés en gris (marquer par la valeur 'true').
Un exemple avec les mots 'a', 'al', 'ali' et 'alik'.
La complexité en espace est donc égale à [N x (taille du tableau + un boolean)] avec N est le nombre de lignes du fichier sans doublons.
Remarque : la taille du tableau dans mon code est de 10241 pour pouvoir lire un grand nombre de caractères .

-Cas pire quand :

- Aucun mot de l'arbre n'est préfixe d'un autre mot de l'arbre.
- La taille des mots est très grande !!
- On a que les feuilles qui sont coloriées en gris (marqué a 'true ').

Un exemple avec les mots :'se', 'cal', 'tar' et 'a'.
Ici il est difficile de donner une complexité en espace dans le cas pire car elle dépend de la taille des mots...
On va donc supposer que la moyenne de la taille des lignes est égale a L, et dans ce cas on peut dire que la complexité en espace dans le pire cas est égale à
[L x N x (taille du tableau + un boolean)] avec N est le nombre de lignes du fichier sans doublons.

-Cas moyen : la complexité en espace dans le cas moyen est donc de
[(N x (taille tableau + un boolean)) x (L + 1) x (1/2)].

3.ABR :
Pas de meilleur cas ni de pire cas pour la complexité en espace d'un ABR, elle est égale à
[N x L] avec N le nombre de lignes dans le fichier sans doublons et L la moyenne de l'espace occupé par une ligne du fichier.

4.AVL :
La complexité en espace dans AVL est la même que celle d'un ABR.

Conclusion :

	Tableau	Arbre dictionnaire	ABR	AVL
Complexité moyenne en espace	somme de (L x i) pour 1<=i<=n	(N x (taille tableau + un boolean)) x (L + 1) x (1/2)	N x L	N x L

Avec :
-N = nombre de ligne du fichier sans doublons. (car on insère pas les doublons)
-n = nombre de ligne du fichier avec doublons.
-L = moyenne de l'espace occupé par une ligne du fichier dans la structure.
On constate donc que AVL et ABR sont les plus économiques, et que Arbre dictionnaire est le plus coûteux si la taille du tableau dans sa structure est très grande .

IV. Des temps d'exécution :

1.Temps de création de la structure :

Table 1 :
Temps de création des structures en seconde pour un petit fichier de taille 19 (le fichier 'dic')

structure	tableau simple	Arbre Dictionnaire	ABR	AVL
une permutation aleatoire des lignes du fichier	0,00028792	0,00327724	0,00035708	0,00039196
lignes du fichier triées par ordre decroissant	0,00025108	0,00338468	0,000399519	0,000372159
lignes du fichier tirées pas ordre croissant	0,000246839	0,00330884	0,00035948	0,00040624

Ce tableau nous montre que la structure naïf (Tableau simple) est plus rapide que les autres structures d'arbre quand il s'agit d'un petit fichier de 19 lignes.
Les structures d'arbre perdent donc du temps en comparant l'élément à rajouter avec quelques éléments déjà introduits dans l'arbre pour savoir à quelle place le mettre. Tandis que le Tableau alloue un nouveau tableau, met dedans l'ancien tableau et le nouvel élément sans perdre beaucoup de temps puisque le tableau est très petit.

Table 2 :

Temps de création des structures en seconde pour un fichier de taille 35728 (le fichier 'descartes.txt')

structure	tableau simple	Arbre Dictionnaire	ABR	AVL
une permutation aleatoire des lignes du fichier	3,10218228	2,134988479	0,10385156	0,187446443
lignes du fichier triées par ordre decroissant	3,103815999	2,0795864	0,61823912	0,16305372
lignes du fichier tirées pas ordre croissant	3,11241216	2,146861001	0,37548576	0,13342072

On remarque ici que pour un gros fichier, le Tableau simple est beaucoup plus lent que les structures d'arbre.
On constate aussi que la structure Arbre dictionnaire n'est pas efficace car quand elle insère un mot, si ce mot n'a pas dans l'arbre un autre mot qui a le même préfixe que lui, il va donc perdre du temps pour allouer un nouveau tableau et initialiser la valeur booleanne à false (voir fonction inserer dans la classe ArbreDictionnaire).
Quand les lignes du fichier sont permutées aléatoirement, on constate que ABR est un peu plus rapide que AVL car AVL équilibre son arbre à chaque insertion d'élément (chose que ABR ne fait pas).
Le pire cas pour un ABR est quand les lignes du fichier sont triées (voir cours Algo L3), on constate que ABR perd deux fois plus de temps quand le fichier est trié par ordre décroissant que quand il est trié par ordre croissant, et ça s'explique quand on revoit la fonction inserer de ABR, cette fonction teste si l'élément à ajouter est plus grand que la racine, si ce teste échoue, elle teste donc s'il est plus petit. Donc quand le fichier est trié par ordre décroissant, on fait deux teste avant d'appeler inserer récursivement, mais quand il est croissant, on ne fait qu'un teste avant de rappelé la fonction inserer..
Enfin on remarque aussi que quand le fichier est trié, la structure de AVL est beaucoup plus efficace que ABR parce que son arbre reste toujours équilibré (ou presque).

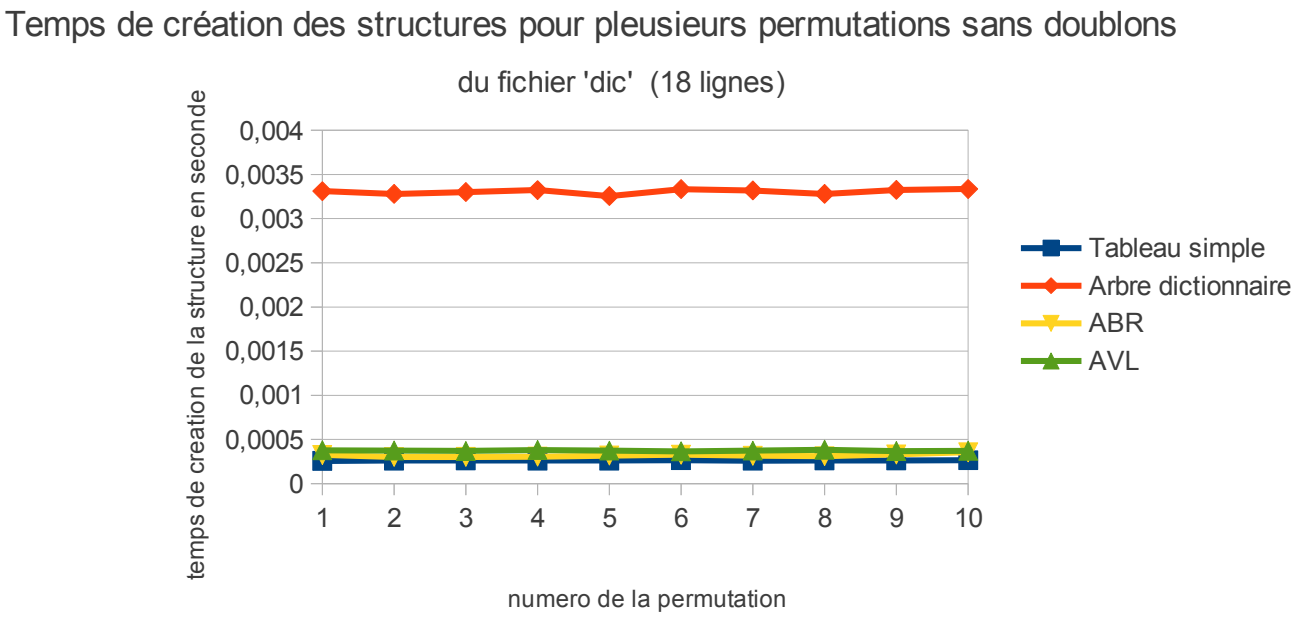
Table3 :

Temps de création des structures en seconde pour un fichier de taille 16029745 (le fichier 'flickr-comments-30.txt')

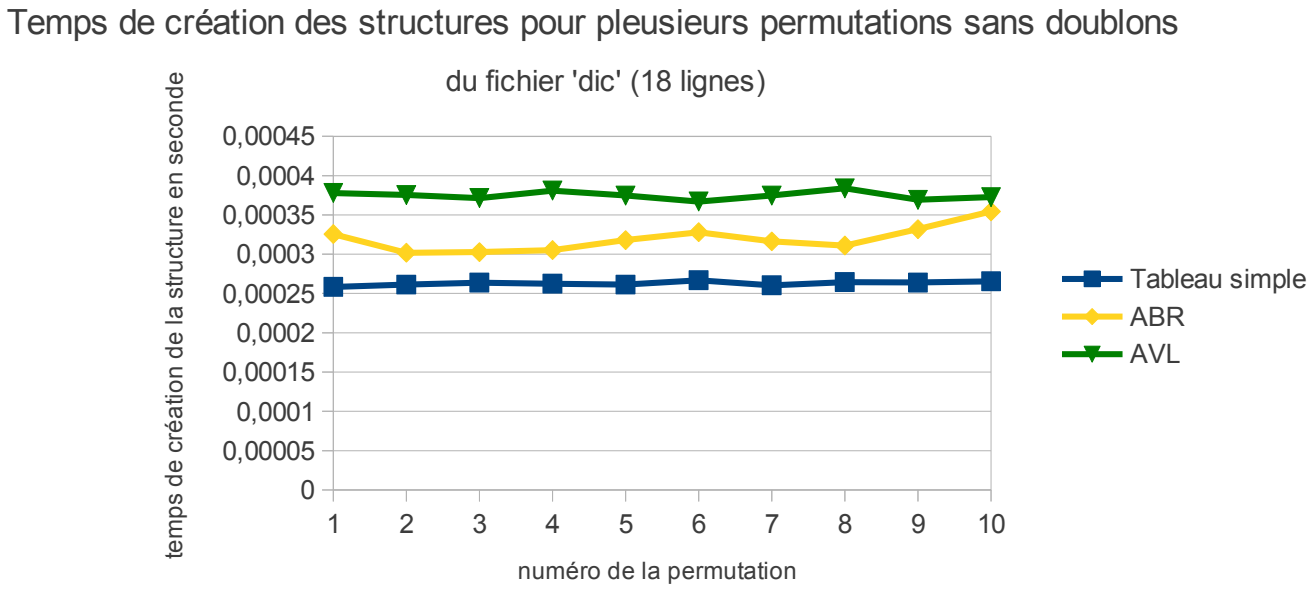
structure	tableau simple	Arbre Dictionnaire	ABR	AVL
une permutation aleatoire des lignes du fichier	????	????	????	106,48580276
lignes du fichier triées par ordre decroissant	????	????	????	64,069334564
lignes du fichier tirées pas ordre croissant	????	????	????	29,821151681

Pour un gros fichier, je constate que AVL est le seul à avoir fini sa création, ce qui prouve a mon avis que la structure AVL est plus efficace que les trois autres pour un fichier très gros.

Graphe 1 et 2 :



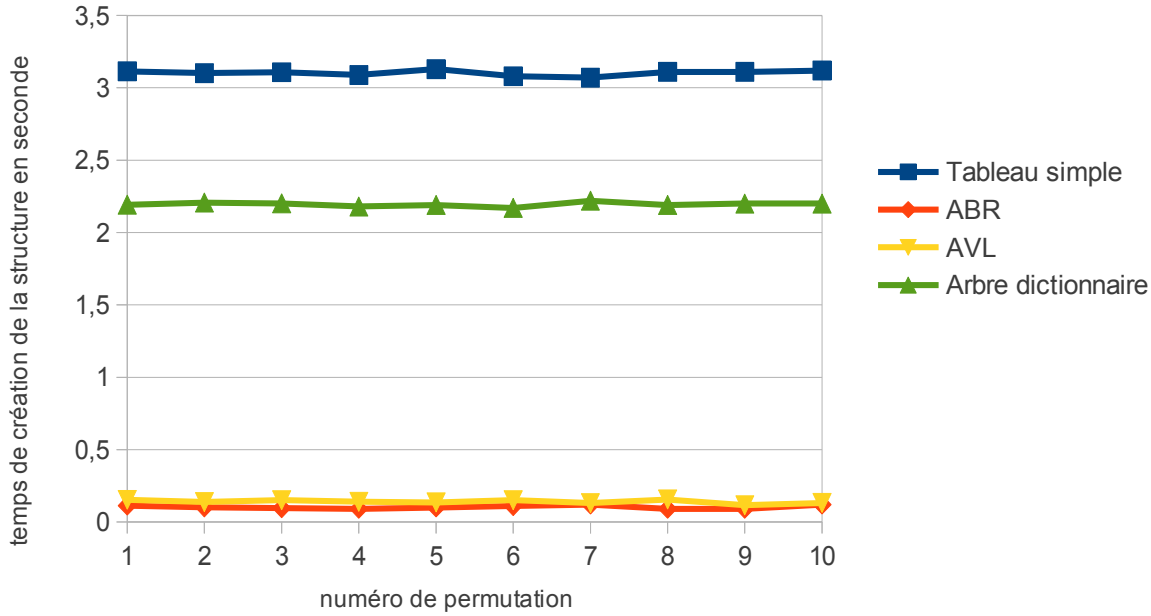
Un Zoom pour les courbes d'en-bas



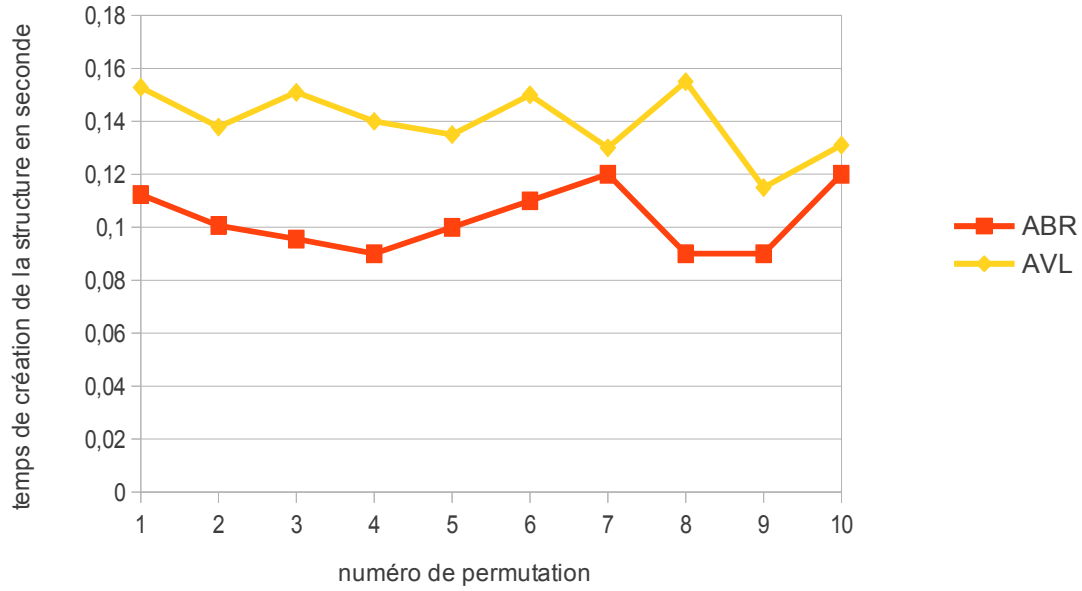
Ces courbes nous confirment une partie des remarques que nous avons tiré de la table 1.

Graphe 3 et 4 :

temps de création des structures en fonction de plusieurs permutations du fichier 'descartes.txt' (35728 lignes)



Zoom pour les les courbes d'en-bas



On remarque aussi sur les courbes que pour n très grand, Tableau est inefficace et que la courbe Arbre dictionnaire passe dessous la courbe Tableau mais il reste loin d'être efficace aussi.
Dans le Zoom on voit que ABR et AVL sont très proche et nous avons expliqué cela dans la partie des tables.

2.Temps de recherche et d'insertion :

- Tableau :

Mot	*'zyzomys'*	*'zyzomys'* x2 (= *'zyzomyszyzomys'*)	*'zyzomys'* x10	*'zyzomys'* x20
Temps d'insertion en seconde	0,000536	0,00014248	0,0001406	0,00015084
Temps de recherche en seconde	0,00047964	0,00028902	0,00044296	0,00052804

D'après la table, l'insertion et la recherche dans un Tableau ne dépendent pas de la taille de la chaîne de caractère.
L'insertion dépend de la taille du tableau, plus il est grand plus l'insertion devient lente, donc la complexité en temps de la fonction insérer de Tableau est de O(n) avec n la langueur du tableau .
La recherche ne dépend que de l'endroit de l'élément à chercher dans le Tableau, donc le pire cas est quand l'élément à chercher se situe à la fin du Tableau, et le meilleur cas est quand l'élément à chercher se situe a la première case du Tableau. Donc on peut dire que ça complexité en temps est de O(n) aussi.
Pour la recherche d'un mot absent dans le tableau, la recherche s'arrête que quand on arrive à la dernière case du tableau .

- Arbre dictionnaire :

Mot	*'zyzomys'*	*'zyzomys'* x2 (= *'zyzomyszyzomys'*)	*'zyzomys'* x10	*'zyzomys'* x20
Temps d'insertion en seconde	0,00007768	0,000124	0,00104344	0,0014826
Temps de recherche en seconde	0,00000422	0,00000788	0,00002952	0,00004712

Pour Arbre dictionnaire, l'insertion et la recherche dans l'arbre dépendent de la taille de la chaîne de caractères.
On constate d'après le tableau que chercher un mot deux fois plus long prend deux fois plus de temps.
Idem pour l'insertion, sauf si le mot à insérer a déjà un autre mot dans l'arbre qui possède un même préfixe que lui, dans ce cas là, l'insertion va gagner un peu de temps (voir explication de Table 2).
La complexité en temps de la recherche et de l'insertion dans Arbre dictionnaire est de O(taille_du_mot). On remarque qu'elle ne dépend pas de la taille du fichier.
Pour la recherche d'un mot absent dans l'arbre, la recherche s'arrête à son plus gros préfixe dans l'abre.

- ABR :

Mot	*'zyzomys'*	*'zyzomys'* x2 (= *'zyzomyszyzomys'*)	*'zyzomys'* x10	*'zyzomys'* x20
Temps d'insertion en seconde	0,00001544	0,00000412	0,0000106	0,00000312
Temps de recherche en seconde	0,00001772	0,00000756	0,00000728	0,00000772

D'après la table, l'insertion et la recherche dans ABR ne dépendent pas de la taille de la chaîne de caractère, mais elles dépendent de la taille de l'arbre.
La complexité en temps de l'insertion et de la recherche dans le pire cas est de O(n) et cela quand le fichier est trié.
La complexité en temps de l'insertion et de la recherche dans le cas moyen pour ABR est de O(log(n))...
Pour la recherche d'un mot absent dans l'arbre, la recherche s'arrête que quand on arrive à une feuille.

- AVL

Mot	*'zyzomys'*	*'zyzomys'* x2 (= *'zyzomyszyzomys'*)	*'zyzomys'* x10	*'zyzomys'* x20
Temps d'insertion en seconde	0,0000158	0,00001236	0,00000644	0,00000648
Temps de recherche en seconde	0,00001852	0,0000072	0,00000588	0,00000624

D'après la table, l'insertion et la recherche dans AVL ne dépendent pas de la taille de la chaîne de caractère, mais elles dépendent de la taille de l'arbre.
La complexité en temps de l'insertion et de la recherche pour AVL est de O(log(n)).
Pour la recherche d'un mot absent dans l'arbre, la recherche s'arrête que quand on arrive à une feuille.

V. Remarques :

Arbre dictionnaire est efficace pour la recherche mais prend beaucoup de temps pour créer sa structures (donc pour l'insertion aussi) et surtout gaspille beaucoup trop de mémoire.
L'insertion dans un ABR est en moyenne un peu plus rapide que dans un AVL mais la recherche dans AVL et plus rapide que dans ABR.