PROGRAMMATION
&
ALGORITHMIQUE

-

Projet “Arbres Lexicographiques”

—

ANIK Myriam - SÉNÉCAL Nicolas

INTRODUCTION

Un arbre lexicographique est un arbre binaire qui permet de traiter des caractères.
Cet outil est privilégié aux listes de mots et aux tableaux de par sa complexité, par son optimisation de la mémoire et de la rapidité du programme.

Il permet effectivement de travailler sur des lexiques et des dictionnaires de mots.
En passant par l'ajout, la recherche, la suppression …

A travers ce projet, il nous a été demandé de créer un programme permettant le tri de mots dans l'ordre alphabétique, de pouvoir sauvegarder ceux-ci dans des fichiers, ou encore de rechercher des mots parmi d'autres.

Etape 01 : Création des fichiers

Pour commencer, il a fallu créer les fichiers dont nous allions nous servir et de les relier avec des includes et un makefile permettant d'exécuter le tout.

Lexique.c : Fichier principale comportant la fonction main. Affiche le menu ou exécute les fonctions en accord avec les lignes de commandes.

arbres.h(.c) : Structure et fonctions de gestion d'arbre binaire lexical.

fichier.h(.c) : Fonctions de gestions (lecture et écriture) des fichiers (L, .DICO et .dot)

ui.h(.c) : Fonctions de gestion de l'interface utilisateurs (commandes et affichages)

Etape 02 : Les lignes de commandes

On commence donc par coder notre fichier Lexique qui comporte notre main.
On fait passer argc et *argv[] en paramètres, qui nous permetteront grace à une boucle for de prendre en considération les commandes saisies par l'utilisateur.

`-l nomdufichier` pour afficher les mots du lexique du fichier nom en ordre alphabétique
`-s nomdufichier` pour sauvegarder les mots du lexique du fichier nom en ordre alphabétique dans un fichier .L
`-S nomdufichier` pour sauvegarder l'arbre dans un fichier .DICO
`-r motàchercher nomdufichier` pour indiquer si Mot apparaît dans le fichier nom.
`-d nomdufichier` pour générer un .dot
`-h` pour pour afficher l'aide

Etape 03 : Écriture des structures / fonctions d'arbres

Définition du nombre de caractère maximal par mot (51 avec un dernier caractère correspondant au \0) : **`#DEFINE MAX_CHAR 51`**

Définition de la Structure de base d'un Noeud dans le fichier arbre.h :

```
typedef struct noeud
{
      unsigned char lettre;
      struct noeuf *filsg,*frered;
}Noeud, *Arbre;
```

Définition des fonctions dans le fichier arbre.h et code de celles-ci dans le fichier arbre.c :

`Arbre alloueNoeud(unsigned char lettre)` : Permet d'Allouer dynamiquement de la mémoire à un Noeud. Avec en paramètre un caractère vu que c'est un arbre lexicographique.

`void ajouteBranche(Arbre *a, char *mot)` : Ajoute un mot sous forme d'une liste de fils gauche, s'arrête quand rencontre un '\0'

`void ajouteMot(Arbre *a, char *mot)` : Permet d'ajouter un mot dans l'arbre selon le cas du mot précédent.

- Si l'arbre est vide, le mot est ajouté sous forme d'une liste de fils (une branche)
- Si la lettre du noeud est plus petite que celle du mot on ajoute le mot dans le frère
- Si la lettre est trouvée et que la fin du mot n'est pas atteinte, on ajoute le reste du mot dans le fils
- Si la lettre du noeud est supérieure à celle du mot, on crée une branche et on l'ajoute avant ce noeud

`void freeArbre(Arbre *a)` : Permet de libérer l'espace mémoire alloué à un arbre.

`int recherche(Arbre *a, char *mot) :` Vérifie l'existence d'un mot dans un arbre.

`void afficheDico(Arbre a)` : Fonction permettant d'afficher les mots.

`void afficheDicoAux(Arbre a, char *buffer, int indice)` : Récursivité de afficheDico

Etape 04 : Écriture des fonctions de gestion des fichiers

`Arbre genereLexiqueTexte(char *nom)` : Fonction qui ouvre le fichier et décompose en mots.

`int genereArbreTexte(Arbre *a, char *nomIn)` : Construit un arbre lexical depuis un fichier texte

`int genereArbreDico(Arbre *a, char *nomIn)` : Construit un arbre lexical depuis un fichier .DICO

`int sauvegardeLexique(Arbre a, char *nomOut)` : Fonction permettant la sauvegarde du lexique dans un fichier

`void sauvegardeLexiqueAux(Arbre a, FILE *out, unsigned char *buffer, int indice)` : Fonction auxiliaire permettant la sauvegarde du texte grace à sauvegardeLexiqueTexte

`int sauvegardeArbre(Arbre a, char *nomOut)` : Fonction permettant la sauvegarde du lexique dans un fichier

`void sauvegardeArbreAux(Arbre a, FILE *out)` : Fonction auxiliaire permettant la sauvegarde du lexique dans un fichier

`int genereDot(Arbre a, char *nomOut)` : Fonction permettant de générer un fichier dot de l'arbre lexicographique

`void ecrireArbreDot(FILE *stream, Arbre a)` : Fonction permettant d'écrire les lignes correspondant a l'arbre dans le fichier DOT.

Améliorations ajoutées :

Modification de l'arbre :

- On sauvegarde le nombre d'occurrences des mots dans l'arbre, au niveau de ses racines ('\0').

Modification des sorties:

- Ajout de la commande "-d" qui créer un fichier .dot qui permet de visualiser l'arbre.
- Ajout d'une aide avec un -h en commande, ou dans le menu.

Modification des entrées :

- On s'assure de l'existence du nom de fichier rentré et on demande de le retaper le cas échéant.
- Si aucun mot n'est renseigné pour la fonction recherche (" -r <mot> "), nous demandons de le rentrer.

Tests effectués :

- Usage de Valgrind pour vérifier la libération total de la mémoire à la fin du programme, pour chaque utilisation possible de celui-ci : *FONCTIONNELLE.*

- Vérification de chaque fonctionnalitées (avec et sans menu) pour un fichier avec peux de texte (10 mots simple) : *FONCTIONNELLE.*

- Idem avec des accents : *FONCTIONNELLE*, mais l'affichage dans la console dépend de la configuration de celle-ci.

- Vérification de chaque fonctionnalitées (avec et sans menu) pour un fichier avec beaucoup de texte (plus de 200 mots) : *FONCTIONNELLE.*

- Vérification de chaque fonctionnalitées (avec et sans menu) pour un fichier vide : *FONCTIONNELLE*, la console informe que le fichier est vide et propose de quitter ou d'en rentrer un nouveau.