SEOC

TP APMAN

Correcteur orthographique

Membres du groupe :
Damien Morlier
Arnaud GARIEL

Grenoble, France
Décembre 2021

Table des matières

Table des figures

Présentation du sujet

L'objectif de ce TP est d'implémenter un correcteur orthographique en C qui passera en revue un texte et pointeras les erreurs d'orthographe présentes dans ce texte. Pour construire ce correcteur, un dictionnaire français est utilisé et le texte qui est vérifié est le roman *A la recherche du temps perdu* de Marcel Proust. Le TP demande donc d'implémenter un correcteur orthographique à l'aide de plusieurs structures qui ont été choisis par les membres du groupe.

Chapitre 1

Structures

1.1 Liste chaînée

Cette structure est la plus basique pour stocker des données. Chaque mot du dictionnaire est stocké dans élément qui est ensuite rattaché à la liste chaînée. Une liste chaînée est en fait un ensemble d'élément (ici char*) reliés un à un par des pointeurs. Très simple à implémenter, elle n'est pas adaptée lorsqu'il faut stocker un nombre important de données puis aller y chercher un élément.

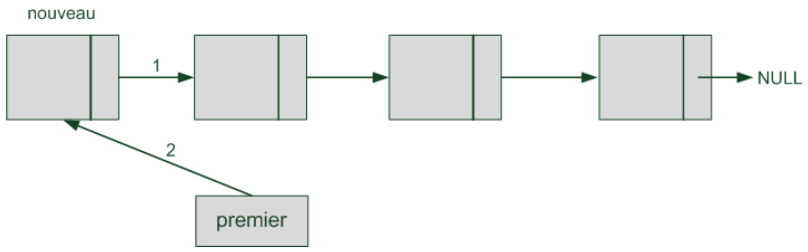

FIGURE 1.1 – Schéma principe d'une liste chaînée

1.2 Table de hachage

Cette structure vient résoudre le problème évoqué précédemment pour éviter de parcourir la totalité de la liste si le mot recherché n'est pas la. Cette structure est composée d'un tableau de liste dans lequel on va stocker les mots et de trois entiers désignant la capacité de la table, le nombre d'éléments et la capacité initiale. Il a fallu aussi construire une fonction de hachage. Il faut aussi calculer le reste de la division avec la capacité de la table de hachage pour pouvoir insérer le mot dans la table. Pour ce TP, nous avons utilisé la fonction de hachage djb2 par *Dan Bernstein* qui est une fonction de hachage qui est efficace avec les chaînes de caractère (plus en anglais qu'en français cependant. Nous avons aussi une fonction de hachage plus simple commentée qui calcule la longueur du mot et la multiplie à la position de sa première lettre. Dans les deux cas, il faut aussi calculer le reste de la division avec la capacité de la table de hachage pour pouvoir insérer le mot dans la table.

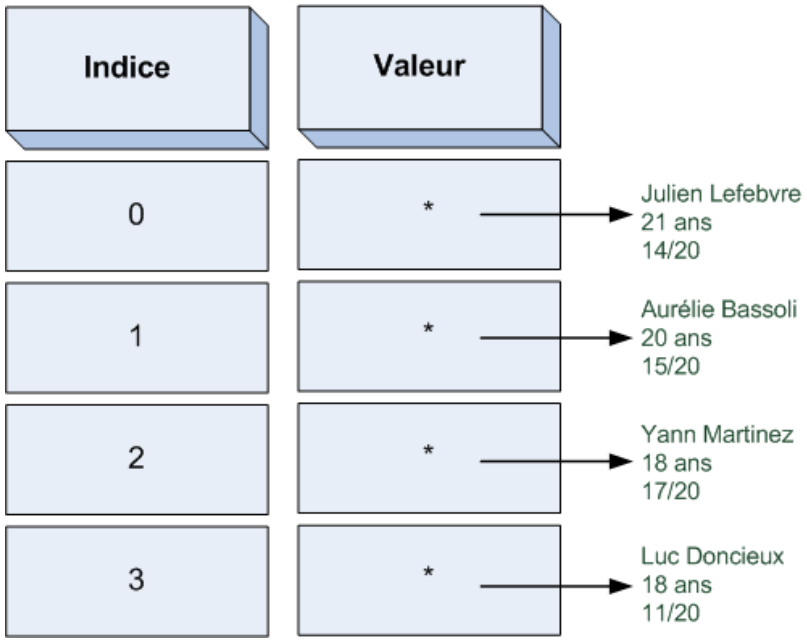

FIGURE 1.2 – Schéma principe de la table de hachage

1.3 Arbre préfixe

L'arbre préfixe que nous avons mis en place se base sur la structure que nous avons vu en cours sur les arbres : un noeud qui contient un caractère, une information s'il s'agit d'un mot ou noeud et plusieurs noeuds fils dans un tableau de noeud fils(alp).Dans notre cas, il y a 26 noeuds fils qui correspondent chacun dans l'ordre a une lettre de l'alphabet(a=0,b=1,etc...). Nous initialisions la première racine de notre arbre à ' ' et nous insérons chacun des mots en découpant les mots en lettres et en calculant l'indice ASCII du caractère moins celui du caractère a pour le ramener a un indice compris entre 0 et 26 et ainsi pouvoir créer un chemin de caractère menant au dernier caractère qui aura son paramètre estunmot a 1 pour la recherche. Cet arbre n'est pas une structure optimisé car a chaque création de noeud, il y a création d'un registre de 26 fils qui ne sont pratiquement jamais tous utilisés, il y a donc trop de mémoire utilisée, mais comme montré dans les tests, il s'agit d'une structure très efficace en recherche.

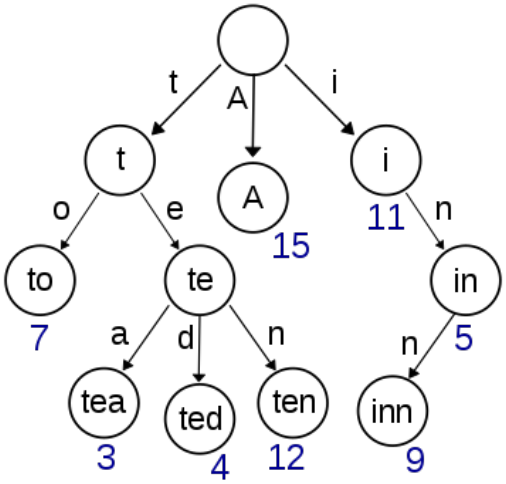

FIGURE 1.3 – Schéma principe de l'arbre préfixe

1.4 Arbre radix

L'arbre radix a pour objectif de compresser l'arbre préfixe pour prendre moins de place en mémoire(sauf si l'arbre préfixe est déjà optimisé en mémoire), il existe deux façons de le construire : partir d'un arbre préfixe et le compresser ou compresser l'arbre a chaque insertion d'un nouveau mot. La figure 1.4 présente un arbre radix finalisé à l'aide de la deuxième méthode de compression a chaque ajout de mot. Nous avons opté pour la première solution car cela nous a permis d'optimiser notre arbre préfixe en mémoire(ne pas allouer 26 fils à chaque création de noeud) et la compression était plus simple à mettre en place qu'une compression de l'arbre à chaque ajout de mot. Pour compresser nous avons utilisé la méthode montrée sur les figures 1.5 1.6 et 1.7 qui suivent.

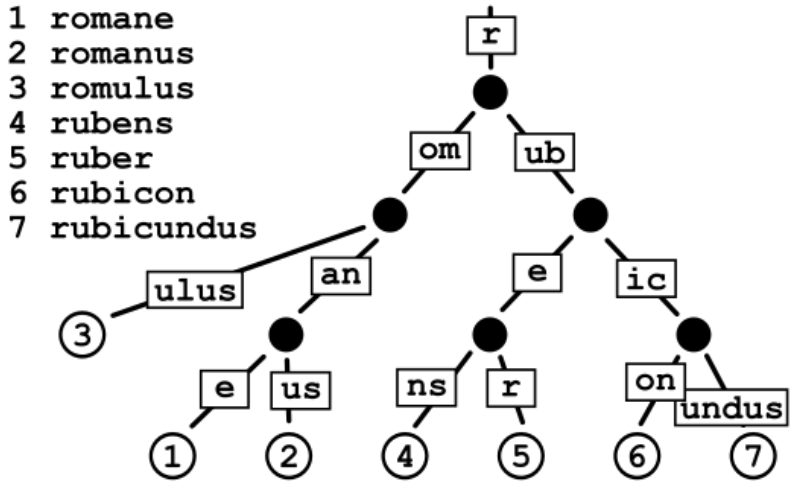

FIGURE 1.4 – Schéma principe de l'arbre radix

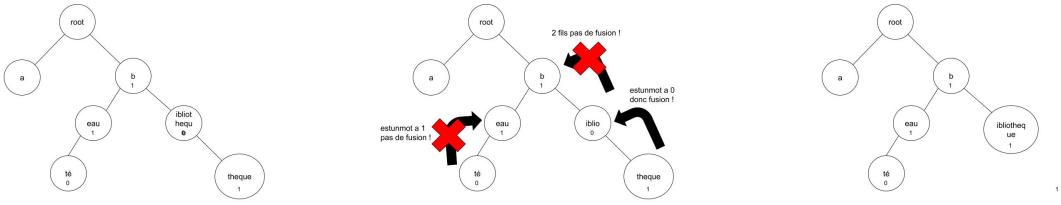

FIGURE 1.5 – Principe de compression de notre arbre radix

Chapitre 2

Complexité des implémentations

Tous les algorithmes présentés précédemment ont des complexités différentes. Notons n le nombre d'éléments que l'on a insérer dans notre dictionnaire qui se trouve sous la forme d'une des structure présenté.

Liste chaînée : Insérer un élément dans la liste chaînée n'as une complexité de O(1) car l'on insère l'élément en début de liste. Cependant, lorsque l'on veut accéder à un élément dans cette liste, on doit parcourir chaque élément de la liste pour espérer trouver l'élément recherché. On a donc dans le meilleur des cas une complexité en O(1) (qui correspond à ce que l'on trouve l'élément dès le début de la liste) et dans le pire des cas une complexité en O(n) (l'élément n'est pas présent dans la liste).

Table de hachage : La complexité d'une table de hachage varie avec l'efficacité de la fonction de hachage. En effet, dans le pire des cas, tous les éléments ont le même hash. Ils sont donc stockés au même endroit dans la table et cela correspond donc simplement à une liste chaînée. La complexité est alors en O(n). Un second problème que l'on trouve dans les tables de hachage intervient lors de la saturation d'une table qui entraîne le redimensionnement de cette dernière. Cela va affecter la complexité de l'algorithme. Cependant, en général et avec une bonne fonction de hachage, les collisions sont moindres et donc l'insertion d'un élément est en O(1) complexité.

Arbre préfixe et radix : Les deux algorithmes traitant les arbres enracinés ont la même complexité en temps. En effet, si un arbre est filiforme, alors sa complexité sera en O(n). Cependant, si il est équilibrés, alors sa complexité sera en O(ln(n)).

Chapitre 3

Tests

3.1 Tests d'exécution

Un des objectifs de ce TP APMAN était de comparer l'efficacité des structures implémentées pour choisir celle qui serait la plus efficace. Pour cela nous avons deux paramètres à étudier :

La construction du dictionnaire : Pour cette partie, il faut extraire tous les mots du fichier *FR.txt* donné et les insérer dans nos structures. Nous avons donc évalué le temps de construction du dictionnaire de chaque structure [Figure 3.1]. On retrouve bien que la liste chaînée et la table de hachage ont un temps de construction faible en raison de la faible complexité de leurs fonctions d'insertion (qui est identique). On observe également que les deux arbres ont un temps de construction similaire et supérieur aux deux structures précédentes. En effet, le rajout d'un élément dans un arbre nécessite le parcours d'une branche de l'arbre. C'est cette différence qui explique le rapport entre nos deux groupes de structures.

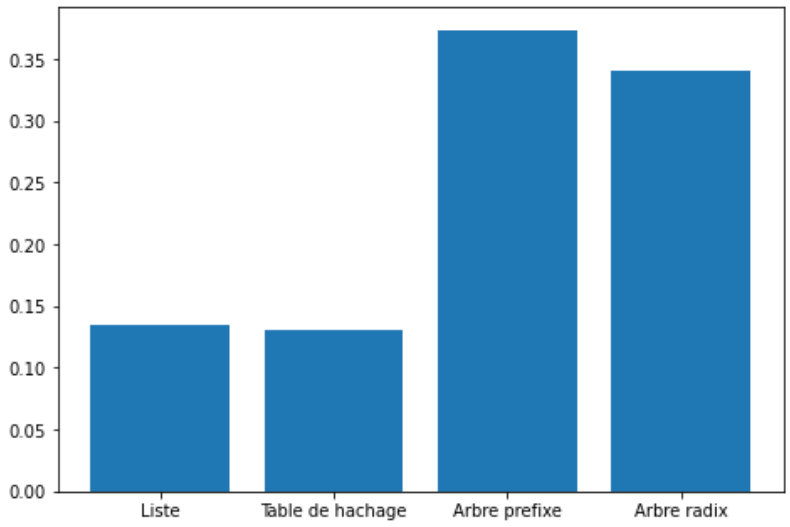

FIGURE 3.1 – Diagramme à bandes des temps de construction du dictionnaire

La recherche d'élément dans le dictionnaire : La fonctionnalité la plus utile de ce TP est la recherche d'un élément dans le dictionnaire. Comme l'on peut le voir sur la figure 3.2, le temps de recherche d'un élément dans une liste est proportionnel au nombre d'éléments dans cette liste.

Les trois autres structures sont toutes bien plus efficaces qu'une liste chaînée comme l'on peut le voir sur la figure 3.3. On observe que la table de hachage est très efficace comparé

aux deux arbres. Sa tendance semble presque linéaire mais avec un coefficient directeur très faible qui rend cette implémentation très efficace. Les arbres ont une courbe de tendance assez similaire en forme mais l'arbre préfixe semble plus efficace.

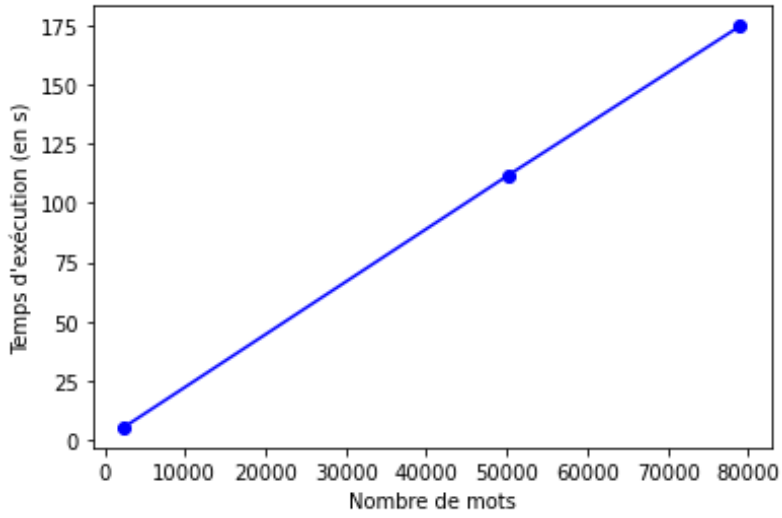

FIGURE 3.2 – Graphique du temps d'exécution de la liste chaînée (construction + vérification)

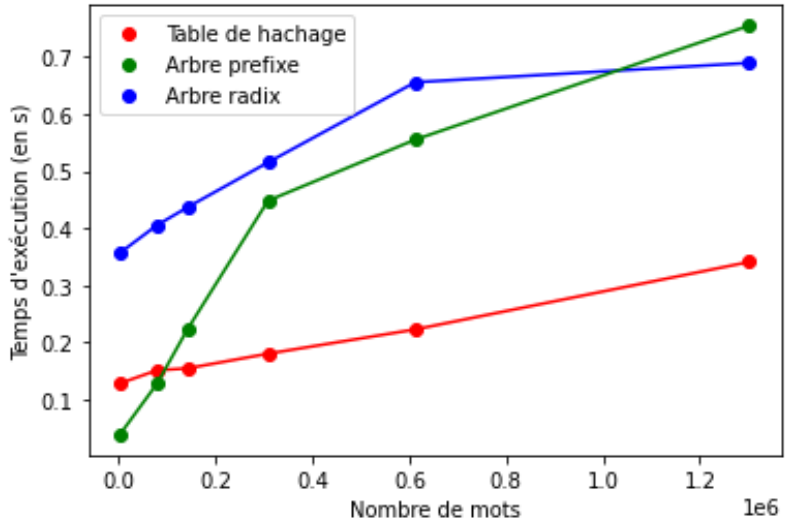

FIGURE 3.3 – Graphique des temps d'exécution (construction + vérification)

3.2 Tests de mémoire

L'outil de programmation *Valgrind* permet de déboguer et savoir s' il y a des fuites de mémoire. L'on peut connaître le nombre d'allocations et de frees dans un programme. Cet outil à été très utile lors des mises en œuvre des algorithmes. Nous avons donc effectué le test pour toutes nos structures avec 11 mots à vérifier.

Comme l'on peut le remarquer, nos programmes nécessitent un nombre important d'allocations mais tout est libéré. Le nombre de bytes alloués est différent selon la structure utilisée. Par exemple, la liste chaînée alloue beaucoup moins de bytes car son implémentation est très simple.

```
==14465== HEAP SUMMARY:
==14465==     in use at exit: 0 bytes in 0 blocks
==14465==   total heap usage: 323,812 allocs, 323,812 frees, 12,962,440 bytes al
located
```

FIGURE 3.4 – Valgrind du programme test pour la liste chaînée

```
==14470== HEAP SUMMARY:
==14470==     in use at exit: 0 bytes in 0 blocks
==14470==   total heap usage: 869,830 allocs, 869,830 frees, 47,908,240 bytes al
located
```

FIGURE 3.5 – Valgrind du programme test pour la table de hachage

```
==14325== HEAP SUMMARY:
==14325==     in use at exit: 0 bytes in 0 blocks
==14325==   total heap usage: 2,777,955 allocs, 2,777,955 frees, 38,474,436 byte
s allocated
```

FIGURE 3.6 – Valgrind du programme test pour l'arbre préfixe

```
==14325== HEAP SUMMARY:
==14325==     in use at exit: 0 bytes in 0 blocks
==14325==   total heap usage: 2,777,955 allocs, 2,777,955 frees, 38,474,436 byte
s allocated
```

FIGURE 3.7 – Valgrind du programme test pour l'arbre radix

3.3 Nombres de mots incorrect

Si toutes nos implémentations fonctionnent correctement, alors nous devrions obtenir le même nombre de mots incorrect pour toutes nos implémentations. C'est bien le cas pour toutes nos implémentations sauf la liste chaînée (le texte est trop long). En effet on trouve **26369** mots incorrect.

Chapitre 4

Problèmes rencontrés par les étudiants

Arnaud : J'ai réalisé les implémentations pour la table de hachage et liste chaînée. Les fonctions à implémenter ressemblaient fortement à celles faites en TD. J'ai commencé directement par la table de hachage qui à été une erreur je pense. J'ai dû déboguer les listes chaînées dans la table. Il aurait été plus simple de régler le problème dans l'implémentation liste plutôt que directement dans la table de hachage. Le plus gros problème à été de stocker une valeur dans la table de hachage puis de rechercher un élément dans cette table. J'ai découvert trop tard la fonction strcpy() qui m'aurait fait gagner beaucoup de temps. Le plus gros problème résidait donc dans la manipulation de char*.

Damien : J'ai réalisé les implémentations de l'arbre préfixe et de l'arbre radix, les fonctions à mettre en place n'étaient pas des fonctions classiques vues en cours mais des adaptations d'arbre à n-fils, il fallait donc faire attention à quelles adresses on accédait et faire attention aux invalid read et invalid write qui sont arrivés fréquemment. L'arbre préfixe était simple à mettre en place avec la création des 26 noeuds fils à chaque création de noeud, pour accéder à une lettre on calculait simplement son code ASCII - celui de la lettre a pour avoir accès au bon fils alors qu'avec l'arbre radix il fallait chercher dans chaque noeud quel noeud correspondait au bon préfixe et la compression de l'arbre était également complexe a mettre en place.

Chapitre 5

Conclusion

Le TP APMAN nous a permis de mettre en application les TD dans un contexte pratique. Il nous à aussi permis de mieux prendre en main différents outils comme Git, gdb ou encore Valgrind. Il faut aussi noter que ce TP nous a poussé à avoir un programme propre et commenté.

Chapitre 6

Sources

Schéma des listes chainées : https ://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/19733-les-listes-chainees

Schéma de la table de hachage : https ://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/19978-les-tables-de-hachage

Schéma des arbres :

https ://fr.wikipedia.org/wiki/Trie$_{(}informatique$)

https ://fr.wikipedia.org/wiki/Arbre$_{r}adix$

Fonction de hachage djb2 : https ://stackoverflow.com/questions/64699597/how-to-write-djb2-hashing-function-in-c