Manar Nssair & Nguyen Sarah

DM - Correcteur orthographique

Étape 3 : Arbres BK

Mode d'emploi :

- Dans le terminal, utiliser la commande `make` pour compiler le projet.
- Pour lancer le jeu avec le fichier texte à corriger (exemple : a_corriger_1.txt) et le dictionnaire de support (exemple : dico_3.dico),

  écrivez `./correcteur_2 a_corriger_1.txt dico_2.dico` .

Modules

Listes

- **Cellule * allouer_Cellule(char * mot)** : Alloue une cellule où l'on y met mot et la retourne.
- **int inserer_en_tete(Liste * L, char * mot)** : Insère en entête une nouvelle cellule dans Liste.
- **void liberer_Liste(Liste * L)** : Libère les nœuds alloués dans Liste.
- **void afficher_Liste(Liste L)** : Affiche les mots présents dans Liste.

ATR

- **ATR creer_ATR_vide()** : Alloue un arbre ternaire de recherche vide et la renvoie.
- **void liberer_ATR(ATR * A)** : Libère les _ATR alloués dans ATR.
- **ajoute_branche(ATR * A, char * mot) :** Ajoute une nouvelle branche à ATR contenant le nouveau mot.
- **int inserer_dans_ATR(ATR * A, char * mot)** : Insère dans ATR un nouveau mot.
- **void supprimer_dans_ATR(ATR * A, char * mot)** : Supprime le mot entré en paramètre dans ATR.
- **void afficher_aux(ATR A, char buffer[], int i)** : Affiche les mots présents dans ATR.

- **void afficher_ATR(ATR A)** : Initialise les valeurs buffer et i pour l'appel de la fonction afficher_aux().
- **int remplir_ATR(FILE * df, ATR * A)** : Rempli l'ATR A avec les mots du fichier.
- **int appartient_dico(ATR t, char * mot)** : Verifie si le mot appartient au dico (s'il est présent dans les mots pouvant être formés dans l'ATR t). S'il y appartient il renverra 1, et dans le cas contraire 0 .

Levenshtein

- **int min(int a, int b)** : Renvoie le plus petit entier entre a et b.
- **int max(int a, int b)** : Renvoie le plus grand entier entre a et b.
- **int Levenshtein(char * un, char * deux)** : calcule la distance de Levenshtein entre les mots un et deux.

Arbre BK

- **ArbreBK creer_ArbreBK_vide(char * mot, int valeur)** : Alloue un arbre BK comportant son mot et sa valeur entrés en paramètre, initialise filsG et frereG à NULL et puis la renvoie.
- **int inserer_dans_ArbreBK(ArbreBK * A, char * mot)** : Insère dans l'arbre BK le nouveau mot.

  Pour cela, dans le cas où il n'existe encore aucun nœud dans l'arbre, il suffit juste de le créer, celui-ci aura par défaut la valeur 0.

  Dans la cas contraire, on calcule sa valeur à l'aide de la fonction de Leveinshtein. On va alors chercher s'il existe un nœud qui est égal à la valeur de Leveinshtein pour se déplacer dans l'arbre et trouver le bon embranchement. On peut alors créer un nouveau nœud qui si celui-ci est plus petit que le nœud du fils gauche, deviendra le nouveau fils gauche et déplacera l'autre dans le fils droit. Dans le cas contraire, on cherche son emplacement parmi les fils droits (rangés par ordre croissant pour leur valeur) et de l'ajouter à cet endroit en décalant les fils droits situés après.
- **ArbreBK creer_ArbreBK(FILE * dico)** : Créer l'arbre BK contenant tous les mots du fichier entré en paramètre en appelant la fonction inserer_dans_ArbreBK().
- **void afficher_ArbreBK_aux(ArbreBK A, int niveau)** : Affiche chaque nœud de l'arbre BK ainsi que sa valeur en prenant en compte son niveau pour effectuer les espaces nécessaires à l'affichage.

- **void afficher_ArbreBK(ArbreBK A)** : Affiche l'arbre BK en appelant la fonction afficher_ArbreBK_aux() initialisé avec le niveau 0.
- **void rechercher_dans_ArbreBK_aux(ArbreBK A, char * m, Liste * corrections, int * seuil)** : Remplie la liste corrections en paramètre avec les mots présent dans A au plus distant de seuil par rapport au mot m. À chaque fois qu'on trouve un mot de distance inférieur à seuil, on vide la liste et on lui ajoute seulement ce nouveau mot, et on met à jour seuil. D'après une propriété du cours, on ne parcourt pas les sous-arbre dont la la valeur absolue de la distance moins la valuation est supérieur au seuil car ils ne contiennent pas de mots au plus à distance seuil de m.
- **Liste rechercher_dans_ArbreBK(ArbreBK A, char * mot)** : Initialise un entier seuil à 2 et une liste corrections à null pour appeler la fonction rechercher_dans_ArbreBK_aux et renvoyer la liste après cet appel.
- **void liberer_ArbreBK(ArbreBK * A)** : Libère les nœuds alloués dans l'arbre BK.

correcteur_0

- **FILE * ouvre_fichier(const char * chemin)** : Ouvre le ficher avec le mode d'accès choisi.
- **Liste algo_1(FILE * df, ATR * A)** : (Implémente l'algorithme 1 de détection de mots mal orthographiés) Renvoie la liste des mots contenant des erreurs orthographiques (mots non présents dans le dictionnaire).

correcteur_1

- **FILE * ouvre_fichier(const char * chemin)** : Ouvre le ficher avec le mode d'accès choisi.
- **Liste algo_1(FILE * df, ATR * A)** : (Implémente l'algorithme 1 de détection de mots mal orthographiés) Renvoie la liste des mots contenant des erreurs orthographiques (mots non présents dans le dictionnaire).
- **Liste algo_2(FILE * df, char * mot)** : (Implémente l'algorithme 2 de détection de mots mal orthographiés par un texte de force brute) Renvoie la liste des mots de corrections proposés pour le mot entré en paramètre.

correcteur_2

- **FILE * ouvre_fichier(const char * chemin)** : Ouvre le ficher avec le mode d'accès choisi.
- **void algo_1_et_correction(FILE * df, ArbreBK * A)** : Implémente l'algorithme 1 de détection de mots mal orthographiés et affiche les corrections pour chaque mot mal orthographié. (vu que la correction est déjà faite et affichée dans la fonction, nous avons décidé de ne pas renvoyer la liste des mots à corriger vu que ce n'est plus nécessaire).

Conclusion

Étape 1 : Nous avons rencontré certaines difficultés lors du codage de la fonction supprimer. En effet comme nous ne possédons pas l'algorithme nécessaire pour celle ci, cela nous a pris plus de temps que prévu puisque nous avons du tester et prendre en compte les différents cas possibles des mots à supprimer en testant plusieurs algorithmes différents.

Pour la répartition des tâches nous avons répartis les différentes fonctions à coder, excepté pour la fonction supprimer où l'on a travaillé dessus ensemble. Pour les autres fonctions, nous nous sommes entraidées quand nous étions bloquées.

Étape 2 : L'étape n°2 était bien rapide et nous n'avons pas rencontré de difficulté à la faire. En effet nous avons simplement appliqué les deux algorithmes de l'énoncé. Pour la répartition du travail, une personne s'est occupé de la fonction Levenshtein et l'autre de celle de l'algorithme 2.

Étape 3 : Pour ce denier rendu, avons rencontré des difficultés pour la fonction d'insertion d'un arbre BK à cause des différentes cas à prendre en compte. En effet, il fallait prendre en compte le cas où il devait se brancher sur le fils gauche et décaler l'ancien nœud en frère droit, ou comparer sa valeur avec les autres nœuds, etc... . Nous avons alors au départ essayé de le coder en récursive mais n'avons pas réussi et au final,nous avons décidé de la coder en itérative ce qui était plus simple pour nous.