Maxime Churin Robin Bria

NF16 - TP 4 – Arbre Binaire

Ce second TP noté de NF16 portait sur les arbres binaires. Après avoir réalisé une démonstration à notre chargé de TD nous allons ici justifier nos choix de programmation pour les différentes structures et fonctions. Enfin nous détailleront leurs complexités à l'aide des variables n (le nombres de nœuds de l'arbre passé en paramètre) et h (la hauteur de l'arbre passé en paramètre) . Dans le pire des cas h = n-1. De même si on a h=0 (un seul nœud la racine) on fait l'hypothèse que O(h) s'apparente à O(1).

Nous avons définit la structure de l'arbre binaire de recherche comme un nœud contenant une clé, un pointeur sur son fils gauche et un pointeur sur son fils droit.

Listes des fonctions supplémentaires et raison de ces choix :

*Noeud * creerNoeud(int n)* : il nous a paru plus propre de créer une fonction qui retourne le nœud à insérer que de le faire directement dans la fonction d'insertion.

*int rechercheArbre(Nœud * root)* : pour sécuriser les entrées dans notre code nous avons créé une fonction qui vérifie que l'arbre existe (sa racine différente de NULL). Elle nous permet de vérifier l'existence ou non de l'objet lors d'une saisie de l'utilisateur par exemple.

*void lire(char *chaine, int longueur)* : il s'agit d'un scanf plus sécurisé. On utilise dans cette fonction un *fgets()* pour lire la chaîne, puis on recherche le retour chariot que l'on remplace par le caractère de fin de chaine. Cette fonction nous permet de redemander la saisie par l'utilisateur tant que la chaine lu ne comporte que des caractères de type retour chariot et/ou espace.

int testMenu(char choix), int testChiffre(char* choix)* : nous utilisons ces deux fonctions pour sécuriser l'entrée dans le menu et la saisie de valeur de nœud l'utilisateur. La chaîne passée en paramètre doit être composé essentiellement de chiffre.

*int maxArbre(Nœud * root), int minArbre(Nœud * root)* : nous retournons à partir d'un arbre passé en paramètre respectivement le max et le min de l'arbre.

Complexité de nos fonctions :

insererNoeud(iterative) **:** Cette fonction vérifie tous d'abord que l'arbre passé en paramètre n'est pas vide, si oui, on retourne directement la fonction *creerNœud()*. Sinon, elle parcourt l'arbre en fonction de la clé de l'élément. Si cette clé est supérieure à celle du nœud courant, on descend d'un niveau dans le sous arbre droit. De même pour la clé inferieure avec le sous arbre gauche. Cela jusqu'à tomber sur un fils gauche ou droit nul. Dans le cas d'une égalité de clé entre l'élément à insérer et la clé du nœud courant, on retourne juste l'arbre initial. Cette fonction contient une seule boucle principale qui dépend de la hauteur h. Elle fait aussi appel à la fonction *creerNoeud()* qui est en O(1). Ainsi la fonction *insererNoeud(iterative)* est en **O(h).**

Entrée : int n, Noeud *root
Sortie : Noeud *root

insererNoeud_rec(récursive) :
Le fonctionnement est quasiment identique à la fonction itérative. En effet on compare le nœud passé en paramètre à NULL, si il y a égalité on retourne directement la fonction *creerNœud(),* sinon on le compare à la clé où, selon le cas, on renvoie le fils gauche ou le fils droit. En cas d'égalité on renvoie l'arbre. Donc la fonction insererNoeud_rec(recursive) est en **O(h) .**

Entrée : int n, Noeud *root
Sortie : Noeud *root

minArbre(), max Arbre() : Dans cette fonction on vérifie tous d'abord que l'arbre passé en paramètre n'est pas vide puis on parcourt l'arbre jusqu'à son nœud le plus à gauche, respectivement le plus à droite pour enfin renvoyer la clé de se nœud qui est le min, respectivement le max. Celles-ci sont donc en **Ω(h)**, car elle parcourt l'arbre par niveau dans tous les cas.

Entrée : Noeud * root
Sortie : int n

verifier (): Dans cette fonction nous appelons deux sous fonctions : *minArbre()* et *maxArbre()* en O(h). Notre fonction va tester pour chaque nœud de l'arbre qu'il n'y a pas d'incohérence sur les sous arbres. A chaque itération, nous vérifions que le min du sous arbre droit du nœud courant est supérieur à la clé du nœud courant et réciproquement. Nous avons donc un appel pour le fils gauche et un appel pour le fils droit. On éxécute donc n-1 itérations représentant le nombre d'élément de l'arbre moins la racine. Cependant, a chaque itération, on calcule le min et/ou le max de l'arbre. La fonction est donc en **O(n*h).**

Entrée : Noeud * root
Sortie : int n

recherche() : Cette fonction possède une condition de boucle identique à celle de *insererNoeud(iterative)*. En effet, elle va comparer la clé recherchée avec la clé du nœud du courant jusqu'à arriver sur le nœud recherché ou un nœud vide. Ainsi la fonction *recherche()* est en **O(h).**

Entrée : int n, Noeud *root
Sortie : Noeud *root

recherche rec() : Cette fonction possède quasiment les mêmes conditions que la fonction *insererNoeud(recursive)*. En effet, elle va comparer la clé recherchée avec la clé du nœud passé en paramètre jusqu'à tomber sur le nœud recherché ou un nœud vide. Ainsi la fonction *recherche_rec()* est en **O(h).**

Entrée : int n, Noeud *root
Sortie : Noeud *root

hauteur() : Par convention un arbre vide à une hauteur égale à -1, on commence donc pas faire ce test. Ensuite la fonction appelle *maximum()* qui réalise un simple test entre ceux deux paramètres grâce à un *if*, *maximum()* est donc en **Ω(1)**. Les deux paramètres de *maximum()* sont des appels récursif de *hauteur()* sur le fils droit et le fils gauche du nœud courant. Cette fonction va, pour chaque nœud, réaliser deux appels à *hauteur()*. La fonction est donc en **Ω(2n+1)= Ω(n).**

Entrée : Noeud * root
Sortie : int n

somme() : Si le noeud passé en paramètre est null on renvoie 0. Sinon on réalise des appels récursif de *somme()* sur le fils droit et le fils gauche du nœud courant. Cette fonction va, pour chaque nœud, réaliser deux appels à *somme()*. La fonction est donc en **Ω(2n+1)= Ω(n).**

Entrée : Noeud * root
Sortie : int n

afficherDecroissant() : Cette fonction fonctionne de la même façons que les deux précédentes. En effet, elle va faire un appel récursif avec comme paramètre le fils droit du nœud, afficher sa clé et faire un appel

récursif avec comme paramètre le fils gauche du nœud. Comme la fonction printf est en O(1). Nous avons donc la fonction *afficherDecroissant()* qui est en **Ω(2n+1)= Ω(n).**

Entrée : Noeud * root
Sortie : void

afficherStructure() : Cette fonction contient beaucoup de condition de vérification, afin de permettre un affichage identique à celui demandé dans la consigne. Cependant, le principe récursif est assez simple, il est en effet inversement identique à celui de la fonction ci-dessus (affichage croissant ici). Cette fonction va donc être appeler pour tous les éléments de l'arbre binaire passés en paramètre excepter les feuilles. La fonction *afficherStructure()* est en **Ω(n).**

Entrée : Noeud * root
Sortie : void

supprimer() : Avant d'effectuer la suppression de l'élément il faut d'abord vérifier son existence dans l'ABR et récupérer son père. Cette première partie s'apparente à la recherche d'un élément dans l'arbre, elle est donc en O(h). Si le nœud à été trouvé on peut passer à la suppression. On distingue 3 cas : le nœud est une feuille, le nœud a un seul fils et le nœud à deux fils. On réalise donc différents tests pour vérifier le cas rencontré :

- Si le nœud est une feuille, on met à jour le lien avec le père si père il y a.
- Si le nœud a un seul fils, on remplace le nœud par son fils soit en mettant à jour le lien avec son père si père il y a, soit en changeant la racine (ce nœud) par son seul fils.
- Si le nœud à deux fils, on va remonter le sous arbre dont la hauteur est la plus grande. Pour cela on appelle la fonction hauteur puis selon les cas on change la clé du nœud par le min ou le max du sous-arbre. Enfin on supprime ce nœud dont on a récupérer la clé et qui a au plus un seul fils.

Enfin on *free()* le nœud. et on retourne l'arbre mis correctement à jour. Dans cette deuxième partie qui réalise majoritairement des opérations élementaires en O(1), on fait dans le cas ou le nœud a deux fils un appel à *supprimer()* et un appel à *minArbre()* ou *maxArbre()* de complexité Ω(h). La fonction à donc une complexité en **O(h).**

Entrée : int n, Noeud *root
Sortie : Noeud * root

detruire() : Cette fonction a pour but de détruire l'arbre passé en paramètre. Elle va donc pour chaque nœud faire un appel récursif avec comme paramètre son fils droit et un appel récursif avec comme paramètre son fils gauche si ceux-ci ne sont pas nuls. Une fois tous ses appels effectués, on free() un par un les nœuds en partant du niveau de valeur le plus élevé grâce aux appels récursifs. La fonction free étant en O(1), la fonction *detruire()* est en **O(n).**

Entrée : Noeud * root
Sortie : void

construire() : Cette fonction a pour but de construire un arbre à partir d'un tableau passé en paramètre. On va utiliser le principe de la dichotomie. On vérifie les conditions sur les deux bornes puis on calcule leur moyenne. On crée alors le nœud correspondant à l'indice moyenne du tableau. On affecte au sous arbre gauche de ce nœud l'appel récursif sur la partie gauche du tableau (borne inf à la moyenne-1) et au sous arbre droit de ce nœud l'appel récursif sur la partie droite du tableau (moyenne+1 à la borne sup). Ainsi la fonction va être appelée pour tous les éléments de ce tableau sauf la moyenne. Comme elle utilise seulement la sous fonction *creerNoeud()* qui est en O(1). La fonction *construire()* est en **O(n)** avec n le nombre d'élément du tableau (accessoirement ce sera aussi le nombre de nœud de l'arbre).

Entrée : Noeud * root
Sortie : void

creerNoeud() : Cette fonction fait appel à la fonction malloc et des opérations élémentaires en O(1). La fonction *creerNoeud()* est donc en **O(1).**

Entrée : int n
Sortie : Noeud *root

Fonction de verification du main test(testChiffre(), testMenu()): Cette fonction fait appel à une boucle *while{}* qui s'exécute tant que la chaine lu respecte les condition du test. Donc la fonction est en **O(k)** avec k la longueur de la chaine.

Entrée : char tab[]
Sortie : int n

lire() : Cette fonction fait appel à une boucle *do{}while{}* qui s'exécute tant que la chaine lu comporte uniquement des caractères de type retour chariot et/ou espace, sa complexité est donc en O(k) avec k le nombre de saisie successives jusqu'à arriver à une entrée valide. De plus à chaque itération de cette boucle on réalise un appel de *fgets()* et *strchr()* qui sont de complexité O(1) car majoré par la constante du maximum de la taille de la chaîne. Pour finir on fait un appel de *fflsush()* de complexité O(1), donc la fonction est en **O(k).**

Entrée : char tab[],int longeur
Sortie : void

rechercheArbre() : Cette fonction est un simple test à l'aide d'un if, elle est donc en **O(1).**

Entrée : Noeud * root
Sortie : int n

Ce TP nous a permis de voir/revoir en détail les fonctions de base liées à un arbre binaire de recherche. Ce code pourrait notamment être réutilisé pour gérer un programme grâce aux arbres binaires. Nous avons décidé de procéder à de nombreuses sécurisation de notre code d'où la présence de ces quelques fonctions supplémentaires.