RAPPORT NF16 TP4

Ce projet nous a fait travailler sur les arbres binaires de recherche, nous devions modéliser une base de données d'une association. Chaque tranche d'âge de bénévole est représentée par un noeud de l'arbre binaire qu'il nous a fallu manipuler en implémentant des fonctions Inserer_ABR, Recherche_ABR, Supprimer_ABR spécifiques au problème.

FONCTIONS TP

Benevole *nouveauBen(char *nom, char* prenom, int CIN, char sexe, int annee);

Tranche *nouvelleTranche(int borneSup);

ListBenevoles *nouvelleListe();

Pour ces 3 fonctions nous allouons de la mémoire pour la structure que nous voulons créer, puis nous initialisons ces structures grâce aux paramètres donnés (pour nouvelleListe nous initialisons ces champs à NULL ou 0 car la liste ne contient pas de bénévole). Nous réalisons un nombre constant d'opération dans chacune de ces fonctions et le nombre d'opération ne dépend pas de la taille des données passées en paramètre donc la complexité de ces 3 fonctions est de l'ordre de O(1).

Tranche *ajoutTranche(Tranche *racine, int BorneSup);

Cette fonction permet d'ajouter une tranche dans l'arbre passé en paramètre grâce à sa racine. Cette fonction parcourt tout l'arbre jusqu'à trouver la tranche père adéquate.

Dans le pire des cas on a donc une complexité de O(h) h étant la hauteur de l'arbre, étant donné que la tranche est à ajouter à la fin de l'arbre mais au minimum elle est de Ω(1), si la tranche est à ajouter juste après la racine.

Benevole* insererBen(Tranche *racine, Benevole *benevole);

Dans cette fonction, nous allons d'abord cherche la tranche dans laquelle doit se trouver le bénévole passé en paramètre, pour cela on va parcourir l'arbre depuis la racine jusqu'aux feuilles en calculant l'âge du bénévole grâce à la fonction anneeActuelle(). Tant qu'on n'a pas trouvé la tranche à laquelle doit appartenir le bénévole et tant qu'on n'est pas arrivé jusqu'aux feuilles, on va parcourir l'arbre, si l'âge du bénévole est supérieur au maximum de la tranche alors on passe au fils droit, sinon on passe au fils gauche. Si la tranche existe, alors on va parcourir la liste des bénévoles de la tranche pour placer le bénévole pour conserver un ordre croissant des âges (donc un ordre décroissant des années de naissance). Si la tranche n'existe pas on l'ajoute dans l'arbre avant d'ajouter le bénévole dans cette nouvelle tranche. Dans le pire des cas, on va parcourir l'arbre en profondeur jusqu'au feuilles, puis on va parcourir la liste des bénévoles de la tranche jusqu'au bout.

Si on prend h comme la hauteur de l'arbre et m comme le nombre de bénévole dans l'arbre (dans le pire des cas, tous les bénévoles se trouvent dans la même tranche), alors la complexité est O(h+m). Dans le meilleur des cas, le bénévole doit être insérer dans la racine et est le premier bénévole de la liste, la complexité devient alors Ω(1).

Benevole * chercherBen(Tranche *racine, int CIN, int annee);

Cette fonction permet de chercher un bénévole particulier selon sa date de naissance et son numéro d'identité nationale.

Dans le pire des cas nous avons une complexité de O(m+h), h étant la hauteur de l'arbre et m le nombre de bénévoles la tranche correspondant à l'âge du bénévole recherché. En effet, dans ce cas-là le bénévole se situe à la fin de l'arbre et à la fin de la liste de bénévoles de cette tranche. Dans le meilleur des cas, le bénévole se situe dans la racine de l'arbre et en première position. On a alors à ce moment-là une complexité de Ω(1)

int supprimerBen(Tranche *racine, int CIN, int annee);

Pour cette fonction, on va commencer par parcourir l'arbre à partir de la racine jusqu'aux feuilles pour retrouver la tranche dans laquelle le bénévole que l'on veut supprimer peut potentiellement se trouver. Si la tranche existe, alors on va parcourir la liste des bénévoles de cette tranche par rapport au CIN. Si jamais le bénévole n'existe pas alors on sort de la. Sinon, on le supprime en réajustant le chainage pour conserver la liste chainée dans le bon ordre. Dans le pire des cas la tranche se trouve au niveau des feuilles et contient tous les bénévoles de l'arbre donc la complexité est de l'ordre de $O(h + m)$ avec h la hauteur de l'arbre et m le nombre de bénévole de l'arbre. Dans le meilleur des cas elle est de Ω(1).

int supprimerTranche(Tranche **racine, int borneSup);

Dans cette fonction, nous allons chercher la tranche à supprimer par rapport à sa borne supérieure. On va parcourir l'arbre en profondeur, depuis la racine jusqu'aux feuilles. Si la borne supérieure passée en paramètre est supérieur à la tranche de l'arbre testé alors on passe à son fils droit, sinon on passe à son fils gauche. On répète ce processus tant qu'on n'a pas trouvé la tranche à supprimer et tant qu'on n'est pas arrivé aux feuilles. Si jamais on a trouvé une tranche à supprimer alors on a différencié 3 cas :

- La tranche n'a pas de fils : On supprime le lien entre la tranche et son père (on donne la valeur NULL au pointeur de fils droit ou gauche, dépendant de la valeur de la borne supérieur de la tranche à supprimer par rapport à celle de son père). Puis on libère la mémoire allouée grâce à la fonction free().
- La tranche a un seul fils (droit ou gauche) : On raccorde le fils de la tranche à supprimer au père de cette tranche (le pointeur fils droit ou fil gauche du père reçoit le fils droit ou gauche de la tranche à supprimer). Puis on libère la mémoire de cette tranche grâce à la fonction free().
- La tranche a deux fils : On va chercher le successeur de la tranche à supprimer grâce à la fonction minimum(Tranche *racine) (on va prendre le minimum du fils droit de la tranche à supprimer) on va placer toutes les informations de ce successeur dans la tranche à supprimer, puis on va rappeler la fonction supprimerTranche avec comme paramètre l'adresse de l'ancienne place du successeur ainsi que sa borne supérieure, pour pouvoir supprimer définitivement cette tranche.

Chaque fois qu'on supprime une tranche, on va parcourir sa liste de bénévoles pour supprimer chaque bénévole et libérer la mémoire alloué.

Dans le pire des cas, si on prend h comme étant la hauteur de l'arbre, et m comme étant le nombre de bénévoles dans l'arbre (dans le pire des cas, tous les bénévoles de l'arbre sont situé dans

la tranche à supprimer), alors la complexité de la fonction est de l'ordre de O(h + m). En effet, dans le pire des cas, la tranche à supprimer se trouve au niveau des feuilles et on doit supprimer tout les bénévoles de l'arbre (qui se trouve dans la tranche à supprimer).

Dans le meilleur des cas, on doit supprimer la racine n'ayant aucun bénévole, la complexité est alors en Ω(1).

On utilise un double pointeur pour la racine pour qu'elle puisse s'actualiser plus facilement.

ListBenevoles *BendHonneur(Tranche *racine);

Cette fonction permet de retourner la liste des bénévoles les plus âgés de l'arbre. Pour cela, la fonction parcours 1 fois l'arbre pour trouver la tranche correspondante. Ensuite, elle parcourt la tranche correspondante deux fois. Une première fois pour déterminer l'âge maximal et une deuxième fois pour ajouter tous ces bénévoles à une nouvelle liste.

Les fonctions utilisés à l'intérieur étant en O(1) nous avons dans le pire des cas une complexité en O(h + 2m), h étant la hauteur de l'arbre correspondant et m le nombre de bénévoles de la tranche correspondant aux bénévoles les plus âgés. Dans le meilleur des cas, nous avons une complexité en Ω(m), la tranche recherchée étant la racine.

int actualiser(Tranche* racine);

Dans cette fonction nous avons eu besoin d'une structure pile pour pouvoir parcourir tout l'arbre pour pouvoir l'actualiser. Pour actualiser l'arbre on va parcourir l'arbre depuis la racine, on va commencer par empiler les fils de la racine, actualiser la racine, puis on va dépiler dans une variable, empiler les fils de cette variable, actualiser la variable et ainsi de suite, tant que la pile n'est pas vide. Pour actualiser une tranche, on va parcourir chaque bénévole de cette tranche, si jamais l'âge du bénévole est supérieur à la borne supérieur de la tranche alors on va retirer le bénévole de la liste des bénévoles puis on va utiliser la fonction insererBen pour réinsérer le bénévole dans l'arbre, donc à la bonne tranche. Si il n'y a plus de bénévole dans cette tranche, alors on supprime la tranche en appelant la fonction supprimerTranche.

Si on prend h comme étant la hauteur de l'arbre et m le nombre de bénévole dans l'arbre, alors dans le pire des cas la complexité est de l'ordre de O(m*(h+m)). En effet, dans le pire des cas on va parcourir chaque bénévole de l'arbre et l'insérer dans une nouvelle branche, donc pour chaque bénévole on va réaliser (h+m) opérations en appelant la fonction insererBen.

int totalBenTranche(Tranche *racine, int borneSup);

Ici on parcourt l'arbre pour trouver la tranche correspondante à la borne supérieur passé en paramètre. Ensuite, on récupère le nombre d'élément de la liste de bénévoles.

Nous avons donc dans le pire des cas une complexité de O(h), h étant la hauteur de l'arbre. Dans le meilleur des cas, nous obtenons une complexité de Ω(1) la tranche recherchée étant la racine.

int totalBen(Tranche* racine);

Dans cette fonction on va encore utiliser la structure de pile pour pouvoir parcourir toutes les tranches de l'arbre. Pour chaque tranche parcourue, on va ajouter le nombre d'élément de sa liste de bénévole dans une variable tot qui va contenir le nombre total de bénévole dans l'arbre.

Si on prend n comme étant le nombre de tranche dans l'arbre, alors dans le pire des cas la complexité de la fonction est de l'ordre de O(n). En effet, on va réaliser un nombre constant

d'opérations pour chaque tranche. Dans le meilleur des cas, la seule tranche de l'arbre est la racine, donc la complexité devient Ω(1).

float pourcentageTranche(Tranche * racine, int borneSup) ;

Cette fonction calcule le pourcentage de bénévoles contenus dans une tranche en s'aidant des deux fonctions précédentes. On a donc dans le pire des cas une complexité de O(h+n) et dans le meilleur des cas une complexité en Ω(1).

void afficherTranche(Tranche* racine, int borneSup);

Dans cette fonction on va parcourir l'arbre de la racine jusqu'aux feuilles pour trouver la tranche à afficher, ensuite on va parcourir sa liste de bénévole pour les affiches un a un. Si la borne supérieure passée en paramètre est supérieur à la tranche de l'arbre testé alors on passe à son fils droit, sinon on passe à son fils gauche. On répète ce processus tant qu'on n'a pas trouvé la tranche à afficher. SI la tranche existe, alors on va parcourir chaque bénévole de la liste des bénévoles et l'afficher.

Si on prend h comme étant la hauteur de l'arbre et m le nombre de bénévole dans l'arbre, alors dans le pire des cas, on doit afficher tous les bénévoles de l'arbre contenus dans une seule tranche se trouvant au niveau des feuilles, la complexité est alors de l'ordre de O(h + m).

Dans le meilleur des cas on doit afficher la racine qui ne contient qu'un seul bénévole, la complexité devient alors Ω(1).

void afficherArbre(Tranche *racine);

Grâce à la pile et au système de marquage, cette fonction permet de parcourir l'arbre de façon infixe et donc d'obtenir toutes les tranches dans l'ordre croissant des bornes supérieurs. De plus, après avoir parcourus l'arbre, on le parcourt de nouveau pour retirer tout marquage.

La complexité de cet algorithme est de Θ(n), n étant le nombre de tranche présente dans l'arbre.

void detruireArbre(Tranche * racine);

Cette fonction parcours grâce à une pile l'arbre en entier pour pouvoir détruire chaque tranche. La complexité est donc la même que précédente Θ(n)

FONCTIONS AJOUTÉES

Tranche* minimum(Tranche* racine);

Pour réaliser ce TP nous avons eu besoin de créer cette fonction qui renvoi le minimum d'un arbre, en partant de la racine, dans le cas où la racine a un fils gauche. La fonction va parcourir l'arbre en allant le plus à gauche possible.

Si on prend h comme étant la hauteur de l'arbre, alors la complexité de la fonction devient de l'ordre de O(h).

```
pileTranche* creerPile();
void empiler(pileTranche* p, Tranche* t);
int pileVide(pileTranche* p);
Tranche* depiler(pileTranche* p);
void marquer(Tranche *t) ;
int elementMarque(Tranche *t) ;
```

Pour certaines fonctions de ce TP nous avons eu recours à une pile (lorsque l'on devait parcourir toutes les tranches de l'arbre). Nous avons donc créé des fonctions pour gérer facilement la structure de pile. Chaque fonction est en O(1) car on réalise un nombre constant d'opérations pour chaque une. Pour la structure pile on a utilisé un tableau de pointeur sur tranche et un entier correspondant à l'indice du sommet de la pile.