Marti Emilie – Sajdak Laurine

Rapport de projet

Arbre lexicographique

CONTENTS

Notre code est organisé comme suit :

- un dossier txt contenant les fichiers .txt pouvant être passés en argument dans notre console et dont le contenu va être traité, affiché, modifié par notre code
- un dossier bin dans lequel on va stocker les fichiers crées par notre programme
- un dossier headers contenant les en-têtes (fichiers .h) de chaque structures et fonctions du code
- à la racine, les fichiers .c ainsi que le Makefile permettant de compiler le code avec la commande « make ».

1 FILEMANAGER.C

1.1 CHAR* JOINSTRING(CHAR* LASTWORD, CHAR* FIRSTWORD)

Cette fonction permet de joindre deux chaînes de caractères et de retourner le résultat grâce à strcat. On va notamment s'en servir pour déterminer le chemin de sauvegarde ou d'ouverture de nos fichiers textes.

Cette fonction a été créée afin d'éviter les erreurs de segmentations.

1.2 VOID GETLEXICON(CHAR* FILENAME, CHAR* COMMANDE, TREE TREE, CHAR* WORD)

Cette fonction permet d'accéder à notre fichier .txt et d'en traiter les informations.

Elle prend en paramètre le nom du fichier déterminé par l'utilisateur dans la console, ainsi que la commande choisie qui détermine l'action à effectuer sur le fichier texte.

On stocke plusieurs chaînes de caractères qui vont déterminer dans quel dossier traiter nos fichiers (txt/ ou bin/) ainsi que le suffixe de nos fichiers (.DICO ou .L).

A l'aide du type FILE*, on va pouvoir accéder à un fichier externe et lire (-r) les données de notre fichier texte, updater ses informations (a+ et a)...

C'est dans cette fonction qu'on va notamment faire appel à createTree() et search() pour créer notre arbre et rechercher un mot spécifique.

On va aussi créer un fichier .DICO si l'utilisateur tape la commande -S ou afficher le contenu de notre arbre si c'est la commande -l.

1.3 INT GETCOMMANDINDEX(INT ARGC, CHAR** ARGV)

On vérifie si l'utilisateur a tapé une commande spécifique (et donc commençant par -) dans la console.

1.4 VOID RUNLEXICON(INT ARGC, CHAR** ARGV)

On vérifie la position de l'argument tapé par l'utilisateur dans la console. S'il n'y a aucun argument, on affiche un menu récapitulant toutes les commandes possibles et on fait appel à getLexicon pour mettre à jour nos fichiers textes et notre arbre selon les choix de l'utilisateur.

1.5 VOID RUNFROMDICO(CHAR** ARGV)

On va faire les mêmes opérations (création, lecture...) mais en créant l'arbre lexicographique directement depuis un fichier .DICO.

2 TREEMANAGER.C

2.1 TREE INITIALIZETREE(CHAR LETTER)

Cette fonction permet de créer notre arbre. On ferme directement le programme s'il y a une erreur d'allocation de mémoire.

2.2 INT CREATETREE(TREE* TREE, CHAR* WORD)

Cette fonction va permettre de créer notre arbre binaire (de forme fils gauche - frère droit) à partir d'un fichier .txt ou d'un mot décidé par l'utilisateur dans la console.

Pour cela, chaque mot est passé en paramètre en plus de l'arbre dans lequel on va les stocker.

On vérifie en premier si la lettre du mot est bien dans l'alphabet (c'est à dire n'est pas un caractère spécial comme un point, une virgule…). Si c'est un caractère interdit, on l'ignore et on fait un appel récursif en prenant en paramètre le caractère suivant du mot. Si on a affaire à une apostrophe, on la supprime et on colle les deux « parties » ensemble.

Si l'arbre ou l'emplacement dans lequel on veut placer notre caractère est vide, on le crée à l'aide de la fonction initializeTree. Sinon, on vérifie pour chaque lettre de notre mot si celle-ci existe déjà dans notre arbre ou si elle est plus grande ou plus petite que celle contenue dans le nœud notre arbre.

Si le mot ajouté est plus à gauche dans le dictionnaire, c'est qu'il faut l'insérer à gauche de notre arbre : on change alors la racine de l'arbre. Pour cela, on stocke notre ancien arbre dans oldTree, on crée un nouvel arbre contenant la première lettre de notre nouveau mot et on lui donne comme frère droit notre ancien arbre. Puis on passe à la lettre suivante en vérifiant à nouveau chaque condition, et ainsi de suite.

On renvoie 1 si la création a bien fonctionné, 0 sinon.

Ainsi, cette fonction permet de trier et d'enregistrer dans un arbre, de manière récursive, chaque mot de notre texte ou tapé par l'utilisateur.

2.3 INT SEARCH(TREE TREE, CHAR* WORD)

On cherche à savoir si un mot passé en paramètre et déterminé par l'utilisateur apparaît dans notre arbre. Pour cela, on compare chaque lettre de ce mot aux valeurs des nœuds de l'arbre. On renvoie 0 si le mot n'est pas dans l'arbre ou si l'arbre est NULL, 1 sinon.

Si la première lettre du mot est supérieur à la racine de l'arbre, on renvoie directement 0 : en effet, comme c'est un arbre trié alphabétiquement, la racine est la première lettre du mot la plus à gauche du dictionnaire. Donc, si on recherche le mot « arbre » et que le premier mot de l'arbre est « charcuterie », comme a > c, inutile de chercher dans les frères de droite.

Si la lettre du mot recherché est égale à celle de l'arbre, on cherche la lettre suivante dans le fils gauche, sinon on cherche cette même lettre dans le frère droit.

Et ainsi de suite, de manière récursive, on s'arrête si on a atteint la fin du mot recherché (=='\0') ou si on ne trouve pas la lettre du mot dans le frère ou le fils de l'arbre.

2.4 VOID PRINTWORD(TREE TREE, CHAR* BUFFER, INT INDEX)

On veut afficher chaque mot de l'arbre par ordre alphabétique.

Pour cela, on stocke chaque lettre l'une à la suite de l'autre dans un buffer qui affiche le mot entier dès qu'il rencontre une fin de mot (=='\0'). L'ordre4 est automatiquement alphabétique puisque notre arbre est trié.

C'est le même principe qu'un parcours préfixe.

2.5 VOID PRINTWORDFULL(TREE TREE)

Permet d'allouer la mémoire de notre buffer (dont on se sert dans la fonction décrite précédemment) en lui donnant une taille max de 51 char. Une fois cet espace alloué, on fait appel à la foncion printWord().

2.6 VOID PREFIXSAVEDICO(TREE TREE)

Cette fonction permet de sauvegarder les mots de notre arbre au format .DICO défini par le sujet.

C'est le même principe qu'un parcours préfixe sauf qu'on remplace les '\0' (donc, à chaque fois qu'on arrive à la fin d'un mot et que le fils gauche est NULL par conséquent) par un espace ' ' et que, pour chaque sous-arbre vide (on calcule le nombre de nœuds à remonter pour passer au frère droit), on marque un '\n'.

2.7 VOID CREATETREEFROMDICO(TREE* TREE, FILE* FILENAME)

On veut crée un arbre à partir de notre fichier .dico. Pour cela, on regarde la nature de chaque caractère : on ignore si on rencontre un '\n', on enregistre notre caractère dans le frère droit si c'est un '\0', dans le fils gauche par défaut.

Malheureusement, même si l'algorithme pour créer l'arbre depuis le fichier .DICO semble correct, l'arbre ne se construit pas ou bien ne se sauvegarde pas correctement ; on n'arrive donc pas à gérer le lexique à partir d'un fichier .DICO

3 LEXICON.C

Il s'agit du fichier principal contenant le main() et donc toutes les commandes principales servant à traiter notre lexique en fonction des choix de l'utilisateur et utilisant les fonctions définies dans les fichiers décrits précédemment. Pour plus de clarté, on récapitule sur la console les choix de l'utilisateur.

4 MAKEFILE

Afin de pouvoir organiser notre code dans plusieurs fichiers (selon le type de fonction) et faciliter la compilation, on a créé un Makefile avec les flags indiqués sur le sujet. Avec la simple commande « make » sur la console, les fichiers objets (*.o) vont être créés à partir des fichiers *.c, pour donner un exécutable final qui s'appellera tout simplement lexicon.

Certaines précisions sur le lancement du binaire et l'organisation des fichiers externes gérés seront également affichées sur la console lors du lancement du Makefile.

Finalement, les fichiers objet seront éliminés avec la commande « clean » afin de préserver un dossier organisé, sans documents inutiles.

Dossier trash : contient les fonctions que nous avons créées sans pour autant les utiliser dans notre code final.

5 USERMANAGER.C

5.1 CHAR* INITIALIZEUSERINPUT()

On alloue la mémoire pour un mot de taille max 51

5.2 CHAR* USERINPUT(CHAR* INPUT)

L'utilisateur rentre un mot de 51 lettres maximum. On vérifie ensuite à l'aide de la fonction fixInput si l'utilisateur a bien rentré un mot correct. Si le mot a été traité par cette fonction (on compare son état avant et après l'appel à la fonction), on prévient affiche le nouveau mot. Sinon, si le mot est le même avant et après traitement, on signale qu'il n'y a aucune erreur.

5.3 VOID FIXINPUT(CHAR* INPUT)

Cette fonction vérifie que l'utilisateur n'a pas rentré de caractères spéciaux. Sinon, elle traite le mot pour ne garder que les caractères corrects.