Cremaschi Guillaume

Compte rendu projet

Détecteur de fautes

Programmation avancée

Sommaire :

Rappel du cahier des charges :

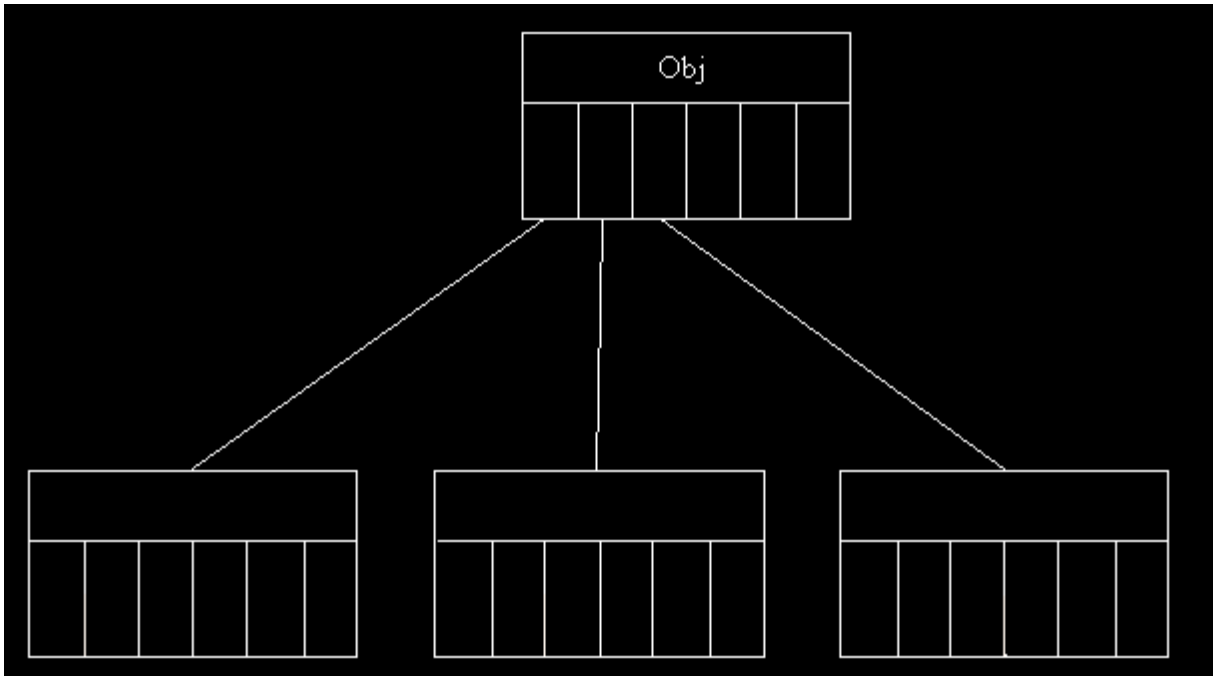
Le travail à réaliser était le suivant:

- Définir et implémenter une structure de données permettant de stocker et de manipuler un dictionnaire sous forme d'arbre préfixe / trie.
- Charger un dictionnaire à partir d'un fichier texte de données. Ce fichier texte pouvant être un texte court, un roman ou une liste de mots.
- Analyser l'orthographe d'une phrase ou d'un texte en indiquant les nombres de mots qui ne sont pas reconnus par le dictionnaire.

Structure choisie :

Afin de stocker le dictionnaire et toutes les possibilités de mots dans un arbre trié, j'ai décidé d'utiliser un arbre de tableau de pointeurs. Chaque cellule de l'arbre est composé d'une valeur, d'un tableau de pointeurs pointants vers une autre cellule et d'une marque de fin de mot qui peut être vraie ou fausse.

Voici un exemple d'arbre de tableau :



La taille du tableau est définie arbitrairement sur 27 pour pouvoir stocker toutes les lettres de l'alphabet plus l'apostrophe.

Pour réaliser ce projet, un arbre de liste chainée était tout aussi possible. L'avantage d'un arbre de tableau et d'une part la vitesse de traitement puisque les tableaux sont ordonnés et que ; chaque indice du tableau correspond à l'indice d'une lettre prédéfinie (Un « a » ou un « g » ne seront jamais en 15ème position mais toujours en 0 et 6). D'une autre part, je trouvais la structure de l'arbre de tableau plus simple à réaliser.

Le désavantage de cette structure, comparé à un arbre de liste chainée, est la place mémoire utilisée. En effet la taille du tableau étant fixe, dans certaines cellule le tableau de sera pas forcément rempli complétement.

Description des fonctions principales:

is_empty(ptcell pt)

return vrai si la cellule est vide, return faux sinon.

feuille(ptcell pt)

return vrai si la cellule ne pointe pas vers une ou d'autres cellule, return faux sinon.

*init_arbre(ptcell *pt)*

Initialise une cellule vide avec un caractère arbitraire. La cellule pointe vers un tableau de pointeur nuls.

indice_lettre(char lettre)

Récupère les codes ascii de la lettre passé en paramètre et regroupe les majuscules et minuscules dans un même index entre 0 et 25 classé par ordre alphabétique(a et A sont l'index 0, e et E sont l'index 5), l'apostrophe est placé en 26ème position.

ajout_arbre(ptcell pt, char mot[])

Tant qu'une fin de mot n'est pas atteinte, si l'indice du tableau qui correspond à l'index de la lettre est vide, l'ajoute à cette indice. Sinon si l'index suivant est une fin de mot, marque vrai sur le marqueur de fin de mot, sinon on passe à l'index suivant et à la ligne suivante de l'arbre.

caract_fin(char lettre)

return vrai si le caractère en paramètre ne correspond pas à une lettre ou un apostrophe, return faux sinon.

verifie_mot(ptcell pt, char mot[], int compteur)*

Vérifie que lettre par lettre que l'index de la lettre est présente dans le tableau de la cellule, si oui on passe à la lettre suivante et à la cellule suivante.

Si l'index d'une lettre n'est pas présent dans le tableau, compte une faute de plus.

Si la dernière lettre du mot est atteinte mais que la lettre correspondant n'a pas d'indicateur de fin de mot, compte une faute en plus.

À chaque fois qu'une faute est comptée, relève le mot concerné.

free_arbre(ptcell arbre)*

Libère la mémoire des cellules de l'arbre passé en paramètre.

Fonctionnement général :

Le programme prend deux fichiers en paramètres, un fichier de référence qui va servir de dictionnaire, et un fichier texte à tester afin de vérifier s'il y a des fautes d'orthographe à l'intérieur.

Voici son fonctionnement général :

- Ouvre le fichier dictionnaire en lecture seule

- Ouvre le fichier à vérifier en lecture seule

- Initialise une première cellule

- Rempli l'arbre

- Compare les mots du fichier à vérifier avec les mots de l'arbre
  et relève les fautes d'orthographe

- Libère la mémoire occupée par l'arbre

- Ferme les fichiers

- Fin

Test de performances :

Afin de tester l'efficacité du programme, sa vitesse d'exécution a été tester. Les résultats moyens obtenus pour un dictionnaire e 99171 mots et un fichier texte de 112 mots sont les suivants :

real : 0.170sec
user : 0.100sec
sys : 0.065sec

Malheureusement, ne disposant pas de *valgrind* sur mon ordinateur, le test des fuites mémoires n'as pas pu être réaliser sur le programme final.
J'ai néanmoins pris soin de libérer les pointeurs utilisés et de fermer les fichiers après utilisation. Il ne devrait donc pas avoir de fuites mémoire, une vérification est tout de même nécessaire.

Limitations :

Ce programme est fonctionnel mais rencontre plusieurs limitations :

Il ne prend pas en compte les accents. Cela signifie que comme il n'y a pas d'accent dans le dictionnaire, et comme il n'y a pas de fonction de traitement d'accent, chaque mot accentué sera traité comme faux.

Il ne différencie pas les majuscules des minuscules. En effet la fonction *indice_lettre* place les lettres majuscules et les minuscules sous le même indice. Cela signifie qu'un « M » ou un « m » sont perçu de la même façon.

Aucun test avec un texte contenant des caractère spéciaux n'a été effectué.

Problèmes rencontrés et solutions apportées :

Durant ce projet, différentes solutions ont été abordées. En premier lieu, j'avais opté pour un arbre de listes chainées, avec une vitesse de traitement plus lente certes mais qui a l'avantage d'être moins conséquent. Pour une application sur un téléphone avec peu de mémoire comme le mien, chaque octet de gagner est important, je me suis donc tourner naturellement vers cette solution. Après quelques échecs et le retard s'accumulant dans le projet, il est apparu nécessaire de me tourner vers une solution plus simple. C'est pourquoi j'ai finalement opté pour un arbre de tableau de pointeurs.

Cette seconde solution est certes plus simple, elle n'en est pas moins efficace puisque la perte en mémoire est compensée par un gain en rapidité d'exécution. Cette solution est donc adaptée pour des machines peu puissante, qui sont d'ailleurs souvent accompagnée d''un espace de stockage faible (comme c'est le cas de mon téléphone).

Une fois l'arbre réalisé, il a fallu coder la détection de fautes. Initialement, j'ai voulu détecter les fautes d'un texte taper directement dans le terminal. N'arrivant pas à résoudre cette nouvelle problématique et le retard s'accumulant encore une fois, j'ai décidé de détecter les fautes dans un fichier texte en lecture seule, cette alternative étant plus intuitive à mon goût.

Conclusion :

Pour conclure, le manque temps et le manque de connaissances ont été un véritable frein pour la réalisation et l'approfondissement du projet.
Ce travail m'a cependant permis d'apprendre à envisager d'autres solutions afin d'avancer et de le réaliser dans le temps imparti et de ne pas m'obstiner sur un obstacle trop complexe pour moi et n'apportant pas une grande différence finale. Cela m'a aussi permis de m'auto-évaluer et de me rendre compte des compétences qu'il me reste encore à acquérir puisque le code fonctionne certes, mais ne ressemble pas à l'idée que je m'en faisait.

Pour approfondir le sujet, on pourrait mettre en place une correction qui propose plusieurs mots pour remplacer le mot erroné et qui le corrige dans le fichier à vérifier (par exmple 3 à la manière d'un smartphone). On pourrait aussi corriger les fautes d'un texte tapé directement par l'utilisateur et l'on pourrait implémenter une fonction qui ajoute un mot dans le dictionnaire si l'utilisateur l'estime correct et qu'il ne se trouve pas dans l'arbre.