RAPPORT DE PROJET
APMAN

Vérificateur orthographique pour documents au format “.txt”

Loïc Falgayrac - Thomas DIJS

SOMMAIRE

I. Présentation du sujet:

L'objectif de ce projet est de trouver un moyen idéal de stocker un dictionnaire et de l'utiliser pour trouver, parmi les mots composant un texte, ceux qui ne sont pas présents dans le dictionnaire. Nous avons pour cela un dictionnaire au format ".txt" contenant 323 295 mots. Le roman à étudier est *A la recherche du Temps Perdu*, de Marcel Proust qui détient le titre du roman le plus long de la langue française (1,5 million de mots). Pour cela, nous avons utilisé 3 méthodes de stockage du dictionnaire différentes pour les comparer : une liste chaînée simple, une table de hachage et un arbre.

II. Lecture du dictionnaire:

Le dictionnaire étant sous format ".txt", sa lecture se fait simplement avec la fonction fgets(). Les mots sont stockés dans des chaînes de caractères qui sont ensuite traitées différemment en fonction des méthodes d'implémentation.

Pour la liste et la table de hachage, on se contente de remplacer le *'\n'* présent à la fin de chaque mot (en dehors de "anticonstitutionnellement") par un *'\0'* qui marque la fin de la chaîne. Pour l'arbre, on doit d'abord récupérer la chaîne puis l'insérer (si cela est nécessaire) caractère par caractère dans l'arbre.

III. Stockage du dictionnaire:

1) Implémentation 1, liste chaînée:

Nous avons créé une simple liste chaînée constituée ou chaque maillon stocke un char de taille 30 (la taille maximum d'un mot étant 26). On remarque que l'exécution de la vérification orthographique sur le livre prend environ 5 minutes, ce qui est très long comparé aux autres méthodes utilisées. La libération mémoire pour cette implémentation se fait simplement maillon par maillon. Aucun problème particulier n'a été rencontré pour cette implémentation.

2) Implémentation 2, table de hachage:

La table de hachage est l'implémentation la plus rapide pour notre cas de figure. Nous avons choisi comme fonction de hachage la formule suivante :

$$hash(clé) = \sum_{i=0}^{strlen(clé)-1} clé[i] * b^{i} \% M$$

- strlen(clé) est une fonction qui renvoie la longueur de la chaîne de caractère "clé"
- b un nombre premier (b=13 dans notre cas)
- M la longueur de la table de hachage

Cette fonction de hachage permet un faible nombre de collisions pour un tableau assez grand (de l'ordre de M = 100 000). Nous avons directement créé un tableau de taille fixe sans gérer le redimensionnement. C'est un tableau de listes chaînées, ainsi pour

chaque valeur de hash différente nous avons une liste chaînée des mots ayant ce hash. Le tableau étant de taille 100 000 (pour 300 000 mots) et la fonction de hachage étant relativement efficace, les collisions sont assez bien réparties sur l'ensemble de la table.

Cette implémentation est la plus rapide des trois dans notre cas de figure. La libération de la mémoire se fait pour chaque case du tableau : liste par liste, chaque liste chaînée est libérée de la même manière que la liste utilisée pour l'implémentation 1.

Nous avons perdu du temps sur cette implémentation à cause des valeurs renvoyées par la fonction de hachage. Au début elle renvoyait un simple "int" signé, les valeurs calculées par la fonction de hachage pouvant atteindre le milliard et le type "int" pouvant seulement stocker des valeurs entre -32 768 et 32 767, nous nous retrouvions avec des valeurs de hash négatives. Ce problème renvoie un "Segmentation fault" car nous essayons d'accéder à un rang de la table inexistante. Nous avons réglé ce problème en retournant un "unsigned long" plutôt qu'un "int".

3) Implémentation 3, arbre préfixe:

La méthode arbre est censée être la plus optimisée mais pas nécessairement la plus rapide. En effet, l'arbre permet théoriquement de stocker 26 puissance n mots de longueur n et de les retrouver pour un coût maximum de 26*n, c'est néanmoins la plus difficile à implémenter.

La méthode de l'arbre était pour nous une découverte, nous n'avions jamais fait cela auparavant (en dehors des arbres binaires). La technique que nous avons utilisée est assez simple, et elle est basée sur le principe *" frères / fils "*.

Deux lettres sont *frères* lorsque toutes les lettres situées avant dans le mot sont identiques et au même nombre, mais que les deux lettres en question sont différentes. Par exemple, le u de *loup* et le n de *long* sont frères dans l'arbre. Le *fils* est quant à lui logiquement la lettre qui suit dans le mot.

La fonction d'insertion de frères et de fils est la même et vérifie automatiquement que l'élément en question n'est pas en double (dans ce cas l'arbre perdrait tout son sens). Un nœud ne peut avoir au maximum qu'un seul frère et qu'un seul fils reliés directement à lui, en revanche, il peut avoir plusieurs frères par l'intermédiaire de son frère ou plusieurs fils par l'intermédiaire de son fils.

Pour repérer la fin du mot, nous avons ajouté à chaque nœud un entier *fin_de_mot* qui est à *'0'* à chaque mot sauf si le chemin créé correspond à un mot, auquel cas on met *'1'*. Cela permet par exemple de conserver le mot *"art"* sans que le mot *"artiste"* vienne l'éliminer. Nous avions cette idée avant même de commencer l'arbre.

Nous avons fait en sorte que l'arbre prenne le moins de taille possible. Pour cela, nous avons rajouté des "frères" que lorsque cela était nécessaire (par exemple, la lettre 'y' du début de l'arborescence n'a que 9 fils (a, e, i, o, p, s, t, u, v) et non 26. Cette méthode est assez peu efficace pour le début des mots mais très efficace pour la fin. Ainsi, nous n'avons que 945 000 allocations pour l'arbre.

En réalisant un valgrind, on voit qu'il n'y a aucune erreur de gestion de mémoire dans l'arbre. Néanmoins, nous n'avons pas réussi à le libérer (problèmes de *"stack overflow"* dans le valgrind, il s'agit d'un problème d'accès mémoire). L'utilisation du GDB ne nous a pas non plus aidé pour la libération. La seule fonction de libération que nous avons trouvée ne fonctionne que pour de petits arbres (environ 5000 allocations).

IV. Lecture du livre:

Le livre étant sous format “.txt”, sa lecture se fait avec la fonction fscanf(). Avec cette fonction nous parcourons le livre par agrégat de mot, elle stocke l’agrégat dans une chaine de caractère de taille 30 que nous traitons ensuite.

Sur chaque agrégat nous convertissons toutes les majuscules en minuscule avec la fonction tolower(). Nous remplaçons ensuite tout caractère qui n’est pas une lettre minuscule par un espace. (Exemple : "Peut-être." devient “peut être “).

Nous utilisons ensuite la fonction strtok() dans une boucle qui a chaque itération renvoie une suite de lettres délimitées par un séparateur que nous rentrons en paramètre, dans notre cas nous choisissons l’espace(Exemple : avec “peut être" nous aurons accès à la première itération à “peut” et à la 2eme itération nous aurons accès à "être". C’est donc dans cette boucle que nous faisons la vérification orthographique avec des fonctions de comparaison propres à chaque implémentation si chaque mot est oui ou non dans le dictionnaire.

V. Tests de rapidité:

Les temps calculés ici sont simplement significatifs, ils sont relatifs à l’ordinateur avec lequel nous avons fait nos tests. Les temps peuvent donc différer d’un ordinateur à un autre. Les temps d’exécution ont été calculés avec la bibliothèque <time.h>.

Temps d’exécution avec la table de hache pour différentes tailles de la table :

	M = 1000	M = 10 000	M = 100 000
Avec doublons	3.455 secondes	0.572 secondes	0.331 secondes
Sans doublons	3.568 secondes	0.682 secondes	0.446 secondes

Nous constatons que le temps d’exécution est plus rapide lorsque la table de hachage est plus grande. Cela se justifie car le nombre de collisions est plus faible. Avec un dictionnaire de 300 000 mots, si on raisonne en moyenne, nous avons 300 collisions par hash avec M = 1000 contre 3 collisions par hash avec M = 100 000. La recherche dans une liste chaînée se fait naturellement beaucoup plus rapidement lorsqu’elle est de taille 3 plutôt que de taille 300. Nous avons essayé avec des valeurs plus grandes que M = 100 000, il est possible d’aller plus vite mais le temps gagné est très faible devant le temps d’exécution.

Temps d’exécution pour chaque implémentation :

	liste chaînée	arbre préfixé	table de hachage (M = 100 000)
Avec doublons	> 5 minutes	0.488 secondes	0.331 secondes
Sans doublons	> 5 minutes	0.587 secondes	0.446 secondes

Nous constatons que la table de hachage est la méthode la plus rapide, suivie de la méthode de l'arbre préfixé, suivie de la méthode par liste chaînée. La méthode par liste chaînée est nettement plus longue que les deux autres, ce résultat était attendu car cette méthode n'est absolument pas optimisée. Nous avons trouvé des temps de l'ordre de 5 à 10 minutes. Ces temps ne sont pas exploitables, c'est pour cette raison qu'ils ne figurent pas dans le tableau.

Si on raisonne toujours en moyenne, pour M = 100 000 nous aurons 3 mots différents par hash, ainsi une fois le hash d'un mot calculé, seulement 3 comparaison maximum sont nécessaire pour savoir si il est dans le dictionnaire, tandis que pour l'arbre il faut parcourir des fils/frères ce qui est plus long. Nous trouvons donc des valeurs qui correspondent à nos attentes.

Les valeurs de temps des exécutions sans doublons sont plus grandes qu'avec doublons car nous stockons les mots qui ne sont pas dans le dictionnaire dans une liste chaînée. Ainsi lorsqu'un mot n'est pas dans le dictionnaire, nous devons vérifier qu'il n'est pas non plus déjà dans la liste chaînée.

VI. Conclusion :

Ce projet a été une expérience enrichissante, il nous a permis d'être plus à l'aise avec les notions abordées (tables de hachage et arbres). Les résultats en termes de temps correspondent à nos attentes, les trois implémentations sont fonctionnelles, seule la libération mémoire de l'arbre manque à notre projet. Enfin, ce projet à souligné l'importance de trouver des structures adéquates (arbres, tables de hachage) pour les utilisations qu'on veut en faire, et même si les ordinateurs sont déjà très puissants (lecture de 1.5 M de mots en 0.5 seconde), une mauvaise gestion des programmes peut rendre sa tâche bien plus compliquée.