rapport du TP : vérificateur orthographique

Baptiste GUYOT Gabriel LEVY

6 décembre 2021

Table des matières

1 Introduction **2**

2 1re méthode : arbre préfixe **2**
2.1 Structure utilisée 2
2.2 avantages et inconvénients de la structure, choix effectués . . . 3

3 2e méthode : arbre radix **4**
3.1 Structure utilisée 4
3.2 avantages et inconvénients de la structure, choix effectués . . . 4

4 3e méthode : la table de hachage **5**
4.1 Structure utilisée 5

5 Comparaison des résultats **6**
5.1 Les méthodes utilisées 6
5.2 Comparaison 7

1 Introduction

Depuis notre arrivée à PHELMA, nous apprenons à manipuler le langage C pour la programmation. Nos professeurs nous ont demandé de faire à la maison, un TP sur un vérificateur orthographique. Ce TP a pour but de programmer un vérificateur orthographique après avoir construit un dictionnaire à partir d'un fichier.

Structure du dossier

Le rendu est un dossier composé de notre travail. Ce dossier est contient 5 fichiers :

-un README qui explique comment compiler.

-un makefile qui sert à compiler.

-un rapport.pdf qui est ce rapport

-FR.txt qui est le dictionnaire utilisé.

-a_la_recherche_du_temps_perdu.txt qui est la texte utilisé pour vérifier que le dictionnaire marche bien.

Le dossier contient également trois autres dossiers :

-src contenant lui aussi trois dossiers. Chaque dossier contient le code d'une méthode utilisé.

-test qui est le dossier contenant les fichiers une fois que ceux-ci sont compilés.

-include contenant toutes les entêtes.

2 1re méthode : arbre préfixe

Un arbre préfixe est une structure arborescente où chaque nœud de l'arbre possède des fils qui sont d'autres nœuds. Chaque nœud servant à désigner une lettre, on peut parcourir l'arbre de lettres en lettres.

2.1 Structure utilisée

Un nœud est une structure composé de :

-lettre : un char qui est la lettre représenté par le noeud.

-fin : un char valant 0 et 1 pour savoir si ce noeud représente une fin de mot ou non.

-suivant : un tableau de 26 noeuds representant toutes les lettres de l'alphabet. Le tableau est classé par ordre alphabetique.

Le tout premier nœud de la structure est un nœud ne contenant pas de lettre. Il contient juste le tableau suivant.

2.2 avantages et inconvénients de la structure, choix effectués

Un arbre préfixe classique est différent du notre : il n'y a pas "suivant". A la place, il contient deux pointeurs sur des nœuds : fils et frère. Fils pointe sur un nœud contenant une des lettres suivantes (pour continuer le mot) alors que frère pointe sur sur un noeud contenant une des lettres ayant le même pere que le nœud sur lequel on se situe.

Notre structure présente un gros défaut par rapport à un arbre préfixe classique : elle prend beaucoup de place en mémoire puisque chaque noeud contient un tableau de 26 pointeurs.

```
==458863== HEAP SUMMARY:
==458863==     in use at exit: 0 bytes in 0 blocks
==458863==   total heap usage: 1,237,093 allocs, 1,237,093 frees, 143,512,368 bytes allocated
==458863==
==458863== All heap blocks were freed -- no leaks are possible
```

memoire occupée par la fonction créant l'arbre préfixe

Neanmoins, ce défaut est à nuancer puisqu'il permet un gain de temps lors de la recherche de présence d'un mot. En effet, pour chaque lettre du mot, on peut se déplacer directement sur la suivante sans avoir a parcourir une liste de nœuds frères. Un autre avantage de cette structure est qu'elle est plus simple à coder, que ce soit pour la création du dictionnaire ou pour le parcours de celui-ci.

Pour gérer le fait qu'un lettre puisse être ou non la fin d'un mot, plusieurs choix s'offraient à nous, nous aurions pû agrandir suivant et faire pointer la 27 case du tableau sur un caractère special (comme $ par exemple) pour signaler qu'une lettre est une fin de mot. Si le pointeur est NULL, alors la lettre n'est pas une fin de mot.

Nous avons choisi d'utiliser une variable fin dans la structure car cette

méthode semblait assez simple a implémenter, assez simple à utiliser et moins coûteuse en mémoire qu'une case supplémentaire du tableau.

3 2e méthode : arbre radix

Un arbre radix est semblable à un arbre préfixe. La seule différence, est que si un nœud possède exactement un fils, ce nœud fusionne avec ce fils dans le but de former un nœud contenant deux lettres. Le nœud fils est alors supprimé et le tableau de pointeurs du père est remplacé par celui du fils. Ce processus peut être répété autant de fois que necessaire ; un nœud peut donc contenir autant de lettres que nécessaire.

Pour construire notre arbre radix, nous avons donc repris l'arbre préfixe construit prècedemment et nous l'avons ensuite modifié afin de fusionner les nœud pouvant l'être.

3.1 Structure utilisée

Nous avons donc repris la structure du nœud de l'arbre préfixe. Nous avons néanmoins effectué un changement. Nous avons remplacé le char lettre, par un char* lettre. Un char* peut contenir plusieurs lettres, ceci permettait donc à la fusion de pouvoir se faire.

Nous avons effctué un autre changement par rapport à l'arbre préfixe, le nœud de base contient un charactère special dans l'arbre radix. Ce charactère est $, et la présence de ce caractère nous permet d'effectuer la transition de préfixe à affixe.

3.2 avantages et inconvénients de la structure, choix effectués

Encore une fois, notre arbre radix présente le problème de la place prise en mémoire par rapport à un arbre radix classique.

L'avantage de l'arbre radix par rapport à l'abre préfixe est qu'il prend moins de place en mémoire puisque de nombreux nœuds sont supprimés. Cependant, nous ne savons pas si cela le rend plus rapide, puisque la fonction de test de la présence d'un mot dans le dictionnaire est plus compliqué que pour l'abre préfixe. Nous verrons donc ceci dans la section test.

Un choix coûteux : la fonction fusion

Notre fonction fusion permettant de fusionner deux nœud, nécessaire pour passer de préfixe à radix est très coûteuse en calcul. En effet, lors de la fusion de deux nœuds, le char* lettre est tout d'abord copié, puis libéré avec la fonction free(), puis réallouer avec la fonction calloc() avec une case supplémentaire. Comme nous savons que les fonctions free() et calloc() sont très coûteuses en calcul, nous nous rendons compte que la transformation d'un arbre à l'autre est assez coûteuse.

```
==458988== HEAP SUMMARY:
==458988==     in use at exit: 0 bytes in 0 blocks
==458988==   total heap usage: 2,088,361 allocs, 2,088,361 frees, 145,583,166 bytes allocated
==458988==
```

memoire occupée par la fonction créant l'arbre radix.

La transformation necessite donc 851 268 allocutions supplémentaires, ce qui est énorme. La partie test explicitera la place gagnée par l'arbre radix par rapport a l'arbre préfixe.

4 3e méthode : la table de hachage

Afin de comparer une autre structure de données, nous avons décider d'implémenter une table de hachage en la remplissant des mots du dictionnaire.

4.1 Structure utilisée

Dans une table de hachage, on utilise tout d'abord un tableau de cellules, ici de taille M qu'on fera varier pour voir son influence. Une cellule est ici composé d'un char* contenant un mot et d'un pointeur vers la cellule suivante. On associe à chaque mot un indice dans le tableau grace à une fonction de hachage. La fonction de hachage est calculée très rapidement en O(1). Ensuite, chaque mot est placé dans le tableau à son indice, les mots ayant le même indices sont placés dans une liste chainée de cellules.

Afin de vérifier si un mot est présent dans la table de hachage, il suffit de calculer son indice à l'aide de la fonction de hachage puis de parcourir la liste chainée correspondant à son indice. On comprend donc rapidement que la valeur de M aura une grande importance sur le temps de recherche d'un

mot car si trop de mots ont le même indice il faudra parcourir de grandes listes chainées ce qui est très couteux en temps.

5 Comparaison des résultats

5.1 Les méthodes utilisées

Afin de vérifier le bon fonctionnement de notre structure, nous avons créé une fonction qui parcourt le texte et vérifie si chaque mot est dans le dictionnaire. La fonction compte le nombre de mots qui ne sont pas dans le dictionnaire. Cette fonction parcours le texte caractère par caractère à l'aide de la fonction fgetc(). Ce n'est pas la solution la plus rapide mais elle permet d'être sûr que tout les mots sont lus car le texte contient de grandes lignes de taille inconnue. On pourra alors ensuite vérifier si chaque solution trouve le même nombre de mots qui ne sont pas dans le dictionnaire. Nous avons aussi affiché les mots afin de s'assurer que les absences étaient cohérentes.

On cherche ensuite à mesurer le temps de création des différentes structure de données représentant le dictionnaire et le temps de recherche dans la structure. Nous avons donc mesuré le temps d'éxecution des fonctions de créations et des fonctions de vérification du texte. Ces fonctions ont la même structure, seul les actions concernant les structures de données diffèrent, on peut donc alors les comparer même si les temps prennent en compte la lecture des fichiers.

Toutes les mesures de temps effectuées sont les temps d'exécution des fonctions mesurés grâce à la bibliothèque time.h, les temps dépendent donc grandement du matériel utilisé et du nombre de processus en arrière plan. C'est pourquoi chaque mesure aura été effectuée sur la même machine avec dans un intervalle de temps court pour essayer au maximum de garder les mêmes conditions et ainsi de pouvoir comparer les résultats. On prendra à chaque fois la moyenne sur un total de 100 mesures.

Dans le but de comparer la place en mémoire que prend chaque méthode, nous avons utiliser valgrind. Nous procédons à la création des dictionnaires selon chaque structure sans désallouer les dictionnaires. Ne pas désallouer les dictionnaires nous permet de voir la mémoire utilisé par la structure une fois construite sans compter la mémoire utilisé pour la construction dans l'arbre

radix en particulier.

5.2 Comparaison

Tout d'abord nous obtenons pour les trois méthodes 26342 mots du texte non compris dans le dictionnaire (les doublons ne sont pas supprimés) sur un total de 1301894 mots dans le texte. Ainsi chaque méthode effectue la tache demandé de la même manière, c'est plutôt rassurant.

Concernant les temps d'exécution moyens :

Pour l'arbre préfixe :

— temps de création : 0,085219s

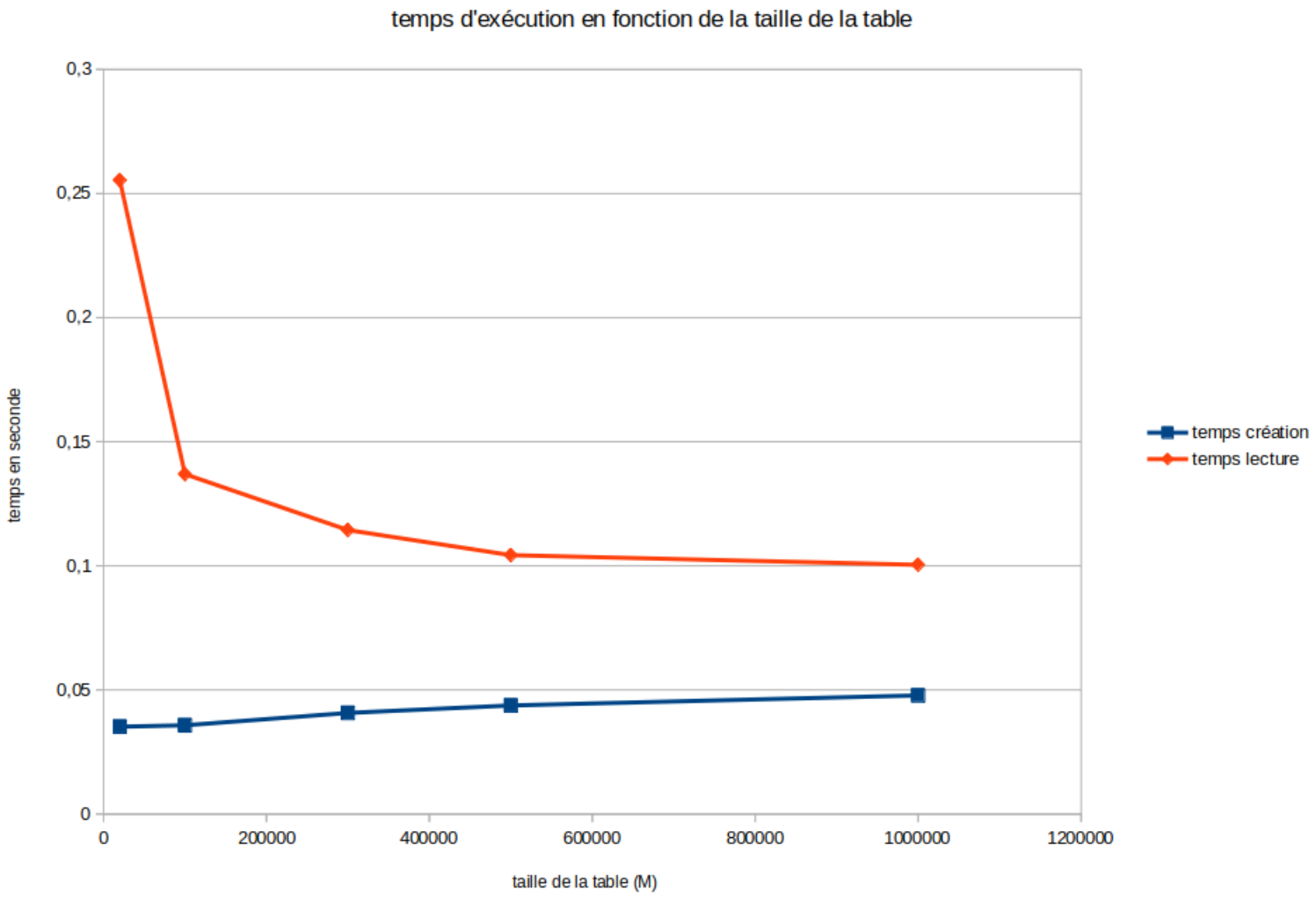
— temps de lecture : 0,113074s

Pour l'arbre radix :

— temps de création : 0,146357s

— temps de lecture : 0,224297s

Pour la table de hachage, cela dépend considérablement de M, la taille de la table. On peut tracer les temps d'exécutions en fonction de la taille M :



On remarque donc que plus la taille de la table est grande plus le temps de lecture est faible, cela est cohérent car en augmentant la taille, on réduit le nombre de liste chainée qui sont couteuses en temps, augmenter la taille revient à réduire la quantité de recherche séquentielle. Cependant cela a tendance à se stabiliser, en effet si la taille est supérieure au nombre de mots à placer dedans, on créer des places inutiles.Ici, le dictionnaire contient environ 325000 mots, le temps de lecture se stabilise à ce niveau. Le temps de

création augmente très légèrement avec la taille, cela est probablement dû à l'allocation du tableau qui est plus grande au départ.

On observe donc qu'en création et en lecture, avec une taille bien choisi, la table de hachage est plus efficace. L'arbre préfixe est plutôt rapide dans les deux catégories mais moins que la table. L'arbre radix est lui couteux en temps dans les deux catégories.

Consernant la lecture, pour l'arbre préfixe la complexité de recherche d'un mot est en O(longueur du mot) car on vérifie chaque lettre du mot. Pour l'arbre radix c'est plus compliqué car on doit savoir combien de lettres vérifiés à chaque étape, ce qui explique que ce soit plus long. La table de hachage, avec une taille bien choisi, effectue pour chaque recherche de mot une action en O(1) pour calculer la valeur de l'index, puis parcours une liste chainée très courte si la taille de la table est grande, il est donc très rapide de vérifier si un mot est présent dans la table.

Concernant les mémoires utilisées :

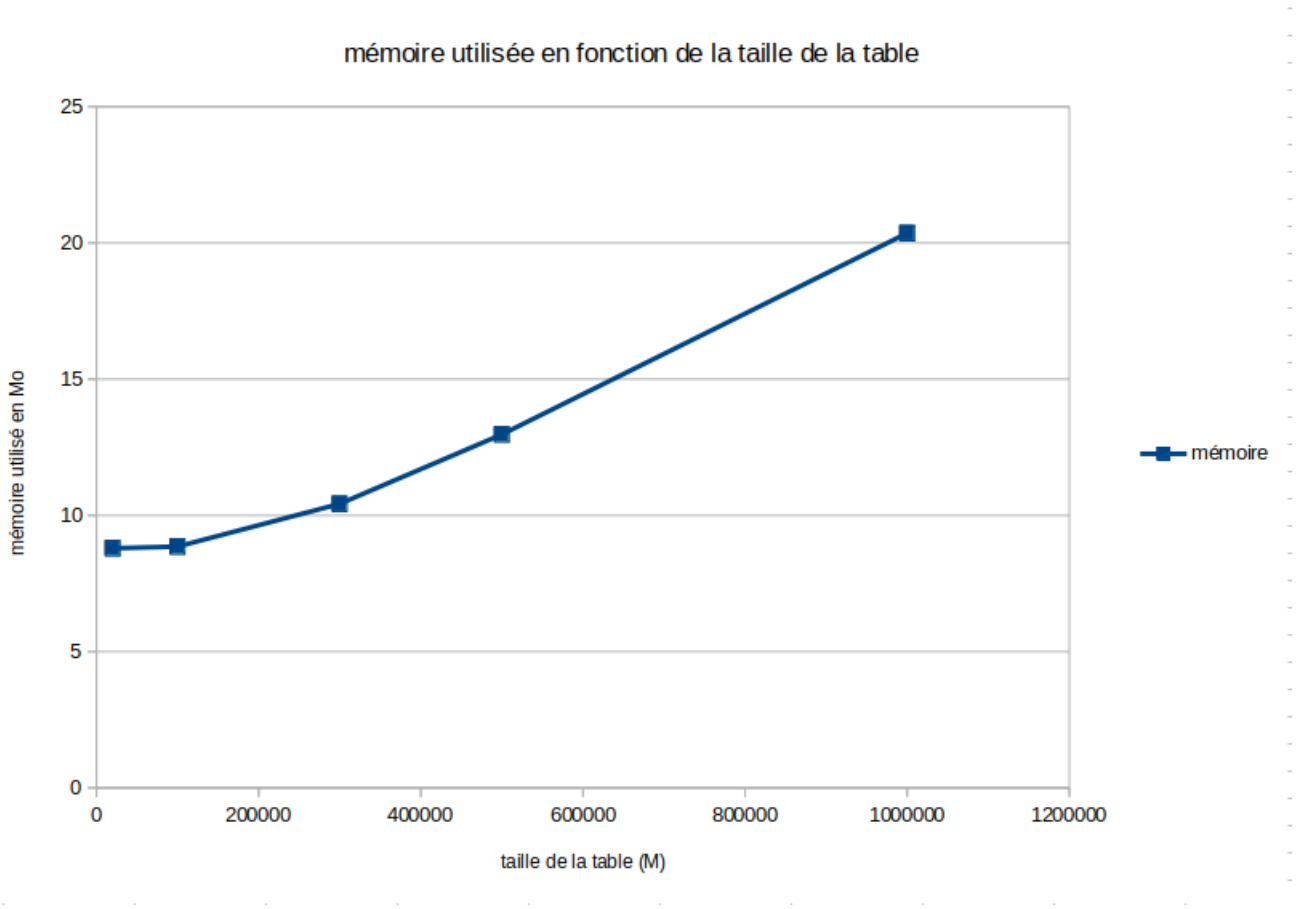
Pour l'arbre préfixe :

— 143,506 Mo

Pour l'arbre radix :

— 90,514 Mo

Pour la table de hachage, on trace la mémoire utilisée en fonction de M :



On remarque donc que les arbres sont très couteux en mémoires à coté d'une table de hachage quelque soit la taille utilisée. Cependant, l'abre radix se trouve être moins gourmand en mémoire que l'arbre préfixe, cela était attendu grâce au regroupement des lettres dans le radix. Il est possible d'optimiser la mémoire utilisée par la table en ne prenant pas une taille démesurée. Même si la consommation en mémoire reste raisonnable avec une taille déraisonnable un compromis est à trouver en fonction des temps d'exécution et de la mémoire utilisée.