Maxime Fontaine
Maxime Claudel

Compte-rendu de projet de programmation avancée

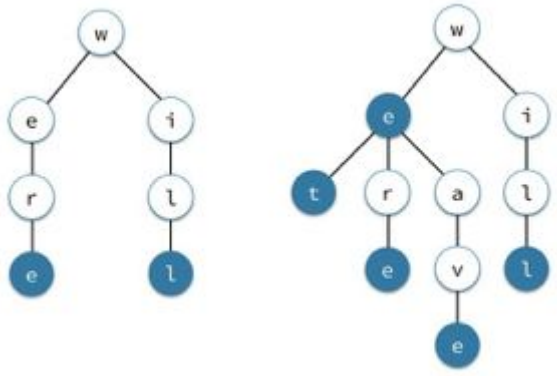
Réalisation d'un correcteur orthographique

- Objectif du projet

L'objectif du projet est de trouver une structure de données adaptée pour ensuite implémenter un détecteur de fautes d'orthographes. Le programme doit ainsi être capable de compter le nombre de mots mal orthographiés dans un texte en le comparant avec un dictionnaire.

- Choix de la structure de données

Nous avons choisi de travailler avec une structure d'arbre ordonné (struct node) composée d'un char l qui représente les différentes lettres d'un certain mot, d'un tableau de pointeurs qui redirige vers la même structure d'arbre et un d'un booléen qui indique une fin de mot.
En ce qui concerne la taille du tableau de pointeurs nous étions partis sur un tableau de 27 pointeurs. Avec ce choix nous étions ainsi limité en ce qui concerne l'emplacement des caractères spéciaux. En effet certains caractères ne pouvaient être comparés à l'aide de leur code ASCII ainsi, nous nous sommes dirigés vers un tableau de 255 pointeurs pouvant placer chaque lettre à l'emplacement désigné par son code ASCII.
Ceci nous permet donc de construire un arbre en fonction du dictionnaire rentré en argument de manière à ce que chaque caractère d'un mot pointe sur le caractère suivant. L'arbre est ainsi construit d'une manière très similaire à celle décrite dans le sujet du projet :



La seule différence avec cet exemple est que nous avons décidé de ranger les lettres d'un même étage par ordre alphabétique de gauche à droite (grâce à notre fonction ajout_alphab décrite plus tard).
Enfin, le booléen fin_de_mot permet au programme de détecter la dernière lettre d'un mot et de passer ensuite au prochain mot à analyser.

- Structuration du programme

Le code du programme est principalement organisé en deux grandes parties : les fonctions qui s'occupent du chargement de l'arbre de dictionnaire et les fonctions qui gèrent l'analyse des mots et la détection de fautes d'orthographe.
On retrouve dans la première partie des fonctions "classiques" (mais adaptées à notre structure et au problème posé) d'algorithmes utilisant des arbres telles que is_empty, ajout, affichage_arbre ou encore detruire_arbre pour gérer les potentielles fuites mémoire.
La deuxième partie est composée de deux fonctions, comparaison et test_erreur. Cette dernière utilise la fonction récursive comparaison pour rendre le résultat final du programme.

- Description des fonctions principales

La première partie sur laquelle nous avons travaillé est la partie chargement de l'arbre. Afin de réaliser cette partie nous avons utilisé plusieurs fonctions comme décrit précédemment. La fonction charger_arbre utilise principalement les fonctions ajout_alphab et ajout. En effet dans cette fonction on parcourt tout notre dictionnaire en format .txt et a chaque fois que l'on détecte un mot on appelle la fonction ajout_alphab. Cette dernière fonction nous permet d'ajouter au fur et à mesure les lettres d'un même mot passé en argument. Pour ce faire on parcourt toutes les lettres du mot et on le positionne à sa place, c'est à dire qu'on récupère son code ASCII et sa position dans notre tableau est ainsi définie. La seule exception étant pour les majuscules que nous transformons en minuscules notamment afin de simplifier l'identification de possibles erreurs dans la deuxième partie du programme. La fonction ajout permet quant à elle de créer une structure si la case située à la position de la lettre n'est pas occupée. Pour finir, lorsque nous détectons le caractère de fin de mot nous mettons la valeur du booléen fin_de_mot à 1.

En ce qui concerne la deuxième partie, le fonctionnement global des algorithmes est le même que dans la première partie, seulement, cette fois-ci, nous vérifions que toutes les lettres du mot sont présentes dans l'arbre chargé précédemment dans le bon ordre jusqu'à ce que le caractère fin_de_mot soit présent au même endroit que dans le mot comparé.

De plus notre programme fonctionne également avec les caractères spéciaux tels que les accents par exemple, mais il détecte aussi les virgules et points situés juste après le mot. Pour les points virgules, points d'exclamations, deux points et points d'interrogations nous avons fait le choix de considérer que ceux-ci sont espacés de part et d'autre d'un espace par rapport aux mots suivant et précédent.

Une fonction affichage_arbre à également était réalisée. Cette fonction affichage nous affiche l'arbre chargé à partir du dictionnaire fourni en affichant les mots de l'arbre en remontant à chaque fois jusqu'à la dernière lettre commune aux mots. Voici un exemple de l'arbre chargé avec les mots Pomme, Poire, Kiwi et Fraise :

```
? -> F
F -> r
r -> a
a -> i
i -> s
s -> e
fin de mot
? -> K
K -> i
i -> w
w -> i
fin de mot
? -> P
P -> o
o -> i
i -> r
r -> e
fin de mot
o -> m
m -> m
m -> e
fin de mot
```

Enfin si le programme remarque que plusieurs mots n'apparaissent pas dans le dictionnaire il affichera le nombre de mots manquant sous forme d'erreur par exemple si 2 mots manquent le code renvoie "2 erreurs".

- Limites de la solution apportée au problème et conclusion

Ce programme permet donc de détecter des fautes d'orthographe dans un texte de manière assez efficace, les temps d'exécution tournant globalement toujours autour de la seconde.
Cependant, notre programme ne nous permet pas de tester des textes écrits dans un alphabet autre que latin (arabe, mandarin, etc.). Ceci pourrait donc être un axe de progression futur. Un autre point de progrès pourrait aussi être de signaler à l'utilisateur les endroits des fautes dans le texte.