Rapport TP4 NF16

Alexis Deseure, Mateo Marcillaud

Implémentation de fonctions et structures supplémentaires

En plus de la fonction viderBuffer() permettant de vider le tampon d'entrée et des fonctions du sujet, nous avons choisi d'implémenter d'autres fonctions et structures décritent ci-dessous :

Pile

Cette structure permet de créer une organisation similaire à une pile mais en utilisant une liste chaînée. Chaque élément de cette structure possède un entier N servant au parcours itératif de l'arbre, un pointeur vers le nœud auquel il fait référence et un pointeur vers l'élément juste en dessous dans la pile.

Phrase

Cette structure est utilisée pour les deux dernières questions du sujet (construireTexte et afficherOccurencesMot). Elle permet de créer une liste chaînée de phrases dans l'ordre auquel elles apparaissent dans le texte. Chaque élément de structure Phrase est alors constitué d'un numéro entier, d'un pointeur vers le premier élément de la liste chaînée des mots constituant la phrase et d'un pointeur vers l'élément de structure Phrase suivant.

Mot

Cette structure est utilisée pour stocker dans l'ordre dans une liste chaînée chacun des mots constituant une phrase (caractérisé par la structure Phrase). Chaque élément de structure Mot est constitué du numéro de ligne, de l'ordre dans la ligne et de la chaîne de caractère associée au mot correspondant. Il comporte aussi le pointeur vers le Mot suivant.

Pile* creerPile(T_Noeud* noeud, int N)

Cette fonction est utilisée dans le cadre des parcours de l'index (parcours préfixe et infixe itératifs) afin de créer une structure similaire à une pile mais sans utilisation d'un tableau avec une taille définie. On utilise ici une structure en liste chaînée avec comme élément au sommet de la pile, le premier élément de la liste chaînée. On aurait pu utiliser un tableau mais n'ayant pas besoin de considérer une pile avec une taille limitée, nous avons fait le choix de procéder avec une liste chaînée.

Mot* creerMot(int ordre,int ligne,char* mot)

Cette fonction fait directement suite à la structure Mot et permet d'initialiser un élément associé en lui allouant dynamiquement sa mémoire et en lui initialisant son ordre, sa ligne et ses caractères associés.

Phrase* creerPhrase(int n)

Cette fonction fait directement suite à la structure Phrase et permet d'initialiser un élément associé en lui allouant dynamiquement sa mémoire et en lui initialisant son numéro.

Phrase* ajouterPhrase(Phrase** phrase, int n)

Cette fonction permet simplement de créer la Phrase de numéro n s'il elle n'existe pas déjà, de l'ajouter au bon endroit dans la liste chaînée dont l'adresse du pointeur vers le premier élément est passé en paramètre puis de retourner un pointeur vers cette phrase.

void ajouterMot(Phrase** phrase, int numeroLigne, int ordre, char* nom, int numeroPhrase)

Cette fonction permet d'ajouter un Mot à la phrase dans laquelle il appartient (celle passée en paramètre). Elle va donc chercher la phrase ou la créer au bon emplacement dans la liste chaînée en utilisant la fonction ajouterPhrase, puis, elle va créer le mot qu'on souhaite ajouter et le placer au bon endroit dans la liste chaînée des mots associés à la phrase en fonction de son ordre et de son numéro de ligne.

void empiler(Pile** pile,T_Noeud* noeud, int N)

Cette fonction fait suite à la structure Pile et à la fonction creerPile, elle permet d'empiler un élément dans la pile associée. Elle crée une structure de type Pile pour l'élément à empiler, puis elle le place en première position de la liste chaînée des éléments de la pile (au sommet).

Pile *depiler(Pile** pile)

A l'inverse de la fonction empiler, celle-ci permet de dépiler l'élément au sommet de la pile puis de renvoyer le pointeur vers celui-ci. La pile passée en paramètre correspond à l'adresse du pointeur vers le premier élément de la liste chaînée des éléments de structure Pile.

void ignorerCasse(char* mot)

Cette fonction permet de modifier le mot passé en paramètre en mettant chacune de ses lettres en minuscule. Elle permet de simplifier l'écriture des autres fonctions.

void libererIndex(T_Index* index), void libererNoeud(T_Noeud* noeud) et void libererPosition(T_Position* position)

Ces fonctions sont utilisées lorsque l'on souhaite libérer la mémoire de l'index pour quitter le programme ou indexer un nouveau texte. Il suffit d'appeler libererIndex pour tout effectuer. A noter que libererNoeud s'effectue de manière récursive.

Complexité des fonctions

T_Position* creerPosition(int ligne, int ordre, int phrase)

Suite d'instructions de complexité constante. La fonction est donc de complexité constante (O(1)).

T_Noeud * creerNoeud(char* mot, int ligne, int ordre, int phrase)

Suite d'instructions de complexité constante. La fonction est donc de complexité constante (O(1)).

T_Index* creerIndex()

Suite d'instructions de complexité constante. La fonction est donc de complexité constante (O(1)).

Pile* creerPile(T_Noeud* noeud, int N)

Suite d'instructions de complexité constante. La fonction est donc de complexité constante (O(1)).

Mot* creerMot(int ordre,int ligne,char* mot)

Suite d'instructions de complexité constante. La fonction est donc de complexité constante (O(1)).

Phrase* creerPhrase(int n)

Suite d'instructions de complexité constante. La fonction est donc de complexité constante (O(1)).

void empiler(Pile** pile,T_Noeud* noeud, int N)

Fait appel à la fonction creerPile, de complexité O(1). Donc elle a une complexité en O(1).

Pile *depiler(Pile** pile)

Fait appel à la fonction creerPile, de complexité O(1). Donc elle a une complexité en O(1).

void ignorerCasse(char* mot)

La boucle while se répète autant de fois qu'il y a de lettres dans le mot passé en entrée. Il y a donc n itérations (n étant le nombre de lettres). La fonction a une complexité en O(n). La complexité de cette fonction sera supposée constante par la suite.

void viderBuffer()

La boucle while se répète autant de fois qu'il y a de caractères dans le buffer. Il y a donc n itérations. La fonction a une complexité en O(n).

Phrase* ajouterPhrase(Phrase** phrase, int n)

Dans le pire des cas, la phrase que l'on veut ajouter se trouve à la fin de la liste chaînée. Ainsi, la boucle while qui place la phrase dans la liste effectue n itérations avant de s'arrêter, avec n le nombre de phrases dans la liste. Les opérations dans la boucle étant constante, la complexité de la fonction est linéaire (O(n)).

void ajouterMot(Phrase** phrase, int numeroLigne, int ordre, char* nom, int numeroPhrase)

Dans le pire des cas, le mot que l'on veut ajouter se trouve à la fin de la liste chaînée de mots. Ainsi, la boucle while qui place le mot dans la liste effectue n itérations avant de s'arrêter, avec n le nombre de mots dans la liste. De plus la fonction fait appel à ajouterPhrase une fois, par conséquent, la complexité de la fonction est O(n + m) avec m le nombre de phrase dans la liste chaînée de phrases.

T_Position *ajouterPosition(T_Position *listeP, int ligne, int ordre, int phrase)

Dans le pire des cas, la position du mot que l'on souhaite ajouter se trouve en dernière position dans la liste chaînée. Ainsi, la boucle while qui place la position dans la liste effectue n itérations avant de s'arrêter, avec n le nombre de positions présentent dans la liste. La complexité de la fonction est O(n).

int ajouterOccurence(T_Index *index, char *mot, int ligne, int ordre, int phrase)

On parcourt l'ABR afin de placer l'occurrence du mot ou de créer le nœud correspondant dans l'index. Dans le pire des cas, il faut l'insérer au niveau de l'élément de profondeur maximale dans l'arbre, c'est-à-dire que la boucle while effectue h+1 itération avec h la hauteur de l'arbre. De plus, la fonction fait appelle, à la fin, à ajouterPosition en O(n), avec n le nombre de positions dans la liste à insérer. La fonction est donc de complexité O(h + n).

int indexerFichier(T_Index *index, char *filename)

On suppose qu'un fichier contient m caractères. L'algorithme parcourt chaque caractère et forme des mots avec les lettres qui se suivent. Pour chacun des mots lus, il va les ajouter dans l'index avec ajouterOccurence. La boucle while se répète autant de fois qu'il y a de caractères dans le fichier, donc m fois. Dans le pire des cas, tous les caractères sont séparés d'un espace, donc ajouterOccurence est appelée toutes les 2 itérations. Ainsi, indexerFichier est en $O((h + n) \times \frac{m}{2}) = O((h + n) \times m)$.

void afficherIndex(T_Index index)

On parcourt l'index (infixe itératif) pour afficher les n mots distincts associés, ainsi que les k occurrences, au total, de ces mots. La première boucle while se répète autant de fois qu'il y a de mots distincts dans l'index et la seconde se répète autant de fois qu'il y a d'occurrences de ce mot. Cependant, chaque mot a au moins une occurrence sinon il n'existerait pas, donc, $k \geq n$. L'algorithme parcourt donc en réalité le nombre de mots total dans le texte (la somme des occurrences de chacun des mots). La fonction afficherIndex est donc de complexité O(k).

T_Noeud* rechercherMot(T_Index index, char *mot)

Dans l'index de hauteur h, on effectue une recherche dans un ABR. La complexité de rechercheMot est donc O(h) avec h la hauteur de l'arbre.

void afficherOccurencesMot(T_Index index, char *mot)

Dans l'index de hauteur h, le programme fait appel à rechercherMot de complexité O(h). Il parcourt ensuite tous les nœuds et les positions de l'index (prefixe itératif) afin de les comparer avec celles du mot recherché et ainsi trouver les phrases dans lesquelles il apparaît. De la même manière que pour afficher index, le parcours de l'index s'effectue $k \times a$ fois avec k le nombre de mots total (la somme des occurrences de chacun des mots) et a, le nombre de positions du mot cherché. Puis, pour chacune de ces itérations, dans le pire des cas, le programme fait appel à ajouterMot de complexité O(k + m), avec m le nombre de phrase où apparaît le mot (on remplace le n par un k car dans le pire des cas, le mot recherché apparaît dans toutes les phrases). Ensuite, le programme doit parcourir les phrases pour les afficher correctement. Ce parcours est en O(k) car dans le pire des cas tous les mots sont dans une même phrase que celle du mot recherché. Finalement, la complexité globale de cette fonction est $O(h + k \times a \times (k + m) + k) = O(k \times a \times (k + m))\, car\, k + m \geq k,\ a \geq 1\, et\, h < k$ (on ne prend pas en compte la complexité pour la libération de mémoire).

void construireTexte(T_Index index, char *filename)

Cette fonction est assez similaire à la précédente mais cette fois on parcourt l'index sans faire de vérification : dans tous les cas on fait appel à ajouterMot. Ce parcours est donc en $O(k \times (k + m))$ avec k le nombre total de mots et m le nombre de phrases. La fonction écrit ensuite le texte dans le bon fichier en parcourant les occurrences de tous les mots, cette boucle s'effectue donc k fois. Finalement la complexité de la fonction est $O(k \times (k + m) + k) = O(k \times (k + m))\, car\ k + m > k$ (on ne prend pas en compte la complexité pour la libération de mémoire).

void libererPosition(T_Position* position)

La fonction s'effectue autant de fois qu'il y a d'occurrences d'un mot dans la liste chaînée de ses positions. La fonction est donc en O(n) avec n le nombre de mots.

void libererNoeud(T_Noeud* noeud)

La fonction parcourt récursivement l'index selon un parcours postfixe. Puis, pour chaque nœud, elle fait appel à libererPosition, de complexité O(m) avec m le nombre d'occurrence du mot.Globalement, la complexité de la fonction est en $O(n \times m)$, avec n le nombre de nœuds de l'index et m le nombre d'occurrences de mots maximum. On peut aussi considérer k le nombre de mots total (en comptant toutes les occurrences des mots identiques) auquel cas la complexité est O(k) (car k ≥nombre de mots distincts = nombre de nœuds de l'index).

void libererIndex(T_Index* index)

Cette fonction fait appel à libererNoeud et réalise des opération de complexité constante donc elle est en $O(n \times m)$.