Xavier Luong

ZAVERIO Lucas

RAPPORT DM

Correcteur orthographique

Sommaire

ETAPE 1

I) Introduction

Ce DM à pour but d'implémenter un mini correcteur orthographique. Dans l'étape 1 le but est de charger en mémoire un dictionnaire servant de référence, puis de lister les mots mal orthographiés d'un texte relativement au dictionnaire chargé.

II) Ecriture du programme

Module Liste

Cellule * allouer_Cellule(char * mot);

Fonction servant à allouer la place mémoire nécessaire à la liste.

int inserer_en_tete(Liste * L, char * mot);

Fonction servant à insérer une valeur en tête de liste.

void liberer_Liste(Liste * L);

Fonction servant à libérer l'espace mémoire utilisée par la liste.

void afficher Liste(Liste L);

Fonction servant à afficher la liste.

Module ATR

ATR creer_ATR_vide()

Fonction servant à créer un Arbre ternaire de recherche vide.
Cette fonction alloue la mémoire nécessaire en même temps.

void liberer_ATR(ATR * A)

Fonction servant à libérer l'espace mémoire utilisé par l'arbre ternaire de recherche.

int inserer_dans_ATR(ATR * A, char * mot)

Fonction servant à insérer un mot dans l'arbre ternaire de recherche.

void supprimer_dans_ATR(ATR * A, char * mot)

Fonction servant à supprimer un mot de l'arbre ternaire de recherche.

void affiche_ATR_auxiliaire(ATR A, char * mot, int i)

Fonction auxiliaire à affiche_ATR.

void afficher_ATR(ATR A)

Fonction servant à afficher les mots présents dans l'arbre ternaire de recherche.

void affiche_ATR_auxiliaire(ATR A, char *)

Fonction auxiliaire à la fonction affiche ATR

int dans_ATR(ATR A, char * mot)

Fonction servant à vérifier si un mot du texte est dans le dictionnaire.

Main (correcteur_0.c)

Nous avons utilisé l'algorithme proposé dans le sujet pour rechercher les mots mal orthographiés.

III) Difficulté rencontrées

Nous avons eu beaucoup de mal sur la fonction dans_ATR car nous avons utilisé une ancienne version de la fonction inserer_dans_ATR qui était trop compliqué ce qui nous avait embrouillé. Nous avons donc refait notre fonction inserer_dans_ATR ce qui nous permet de réussir à faire la fonction dans_ATR.

Nous avons également du mal sur la fonction afficher_ATR car nous ne réussissions pas à afficher tous les mots, puis nous avons eu l'idée d'utiliser une fonction auxiliaire.

Nous n'avons pas réussi à faire la fonction supprimer_dans_ATR, elle n'est cependant pas utile dans la l'étape 1, nous essayerons donc de la faire dans l'étape 2.

IV) Conclusion

Nous n'avons pas eu de difficultés à faire les fonctions du modules Listes, car ce sont des fonctions vu en cours plusieurs fois.

Cependant le module ATR nous a donné beaucoup de mal car nous n'avions pas manipulé beaucoup d'arbre ternaire, nous avons tout de même réussi à faire l'étape 1 du sujet.

Compilation

Pour compiler le programme il vous suffit de vous rendre dans le dossier source, d'ouvrir un terminal à partir de ce dossier, puis d'entrer la commande : **make**

Vous pouvez aussi utiliser la commande : **make mrproprer**
qui permet de supprimer tous les .o et l'exécutable générés par le Makefile.

Exécution

Pour exécuter le programme (après l'avoir compilé bien-sûr), il vous suffit d'entrer la commande : **./correcteur_0 a_corriger_0.txt dico_1.dico**

Pour tester le programme avec un autre dictionnaire et un autre texte, il vous suffit de mettre votre dictionnaire et votre texte dans le dossier **src** puis de remplacer les arguments "a_corriger_0.txt" et "dico_1.dico" lors de l'exécution par le nom de votre texte et de votre dictionnaire.

ETAPE 2

I) Introduction

Dans l'étape 2 le but est de trouver la correction des mots mal orthographiés à l'aide de l'algorithme de Levenshtein.

II) Ecriture du programme

Tout ce qui était déjà présent à l'étape 1

Module Levenshtein

Le module Levenshtein comporte une seule fonction, la fonction Levenshtein :

```
int Levenshtein(char * un, char * deux);
```

Fonction servant à calculer la distance de Levenshtein entre deux mots.

Fonction ajoutés

```
Liste parcourir(FILE * f, char * mot);
```

Fonction qui prend un mot mal orthographié et calcule sa distance pour chaque mot du dictionnaire. (Fonction ajouter dans le module Liste)

Main (correcteur_1.c)

Nous avons utilisé l'algorithme proposé dans le sujet pour rechercher la la correction des mots mal orthographiés.
Cet algorithme utilise l'algorithme de Levenshtein et propose une correction des mots mal orthographiés en fonction de la distance de Levenshtein entre le mot mal orthographié et les mots présents dans le dictionnaire.

III) Difficulté rencontrées

Nous avons eu du mal à comprendre l'algorithme de Levenshtein mais nous avons finalement pu le faire.

IV) Conclusion

Cette partie du projet était moins compliquée que l'étape une. Nous avons réussi à faire tout ce qui était demandé.

Compilation

Pour compiler le programme il vous suffit de vous rendre dans le dossier source, d'ouvrir un terminal à partir de ce dossier, puis d'entrer la commande :

make correcteur_1

Vous pouvez aussi utiliser la commande : **make mrproper**
qui permet de supprimer tous les .o et l'exécutable générés par le Makefile.

Exécution

Pour exécuter le programme (après l'avoir compiler bien-sûr), il vous suffit d'entrer la commande : **./correcteur_1 a_corriger_0.txt dico_1.dico**

Pour tester le programme avec un autre dictionnaire et un autre texte, il vous suffit de mettre votre dictionnaire et votre texte dans le dossier **src** puis de remplacer les arguments "texte_0.txt" et "dico_1.dico" lors de l'exécution par le nom de votre dictionnaire et de votre texte.

ETAPE 3

I) Introduction

Dans l'étape 3 le but est de trouver la correction des mots mal orthographiés à l'aide de l'algorithme de l'arbre de Burkhard and Keller .

II) Ecriture du programme

Tout ce qui était déjà présent à l'étape 1 et l'étape 2

Module ArbreBK

int inserer_dans_ArbreBK(ArbreBK * A, char * mot);

Fonction servant à insérer un mot au bon endroit dans un arbre BK.

Liste rechercher_dans_ArbreBK(ArbreBK A, char * mot);

Fonction servant à rechercher les mots les plus proches d'un mot.

void recherche_aux(ArbreBK A, Liste * L, char * mot, int *seuil);

Fonction auxiliaire de la fonction rechercher_dans_ArbreBK.

char * prendre_mot(FILE * dico, int * ok);

Fonction servant à lire un mot dans un texte.

ArbreBK creer_ArbreBK(File * dico);

Fonction servant à créer un arbre BK à partir d'un fichier.

void liberer_ArbreBK(ArbreBK * A);

Fonction servant à libérer un arbre BK.

void afficher_ArbreBK(ArbreBK A);

Fonction affichant l'arbre BK.

dans_arbre(arbreBK, char * mot);

Fonction vérifiant si un mot est dans l'arbre BK.

III) Difficulté rencontrées

La fonction inserer_dans_arbreBK nous a posé quelques problèmes, nous avons cependant réussi à la faire.

IV) Conclusion

Nous avons réussi à obtenir le bon affichage de l'arbre BK ainsi que les bonnes corrections pour les mots, cependant nous nous sommes aperçu que pour une correction plus poussé il faudrait prendre en compte le contexte (exemple : pour "vil" la correction est "il" cependant on voudrait plutôt avoir "ville"), c'est pourquoi la distance de Levenshtein est un très bon indicateur mais qui à tout de même ses limites.

Compilation

Pour compiler le programme il vous suffit de vous rendre dans le dossier source, d'ouvrir un terminal à partir de ce dossier, puis d'entrer la commande :

make correcteur_2

Vous pouvez aussi utiliser la commande : **make mrproper**
qui permet de supprimer tous les .o et l'exécutable générés par le Makefile.

Exécution

Pour exécuter le programme, il faut entrer dans le terminal :

./correcteur_2 a_corriger_1.txt dico_2.dico

Pour tester le programme avec un autre dictionnaire et un autre texte, il vous suffit de mettre votre dictionnaire et votre texte dans le dossier **src** puis de remplacer les arguments "a_corriger_1.txt" et "dico_2.dico" lors de l'exécution par le nom de votre dictionnaire et de votre texte.