SCHNEIDER Martin
PICART Damien
NF16

NF16: Compte Rendu TP4 - Les Arbres Binaires de Recherche

I. Liste des fonctions supplémentaires:

Dans cette partie, nous allons expliciter les différentes fonctions que nous avons choisi d'ajouter à notre script afin de faciliter l'élaboration de certaines fonctions plus complexes.

Pour les structures, nous avons uniquement choisi de modifier la structure t_Index en lui ajoutant un champ. Ce champ est un **tableau de phrases** dans lequel nous allons stocker toutes les phrases du fichier texte. Cela nous permettra par la suite de pouvoir récupérer puis afficher les phrases dans lesquelles un mot apparaît lors de l'exécution de la fonction *afficher_occurences_mot.*

Pour les fonctions, nous avons tout d'abord ajouté la fonction *creer_noeud*, car nous allons créer des index qui vont contenir des nœuds. Il faut donc **les créer en leur allouant de la mémoire** et en **complétant les champs** d'un nœud avec les bonnes informations.

Nous avons ajouté les différentes fonctions de base nécessaires pour **manipuler des piles**: *creer_pile*, *pile_vide*, *pile_pleine*, *empiler*, *depiler*, *libere*. Ces fonctions permettent respectivement de créer une pile, tester si une pile est vide ou pleine, empiler un élément, dépiler un élément et enfin libérer l'espace mémoire alloué pour une pile. Elles sont utilisées pour **parcourir un arbre** de façon itérative. Ce parcours est notamment utilisé dans les fonctions *afficher_index* et *equilibrer_index*.

La fonction *addLetter* va être utilisée lors de l'indexation du fichier. Elle permet d'**ajouter un caractère à un mot**, en gardant le caractère '\0' en fin de mot. Lorsqu'on va lire le fichier caractère par caractère, *addLetter* sera donc utilisée pour ajouter le caractère (si c'est une lettre) au mot qui est en train d'être lu. Elle permet donc de reconstituer les mots inscrits dans le fichier texte.

La fonction *minuscule* permet de **transformer toutes les lettres d'un mot en lettres minuscules**. Elle est utilisée lors de l'indexation du fichier afin de n'avoir que des mots en minuscule dans notre index. Cela permettra de comparer plus facilement les mots entre eux (par ordre alphabétique) sans se préoccuper de la casse des mots, lorsque l'on doit placer un nœud au bon endroit dans l'index par exemple (avec *traiter_noeud*).

La fonction *majuscule* permet de **transformer la première lettre d'un mot en lettre majuscule**. Nous en avons besoin lorsqu'on affiche l'index, pour afficher chaque mot avec une lettre majuscule en début de mot.

La fonction *traiter_noeud* va être utilisée dans le **processus d'indexation du fichier**. En effet, elle prend en paramètre un index, un mot et les informations sur la position de ce mot. Elle va chercher si ce mot existe déjà dans l'arbre. Si oui, elle va mettre à jour les informations de l'index et la liste des positions du nœud correspondant. Sinon, elle va ajouter ce mot à l'index en créant un nœud et en le plaçant au bon endroit.

La fonction *afficher_lettre* permet, lors de l'affichage de l'index, d'**afficher tous les mots commençant par une certaine lettre**, ainsi que leurs **informations de position**. Cette fonction va renvoyer un caractère qui sera la première lettre du mot du nœud traité. Cela permet de garder cette lettre en mémoire et ainsi de savoir si nous devons afficher une nouvelle lettre lorsqu'on passe au noeud suivant:

- soit on passe à un mot commençant par une lettre différente du mot correspondant au noeud précédent. Dans ce cas, on affiche cette première lettre qui est différente.
- soit le mot du nœud à traiter commence par la même lettre que le mot du nœud précédent. Dans ce cas là nous n'avons pas besoin d'afficher de nouveau la première lettre du mot.

Les fonctions *placer_noeud* et *liberer_index* (qui fait appel à *suppression_postfix*) sont utilisées pour la fonction *equilibrer_index*. Toutes ces fonctions seront donc utilisées pour répondre à la question B.9 et ainsi **équilibrer l'arbre d'un index**. Nous expliquerons le choix de leur implémentation en partie II).

La fonction *arbre_equilibre* est également utilisée pour répondre à la question B.9. Cette fonction ainsi que les fonctions *max* et *hauteur_arbre* qu'elle appelle seront donc expliquées en partie II).

La fonction *indexer_phrase* est utile pour répondre à la question B.8 et ainsi **afficher les occurrences d'un mot**. Nous expliquerons la raison de l'implémentation de cette fonction dans la partie II).

Enfin, la fonction *menu_principal* permet, via un switch, de **traiter tous les choix** de notre programme disponibles pour l'utilisateur. Elle fait appel à la plupart des fonctions implémentées.

II. Solutions et algorithmes pour les questions B.8 et B.9:

Pour répondre à la question B.8, nous avons créé une fonction *afficher_occurences_mot*. Le principal problème auquel nous avons fait face lors de l'élaboration de cette fonction était de réussir à **récupérer les phrases** dans lesquelles se trouvait le mot recherché. Nous avons donc décidé d'ajouter un champ à la structure t_Index. Ce champ est un **tableau de phrases**. Ainsi, nous pouvons remplir ce tableau en y insérant dans chaque case du tableau une phrase du fichier texte. Nous avons défini une constante NB_PHRASES qui permettra de modifier rapidement le nombre de phrases que l'on peut rentrer dans le tableau. De même, nous avons défini la constante NB_LETTRES_PHRASE afin de modifier facilement le nombre de caractères que l'on peut rentrer dans une phrase.
La récupération de chaque phrase du fichier texte dans le tableau de phrases se fait grâce à la fonction *indexer_phrase*. Cette fonction va lire le fichier caractère par caractère et lorsqu'elle rencontre un point, elle stock la phrase dans un tableau passé en paramètre. Nous avons choisi de faire cette indexation dès lors que l'utilisateur importe le fichier. Ainsi, tous les champs de la structure t_Index sont remplis dès la première indexation.
Une fois que ce nouveau champ de la structure t_Index est rempli, il est facile d'accéder aux informations voulues car elles se trouvent toutes dans l'index passé en paramètre. On peut ainsi récupérer la **position** du mot en cherchant ce mot dans **l'arbre** correspondant à l'index, et à chercher les **phrases** correspondantes à chaque position dans le **tableau de phrases** (grâce au numéro de phrase stocké dans la position du mot).

Pour répondre à la question B.9 et créer la fonction *equilibrer_index*. Tout d'abord, cette fonction doit tester si l'arbre correspondant à l'index donné en paramètre est équilibré ou non. Pour cela, on fait appel à la fonction *arbre_equilibre* qui permet de **tester si l'index est équilibré**. Cette fonction fait appel à la fonction *max*, qui retourne le maximum entre deux entiers, et à la fonction *hauteur_arbre*. Cette dernière permet de **fournir la hauteur d'un arbre**. Elle est utilisée dans la fonction *arbre_equilibre* afin de déterminer la hauteur des sous arbres droit et gauche de la racine de l'index et ainsi savoir si l'index est équilibré.
Une fois que l'on a cette information, on peut équilibrer l'arbre s'il ne l'est pas déjà. Pour cela, on commence par parcourir tout l'arbre (parcours infixe) à l'aide d'une pile, afin de **stocker tous les nœuds** dans un tableau. Grâce au parcours infixe, le tableau sera alors trié dans l'ordre alphabétique. On peut ensuite créer un nouvel index et faire appel à la fonction *placer_noeud* qui prend en argument le nouvel index créé et un tableau de noeuds. Elle va ensuite placer les nœuds de façon à obtenir un arbre équilibré: la moitié gauche du tableau de nœuds va être placée dans le sous arbre gauche de la racine et la moitié droite du tableau sera placée dans le sous arbre droit de la racine. On effectue cette mécanique de placer la moitié d'un tableau dans le sous arbre d'un nœud et l'autre moitié dans l'autre sous arbre sur toute la hauteur de l'arbre grâce à des appels récursifs.
Une fois qu'on a ce nouvel arbre équilibré, on peut libérer la mémoire allouée à l'ancien arbre non équilibré. Pour cela on fait appel à la fonction *liberer_index*. Cette fonction fait elle même appel à la fonction *suppression_postfix* qui **supprime tous les noeuds d'un index** en parcourant cet index selon la procédure **postfixe** afin de ne pas perdre l'accès à chaque nœud (on ne peut pas commencer par supprimer la racine sinon on perd l'accès à ses sous arbres droits et gauches). La fonction *liberer_index* va donc permettre de libérer tous les nœuds d'un index puis, grâce à une boucle for de libérer le tableau de phrases de l'index et enfin de libérer la mémoire allouée à la structure index.

III. Exposé succinct de la complexité des fonctions implémentées:

Tout d'abord, les fonctions *creer_index*, *creer_liste_positions*, *creer_pile*, *pile_vide*, *pile_pleine*, *empiler*, *depiler*, *libere* et *max* sont toutes en **O(1)** car elles ne présentent que des opérations élémentaires.

addLetter : cette fonction fait uniquement appel à la fonction *strcat* qui concatène deux chaînes de caractères. Notons O(a) la complexité de la fonction *strcat*. Ainsi, la complexité de la fonction *addLetter* est également en **O(a)**.

majuscule : cette fonction fait appel à la fonction *toupper*. Cette fonction transforme une lettre minuscule en majuscule. On suppose donc que sa complexité est en O(1). La complexité de *majuscule* est donc également en **O(1)**.

minuscule : cette fonction fait appel à la fonction *tolower* qui transforme une lettre majuscule en minuscule. On supposera donc que se complexité est en O(1). De plus, elle présente une boucle for qui parcourt toutes les lettres d'un mot. En notant n le nombre de lettre dans le mot traité, la complexité de la fonction *minuscule* est en **O(n)**.

afficher_lettre : dans les deux tests de cette fonction, on appelle la fonction majuscule et on a une boucle for du même nombre d'itération pour les deux tests: le nombre d'occurrences du mot traité. En notant o le nombre d'occurence du mot traité, la complexité de la fonction *afficher_lettre* est en **O(o)**.

afficher_index : cette fonction présente deux boucles while dans lesquelles les fonctions *empiler* (O(1)), *dépiler* (O(1)) et *afficher_lettre* (O(t+o)) sont appelées. La fonction *libere* (O(1)) est également appelée. Les deux boucles while permettent de parcourir tout l'arbre. On appelle donc la fonction *afficher_lettre* pour chaque nœud de l'arbre. En notant n le nombre de noeud dans l'arbre, la complexité de la fonction *afficher_index* est en **O(n*(t+o))**.

ajouter_noeud : cette fonction fait appel à une boucle while qui permet de parcourir toute la hauteur de l'arbre, notée h. De plus, on appelle la fonction strcmp, dont la complexité sera notée en O(a). La complexité de la fonction *ajouter_noeud* est donc en **O(h*a)**.

ajouter_position : cette fonction va parcourir une liste de position. Dans le pire des cas, elle va parcourir toute la liste. Notons p le nombre de positions dans la liste considérée, la complexité de cette fonction sera en **O(p)**.

creer_noeud : cette fonction fait appel aux fonctions *strcpy*, *creer_liste_positions* et *ajouter_position*. Supposons que la complexité de la fonction *strcpy* est en O(b). En utilisant les complexités des fonctions *creer_liste_positions* et *ajouter_position*, on obtient que la complexité de la fonction *creer_noeud* est en O(n+b) avec n le nombre de positions à ajouter. Or ici une seule position est ajoutée. La complexité de la fonction *creer_noeud* est donc en **O(b)**.

liberer_position : cette fonction est récursive. Le nombre d'appels est égale au nombre de positions dans la liste de positions. En posant n le nombre de positions dans la liste, la complexité de la fonction *liberer_position* est en **O(n)**.

suppression_postfix : cette fonction est récursive. Elle présente autant d'appels récursifs que de nœuds présents dans l'index, car il s'agit ici de supprimer toutes les positions de chaque nœud. Cette fonction fait également appel à la fonction *liberer_position.* En notant n le nombre maximum de positions pour un noeud et m le nombre de noeuds dans l'index, la complexité de la fonction *suppression_postfix* est en **O(m*n)**.

liberer_index : cette fonction fait seulement appel à la fonction *suppression_postfix.* En prenant les mêmes notations que précédemment, la complexité de cette fonction est en **O(m*n)**.

placer_noeud : cette fonction est récursive. Elle permet de placer les nœuds de façon à former un arbre équilibré. Il y aura donc autant d'appels récursifs que de nœuds à placer. De plus, cette fonction fait appel à

creer_noeud (complexité en O(b) qui est la complexité de la fonction de *strcpy*) et *ajouter_position* (complexité en O(p) avec p le nombre de positions maximum dans une liste considérée). On va ainsi devoir créer chaque nœud et ajouter leurs positions. Notons p le nombre maximum de positions pour un nœud et n le nombre de nœuds à placer. La complexité de la fonction *placer_noeud* est donc en **O(n*(b+p))**.

hauteur_arbre : cette fonction est récursive. Pour chaque nœud, on appelle deux fois la fonction. Le nombre d'appels est donc égal à la somme sur k des 2^k avec k allant de 0 à h la hauteur de l'arbre. La complexité de la fonction *hauteur_arbre* est donc en **O(2^h)**.

arbre_equilibre : cette fonction contient deux appels à la fonction *hauteur_arbre*. La complexité de la fonction *arbre_equilibre* est donc également en **O(2^h)**.

equilibrer_index : cette fonction contient notamment deux boucles while imbriquées qui servent à parcourir tous les nœuds de l'arbre, et une boucle for qui permet de recopier le tableau de phrases de l'ancien index vers le nouveau. De plus, elle fait appel à *creer_pile* et *creer_index* qui sont en O(1), mais aussi à *placer_noeud* qui est en O(n*(b+p)) et à *liberer_index* qui est en O(m*n) avec m le nombre de nœuds dans l'index et n le nombre maximum de positions pour un nœud. En notant n le nombre de nœuds dans l'arbre et y le nombre de phrases dans le texte, la complexité de la fonction *equilibrer_index* est donc en **O(n+y+n*(b+p)+m*n) = O(y+n*(b+p+m)).**

rechercher_mot : cette fonction permet de trouver un nœud dans l'arbre. Dans le pire des cas, il se trouve au niveau d'une feuille. On va donc parcourir au maximum la hauteur de l'arbre avant de trouver le nœud. Cette fonction fait également appel à la fonction *strcmp*. En notant O(a) la complexité de cette dernière fonction et en notant h la hauteur de l'arbre traité, la complexité de la fonction *rechercher_mot* est en **O(a*h)**.

traiter_noeud : cette fonction commence par appeler la fonction *rechercher_mot* qui est en O(a*h) avec les notations précédentes. Ici le pire des cas est lorsqu'on ne trouve pas le mot considéré. Il faut alors créer un nœud grâce à la fonction *creer_noeud* qui est en O(b). Puis on ajoute le noeud à l'arbre avec la fonction *ajouter_noeud* qui est en O(h*a). Ainsi, la complexité de la fonction *traiter_noeud* est en **O(a*h+b+h*a) = O(h*a+b)** car le a correspond dans les deux cas ici à la complexité de la fonction *strcmp.*

indexer_fichier : dans cette fonction, la boucle do-while permet de parcourir le fichier texte caractère par caractère. La fonction *addLetter* est appelée si le caractère est alphanumérique. Sinon, si le caractère est un espace, un retour à la ligne ou un point, on fait appel aux fonctions *minuscule* et *traiter_noeud*. Posons **c** le nombre de **caractères alphanumériques** et **x** le nombre d'**espaces**, **retours à la ligne** ou **points**. La complexité de la fonction *indexer_fichier* est donc en **O(c*(t+o) + x*((h*a+b)+(n)))** avec les notations précédentes.

indexer_phrase : cette fonction va permettre d'indexer un fichier en stockant les phrases dans un tableau. Ainsi, les boucles for et do-while permettent de parcourir tout le fichier, caractère par caractère et de stocker chaque phrase dans une case d'un tableau. Cette fonction fait également appel à la fonction *addLetter* et *strcpy*. En notant O(a) la complexité de la fonction *addLetter,* O(s) la complexité de la fonction *strcpy,* n le nombre de caractères dans le fichier et p le nombre de phrases, la complexité de la fonction *indexer_phrase* est en **O(p*s+n*a)**.

afficher_occurences_mot : cette fonction fait d'abord appel à *rechercher_mot* (O(a*h)). Puis une boucle while permet d'afficher les informations pour chaque occurrence d'un mot. En notant o le nombre d'occurrence, la complexité de la fonction *afficher_occurences_mot* est en **O(o+a*h)**.