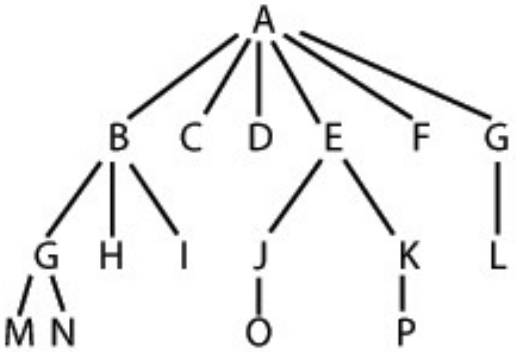
Simonin Richard
2019

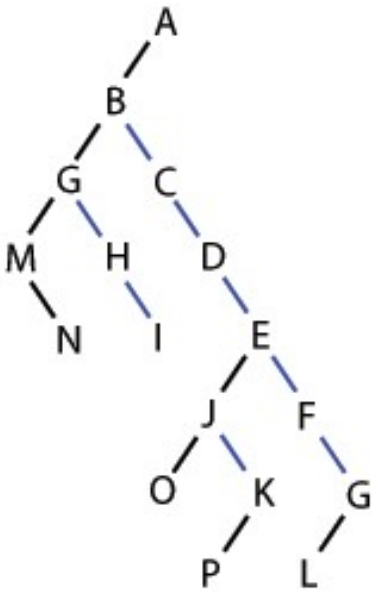
Correcteur Orthographique

COMPTE RENDU PROJET IMA3

Programmation C

Table des matières





Introduction

Le but de ce projet est de réaliser un correcteur orthographique. Lors de ce projet, la solution technique proposé est un arbre. Nous sommes libres des choix et de l'adaptation de notre projet. Je vais expliquer dans la suite du rapport les choix ainsi que les solutions techniques choisis. Le projet a pour but de charger un dictionnaire dans un arbre et d'analyser un fichier afin de savoir le nombre de mot juste ainsi que de mot faux.

Choix principaux du projet

Pour la réalisation de ce projet, j'ai décidé de ne pas considérer les accents car cela aurait engendré une complexité trop importante par rapport au faible intérêt du résultat engendré et au peu de temps pour l'élaboration du programme. Ainsi, j'ai travaillé uniquement avec l'alphabet et l'apostrophe.

En supposant qu'un dictionnaire est chargé une seule fois au début de l'exécution du programme, réduire les vitesses de lecture de fichier et de lecture dans la structure de données sont des contraintes primordiales. J'ai donc choisi d'utiliser un arbre qui permet d'optimiser à la fois l'espace mémoire et le temps d'exécution. En effet, à l'aide des commandes *time* et *valgrind*, pour un dictionnaire de 100 000 mots, le programme s'exécute en 0.040s *sys* et alloue 52 millions d'octets.

Structure

Choix de la structure

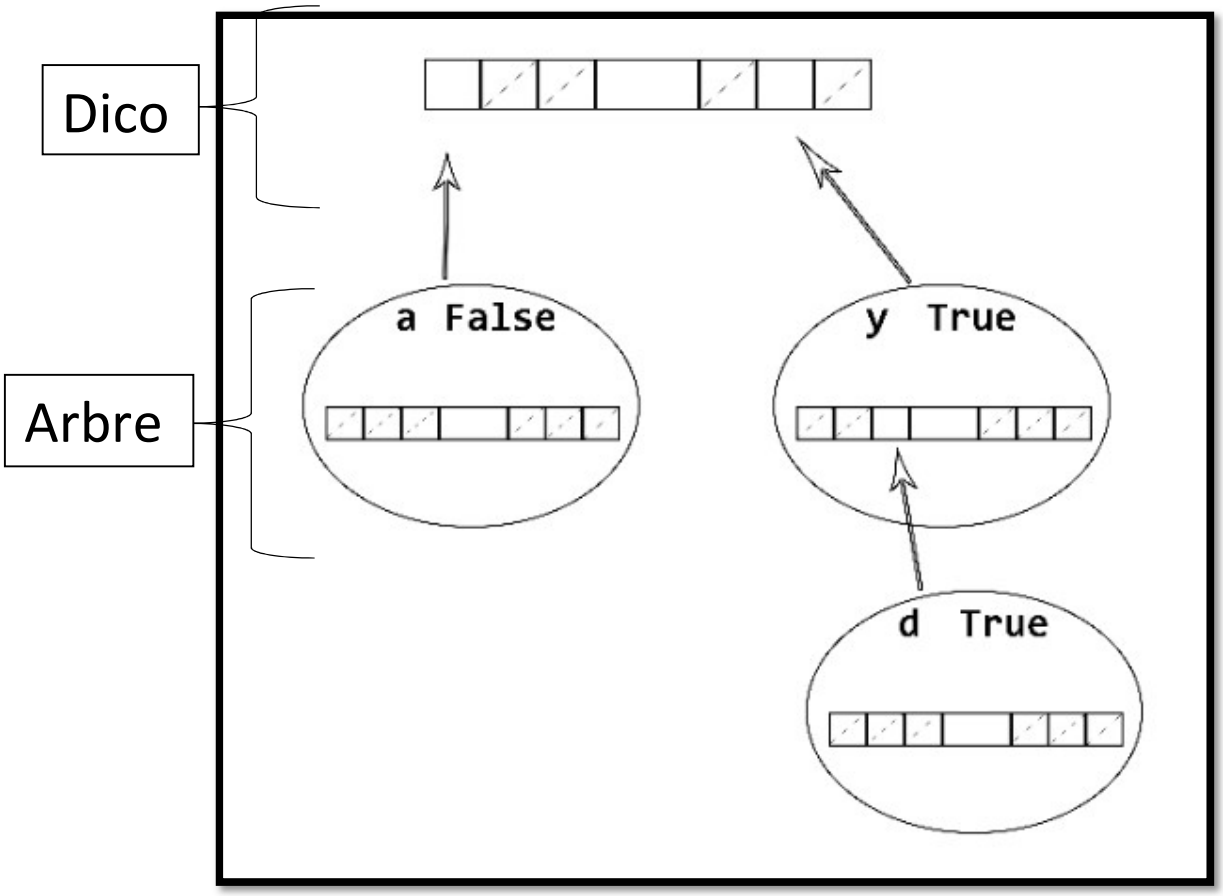
La structure globale du dictionnaire comprend deux sous structures :

- La première structure, *Dico*, est un tableau de taille 27 de pointeurs sur la seconde structure, *Arbre*. Elle comprend donc une case par lettre de l'alphabet et une pour l'apostrophe.
- La seconde structure, *Arbre*, contient un caractère, un booléen et un tableau de taille 27 de pointeurs sur sa propre structure. Le caractère permet de connaître la lettre actuelle. Le booléen est un indicateur de fin de mot, il a pour valeur *True* si le caractère est une fin de mot et *False* sinon. Enfin, le tableau permet de pointer sur la structure suivante si elle existe, sinon sa valeur est *NULL*.

J'ai d'abord pensé à utiliser une seule structure, *Arbre*, pour modéliser le dictionnaire, mais il me fallait un point de départ lors de l'implémentation d'un mot. Dans ce cas, il aurait fallu que j'attribue un caractère initial par défaut.

Finalement, j'ai préféré utiliser les deux structures présentées ci-dessus afin de représenter au mieux un dictionnaire.

Dessin



Initialisation

Lors de l'initialisation du dictionnaire, chaque pointeur de la structure *Dico* prend comme valeur *NULL*. Ainsi, dans la suite du programme, seul les structures arbres seront à créer.

Ci-dessous une structure dico dont toutes ses cases, pointeurs, sont nuls.

Fonctions basiques

Calcul de case

Le but de cette fonction est d'ordonner tous les tableaux afin de connaître l'emplacement d'un caractère donné. Elle retourne donc un entier représentant la place du caractère dans le tableau de pointeurs. Les cases du tableau sont numérotées de 0 à 26, et nous utilisons la table du code ASCII pour effectuer nos calculs et retrouver les caractères.

Dans cette fonction, nous considérons trois possibilités pour la recherche de notre caractère : une minuscule, une majuscule ou bien l'apostrophe. Le premier cas est le plus simple car il consiste à renvoyer la différence entre le code ASCII correspondant au caractère 'a' avec celui de notre caractère. De même pour le deuxième cas, cette fois ci avec le code du caractère 'A'. Dans le dernier cas, elle renvoie automatiquement l'entier 26 correspondant à la dernière case du tableau.

Mot_existe

Cette fonction va tester l'existence d'un mot dans un dictionnaire. Elle a besoin d'un mot, d'un entier et d'un arbre, et va retourner un booléen de valeur *True* si le mot existe, sinon *False*.

Cette fonction est récursive et teste si le caractère suivant est une fin de chaine afin de retourner la valeur de *finmot* de l'arbre courant. L'entier est utilisé afin de savoir sur quel caractère on est placé, il est donc incrémenté à chaque appel de fonction.

Création arbre

Cette fonction est très importante dans la phase de l'implémentation de mot dans le dictionnaire. Cette fonction va créer un arbre. Son caractère va prendre comme valeur le caractère entré en paramètre. Puis initialise ses pointeurs d'arbre à *NULL*. Elle va modifier le pointeur de pointeur de l'arbre entré en paramètre afin de modifier notre arbre courant.

Ajout mot

Cette fonction prend en paramètre un arbre ainsi qu'un mot. Elle va parcourir les arbres existants et faire appel à *Création arbre* pour les arbres *NULL* rencontrées nécessaires à l'implémentation du mot. Elle va aussi tester le caractère suivant pour savoir si c'est une fin de chaine et ainsi changer la valeur de *finmot* à *TRUE*.

Charger dico

Cette fonction va parcourir un fichier jusqu'à la fin, au caractère : EOF, et elle lit toutes les chaines de caractères, c'est-à-dire les mots qui composent le fichier. Et elle fait appel à la fonction *ajout mot* pour implémenter notre dictionnaire.

Free_dico et *Free_arbre*

Ces fonctions sont très importantes car nous avons choisi de travailler avec des pointeurs et nous devons ainsi rendre disponible les espaces mémoire alloués à la fin du programme. Puisque nous avons choisi d'utiliser deux sous structures d'arbres, nous devons utiliser deux fonctions différentes

afin de libérer le dictionnaire dans sa totalité. Premièrement, *Free_arbre* teste si l'arbre est différent de *NULL*. Dans ce cas, la fonction parcourt toutes les branches de l'arbre de manière récursive afin de libérer tous les pointeurs différents de *NULL*. Enfin, *Free_dico* fait appel à *Free_arbre* dans le but de libérer tous ses pointeurs d'arbre.

Affiche Dico/ Affiche Arbre

Ces deux fonctions ont seulement été utiles lors de la réalisation du projet afin de tester les résultats des autres fonctions. Cependant, elles sont inutiles dans la version finale du projet car il n'est jamais demandé de réaliser ces affichages.

Analyse Fichier

Cette fonction prend pour paramètre un nom de fichier et modifie les deux pointeurs d'entier en paramètre. Elle parcourt le fichier correspondant dans le but de comptabiliser, à l'aide de la fonction *Mot Existe*, le nombre total de mots existants et de mots faux.

Makefile

Le Makefile compile les fichiers *arbre.c*, *arbre.h* et le *main.c* qui permet d'exécuter le cœur du programme. Comme nous l'avons vu en cours et en TP, j'ai choisi de créer un Makefile puisqu'il est très utile pour faciliter la compilation des fichiers et m'a permis de gagner beaucoup de temps lors de la phase de programmation. J'ai décidé de réaliser mon code en créant des fichiers '.h' pour m'entraîner à utiliser cette structure de programmation.

Main.c

Le fichier 'main.c' contient toutes les étapes de la réalisation du projet en prenant en compte les différentes possibilités associées.

Dans un premier temps, il vérifie l'existence des deux arguments nécessaires à la bonne exécution du programme, à savoir les noms du dictionnaire et du fichier à analyser. Dans le cas où les arguments sont vides lors de l'exécution du programme, une fonction *scanf* permet de les récupérer en sollicitant l'utilisateur. S'il renvoie ''no'' ou bien un nom de dictionnaire inconnu, alors le programme charge le dictionnaire par défaut. Puis, un second *scanf* récupère le nom du fichier à analyser.

Le main est structuré en différentes fonctions locales au fichier Main.c, car il effectue de l'affichage console à l'aide de *printf*, ainsi il est adaptable à différente technologie. Il faudra simplement modifier les fonctions mais la structure restera la même.

Il y a une fonction qui permet de vider le Buffer, car quand nous sommes dans le cas où aucun fichier n'est passé en paramètre, j'effectue un *scanf* afin de récupérer les fichiers manquants à une bonne exécution. Alors le buffer ne sera pas vide, et le g*etchar()* ne fonctionnera pas.

Enfin, si l'exécution s'est correctement lancée, le programme affiche le nombre total de mots contenus dans le fichier, le nombre de mots justes ainsi que celui de mots faux. A noter que le correcteur effectue ses tests par rapport au dictionnaire entré par l'utilisateur. Ainsi, un même fichier peut obtenir des ratios de mots correctement orthographiés différents selon le dictionnaire de référence ajouté par l'utilisateur.

Conclusion

L'objectif de ce projet est de réaliser un correcteur orthographique en utilisant une structure de données de type : arbre. Cet objectif a été réalisé avec succès tout en optimisant la rapidité ainsi que l'espace mémoire. Les choix faits au début du projet ont été concluants et malgré quelques difficultés tout fonctionne.

Pour conclure, ces choix m'ont permis de finaliser ce projet en accomplissant mes objectifs, de rapidité et d'espace. Ce projet m'a donné l'occasion de travailler seul et ainsi prendre confiance en mon code et mes capacités. J'ai dû approfondir certains points lors de ce projet, qui m'ont permis de mieux comprendre certains aspects techniques, comme le buffer.

Pour améliorer ce projet, la prise en compte de tous les caractères, accents, est envisageable. De plus nous pourrions offrir un plus large choix de dictionnaire par default, notamment en proposant différentes langues.

Pour une simplicité d'utilisation, une interface graphique aurait été préférable pour l'utilisateur. Une programmation orientée objet est envisageable pour répondre à ces exigences.