CHAZELAS Clément
PASZTOR Amélie

Rapport TP4

Structure ajoutée :

Nous avons choisi d'implémenter une structure de données phrase qui nous permet d'optimiser grandement la complexité des deux dernières fonctions. La structure index comportera également un champ liste_phrases qui contiendra les phrases du texte.

Fonction ajouterPosition:

Fonctionnement : La fonction prend en paramètre un pointeur vers le premier élément de la liste chaînée des positions, qui peut être nul ou une position. Il prend aussi en paramètre les éléments nécessaires pour créer un élément de la structure position. On commence par créer une position. Ensuite, si le premier élément de la liste chaînée est nul, ou si le numéro de ligne est inférieur, on ajoute la position en tête de liste. Sinon, on parcourt la liste pour trouver où insérer le nouvel élément. Enfin, on insère le nouvel élément dans la liste, puis, on renvoie la liste.

Complexité : La fonction utilise une boucle while pour parcourir la liste des positions, jusqu'à trouver l'emplacement où insérer la nouvelle position. Dans le pire des cas, la boucle doit parcourir toutes les positions de la liste, donc la complexité est de O(n). Cette fonction n'en appelle pas d'autres et toutes les autres opérations (tester si la liste n'est pas vide, insérer le rayon) ont un coût constant. La complexité de cette fonction est donc O(n), avec n le nombre d'éléments déjà présents dans la liste de positions.

Fonction ajouterOccurence:

Fonctionnement : La fonction prend en paramètre un index, un mot, et les paramètres nécessaires pour créer une position. Elle commence par transformer chaque lettre en majuscule du mot en minuscule. Ensuite, si l'index est vide, on crée un élément Noeud et on l'ajoute à la racine de l'index. On appelle la fonction ajouterPosition pour créer la première position du nœud et l'ajouter. Si l'index est non vide, on cherche l'emplacement d'insertion de l'occurrence : si le mot que l'on veut ajouter est égal au mot testé, on utilise simplement la fonction ajouterPosition et on incrémente le nombre d'occurrences de ce mot, ainsi que le nombre de mots total de l'index. Sinon, si le mot est avant le mot testé dans l'ordre alphabétique, on teste le fils gauche, si il est après, on teste le fils gauche. Enfin si le mot testé est nul, cela veut dire que le mot est rencontré pour la première fois, donc on crée un noeud pour ce mot et on incrémente le nombre de mots distinct en plus d'incrémenter le nombre de mot total et d'initialiser le nombre d'occurrences de ce mot à 1.

Complexité : La fonction commence par parcourir le mot pour transformer toutes les majuscules en minuscules, la complexité de cette opération est en O(m), avec m la longueur du mot. La fonction fait aussi appel à un parcours d'arbre, à chaque itération, on descend d'un cran dans l'arbre, au pire jusqu'à ce qu'on atteigne une feuille. La complexité de ce parcours d'arbre est donc O(h) avec h la hauteur de l'arbre. On appelle également la

fonction ajouterPosition, de complexité O(n) avec n le nombre de positions dans la liste, une seule fois. Toutes les autres opérations sont de complexité constante. Donc la complexité de ajouterOccurence est O(n+h+m) avec n le nombre de position dans la liste de position, h la hauteur de l'arbre et m la longueur du mot qu'on ajoute.

Fonction indexerFichier:

Fonctionnement : La fonction prend en paramètres un index, ainsi que le nom du fichier à indexer.
On va tout d'abord ouvrir le fichier en mode lecture à l'aide de la fonction. Si l'ouverture du fichier échoue, un message d'erreur est affiché et la fonction retourne 0 pour indiquer l'échec.
On parcourt ensuite chaque ligne du fichier Et on stocke chaque ligne dans la variable 'ligneCourante'.
Une fois la ligne récupérée on la divise en mots à l'aide de la fonction `strtok`. Le délimiteur utilisé pour séparer les mots est l'espace, la tabulation et le saut de ligne afin de prendre en compte tous les cas possibles.
Ensuite, pour chaque mot extrait, on vérifie si le mot se termine par un point, auquel cas on supprime le point dans la chaîne de caractère, on ajoute le mot au contenu de la phrase à l'aide de la fonction 'strcat', on ajoute le mot à l'index à l'aide de la fonction 'ajouterOccurence', on incrémente le nombre de mots lus, on augmente le nombre de phrases lues et on passe à la phrase suivante. On crée ensuite une nouvelle structure `T_Phrase` pour représenter la phrase courante avec son contenu, son numéro et un pointeur vers la phrase suivante et on l'ajoute à la liste des phrases de l'index. On réinitialise enfin le contenu de la variable 'contenu' pour pouvoir prendre la prochaine phrase.
Si le mot n'est pas un point on va l'ajouter à l'index à l'aide de la fonction 'ajouterOccurence', on incrémente le nombre de mot lut, on incrémente l'ordre et on ajoute le mot au contenu de la phrase actuelle en utilisant la fonction 'strcat'.
On passe ensuite à la ligne suivante et on réinitialise donc l'ordre à 1.

Complexité : La fonction parcourt toutes les lignes du fichier, la première boucle while est donc de complexité O(l), avec l le nombre de lignes du fichier. Dans la deuxième boucle on fait appel à la fonction ajouterOccurence qui est en O(p+h+m) avec p le nombre de position dans la liste de position, h la hauteur de l'arbre et m la longueur du mot qu'on ajoute sur chaque mot de la ligne. Cette deuxième boucle est donc en O(n*(p+h+m)).
On a donc une complexité de O(l+n*(p+h+m)), avec l le nombre de lignes du fichier, p le nombre de position dans la liste de position, h la hauteur de l'arbre et m la longueur du mot qu'on ajoute sur chaque mot de la ligne.

Fonction afficherIndex:

Fonctionnement : La fonction crée un tableau qui est rempli avec chaque nœud dans l'ordre alphabétique en appelant la fonction parcoursInfixe. Ensuite elle parcourt ce tableau avec une boucle for et affiche chaque mot. Pour chaque mot, la fonction teste si le mot précédant commence par la même lettre que celui qui va être affiché. Si les premières lettres sont

différentes, l'affichage est différent. On affiche également toutes les positions des mots grâce à une boucle while.

Complexité : La fonction appelle la fonction parcoursInfixe, de complexité O(n), avec n le nombre de nœuds dans l'index. La fonction utilise également une boucle while pour afficher chaque nœud un par un, et pour chaque nœud, on affiche chaque position associée avec une autre boucle while. La complexité la partie affichage est donc en O(n*m) avec m le nombre de positions. Toutes les autres opérations ont une complexité constante. La complexité de cette fonction est donc O(n + n*m).

Fonction rechercherMot:

Fonctionnement : La fonction commence par convertir le mot en minuscule pour éviter les erreurs potentielles. Ensuite elle parcourt l'arbre pour rechercher le mot de la même manière que dans la fonction ajouterOccurence. Si on trouve le mot, on renvoie le mot, sinon on renvoie NULL.

Complexité : La fonction parcourt l'arbre. A chaque occurrence de la boucle while, on descend d'un cran dans l'arbre, la complexité est donc O(h) avec h la hauteur de l'arbre.

Fonction afficherOccurrencesMot:

Fonctionnement : La fonction fait d'abord appel à la fonction rechercherMot pour retrouver le mot cherché. Si le nœud renvoyé par rechercherMot n'est pas nul, alors on parcourt la liste des positions de ce mot et on cherche les phrases dans lesquelles il apparaît. Une fois la phrase trouvée on retire les éventuels caractères retour chariot (ce n'est pas obligatoire et ajoute de la complexité à la fonction mais nous l'avons fait par souci d'esthétisme). On met ensuite une majuscule à la première lettre de la phrase et on l'affiche. On passe ensuite à la position suivante.

Complexité : La fonction va d'abord rechercher le mot à l'aide de la fonction rechercherMot qui a une complexité de O(h), où h est la hauteur de l'arbre binaire de recherche dans l'index.
Ensuite, la fonction parcourt les phrases et vérifie si le numéro de phrase correspond à celui des positions du mot recherché. Cela nécessite de parcourir toutes les phrases et toutes les positions du mot, ce qui a une complexité de O(p*n), où p est le nombre de phrases et n est le nombre d'occurrences du mot.
Avant d'afficher la phrase on va enlever tous les retours à la lignes ce qui se fait en O(m), avec m la longueur de la phrase.
Les autres opérations sont effectuées en temps constant.
On a donc une complexité totale de O(h + p*n*m), avec h la hauteur de l'arbre de recherche, p le nombre de phrases, n le nombre d'occurrences du mot et m la longueur de la phrase.

Fonction construireTexte:

Fonctionnement : La fonction va parcourir la liste des phrases de l'index avant de les afficher successivement.

Complexité : La fonction comprend une boucle while qui parcourt la liste des phrases de l'index.
Les autres opérations sont réalisées en temps constant.
On a donc une complexité de O(n), avec n le nombre de phrases de l'index.

Fonction afficherPositions:

Fonctionnement : La fonction parcourt la liste de position reçue en paramètres. Elle affiche chaque position une par une.

Complexité : La fonction fait un parcours de liste chaînée. Dans tous les cas, la liste est parcourue en entier. La complexité est donc O(n) avec n le nombre de positions dans la liste chaînée.

Fonction parcoursInfixe:

Fonctionnement : La fonction est presque une copie exacte de celle vue en cours, sauf qu'au lieu d'afficher directement le texte, elle enregistre chaque nœud dans un tableau, dans l'ordre alphabétique.

Complexité : La fonction parcourt une fois chaque nœud de l'index, la complexité est donc en O(n), avec n le nombre de nœuds dans l'index.

Problèmes résolus

Ordre dans la ligne : Ce problème vient d'un problème de compréhension du sujet, nous pensions qu'il fallait indiquer l'ordre dans la phrase au lieu de l'ordre dans la ligne. Il nous a suffit de déplacer la remise à 1 de l'ordre du cas où on est à la fin d'une phrase à la fin d'une ligne.

Problèmes liés à l'affichage des phrases : Lors de l'affichage des phrases nous avions constaté que les phrases pouvaient êtres coupées. Ce problème venait de la manière dont nous récupérions les phrases du texte. Nous avons donc refait cette partie du code en fusionnant la récupération des phrases à celle des mots en concaténant ces derniers.