NF16 - TP4
Les ABR

Automne 2014 – 22 décembre 2015

Enseignant : Mohamed Akheraz

Thomas COUTANT **GI01**
Eric GOURLAOUEN **GI01**

Introduction

Ce TP avait pour objectif de nous faire progresser dans la maîtrise de structures de données, en nous faisant concevoir un programme exploitant à la fois des données sous forme de liste chaînées, et sous la forme d'arbre binaire trié.

Dans ce projet, l'objectif était de représenter, pour un texte donné, les mots dans ce texte ; chaque mot était ajouté dans l'arbre selon l'ordre lexicographique, et chaque position de ce mot est ajoutée à une liste de positions propre à ce mot.

Algorithmes utilisés

Dans le cadre de ce programme, nous avons conçu la structure de données sous la forme d'un arbre, chaque nœud représentant un mot, et menant à une liste chaînée de positions (de forme **ListePosition**, contenant des positions de forme **struct Position**) contenant les identificateurs de chacune de ses positions dans le texte.

Nous avons conçu d'abord, dans l'objectif de pouvoir ajouter des positions à des nœuds, la fonction **ajouter_position**. Cette fonction prend en entrée la liste des positions du mot étudié, et les paramètres de la position (la ligne, l'ordre et le numéro de phrase). Si la liste de positions est vide, on ajoute une instance de position avec les paramètres voulus en tête de liste ; sinon, on parcourt la liste jusqu'à tomber soit sur la fin de la liste, soit sur une position précédente. On ajoute ensuite une instance de la position avec les paramètres voulus.

La deuxième fonction est une fonction d'ajout de nœuds, appelée **ajouter_noeud**. Elle prend en paramètre le mot, sa position dans le texte, et l'arbre où on veut l'insérer. Elle parcourt l'arbre en suivant l'ordre lexicographique ; si elle trouve une instance du mot dans l'arbre, elle ajoute juste une position supplémentaire à ce mot, et sinon, elle rajoute ce mot à l'arbre, avec une première position.

L'arbre était construit selon l'ordre lexicographique, il est facile de concevoir une fonction de recherche, appelée **rechercher_noeud**. Cette fonction parcourt l'arbre dans l'ordre lexicographique en prenant en paramètre un mot voulu, et rend l'adresse mémoire du nœud de ce mot s'il est présent dans l'arbre ; sinon, il rend un pointeur nul.

Pour afficher tous les mots de l'arbre et toutes leurs positions, on utilise deux fonctions, **afficher_noeud** e t **afficher_noeuds**. afficher_noeuds est une fonction récursive qui s'appelle d'abord pour le nœud fils gauche s'il n'est pas vide, puis qui appelle afficher_noeud pour le nœud en question, puis qui appelle affiche_noeuds pour le nœud fils droit. afficher_noeud, elle, affiche d'abord le mot, puis parcourt la liste des positions pour afficher toutes les positions.

Pour l'équilibrage, bien que les fonctions ne fonctionnent pas complètement, nous avons conçu une fonction **RotDroite** qui effectue une rotation droite, et qui rend le nœud père

s'il y a eu rotation droite, et un pointeur NULL sinon ; et une fonction **RotGauche** qui fait la même chose pour une rotation gauche. La fonction d'équilibrage était sensée appeler les fonctions RotGauche et RotDroite en fonction des différences de hauteur des branches ; mais il semblerait qu'il y ait une erreur dans le processus, puisque la fonction equilibrage finit par couper des nœuds de l'arbre...

La fonction **hauteur** est une fonction récursive qui évalue la hauteur d'un sous-arbre dont le nœud en paramètre est la racine, en parcourant les sous-arbres gauche et droit du nœud, et en incrémentant la hauteur reçue à chaque fois.

Enfin, pour l'équilibre, on utilise un ensemble de fonctions **equNoeud** et **equNoeudRec**. On part de la définition selon laquelle un arbre est équilibré si en tout nœud x, |hauteur(droit(x))-hauteur(gauche(x))| <= 1. Ce que fait la fonction equNoeudRec, (qui est appelée par la fonction de base **equilibre**), c'est d'abord vérifier la propriété de différence de hauteur pour le nœud étudié, puis de s'appeler récursivement pour tous les nœuds descendants de cet arbre. De ce fait, equilibre appelant equNoeudRec en la racine de l'arbre, la propriété est vérifiée pour tous les nœuds.

Fonctions et structures de données supplémentaires

Pour ce qui est en rapport avec la libération de la mémoire avant de quitter le programme, nous avons créé plusieurs fonctions :

- **supprimer_arbre** : supprime la structure correspondant à la racine de l'arbre. Cette fonction appelle supprimer_noeuds.
- **supprimer_noeuds** : libère la mémoire allouée pour un nœud. Cette fonction s'appelle récursivement sur les fils gauche et droit du nœud si ils existent. De plus, pour chaque nœud, on appelle la fonction supprimer_listePositions.
- **supprimer_listePositions** : cette fonction libère la mémoire en supprimant chaque élément de la liste chaînée.

Pour réaliser la fonctionnalité d'équilibrage de l'arbre, nous avons du mettre en place plusieurs fonctions :

- **RotDroite** : effectue une rotation à droite et renvoie le nouveau père de nœud.
- **RotGauche** : effectue une rotation à gauche et renvoie le nouveau père de nœud.
- **pere** e t **recupPere** : renvoient le père du nœud dans l'arbre si celui-ci existe ou NULL dans le cas opposé.
- **equilibrage** : equilibre l'arbre en effectuant une rotation à droite où une rotation à gauche selon la hauteur des sous arbres au noeud donné.

Différentes fonctions d'affichage ont du être réalisées pour permettre à l'utilisateur d'utiliser le programme qui viens d'être réalisé :

- **afficher_noeud** : parcours la liste chaînée contenant la ou les positions d'un mot dans le texte et affiche ces positions.
- **afficher_noeuds** et **afficher_arbre** : parcours récursivement tous les fils d'un nœud ou à partir de la racine et appelle la fonction afficher_noeud pour afficher les positions des mots correspondants à ces nœuds.
- **afficher_phrase** et **rechPhrase** : permet de rechercher les phrases contenant deux mots donnés.

La fonctionnalité nous disant si l'arbre est équilibré ou non a aussi nécessité plusieurs fonctions :

- **nb_descendants** : retourne le nombre de fils d'un nœud.
- **hauteur** : donne la hauteur d'un nœud donné.
- **equNoeud** : détermine si un nœud donné est équilibré.
- **equNoeudRec** : détermine si un nœud donné et tous ses descendants sont équilibrés.
- **equilibre** : détermine si un arbre est équilibré.

Difficultés rencontrées

La fonction **charger_fichier** nous a posé problème. En effet, nous chargions au début le fichier en récupérant mot par mot avec la fonction fscanf. Cette solution ne nous permettait pas de savoir lorsque nous passions à la ligne suivante car le caractère \n n'était jamais récupéré. Au lieu de cela, nous avons opté pour la solution qui consiste à récupérer caractère par caractère. Ainsi rencontrer un espace signifie arriver à la fin d'un mot, rencontrer un \n correspond à une fin de ligne et un point correspond à une fin de phrase.

Nous n'avons pas réussi à faire fonctionner la fonction d'**équilibrage de l'arbre**. Les algorithmes du cours ont été implémentés mais ceux-ci ne sont pas fonctionnels ; en effet, bien que la fonction de mesure de l'équilibrage est correcte, nous n'avons pas été en mesure de concevoir un algorithme d'équilibrage qui fonctionne. L'algo conçu fonctionnait à base de rotations gauche ou droites en fonction de la situation ; mais dans notre implémentation, des nœuds disparaissaient à chaque tentative d'équilibrage, vraisemblablement à cause d'une erreur dans la reconstitution des liens. L'algorithme d'équilibrage de l'arbre étant en bonus dans le cahier des charges et ayant été pris par le temps, nous avons pris la décision de ne pas garder l'équilibrage dans le programme final.

Conclusion

Ce TP a représenté un challenge pour nous car il nous a demandé de gérer plusieurs

structures de données dans un même programme ; il nous a cependant permis de réaliser des choses ambitieuses, comme la gestion de tout un texte dans une structure mémoire tout en conservant une façon de procéder ayant pour objectif d'optimiser le nombre d'appels et de réduire la complexité. De plus, ce TP nous a permis de comprendre en pratique comment utiliser un arbre binaire de recherche d'une autre manière que celle abordée en cours où chaque nœud contient une simple clé.