Rapport de projet

Recensement des mots d'un corpus

Analyse des structures de données

Florent HENRY
<florent.henry@etudiant.univ-nancy2.fr>

Mathieu CANZERINI
<mathieu.canzerini@etudiant.univ-nancy2.fr>

Médéric HURIER
<mederic.hurier@etudiant.univ-nancy2.fr>

Omar EDDASSER
<omar.eddasser@etudiant.univ-nancy2.fr>

Quentin BURTSCHELL
<quentin.burtschell@etudiant.univ-nancy2.fr>

Année 2011 - 2012

Table des matières

1. Contexte du projet

1.1. Conception

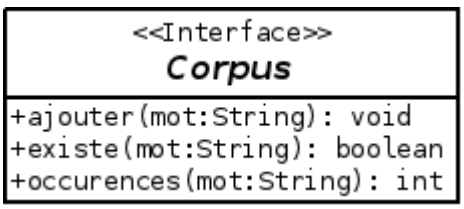
Le but du projet est d'étudier les performances théoriques et pratiques de plusieurs algorithmes et structures de données, dans l'objectif de recenser les mots d'un corpus textuel ainsi que la fréquence de chacun.

Les opérations nécessaires pour l'implémentation des structures sont:

- recherche de l'existence d'un mot
- recherche du nombre d'occurrences d'un mot
- insertion d'un nouveau mot

Il n'est pas nécessaire de stocker un mot plusieurs fois. On peut simplement incrémenter un compteur pour obtenir la fréquence d'utilisation. Nous avons fait ce choix pour limiter la taille en mémoire des structures et rendre nos tests plus faciles et rapides.

L'interface de programmation des structures de corpus est la suivante:



Pour réaliser nos tests une fonction de lecture est chargée de lire un mot dans le fichier de données pour l'insérer dans le corpus.
Dans notre projet, un mot défini comme une suite de caractère unicode, pouvant comprendre des chiffres, des majuscules, des caractères accentués ou non occidentaux et des signes de ponctuation. Nous avons considéré les espaces, tabulations et sauts de lignes commet étant des séparateurs de mots.

Pour ne pas influencer les tests de performance, un programme "Cleaner.java" permet de réécrire le fichier en une suite de mot, séparée par des sauts de ligne.
Après nettoyage, le fichier comprend 226 492 647 mots (dont 1 961 101 différents) pour 1 250 218 278 caractères soit une longueur moyenne des mots de 5.5 caractères.
La taille d'un fichier ne contenant qu'une fois chaque mot du corpus est de 20 Mo.

1.2. Procédure de test

Dans les environnements d'exécution actuels, les performances d'un programme sont influencées par plusieurs facteurs:

- Caractéristiques techniques de la machine (CPU / RAM, OS)
- Système multi-thread, concurrence des processus
- Vitesse des Entrée/Sortie
- Spécificités du langage Java (JVM, Garbage Collector, Version du langage)
- Expérience du programmeur de l'application

Afin de réduire ces effets de bord, nous avons spécifié une procédure de test dont les paramètres sont:

- Répéter le test pour faire une moyenne (5 par défaut)
- Préférer la mesure du temps CPU plutôt que du temps réel (horloge de bureau) qui dépend fortement de l'environnement extérieur au programme
- Avant chaque mesure, lancer manuellement le Garbage Collector (System.gc)
- Charger le fichier de test en RAM pour les machines qui le permettent
- Utiliser la version 7 de Java, dont l'algorithme de gestion de la mémoire est plus performant (voir cet article : http://blog.xebia.fr/2008/03/12/gc-generationnels-traditionnels-jdk6-vs-gc-garbage-first-jdk7/)
- Lancer les tests sur plusieurs machines

Les résultats de nos tests sont présentées dans la section 3 de ce document.

La configuration minimale pour lancer le programme est de 6GO de RAM, sans lancer d'autres programmes.

1.3. Utilisation du programme

Notre programme de test complet se trouve dans le répertoire "Tests" du projet. Il s'utilise de la façon suivante:

```
Usage: java TestComplet <fichier en entrée> <nombre de mesure>
Exemple: java TestComplet dump.txt 3
```

Le résultat de cette commande est la génération d'un fichier au format CSV.
Si les arguments ne sont pas précisés, le fichier par défaut sera "fichier.txt" et le nombre de mesure fixé à 5.

Pour générer le fichier de test, nous utilisons le fichier Cleaner.java de cette façon :

```
Usage: java Cleaner <fichier en entrée> <fichier en sortie>
Exemple: java Cleaner dump.txt fichier.txt
```

Les résultats bruts de nos exécutions se trouvent dans le dossier "Tests/résultats".

2. Structures de données

2.1. Table de hachage

2.1.1. Structure

La table de hachage est une structure de données qui permet une association clé-élément. On accède à chaque élément de la table via sa clé. Il s'agit d'un tableau ne comportant pas d'ordre (un tableau est indexé par des entiers). L'accès à un élément se fait en transformant la clé en une valeur de hachage (ou simplement hachage) par l'intermédiaire d'une fonction de hachage.

Détail des attributs de la classe :

```
Vector valeurs          // Tableau de taille variable contenant les enregistrements (Un
                            // enregistrement est une liste chaînée qui contient le mot dans
                            // une chaîne de caractère et // le nombre d'occurences de ce mot)

boolean java            // Booléen indiquant si on utilise la fonction de hachage de java
                        // ou celle redéfinie
```

2.1.2. Insertion

L'insertion d'une valeur dans une table de hachage comprend plusieurs étapes (du hachage à l'ajout) :

```
fonction ajouter(mot)
début
    indice <- hachage(mot)
    si valeurs.taille <= indice alors // si on dépasse la taille du tableau
        valeurs.taille <- indice + 1
    fsi
    si valeurs.get(indice) = null alors // si le champ est vide
        valeurs.set(indice,mot)         // on ajoute le mot
    sinon si valeurs.get(indice).contient(mot) alors // si le mot existe
            mot.occurence ++ // on incrémente l'occurence
        sinon
            valeur.get(indice).ajouter(mot) // sinon on l'ajoute dans la liste
        fsi
    fsi
fin
```

2.1.3. Recherche

```
fonction rechercher(mot)
début
    indice <- hachage(mot)
    si valeurs.get(indice) = null alors // si le champ est vide
        retourne faux         // le mot n'existe pas
 sinon si valeurs.get(indice).contient(mot) alors // si le mot existe dans la liste
            retourne vrai
        sinon
            retourne faux
        fsi
    fsi
fin
```

2.1.4. Analyse

Les paramètres ayant de l'influence sont :

- la choix de la fonction de hachage, qui influe sur la répartition des données et la vitesse de calcul de l'index
- la méthode d'accroissement de la table, qui peut ajouter un nombre fixe de case ou multiplier la taille de la structure. On peut aussi préciser une valeur fixe, et gérer les collisions.

Complexité pour un n-ième élément dans le cas le pire :

- Insertion : O(1)
- Recherche : O(1)
- La réorganisation a une influence difficile à mesurer car elle dépend du nombre de valeurs déjà insérées. Si l'on choisit de doubler la table, on doit d'abord la recopier dans une nouvelle structure. Cette étape est en O(n), où n est la taille de la structure lors de la recopie.

2.2. Arbre binaire non équilibré

2.2.1. Structure

Les Arbres binaires non équilibrés permettent d'utiliser une structure récursive. Ainsi, les sous arbres droit et gauche sont également des arbres binaires de recherche non équilibrés.

Détail des attributs de la classe :

String val	// Valeur stockée dans la racine
	// Mis à jour dans le cas d'une feuille ou directement intialisé à la
	// construction du noeud.

int occurrences	// Nombre d'occurrences de la valeur
	// Intialisé à la création du noeud puis ensuite mis à jour à chaque insertion
	// (cas où la valeur existe déjà)

ArbreBinaire fg, fd	// fils gauche et fils droit de l'Arbre Binaire
	// Ils peuvent être null ou de type Arbre Binaire, dépendant si la
	// racine (le parent) est une feuille ou non

2.2.2. Insertion

L'insertion d'une valeur dans un arbre binaire de recherche se fait de façon récursive, en appelant la fonction d'insertion au niveau des fils du noeud parent courant

```
fonction ajouter(mot)
début
    diff <- difference entre mot et arbreb
    si diff < 0 alors // si le mot à inserer est plus grand que le noeud courant
        si arbreb.fd != null alors
            arbreb.fd.ajouter(mot) // Début de la récursivité
        sinon
            arbreb.fd = arbreb(mot) // création du noeud
        fsi
     sinon si deff > 0 alors // si le mot à insérer est plus petit que le noeud courant
        si arbreb.fg != null alors
            arbreb.fg.ajouter(mot) // Début de la récursivité
        sinon
            arbreb.fg = arbreb(mot) // création du noeud
        fsi
    sinon // si le mot à insérer est égal au noeud courant
        arbreb.occurrences ++
    fsi
fin

Légende :
ajouter : <fonction> fonction qui permet d'ajouter un mot à l'arbre
mot : <chaine> mot à ajouter dans l'arbre
arbreb : <arbre> noeud courant dans l'arbre
<arbre> : type Arbre Binaire décrit précédemment. Les attributs utilisés par cette fonction sont vide, val, occurrences, fg, fd
```

2.2.3. Recherche

```
fonction rechercher(mot) : entier
début
```

```
    si arbreb.vide alors // Fin de la récursivité si le mot n'est pas trouvé
        retourner 0
    fsi

    si arbreb.val = mot alors
        retourner arbreb.occurrences
    sinon
         si arbreb.val < mot alors // Recherche récursive sur le sous arbre gauche
ou droit
            retourner arbreb.fg.rechercher(mot)
        sinon
            retourner arbreb.fd.rechercher(mot)
        fsi
    fsi
fin

Légende :
rechercher : <fonction : entier> fonction de recherche qui retourne le nombre
d'occurrences du mot ou 0 s'il n'existe pas dans l'arbre
mot : <chaine> valeur recherchée dans l'arbre
arbreb : <arbre> noeud courant dans l'arbre
<arbre> : type ArbreBinaire décrit précédemment. Les attributs utilisés par cette
fonction sont vide, val, occurrences, fg, fd
```

2.2.4. Analyse

L'ordonnancement des mots en entrée aura un impact direct sur le temps de calcul. Par exemple, une entrée déjà triée aura pour conséquence d'avoir des insertions uniquement dans les noeuds de droite. De cette manière, on construit l'équivalent d'une liste.

Complexité pour un n-ième élément dans le cas le pire :

- Insertion : O(n)
- Recherche : O(n)
- Dans le cas le pire, les valeurs sont toujours sur la même branche de l'arbre (uniquement à gauche ou droite). L'arbre binaire se comporte alors comme une liste chaînée.

2.3. AVL

2.3.1. Structure

Les AVL permettent d'utiliser une structure récursive. Ainsi, on peut voir les sous arbres gauche et droit d'un AVL comme étant eux même des AVL.

Détail des attributs de la classe :

short h	// Hauteur de l'arbre
	// Mis à jour sur tous les noeuds de la branche parcourue pour accéder au
	// noeud concerné par l'ajout uniquement si un noeud est créé.

short deseq	// Différence des hauteurs des sous arbres gauche et droit // Mis à jour lors de l'insertion d'une nouvelle valeur uniquement.

String val	// Valeur stockée dans la racine
	// Mis à jour dans le cas d'une feuille ou directement intialisé à la
	// construction du noeud.

int occurrences	// Nombre d'occurrences de la valeur
	// Intialisé à la création du noeud puis ensuite mis à jour à chaque insertion
	// (cas où la valeur existe déjà)

boolean vide	// false s'il s'agit d'une feuille
	// Cet attribut permet d'éviter de faire des opérations booléennes complexes
	// lors des parcours de l'arbre (affichage, recherche, insertion)

AVL fg, fd	// fils gauche et fils droit de l'AVL
	// Ils peuvent être null ou de type AVL, dépendant si la racine (le parent)
	// est une feuille ou non

2.3.2. Insertion

A l'inverse de la recherche, cet algorithme n'est pas récursif. Il utilise des boucles.
La structure AVL, pour rester équilibrée, utilise 4 rotations : rotation Gauche, DroiteGauche, Droite, GaucheDroite. Ces rotations dépendent de la valeur du déséquilibre du noeud, ainsi que la comparaison par rapport au fils droit ou gauche qui détermine si la rotation est simple ou double.

Algorithme :

```
fonction ajouter(mot)
début
    pos <- avl
    prec <- null
    tant que non pos.vide faire
        si pos.deseq != 0 alors
            prec <- pos
        fsi
        si mot = pos.val alors
                        // Si la valeur existe, on augmente le nombre d'occurrences
et on termine là
            // fonction
            pos.occurrences = pos.occurrences + 1
            fin
        sinon
```

```
            si mot < pos.val alors
                pos = pos.fg
            sinon
                pos <- pos.fd
            fsi
        fsi
    ftant
    // Reste de l'algorithme
    // Sinon on ajoute un noeud pour sauvegarder la nouvelle valeur dans l'arbre
      // On recalcule les hauteurs depuis le dernier déséquilibre différent de 0 (prec)
    // Si le déséquilibre de prec est inférieur à -1 ou supérieur à 1 alors
    // on rééquilibre l'arbre par une rotation appropriée
     // (dépend de la valeur du déséquilibre et de la valeur de la racine droite ou gauche
    // par rapport au mot à insérer)

fin

Légende :
ajouter : <fonction> fonction qui permet d'ajouter un mot à l'arbre
mot : <chaine> mot à ajouter dans l'arbre
avl : <arbre> racine de l'arbre
pos : <arbre> noeud courant dans l'arbre
prec : <arbre> dernier noeud dont le déséquilibre est différent de 0
<arbre> : type AVL décrit précédemment. Les attributs utilisés par cette fonction sont vide, val, occurrences, fg, fd
```

Les calculs importants ne sont effectués que si nécessaire (calcul de la hauteur, rotations).

2.3.3. Recherche

La structure de l'algorithme de recherche est récursive. Celle-ci est permise grâce à la structure récursive des sous arbres qui sont eux mêmes des AVL.

Algorithme :

```
fonction rechercher(mot) : entier
début
    si avl.vide alors // Fin de la récursivité si le mot n'est pas trouvé
        retourner 0
    fsi

    si avl.val = mot alors
        retourner avl.occurrences
    sinon
         si avl.val < mot alors // Recherche récursive sur le sous arbre gauche ou droit
            retourner avl.fg.rechercher(mot)
        sinon
            retourner avl.fd.rechercher(mot)
        fsi
```

```
    fsi
fin

Légende :
rechercher : <fonction : entier> fonction de recherche qui retourne le nombre d'occurrences du mot ou 0 s'il n'existe pas dans l'arbre
mot : <chaine> valeur recherchée dans l'arbre
avl : <arbre> noeud courant dans l'arbre
<arbre> : type AVL décrit précédemment. Les attributs utilisés par cette fonction sont vide, val, occurrences, fg, fd
```

L'algorithme s'arrête après avoir parcouru toute la hauteur de l'arbre ou s'il a trouvé la valeur.

2.3.4. Analyse

Comme pour les arbres binaires, l'ordonnancement des mots en entrée a une influence directe sur les performances en terme de temps de calcul. Néanmoins, l'influence sur les performances ne concerne que l'insertion.

Complexité pour un n-ième élément dans le cas le pire :

- Insertion : O(log(n))
- Recherche : O(log(n))
- Un AVL étant équilibré, la recherche et l'insertion sont toujours en log(n)

2.4. Structure originale

2.4.1. Structure

Structure interne d'un nœud de la structure originale :

NŒUD :
- **table <Caractère, Nœud >** : Il s'agit d'une table de hachage qui pour un caractère associe un nœud fils
- **occurrence** : Il s'agit d'un entier qui représente le nombre d'occurrence du mot composé des caractères allant de la racine jusqu'au nœud courant

La structure originale étant en fait un pointeur vers le nœud racine, les fonctions d'ajout, de recherche, … s'effectuent sur ce nœud racine qui par récursivité les exécute sur ses enfants.

2.4.2. Insertion

Algorithme de la fonction d'ajout d'un mot dans la structure originale:

```
Fonction : ajouter(String mot) :

SI mot.longeur > 0 ALORS
              // on récupère la première lettre du mot
              c = mot.charAt(0);
              // on supprime la première lettre du mot
              mot = mot.substring(1);
              // on récupère le noeud dans la table correspondant à la première
lettre du mot
              nœud = table.get(c);
              // si le noeud n'existe pas, on le crée et on l'ajoute à la table de
hachage
              SI nœud EST null
                     noeud = creerNoeud()
                     table.put(c,n);
              FINSI
              // on effectue un appel récursif sur le noeud fils avec le mot
(dépourvu de sa première lettre)
              nœud.ajouter(mot);
SINON
              // le mot n'a plus de lettre : on se situe dans le bon noeud :
              // on incrémente donc le nombre d'occurrence du nœud courant
              occurrence += 1
FINSI
```

2.4.3. Recherche

Algorithme de la fonction recherche dans la structure originale:

```
Fonction : rechercheOccurrence(String mot) :
SI mot.longeur > 0 ALORS
              // on récupère la première lettre du mot
              c = mot.charAt(0);
              // on supprime la première lettre du mot
              mot = mot.substring(1);
              // on récupère le noeud dans la table correspondant à la première
lettre du mot
              nœud = table.get(c);
              SI nœud EST null
                     //  si le noeud n'existe pas, il n'y a pas d'occurrence du mot
                     RETOURNER 0
              SINON
                     // on effectue un appel récursif sur le noeud fils avec le mot
```

```
(dépourvu de sa première lettre)
                    RETOURNER nœud.rechercheOccurrence(mot)
            FINSI
SINON
            // le mot n'a plus de lettre : on se situe dans le bon noeud :
            // on retourne donc le nombre d'occurrence du nœud courant
            RETOURNER occurrence
FINSI
```

L'algorithme de recherche (de nombre d'occurrence d'un mot) est en fait quasiment identique à celui de l'ajout :

- au lieu de créer un nœud dans le cas où il n'existe pas dans la table, on retourne 0 (en effet, s'il n'existe pas de nœud pour un caractère du mot dans l'arbre, c'est que le mot n'est pas présent)
- et au lieu d'incrémenter le nombre d'occurrence du nœud courant, on retourne ce nombre

2.4.4. Analyse

Les paramètres qui vont avoir un impact sur les performances en temps de calcul et en coût en mémoire sont :

- L'alphabet autorisé : avec un petit alphabet pour les mots, il est plus optimal (à la fois en temps de calcul qu'en coût en mémoire) d'utiliser une liste chainée qu'une table de hachage au niveau des nœuds (les listes des nœuds de niveau inférieur – proche des feuilles – seraient plus petite que celles des nœuds proche de la racine).
- La longueur moyenne des mots.

Complexité pour un n-ième élément de longueur "m" dans le cas le pire :

- Insertion : O(m)
- Recherche : O(m)
- L'ajout et la recherche dans la structure originale se faisant lettre par lettre (ou noeud par noeud), la complexité dépend de la longueur du mot. La structure utilisée par les noeuds étant une table de hachage, les opérations de base sont en O(1). Ainsi, on obtient une recherche et une insertion en O(m)

3. Analyse des résultats

3.1. Comparaison des performances pour la création du corpus

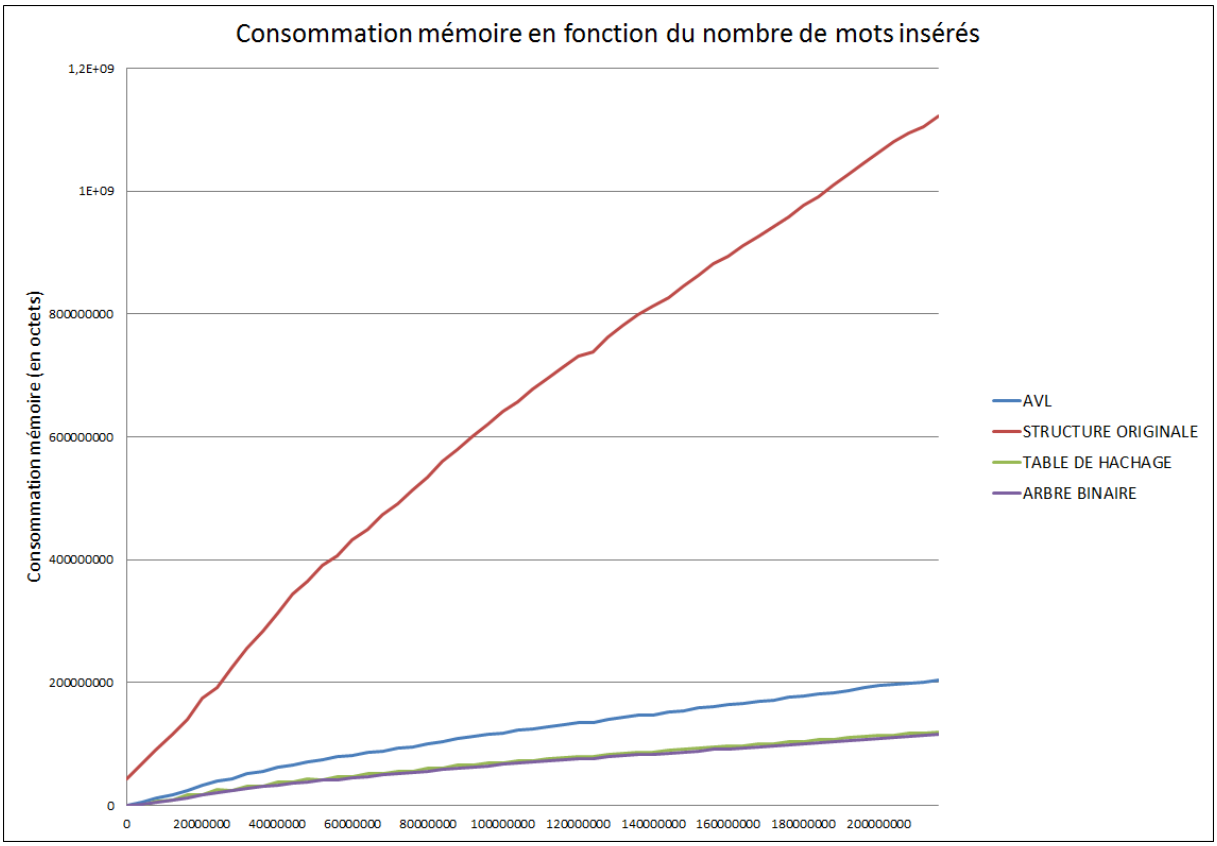
	Arbre Binaire	AVL	Structure Originale	Table de hachage
Temps de calcul	1min 30s	1min 59s	1min 3s	23min 4s
Coût mémoire	108 Mo	192 Mo	1 053 Mo	112 Mo

La comparaison du temps de calcul montre une grande différence entre la table de hachage et les autres structures. Malgré une complexité algorithmique en O(1) pour les opérations de base, c'est la structure qui met le plus de temps à créer le corpus en 23 minutes et 4 secondes.
Les autres structures ont des temps de calcul semblable mais la structure la plus rapide est au final la structure originale avec un temps de traitement CPU de 1 minute et 3 secondes.

Au niveau du coût en mémoire, la structure originale a le revers d'être très gourmande avec 1053 Mo de mémoire vive consommée. Sa représentation est quasiment dix fois plus élevée que celle de la table de hachage (112 Mo), et plus de cinq fois celle de l'AVL (200 Mo).
Sur ce paramètre, on voit que les arbres binaires sont les plus performants, avec une consommation mémoire de 108 Mo. La table de hachage a cependant un coût en mémoire très proche.

En considérant que le corpus contient plus de 226 492 647 mots (dont 1 961 101 mots différents) et que la taille du fichier ne contenant que des mots différents fait 20 Mo, on peut considérer qu'un temps de calcul de l'ordre de 1 à 2 minutes et une consommation mémoire de 100 à 200 Mo sont satisfaisantes.

3.2. Coût en mémoire en fonction de la taille du fichier



Ce graphique illustre l'état de la mémoire en fonction du nombre de mots déjà insérés. Comme le résume de la tableau de la partie 3.1, la structure originale est beaucoup plus gourmande en mémoire que les autres structures, qui ont des accroissement semblables. L'AVL est légèrement plus gourmand que l'arbre binaire et que la table de hachage.

Le coût en mémoire de la structure originale s'explique par notre choix de conception pour la structure interne des noeuds. Nous avons privilégié la table de hachage Java pour stocker les caractères, en raison de ses performances pour un nombre d'entrée limité (alphabet unicode).
Les tables de hachage sont performantes en temps de calcul, mais leurs répartitions sont inégales et donnent des taux de remplissage variables. Ces espaces vides ont donc un coût très important en mémoire.

Si on compare la table de hachage que nous avons implémenté pour le corpus, son accroissement est plus faible que la structure originale. En effet, le nombre important d'entrées et la diversité des mots donnent un taux de remplissage plus important. La mémoire est donc davantage rentabilisée.

Concernant les arbres, il n'est pas étonnant que l'AVL soit plus gourmand que l'arbre binaire. De par son fonctionnement, l'AVL requiert plus d'attributs que l'arbre binaire. Cette différence explique l'écart de coût en mémoire.

En résumé, les résultats du graphique semblent cohérents avec nos estimations théoriques. Le rapport entre la structure originale et les autres structures dépasse nos attentes en proportion.

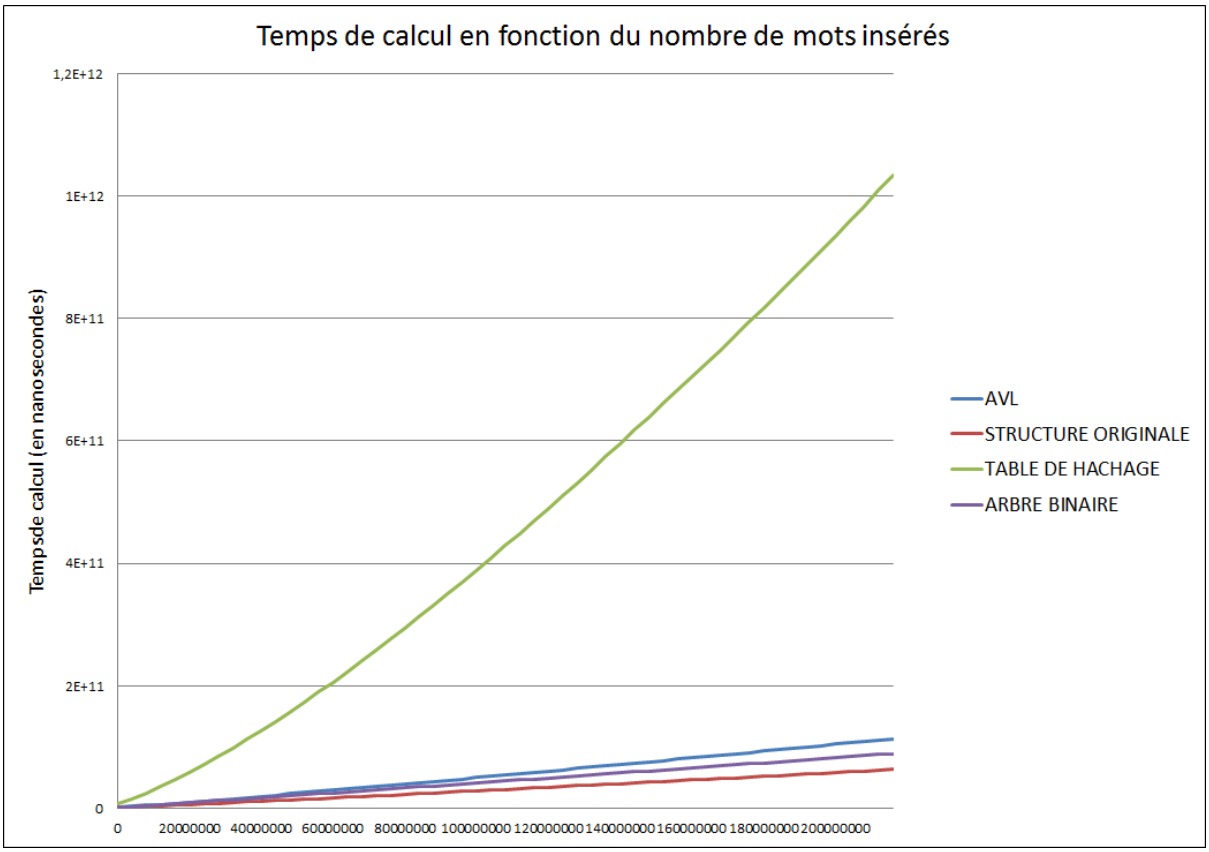
3.3. Coût d'appel à la fonction d'insertion en fonction de la taille de l'entrée

Les opérations nécessaires pour insérer un nouveau mot sont très différentes selon la structure employée, mais des principes de fonctionnement peuvent être dégagés.

Pour la table de hachage, l'opération est en O(1) et ne nécessite pas de recherche. Toutefois, ces deux opérations peuvent être assimilés car elles sont toutes deux en O(1), et la différence entre l'insertion d'un nouveau mot et le simple test de l'existence d'un mot est négligable.
L'insertion dans un arbres binaire ou AVL nécessite au préalable une recherche pour localiser le noeud. Une fois la recherche effectuée, l'insertion est en O(c), où c est une constante.
Le coût d'appel à la fonction d'insertion est presque uniquement lié à celui de la recherche, en O(log n) pour les AVL et en O(n) pour les arbres binaires dans le cas le pire.
Enfin, la structure originale a un mode de fonctionnement similaire. Une recherche est nécessaire pour parcourir les lettres du mots, et créer les noeuds quand ceux ci n'existent pas déjà. La création d'un nouveau noeud ou le parcours d'un noeud est en O(1).
Ainsi, on en conclut que le coût d'appel à la fonction d'insertion dépend principalement du coût d'appel à la fonction de recherche. Comme nous l'avons étudié dans le cas général, il faut parcourir ou rechercher dans la structure avant d'ajouter une nouvelle valeur.

Même si l'on peut s'attendre à ce que le coût de l'insertion dépend de la taille de la structure, on voit que tous les algorithmes d'insertion requièrent une recherche au préalable, et que l'insertion en elle même est en O(c). Ce n'est donc pas l'existence d'un mot qui conditionne le temps de calcul, mais bien la taille du fichier d'entrée (en fonction de n).

graphique sur la page suivante



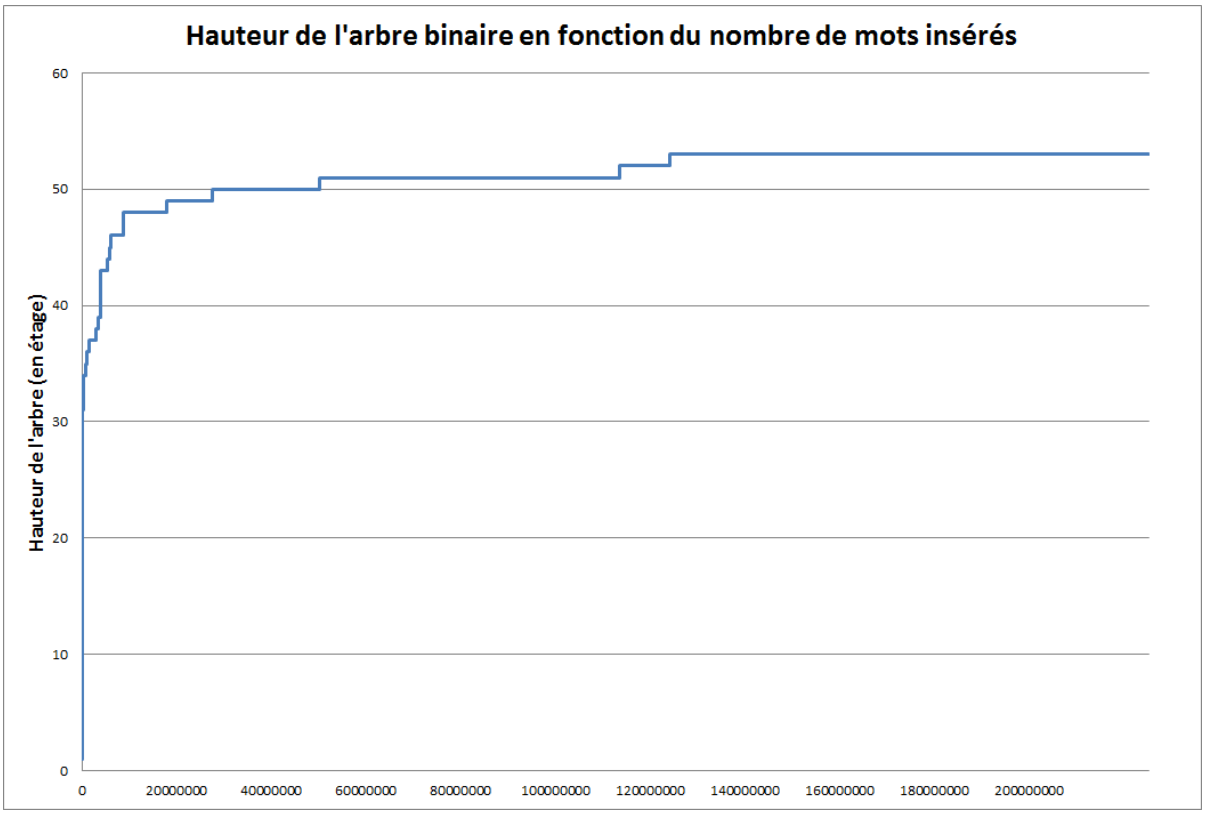
Le graphique ci-dessus montre le temps de calcul des structures en fonction du nombre de mots déjà insérés.

L'analyse des courbes montre que les réorganisations massives compromettent sérieusement les performances des tables de hachage. Chaque recopie étant en O(n), le temps de calcul est sérieusement dégradé.

La structure originale est ici la plus performante. En effet, chacune des opérations dépend de la taille du mot en entrée, et non de la taille de la structure. Avec une longueur moyenne de mot de 5.5 caractères, c'est une méthode très efficace pour insérer et recherche de nouveaux mots avec ce jeu de donnée.

Enfin, les arbres binaires et AVL ont un temps de calcul semblable pour le corpus. Même si la complexité algorithmique dans le cas le pire est plus grande pour l'arbre binaire, on voit que cette structure est plus rapide que l'AVL pour ce jeu de donnée. Cela illustre la différence entre cas pratique, et cas théorique.

3.4. Performances des arbres



On observe sur le graphique que la hauteur effective de l'arbre binaire est de 53 pour le corpus. Nous avons mesuré que les AVL ont une hauteur de 26 pour le même jeu de donnée. On a donc un facteur multiplicatif de deux entre les structures.
En considérant le cas le pire pour l'arbre binaire, on pourrait avoir une hauteur d'arbre de 226 492 647 mots. On se retrouverait alors avec les performances d'une liste chaînée. Une hauteur de 53 est donc un très bon résultat !

D'après nos tests sur le temps de calcul, on voit que ce déséquilibre n'a pas un impact fort sur les performances. Même si les arbres binaires ont des hauteurs fortement variables selon le jeu de données, ils n'en restent pas moins plus simple dans leur gestion que les AVL, qui nécessitent des réorganisations fréquentes. La différence donnant l'avantage aux arbres binaires ne s'explique pas par la hauteur de l'arbre, mais par la simplicité des structures.

4. Conclusion

4.1. Estimations et expérimentations

L'approche théorique et empirique de l'analyse des algorithmes sont deux méthodes distinctes.
Dans le premier cas, il faut déterminer par la logique une expression mathématique en fonction de l'entrée. Cette méthode a l'avantage d'être assez rapide, mais pas toujours simple. Bien souvent, il est impossible d'arriver à un résultat précis en $\Theta(n)$ voir en $O(n)$.
Au contraire, l'approche empirique permet d'obtenir un résultat dans tous les cas, mais la rédaction et l'exécution des tests est longue et fastidieuse. De plus, il faut avoir une méthode rigoureuse pour mesurer la bonne donnée.

Lors de la réalisation du projet, nous avons constaté plusieurs incohérences entre les deux méthodes. Bien souvent, ces écarts sont du à l'omission des constantes dans la partie théorique. La hauteur des arbres binaires étant seulement deux fois plus grande que celle des AVL, le nombre de calculs moyens pour une insertion est augmenté par 2. Pour le jeu de tests donné, la constante des AVL est supérieure à celles des arbres binaires. Ce comportement n'est pas naturel par rapport à l'étude théorique.
Dans tous les cas, il faut se montrer critique envers les résultats obtenus. Un bon résultat pratique dépend largement des conditions du test, dont les caractéristiques du jeu d'entrée. Nous avons vu ici que l'arbre binaire est plus performant que l'AVL, mais cela aurait put être très différent avec un jeu trié de données.

Pour un concepteur de système, ces deux approches sont toutefois complémentaires. Une bonne analyse de la complexité permet d'éviter des développements coûteux en ressource, mais ne remplace pas totalement l'étude d'un cas concret.
Un esprit critique est indispensable dans les deux cas, et permet d'éviter que le cas le pire ne se produise.

4.2. Forces et faiblesses des structures

Nous avons étudié dans la section 2 les choix de conception ayant un impact sur les performances en terme de mémoire et de temps de calcul. Ces paramètres permettent de formuler des hypothèses concernant les cas où ces structures seront pleinement efficaces.

Dans le cadre général, aucune structure n'est meilleur qu'une autre !

Le sujet citait en exemple l'utilisation du mode T9 pour les téléphones mobiles. Pour ces environnements où les ressources sont limitées et où les réponses doivent être rapides, l'AVL semble être le meilleur choix. Il est très indépendant du jeu de données initial, et a un bon rapport entre temps de calcul, et coût en mémoire. L'AVL est un excellent compromis entre le temps d'insertion, la

recherche et la consommation mémoire.
Si la machine a des ressources très importantes, comme c'est le cas pour un super-calculateur, alors la structure originale est un bon choix, à condition que la longueur moyenne des mots soit raisonnable. La consommation mémoire sera plus élevée avec les choix de conception que nous avons fait, mais les performances seront meilleurs.
Même si les tables de hachage ont montré des performances décevantes lors de la création du corpus, elles n'en restent pas moins très puissantes pour un système orienté vers la recherche d'information plus que pour l'insertion. Elle peut être utilisée dans des bases de données où les informations évoluent peu.
Enfin, l'arbre binaire offre l'avantage d'être très simple à implémenter. Cela limite les gestions, telles que les réorganisations, mais cela fait dépendre fortement la structure du jeu de données en entrée. La simplicité est l'avantage ainsi que le désavantage de cette structure.

L'important à retenir est qu'il n'existe aucune structure gagnante. Tout dépend des hypothèses qui se rajoutent au problème de base : définition des entrées (triées, aléatoires...), but de la structure (stockage ou recherche), utilisation du programme...

4.3. Difficultés rencontrées

Nos principales discussions ont porté sur les choix de conception concernant les structures de données. Même si le choix des structures était imposé, une grande liberté nous était laissée pour l'implémentation.
Il n'est pas facile de s'orienter vers une solution plutôt que vers une autre. Dans le cadre de ce projet, nous avons privilégié les compromis en essayant tout de même de maximiser les performances. C'est la voie du juste milieu.
Néanmoins, nous avons pour chaque structure dû faire plusieurs essais afin de trouver la meilleure version.

Notre autre grande difficulté a été de comparer les résultats théoriques et empiriques. La complexité est souvent assez facile à déterminer dans le cas le pire, mais il est impossible de formuler des hypothèses à partir d'un cas précis. Ces différences ont été remarquées par notre analyse empirique.

Enfin, la rédaction de la procédure de test a été une partie très délicate. Java n'offre pas un standard pour estimer les performances globales d'une structure, et chaque développeur est souvent contraint d'implémenter sa propre procédure. Des outils de profiling existent, mais ne permettent pas de tester les comportements asymptotiques des fonctions.
A l'aide de plusieurs forums et documentations, nous avons put écrire une méthode la plus stricte et indépendante possible de l'environnement. Nous n'avons cependant pu réaliser ces tests que sur une machine très performante.