Rapport de projet APMAN

Projet : Vérificateur orthographique

ROUSSEAU Cloé
HEITZ Sacha

Phelma, SEOC, 2021-2022

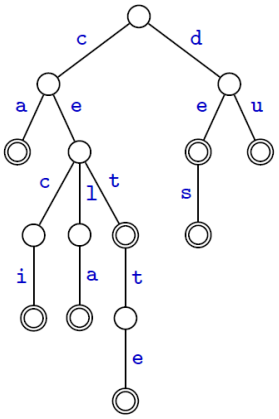
Sommaire :

1. Introduction :

L'objectif de ce projet est de concevoir un vérificateur orthographique en C. Nous allons construire donc un dictionnaire à partir d'un fichier .txt puis parcourir un texte donné afin de vérifier mot par mot s'il appartient ou non au dictionnaire. Les mots n'appartenant pas au dictionnaire seront affichés dans le terminal.
Nous avons implémenté le dictionnaire de quatre manières différentes afin de comparer les performances CPU et mémoires de plusieurs structures.

2. Les différentes implémentations :

a. Les arbres préfix :



Nous avons décidé d'implémenter cette solution avec deux structures légèrement différentes afin de comparer leurs performances. Dans la première structure proposée, chaque nœud contient une lettre, une liste chaînée de fils et un booléen qui permet d'indiquer si le nœud correspond à la fin d'un mot ou non. Dans la deuxième structure, chaque nœud contient une table de hachage de fils ainsi qu'un booléen qui à la même fonction que dans la première structure.

La première structure devrait être moins gourmande en mémoire que la seconde puisqu'on ajoute un fils seulement lorsqu'il y en a besoin alors qu'un tableau de 256 caractères est alloué même si on n'utilise qu'une seule case.

Toutes ces allocations mémoires sont également coûteuses en temps, on s'attend donc à créer le dictionnaire plus vite avec la première structure qu'avec la seconde.

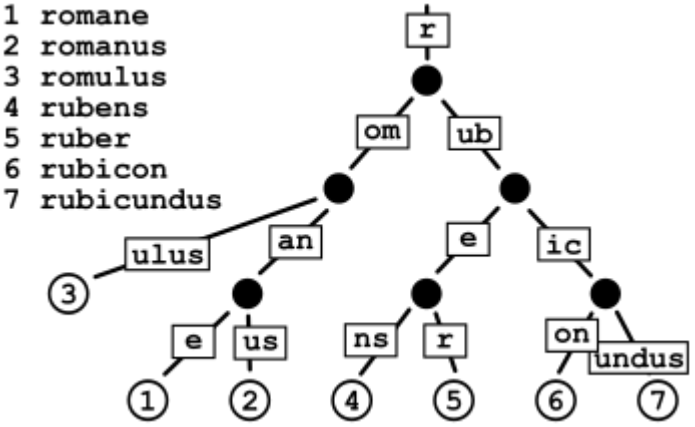
Cependant la deuxième structure devrait être beaucoup plus rapide que la première lors de la recherche d'un mot puisqu'il n'y aucune collision dans les tables de hachage représentant les fils (le code ACSII correspond à une fonction de hachage parfaite) .

b. La table de Hachage :

Nous avons également implémenté un dictionnaire à l'aide d'une table de hachage. La structure consiste en un tableau de listes chaînées. On ajoute dans la liste à l'indice de la table correspondant au hash du mot le mot en entier. La fonction

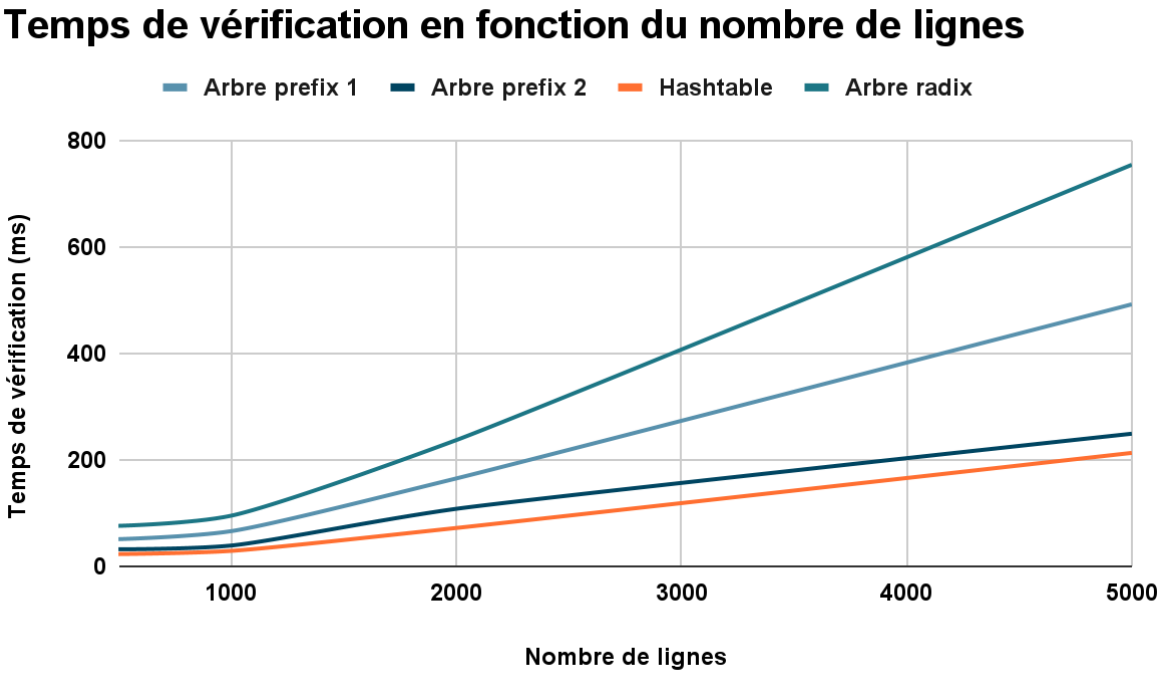
de hachage à été choisie dans le but de limiter les collisions et donc d'optimiser le temps de recherche d'un mot dans la table (La plus longue liste contient 13 mots dans notre table de hachage) . Cependant il existe sûrement des fonctions de hachage plus performantes que celle utilisée dans notre projet.

c. L'arbre radix:



Nous avons choisi d'implémenter l'arbre radix avec une structure proche de l'arbre prefix. En effet, l'arbre radix est un arbres ou les noeuds contiennent un mot (char *), un booléen pour indiquer la fin d'un mot et une liste de fils qui sont des nœuds. Le but de l'arbre radix est de compresser les nœuds avec leurs fils lorsqu'ils en possède qu'un seul. On s'attend donc à ce que l'arbre radix soit moins coûteux en mémoire qu'un arbre radix.

3. Comparaison des différentes structures :



	arbre prefix 1	arbre prefix 2	arbre radix	table de hachage
Espace mémoire occupé	24,7 Mo	1 272 Mo	16 Mo	9,5 Mo

Comme nous nous y attendions, l'arbre prefix ayant pour fils des listes chaînées occupe moins de place en mémoire que celui utilisant des tables de hachage. Cependant il est aussi bien moins performant lorsqu'il faut comparer un grand nombre de mots.
La tendance selon laquelle moins une structure occupe de place en mémoire, moins elle est rapide lors de la recherche d'un mot se confirme avec l'arbre radix.
Enfin nous constatons que la table de hachage est la structure la plus rapide pour rechercher un mot alors que c'est aussi celle qui occupe le moins de place en mémoire.

Cependant les arbres présentent un gros avantage sur la table de hachage puisqu'ils permettent de mettre en place les fonctionnalités d'auto-complétion que nous utilisons régulièrement (proposition lors d'une recherche sur internet, tabulation dans le terminal...).

4. Problèmes rencontrés :

Concernant l'arbre radix nous avons eu de nombreux problèmes liés à la gestion des chaînes de caractères. En effet nous avons dû utiliser des fonctions comme strdup pour copier une chaîne de caractère sans modifier tout l'arbre. Cependant cette fonction alloue en mémoire la chaîne de caractère. Il faut donc penser à la libérer au moment opportun. L'arbre Radix ne fonctionne pas totalement et nous n'avons pas réussi à débugger les dernières erreurs. Il n'y a aucun problème dans la formation du dictionnaire, cependant il y a un problème dans le remplissage de l'arbre radix des mots déjà rencontrés. A certains moment du texte (environ 2 fois pour 1000 mots) l'insertion d'un mot précis provoque l'insertion d'un paragraphe entier. Au final cela à des conséquences lors de la libération de la mémoire qui ne peut plus s'effectuer. De plus quand nous affichons au fur et à mesure les mots non présents certains se retrouvent alors en plusieurs fois. Nous avons donc décidé d'utiliser une table de hachage pour stocker les mots non présents afin de pouvoir quand même avoir un résultat.

5. Conclusion :

Ce projet nous a tous les deux beaucoup intéressé et nous a permis de découvrir en profondeur les notions d'arbres préfixes et radix que nous ne connaissions pas du tout. Il nous a permis de développer aussi bien nos compétences en informatique que nos compétences transversales telles que l'organisation, la gestion de projet ou la communication. Il nous a demandé beaucoup de rigueur et une certaine efficacité dans notre collaboration. Nous avons beaucoup appris et avons pris beaucoup de plaisir à construire et développer l'ensemble de ce projet, et avons notamment apprécié la satisfaction lorsque nous arrivions à un résultat concret et un code opérationnel.