Julien PILLIS

Rapport TP4-NF16 : Arbres Binaires de Recherche

I. Introduction :

L'objectif du TP 4 est de créer un programme qui permet d'indexer les mots d'un texte lu à partir d'un fichier. Pour cela, nous utiliserons les arbres binaires de recherche (mots triés selon l'ordre lexicographique), et tenterons d'optimiser au mieux l'indexation. L'ABR nous permettra de manipuler efficacement l'index pour pouvoir récupérer des informations relatives au texte et aux mots (occurrences, numéro de ligne, ordre…), mais aussi pouvoir reconstruire le texte dans un autre fichier.

II. Liste des structures :

Structure t_Position :

```
typedef struct t_position{
    int numero_ligne;
    int ordre;
    int numero_phrase;
    struct t_position* suivant;

}t_Position;
```

Structure t_Noeud:

```
typedef struct t_noeud{
    char* mot;
    int nb_occurences;
    t_ListePositions positions;
    struct t_noeud* filsGauche;
    struct t_noeud* filsDroit;
}t_Noeud;
```

Structure t_ListePositions :

```
typedef struct t_listePositions{
    t_Position* debut;
    int nb_elements;
}t_ListePositions;
```

Structure t_Index :

```
typedef struct t_index{
    t_Noeud* racine;
    int nb_mots_differents;
    int nb_mots_total;
    int nb_phrases; // attribut ajouté pour la construction du texte à partir de l'index
}t_Index;
```

Structures ajoutées

Structure t_Mot :

```
typedef struct t_mot{
    char *mot;
    int numero_ligne;
    int ordre;
    struct t_mot* suivant;
}t_Mot;
```

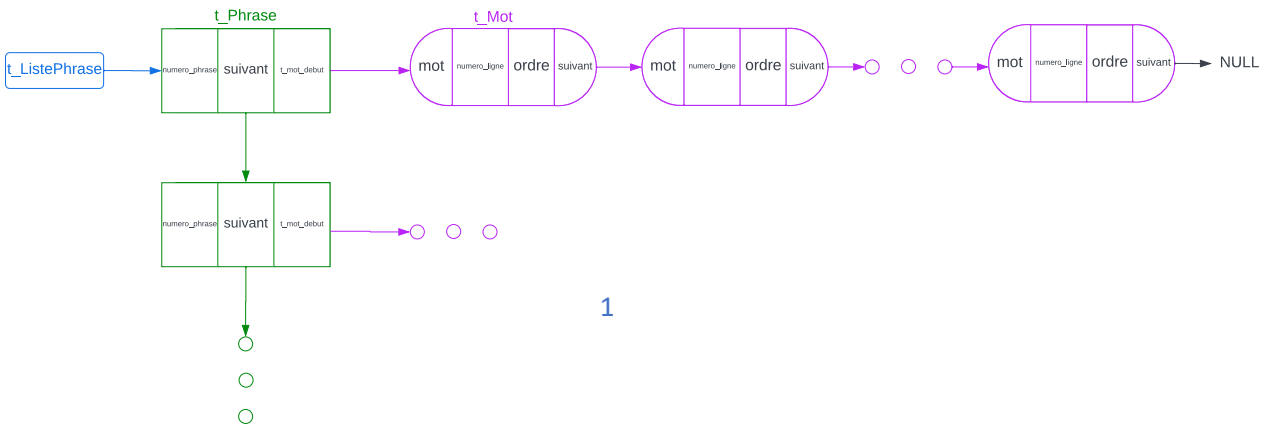
Structure t_Phrase:

```
typedef struct t_phrase{
    t_Mot *mot_debut;
    int numero_phrase;
    struct t_phrase*
suivant;
}t_Phrase;
```

Structure t_ListePhrase:

```
typedef t_Phrase *
Liste_phrase;
```

Schéma des strucutres de donnés supplémentaires



Nous avons fait le choix de rajouter 3 structures supplémentaires pour faciliter la reconstruction des phrases et du texte. En effet, cette disposition nous évite de relire plusieurs fois l'index pour reconstruire différentes phrases.

III. Fonctions supplémentaires ajoutées :

t_Noeud* creer_noeud(char *mot) : Cette fonction permet d'initialiser et de créer un nœud. Elle est utile car nous créons des nœuds à plusieurs endroits du programme (éviter d'allonger le code).

t_Mot* creer_mot() : : Cette fonction permet d'initialiser et de créer un mot. Elle est utile car nous créons des mots à plusieurs endroits du programme (éviter d'allonger le code).
t_Phrase* creer_phrase() : : Cette fonction permet d'initialiser et de créer une phrase. Elle est utile car nous créons des phrases à plusieurs endroits du programme (éviter d'allonger le code).

void afficher_mot(t_Mots l) : Cette fonction est pratique car elle permet de parcourir et d'afficher les mots d'une liste chaînée de mots composant une phrase.

void afficher_liste_positions(t_ListePositions *listeP) : Cette fonction est pratique car elle permet de parcourir et d'afficher les positions d'un nœud (utile lors de l'affichage de l'index).

int ajouter_mot(t_Mot ** tMot,int numero_ligne,int ordre, char* mot) : Cette fonction est utile car elle permet de rajouter un mot à la fin d'une liste chaînée de mots d'une phrase.

void ajouter_mots_phrase(t_Noeud *racine, Liste_phrase* liste_phrases) : Cette fonction est pratique car elle parcourt l'index (INFIXE), et ajoute le mot à la bonne phrase si son numéro correspond à une phrase présente dans la liste.

int ajouter_phrase(Liste_phrase * ListePhrases,int numero_phrase) : La fonction permet de créer et d'ajouter une phrase de numéro numero_phrase à la liste de phrases.

t_Phrase* recherche_phrase(t_Phrase* tPhrase,int numero_phrase) : La fonction est utile car elle permet de vérifier si une phrase de numéro numero_phrase est déjà présente dans la liste ou non. (On évite de créer 2 phrases de même numéro).

void transforme_to_min(char *str) : La fonction permet de transformer tous les caractères d'une chaîne en caractères minuscules.

void supprimer_caractere(char *str, char c) : La fonction supprimer le caractère c de la chaine str. Utile lors de la lecture du fichier pour supprimer les points et les sauts de ligne.

void parcours_affichage_index(t_Noeud *racine, char* c) : La fonction parcours les nœuds de l'index (INFIXE, pour conserver l'ordre lexicographique) et affiche les mots et leurs positions

void parcours_recherche_max(t_Noeud *racine, t_Noeud *nd) : La fonction cherche à nd le nœud du mot ayant la plus grande occurrence (PREFIXE). Utile pour le menu.

void libereNoeud(t_Noeud *racine), void libereIndex(t_Index *index), void libereListe(t_Position *liste), void libereMot(t_Mot* liste), void liberePhrase(t_Phrase *phrase) : permettent de libérer l'allocation mémoire des structures.

int hauteur(t_Noeud *racine) : Déterminer la hauteur de l'arbre associé au nœud.

int equilibre(t_Noeud *racine) : Détermine si un arbre est équilibré ou non.

IV. Complexité des fonctions implémentées :

t_ListePositions* creer_liste_positions() : La fonction ne fait qu'allouer la mémoire au nouvel objet et initialiser les attributs. Le nombre d'instructions est constant➔ **O(1)**

`int ajouter_position(t_ListePositions *listeP, int ligne, int ordre, int num_phrase)` : La fonction crée (O(1)) et ajoute une position dans une liste chaînée de position triée selon le numéro de ligne, l'ordre et le numéro de phrase. Dans le pire des cas, la position est à placer à la fin de la liste ➔ **O(n),** n : nombre de positions de la liste

`t_Index* creer_index()` : La fonction ne fait qu'allouer la mémoire au nouvel objet et initialiser les attributs. Le nombre d'instructions est constant➔ **O(1)**
`t_Noeud* rechercher_mot(t_Index *index, char *mot)` : La fonction permet de savoir si un mot se trouve déjà dans l'arbre. Pour cela elle suit le chemin en fonction de la lexicographie du mot. Pire des cas, le mot n'est pas dans l'arbre et on a atteint la feuille la plus basse ➔ **O(h),** h : la hauteur de l'index (arbre)

`int ajouter_noeud(t_Index *index, t_Noeud *noeud)` : La fonction ajoute un nœud dans l'arbre. Dans tous les cas, il faut vérifier si le mot n'appartient pas déjà à l'arbre (O(h)). S'il n'appartient pas à l'arbre, dans le pire des cas, il s'agit de la feuille la plus basse (O(h), h : la hauteur de l'arbre)). ➔ Règle du maximum : **O(h)**
int indexer_fichier(t_Index *index, char *filename) : Pour chaque ligne du fichier (m lignes) et pour chaque mot(n mots), on ajoute le nœud associé à l'arbre (O(h), h : la hauteur de l'arbre (index)) ➔ **O(m*n*h),** ici n représente le nombre de mot de la ligne la plus grande
`void afficher_index(t_Index *index)` : La fonction affiche tous les mots de l'index et leurs positions. Elle fait appel à parcours_affichage_index ➔ **O(n*m),** avec n le nombre de nœuds et m le plus grand nombre de positions associées à un nœud de l'index
`void afficher_max_apparition(t_Index *index)` : La fonction parcourt l'arbre (PREFIXE) pour trouver le nœud avec la plus grande occurrence. On est obligé de vérifier chaque nœud ➔ **O(n),** n : le nombre de nœuds ⇔ nombre d'appels récursifs imbriqués
void afficher_occurrences_mot(t_Index *index, char *mot) : On recherche le mot O(n), n : le nombre de nœuds de l'index). Si le mot est présent dans l'index, on récupère ses positions (O(p), p : le nombre de positions du mot et on ajoute à la liste de phrase l'objet de t_Phrase avec le numéro de phrase de la position (O(p), p le nombre de positions du mot>= nombre de phrases). Puis on fait appel à ajouter_mots_phrase (O(n*max(p+l), : l le nombre de mots dans la liste chaînée de la phrase associée). A ce moment, la complexité est O(n*max(p+l) d'après la règle du maximum. Enfin, on affiche chaque mot de chaque phrase (O(p*l)).
Ainsi la complexité de la fonction ➔ **O(max((p*l), n*max(p+l)))**

`void construire_texte(t_Index *index, char *filename)` : Pour chaque phrase, on crée un objet de type t_Phrase et on l'ajoute à la liste chaînée de phrases. ($O(m^2)$, m : le nombre de phrases). Puis, on parcourt l'index et ajoute tous les mots à leurs phrases correspondantes (ajouter_mots_phrase : O(n*max(m+l)), avec n le nombre de nœuds de la liste, l le nombre de mots dans la liste chaînée de la phrase associée.). Enfin, pour chaque mot de chaque phrase, on l'ajoute à la phrase que l'on va écrire sur le texte. => O(m*l). La règle du maximum ➔ **O(max(m^2, n*max(m+l), m*l)**

```
//-------------------FONCTIONS AJOUTEES--------------------------
```

`void ajouter_mots_phrase(t_Noeud *racine, t_ListePhrase* liste_phrases)` : La fonction parcourt l'index (INFIXE) (O(n)), et ajoute le mot à la bonne phrase si son numéro correspond à une phrase présente dans la liste (recherche_phrase (O(m)),

ajouter_mot O(l)) ➔ **O(n*max(m+l)),** avec n le nombre de nœuds de la liste, m le nombre de phrases dans liste_phrase et l le nombre de mots dans la liste chaînée de la phrase associée.

t_Phrase* recherche_phrase(t_Phrase* tPhrase,int numero_phrase) : Parcourt la liste des phrases et compare leurs numéros à celui mis en paramètre ➔ **O(n),** n : le nombre de phrases de la liste
void afficher_mot(t_Mot *l) : parcourt et affiche les mots de la liste de mots ➔ **O(n),** n : le nombre de mots de la liste.
t_Phrase* creer_phrase() : La fonction ne fait qu'allouer la mémoire au nouvel objet et initialiser les attributs. Le nombre d'instructions est constant➔ **O(1)**
int ajouter_mot(t_Mot ** tMot,int numero_ligne,int ordre, char* mot) : La fonction ajoute un mot à une phrase. La liste chaînée est triée selon la ligne et l'ordre. Dans le pire des cas, il faut parcourir toute la liste ➔ **O(n),** n : le nombre d'éléments de la liste

t_Mot* creer_mot() : La fonction ne fait qu'allouer la mémoire au nouvel objet et initialiser les attributs. Le nombre d'instructions est constant➔ **O(1)**
int ajouter_phrase(t_ListePhrase * ListePhrases, int numero_phrase) : La fonction fait appel à recherche_phrase (O(n)) et crée et ajoute une phrase en fin de liste si la phrase n'existe pas ➔ **O(n),** n : le nombre de phrases de la liste.
void afficher_liste_positions(t_ListePositions *listeP) : Parcours et affiche la liste des positions de la liste mise en paramètre ➔ **O(n),** n : le nombre de positions de la liste.
t_Noeud* creer_noeud(char *mot) : La fonction ne fait qu'allouer la mémoire au nouvel objet et initialiser les attributs. Le nombre d'instructions est constant➔ **O(1)**
void supprimer_caractere(char *str, char c) : On parcourt la chaine str à la recherche du caractère c ➔ **O(n),** n : le nombre de caractères de la chaine
void parcours_affichage_index(t_Noeud *racine, char* c) : Parcours INFIXE et affichage des positions dans chaque appel récursif ➔ **O(n*m),** avec n le nombre de nœuds et m le nombre de position associé à chaque noeud
void parcours_recherche_max(t_Noeud *racine, t_Noeud *nd) : On effectue un parcours préfixe de l'index ➔ **O(n),** n : le nombre de nœuds
void libereNoeud(t_Noeud *racine) : La fonction ne fait que libérer la mémoire des de racine, de ses nœuds fils et de la liste de positions du nœud racine ➔ **O(n*m),** m étant le nombre de positions du mot associé au nœud (car appel à la fonction LibererListe) et n le nombre de noeud
void libereIndex(t_Index *index) : La fonction ne fait que libérer la mémoire des nœuds de l'index et de l'objet index ➔ **O(n)**
void libereListe(t_Position *liste) : La fonction ne fait que libérer la mémoire de la liste des positions d'un nœud, on itère n fois, n étant le nombre de positions du mot ➔ **O(n)**
void libereMot(t_Mot* liste) : La fonction ne fait que libérer la mémoire des mots de la liste chaînée, on itère n fois, n étant le nombre de mot de la phrase ➔ **O(n)**
void liberePhrase(t_Phrase *phrase) : La fonction ne fait que libérer la mémoire dans laquelle est l'objet. Le nombre d'instructions est constant➔ **O(1)**
int hauteur(t_Noeud *racine) : Calcule la hauteur de l'arbre dont la racine est racine , donc passe par chaque nœud fils (POSTFIXE) ➔ **O(n)**, n : le nombre de nœuds de l'arbre
int equilibre(t_Noeud *racine) La fonction détermine si l'arbre (index) de racine racine est équilibré ou non. On calcule la hauteur du fils droit (O(n), n : le nombre de nœuds de l'arbre droit) et la hauteur du fils gauche (O(m),m : le nombre de nœuds de l'arbre gauche) ➔ **O(max(n+m))**

void transforme_to_min(char *str) : La fonction vérifie que chaque caractère de la chaîne est en minuscule ➔ **O(n),** n : le nombre de caractères de la chaîne