BOCQUELET Théo
GAO Fan

RAPPORT TP4 NF16
-
ARBRES BINAIRES DE RECHERCHE

Durant ce TP, nous avons réalisé un programme permettant d'effectuer la plupart des opérations sur les arbres binaires de recherche. Un arbre binaire de recherche étant un arbre binaire (ayant au plus deux fils) dont la valeur du fils gauche est inférieure à la valeur du père, et la valeur du fils droit est supérieure à la valeur du père.

Pour représenter l'arbre, nous avons utilisé la structure *Nœud* qui permet d'accéder au père, au fils gauche, au fils droit et aussi à la clé du nœud actuel.

- **Nœud *insererNoeud(int n, Nœud *root)**

Spécifications :

Entrées : la racine de l'arbre (root) et la valeur de la clé (n) que l'on veut insérer
Sortie : la racine de l'arbre (root)

Fonctionnement : si l'arbre est vide, alors il est créé avec la clé indiquée en entrée, sinon, la fonction insère un nouveau nœud qui sera une feuille de l'arbre de racine root, ce nouveau nœud aura pour clé n. Ce nœud sera bien sur inséré en respectant les spécificités des arbres binaires de recherche, c'est-à-dire pas plus de deux fils par nœuds et clé(fils gauche)<clé(père)<clé(fils droit). Il faut de plus que ce nœud ne soit pas déjà présent dans l'arbre, autrement il n'est pas inséré.

Technique : l'utilisation d'une variable « temp » nous a permis ici de pouvoir parcourir l'arbre sans perdre la valeur de la racine de l'arbre. Temp a pour but d'aller jusqu'à la fin de l'arbre, il vaut donc « NULL » à la fin de la boucle. Une autre variable, « pred », prenant la valeur du nœud précédent temp, nous a permis de faire nos insertions.

Complexité :

Ici, nous ne parcourons pas tout l'arbre, uniquement un chemin spécifique dépendant de la hauteur de l'arbre. En effet, nous insérons à la fin de l'arbre. La complexité de la boucle while est en O(h). Le reste des opérations étant en O(1), la fonction est en O(h).

- **Noeud * insererNoeud_rec (int n, Noeud *root)**

Spécifications :

Les entrées et sorties sont les mêmes que pour la fonction précédente.

Fonctionnement : si l'arbre est vide, alors il est créé immédiatement avec la clé indiquée en entrée, sinon, suivant la valeur des fils du nœud, des sous appels seront effectués, jusqu'à atteindre la fin de l'arbre où sera inséré le nouveau nœud. A chaque sous appel, on envoie la valeur de clé que l'on veut entrer et le sous arbre concerné. Ce sous arbre correspond à l'un des fils du nœud de l'appel précédent.

Technique : il n'y a pas de technique particulière à noter ici. Si la valeur de la clé est inférieure au nœud actuel nous lançons un sous appel sur le sous arbre gauche sinon nous lançons un sous appel sur le sous arbre droit. Il faut évidemment que la valeur de ce sous arbre ne soit pas NIL.

Complexité :

La complexité est identique à celle de la fonction itérative mais la justification diffère. Là encore, le chemin parcouru dépend de la hauteur de l'arbre. Nous effectuerons dans le pire des cas h appels en O(1), ce qui représente une complexité en O(h).

- **int verifier (Noeud *root)**

Spécifications :

Entrée : la racine de l'arbre à tester
Sortie : la fonction retourne 1 si l'arbre est un arbre binaire de recherche et 0 sinon

Fonctionnement : la fonction retourne 0 directement si l'arbre est vide. Si l'arbre n'est pas vide, la fonction teste récursivement chaque nœud de manière à vérifier si l'on a bien clé(fils gauche)< clé(père)<clé(fils droit). Si cette condition est vérifiée pour tous les nœuds alors la fonction retourne 1 autrement, dès qu'elle trouve un nœud qui ne vérifie pas la condition, elle retourne 0.

Technique : étant donné que root est de type pointeur sur *Nœud*, nous n'avons pas à tester si l'arbre est un arbre binaire, *Nœud* ne représente que des arbres binaires. Nous avons utilisé un système de deux variables, une pour le sous arbre gauche et une autre pour le sous arbre droit. Ces variables prennent la valeur 1 lorsque la fin de l'arbre est atteinte et que la condition ci-dessus est respectée. Ainsi, la fonction ne retourne 1 que lorsque tous les nœuds on été testés.

Complexité :

Pour cette fonction, il faut tester tous les nœuds de l'arbre, il y aura donc n-1 sous appel en O(1) ce qui fait une complexité en O(n).

- **Noeud * recherche (int n, Noeud * root)**

Spécifications :

Entrées : la racine de l'arbre et la valeur n que l'on souhaite rechercher
Sortie : le nœud possédant cette valeur ou NIL si ce nœud n'existe pas

Fonctionnement : si l'arbre est vide, on retourne NIL directement. Sinon, on parcourt l'arbre tant que la fin de celui-ci n'est pas atteinte ou que le nœud actuel n'est pas égal à n. Le parcourt se fait comme pour les autres fonctions, si n<clé du nœud actuel on parcourt le sous arbre gauche, sinon on parcourt le sous arbre droit. Une fois sorti de la boucle, il n'y a plus qu'à retourner le nœud correspond à n s'il existe, ou NIL s'il n'existe pas.

Complexité :

Là encore, tout comme l'insertion, on ne parcourt pas la totalité de l'arbre. On ne parcourt qu'un chemin spécifique. Dans le pire des cas, ce chemin sera de longueur h si l'on atteint la fin de l'arbre. La fonction est donc en O(h).

- **Noeud * recherche_rec (int n, Noeud * root)**

Spécifications :

Les entrées et sorties sont les mêmes que pour la fonction précédente.

Fonctionnement : si le nœud actuel vaut NIL ou si la valeur du nœud est égale à la clé recherchée alors on retourne ce nœud. Sinon si la clé est inférieure à la valeur du nœud actuel on lance un sous appel sur le sous arbre gauche et si la clé est supérieure à la valeur du nœud actuel on lance un sous appel sur le sous arbre droit.

Complexité :

La complexité de cette fonction sera la même que pour la fonction « recherche » itérative. En effet, dans le pire des cas nous allons parcourir tout un chemin dépendant de la hauteur où nous lançons h sous appels en O(1). Cela donne ainsi une complexité dans le pire des cas en O(h).

- **Int * hauteur (Noeud * root)**

Spécifications :

Entrées : la racine de l'arbre
Sortie : la hauteur de l'arbre

Fonctionnement : si l'arbre est vide ou s'il n'y a qu'un seul noeud (root), on retourne 0. Si les sous-arbres du nœud actuel ne sont pas égals à nul, on retourne 1. Sinon, on calcule la hauteur avec une fonction qui retourne la valeur maximale entre la hauteur du sous-arbre gauche et la celle du sous arbre droit de la racine *root*.

Complexité :

On devra parcourir tout l'arbre jusqu'à la fin. Dans le pire des cas, tous les noeuds de l'arbre seront testés et la hauteur de chaque hauteur sera retournée. La fonction est donc en O(n).

- **Int * maxi (int a, int b)**

Spécifications :

Entrées : La valeur a et la valeur b
Sortie : La valeur maximale entre les deux entrées

Fonctionnement : Si a > b, on retourne a. Sinon on retourne b.

Complexité :

Là, il n'y a qu'une seule comparaison donc la fonction est en O(1).

- **Int * somme (Noeud * root)**

Spécifications :

Entrées : la racine de l'arbre
Sortie : la somme des noeuds

Fonctionnement : La fonction est récursive. Si l'arbre est vide, on retourne 0. Sinon, on calcule la somme de la racine et de son sous arbre gauche et droit.

Complexité :

Comme la hauteur, on devra parcourir tout l'arbre jusqu'à la fin. Dans le pire des cas, tous les noeuds de l'arbre seront testés. La fonction est donc en O(n).

- **Void afficherDecroissant (Noeud * root)**

Spécifications :

Entrées : la racine de l'arbre
Sortie : l'affichage de l'arbre en ordre décroissant

Fonctionnement : La fonction est récursive. Si l'arbre n'est pas vide, on affiche d'abord le sous arbre gauche puis la racine et enfin le sous arbre droit. Si l'arbre est vide, on sort de la fonction.

Complexité :

Dans la fonction, on devra parcourir tout l'arbre jusqu'à la fin pour un affichage. Dans le pire des cas, tous les noeuds de l'arbre seront testés. La fonction est donc en O(n).

- **Void afficherStructure (Noeud * root)**

Spécifications :

Entrées : la racine de l'arbre
Sortie : l'affichage de l'arbre de façon à visualiser la structure de l'arbre

Fonctionnement : La fonction est récursive. Si le noeud n'existe pas, on affiche "_". Sinon on affiche la valeur de chaque noeud existant. L'ordre est comme l'affichage décroissant. On affiche d'abord le sous-arbre gauche puis la racine et enfin le sous-arbre droit.

Complexité :

Dans la fonction, on devra parcourir tout l'arbre jusqu'à la fin pour un affichage. Dans le pire des cas, tous les noeuds de l'arbre seront testés. La fonction est donc en O(n).

- **Noeud* supprimer (int n, Noeud * root)**

Spécifications :

Entrées : la racine de l'arbre et la valeur n que l'on souhaite supprimer
Sortie : le nœud possédant cette valeur ou NIL si ce nœud n'existe pas

Fonctionnement : si le noeud actuel vaut NIL, on retourne NIL. Si la valeur du nœud est égale à la clé à supprimer alors on va ensuite vérifier si son sous-arbre gauche existe. Si oui, on passe au sous-arbre gauche et on parcourt le sous-arbre droit jusqu'à ce que le sous-arbre droit soit nul. Cela sert à chercher un bon endroit pour mettre le sous-arbre droit du noeud à supprimer. Enfin, on retourne le noeud gauche du noeud à supprimer. Sinon, on retourne le noeud droit du noeud à supprimer. Si n n'est pas égal à la clé du noeud root, on parcourra son sous-arbre gauche ou son sous-arbre droit selon une comparaison de n et la clé de la racine.

Complexité :

La complexité de cette fonction sera la même que pour la fonction « recherche » récursive. En effet, dans le pire des cas nous allons parcourir tout un chemin dépendant de la hauteur où nous lançons h sous appels en O(1). Cela donne ainsi une complexité dans le pire des cas en O(h).

- **Noeud* detruire (Noeud * root)**

Spécifications :

Entrées : la racine de l'arbre
Sortie : le nœud possédant NIL si l'arbre est bien détruit.

Fonctionnement : La fonction est itérative. Si le noeud actuel n'est pas nul, on le supprime jusqu'à ce qu'il n'y ait plus de noeud.

Complexité :

Dans la fonction, on devra parcourir tout l'arbre jusqu'à la fin pour bien détruire l'arbre. Dans le pire des cas, tous les noeuds de l'arbre seront testés et détruits. La fonction est donc en O(n).