Saisie intuitive

Projet Algorithmique et programmation
Programmation C

Authors: *Steeven KENNY & Jean Eudes KONAIN*

Instructor: *Marco SILVA*
Department: Informatique
Date: April 15, 2023

Contents

1 Choix de la structure de donnée

La structure de données utilisée dans ce projet est la structure Trie. cette structure permet de stocker et de rechercher efficacement un grand nombre de chaînes de caractères. Les Tries sont des arbres n-aires où chaque nœud représente un caractère dans une chaîne et chaque chemin de la racine à un noeud feuille représente une chaîne dans la structure.

2 Fonction TrieNode

2.1 Prototype

La fonction create_node() cree et initialise un pointeur vers un nouveau TrieNode.

```
TrieNode *create_node();
```

2.2 Complexité

La fonction create_node() a une complexité en temps constante, O(1), car elle ne dépend pas de la taille de la structure de données ou de l'entrée.

3 Fonction char_to_index

3.1 Prototype

La fonction char_to_index() retourne l'indice de l'alphabet etendu pour un caractère donne en utilisant la conversion de type cast

```
int char_to_index(char c);
```

3.2 Complexité

La fonction char_to_index() a également une complexité en temps constante, O(1), car elle ne dépend que d'un seul caractère.

4 Fonction insert

4.1 Prototype

La fonction insert() insere un mot donné dans le trie en parcourant le trie a partir de la racine, en utilisant chaque caractere du mot pour indexer les nœuds enfants et en creant des nœuds pour les caracteres qui ne sont pas encore presents. Une fois que le dernier caractere est atteint, is_end_of_word est defini sur true et frequency est incrémente de 1.

```
void insert(TrieNode *root, const char *word);
```

4.2 Complexité

La fonction insert() a une complexité en temps de O(L), où L est la longueur du mot à insérer. Cette complexité est due à la boucle de parcours du mot à insérer et à la création de nouveaux noeuds dans l'arbre si nécessaire.

5 Fonction find_top_n_predictions_helper

5.1 Prototype

La fonction find_top_n_predictions_helper() est une fonction auxiliaire qui permet de trouver les n meilleures prédictions pour un préfixe donné.

```
void find_top_n_predictions_helper(TrieNode *node, char *prefix,
    int n, int level, int *count);
```

5.2 Complexité

La fonction find_top_n_predictions_helper() a une complexité en temps de O(SNlog(N)), où S est le nombre total de mots stockés dans l'arbre et N est le nombre de prédictions souhaité. Cette complexité est due à la recherche et au tri des prédictions dans l'arbre.

6 Fonction find_top_n_predictions

6.1 Prototype

La fonction find_top_n_predictions prend en entree un pointeur vers la racine d'un arbre Trie, un prefixe de mot et un entier n qui indique le nombre de predictions a afficher. Elle recherche l'arbre Trie pour trouver les n mots les plus frequents qui commencent par le prefixe donné et les affiche a l'aide de la fonction find_top_n_predictions_helper.

```
void find_top_n_predictions(TrieNode *root, const char *prefix,
    int n);
```

6.2 Complexité

La complexité de la fonction find_top_n_predictions dépend de la longueur de la chaîne de caractères prefix, qui est O(m), où m est la longueur de prefix, ainsi que du nombre de résultats souhaité, qui est O(n). Dans le pire des cas, la fonction parcourt tout l'arbre à partir du nœud correspondant à prefix, ce qui donne une complexité O(s * n), où s est la somme des longueurs des chaînes de tous les mots dans l'arbre, c'est-à-dire la taille de l'arbre.

7 Fonction load_words_into_trie

7.1 Prototype

La fonction load_words_into_trie prend en entree un pointeur vers la racine d'un arbre Trie et un chemin de fichier. Elle ouvre le fichier et lit chaque mot du fichier en utilisant la fonction fgets, retire le caractere de nouvelle ligne a l'aide de strcspn, puis insere chaque mot dans l'arbre Trie en appelant la fonction insert.

```
void load_words_into_trie(TrieNode *root, const char *file_path);
```

7.2 Complexité

La complexité de la fonction load_words_into_trie dépend de la taille du fichier d'entrée. Dans le pire des cas, la fonction doit lire et insérer chaque mot dans l'arbre, ce qui donne une complexité O(s), où s est la somme des longueurs des chaînes de tous les mots dans le fichier d'entrée.

8 Fonction load_words_into_trie

8.1 Prototype

La fonction search prend en entree un pointeur vers la racine d'un arbre Trie et un mot a chercher. Elle recherche l'arbre Trie pour voir si le mot donné est present dans l'arbre ou non et retourne vrai s'il est présent et faux sinon.

```
bool search(TrieNode *root, const char *word);
```

8.2 Complexité

La complexité de la fonction search dépend de la longueur de la chaîne de caractères word, qui est O(m), où m est la longueur de word. Dans le pire des cas, la fonction parcourt tout l'arbre à partir de la racine jusqu'au dernier caractère de word, ce qui donne une complexité O(s), où s est la somme des longueurs des chaînes de tous les mots dans l'arbre, c'est-à-dire la taille de l'arbre.

9 Fonction process_input_word

La fonction process_input_word a pour but de chercher si un mot existe dans un dictionnaire, qui est représenté par un arbre Trie et passé en paramètre sous forme de pointeur.

9.1 Prototype

```
void process_input_word(TrieNode *dictionary, TrieNode
   *prediction, const char *word);
```

9.2 Complexité

La complexité de la fonction process_input_word est la même que celle de la fonction search, car elle appelle simplement la fonction search pour savoir si word est présent dans l'arbre dictionary, puis insère word dans l'arbre prediction si c'est le cas.

10 Fonction add_custom_word

10.1 Prototype

```
void add_custom_word(TrieNode *prediction, const char *word)
```

10.2 Complexité

La complexité de la fonction add_custom_word dépend de la complexité de la fonction insert, qui a une complexité en temps de $\Theta(L)$, où L est la longueur du mot à insérer. La complexité en temps de la fonction add_custom_word est donc également $\Theta(L)$.

11 Fonction delete_word_helper

La fonction delete_word_helper a pour but de supprimer un mot spécifié d'un trie en effectuant une recherche à partir de la racine

11.1 Prototype

```
bool delete_word_helper(TrieNode *node, const char *word, int
    level)
```

11.2 Complexité

La complexité de la fonction delete_word_helper dépend de la longueur du mot et de la profondeur à laquelle le mot est présent dans l'arbre. Dans le pire des cas, où le mot à supprimer est le seul mot présent dans l'arbre, la complexité en temps de la fonction est $\Theta(L * N)$, où N est le nombre total de mots dans l'arbre et L est la longueur du mot. Dans le meilleur des cas, où le mot à supprimer n'est pas présent dans l'arbre, la complexité en temps est $\Theta(L)$, où L est la longueur du mot à supprimer.

12 Fonction delete_word

12.1 Prototype

```
void delete_word(TrieNode *root, const char *word);
```

12.2 Complexité

La complexité de la fonction delete_word dépend de la complexité de la fonction delete_word_helper, qui a une complexité en temps dépendant de la longueur du mot et de la profondeur à laquelle le mot est présent dans l'arbre. La complexité en temps de la fonction delete_word est donc également $\Theta(L * N)$ dans le pire des cas, où N est le nombre total de mots dans l'arbre et L est la longueur du mot à supprimer.

13 Fonction edit_word

La fonction edit_word a pour but de modifier un mot dans l'arbre Trie

13.1 Prototype

```
void edit_word(TrieNode *root, const char *original, const char
    *modified)
```

13.2 Complexité

La complexité de la fonction edit_word dépend de la complexité des fonctions delete_word et insert, qui ont toutes deux une complexité en temps de $\Theta(L)$, où L est la longueur du mot. La complexité en temps de la fonction edit_word est donc $\Theta(L * N)$

14 Fonction display_menu

14.1 Prototype

```
int display_menu();
```

14.2 Complexité

Complexité linéaire $\Theta(1)$.