Marion CHAN-RENOUS

RAPPORT NF16 -TP4

Les Arbres Binaires de Recherche

Introduction

Le but de ce TP4 était de créer un programme permettant de structurer les informations sur les bénévoles d'une association, en forme d'Arbre Binaire de Recherche. Chaque nœud de cet ABR est la borne supérieure d'une tranche d'âge donnée.

Fonctions requises

Fonctions d'initialisation des structures

- `Benevole *nouveauBen(char *nom, char *prenom , int CIN, char sexe, int annee);`
- `Tranche *nouvelleTranche(int borneSup);`
- `ListBenevoles *nouvelleListe();`

Ces trois fonctions servent à initialiser les différents types de structure. Les **entrées** de celles-ci sont les éléments nécessaires à leur initialisation : par exemple, pour la création d'un nouveau bénévole, les entrées sont : son nom, son prénom, son numéro de carte d'identité, son sexe et son année de naissance. Les **sorties** de ces trois fonctions sont des pointeurs sur le type de structure crée : par exemple, la création d'un nouveau bénévole via la fonction nouveauBen renvoie un pointeur sur le type Benevole. Les instructions dans chacune de ces trois fonctions sont des instructions élémentaires (affectation, allocation mémoire), leur complexité est donc la même dans le meilleur et dans le pire des cas. Ainsi, leur **complexité** est en $\Omega(1)$ et $O(1)$, donc en $\Theta(1)$.

Fonctions d'ajout

- `Tranche *ajoutTranche(Tranche *racine , int borneSup);`

Cette fonction permet d'ajouter un nœud dans l'ABR. Les **entrées** de cette fonction sont la racine de l'ABR qui correspond à un pointeur sur le type Tranche et la valeur de la borne supérieure du nœud à ajouter. La **sortie** de cette fonction est un pointeur sur la tranche crée. Afin d'ajouter le nœud dans l'ABR, il est nécessaire de trouver la place adéquate dans celui-ci pour insérer le nœud selon les propriétés d'un ABR. Dans le pire des cas, il faudra parcourir les nœuds de chaque profondeur de l'arbre. Ainsi, la **complexité** de cette fonction est donc en $O(h)$, h étant la hauteur de l'arbre.

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

Cette fonction permet d'ajouter un nouveau bénévole dans l'ABR. Les **entrées** de cette fonction sont la racine de l'ABR et un pointeur sur le bénévole à insérer. La **sortie** de cette fonction est un pointeur sur le bénévole inséré. Si le bénévole est déjà présent dans la liste, cette fonction renverra le pointeur nul. Afin de programmer cette fonction, nous avons décidé d'utiliser plusieurs fonctions supplémentaires (détaillées plus loin dans ce rapport). « calculBorne » dont la complexité est en $O(1)$, « chercherTranche » dont la complexité est en $O(h)$ (h étant la hauteur de l'ABR), et « insererListBen » dont la complexité est en $O(n)$ (n étant le nombre de bénévoles dans la tranche). La fonction ajoutTranche (en $O(n)$) est également appelée dans le cas où la tranche d'âge du bénévole n'est pas encore présente dans l'ABR. Les autres instructions sont des instructions élémentaires ($O(1)$). Nous pouvons considérer que le nombre n de bénévoles sera toujours supérieur ou égal à la hauteur de l'arbre. En effet, selon l'énoncé, une tranche aura forcément au moins un bénévole dedans, sinon celle-ci devra être supprimé. Ainsi, nous pouvons conclure que la **complexité** de la fonction insererBen est en $O(n)$.

Fonction de recherche

- `Benevole *chercherBen(Tranche *racine ,int CIN, int annee)`

Cette fonction permet de chercher un bénévole dans l'ABR. Les **entrées** de cette fonction sont donc la racine de l'ABR, le CIN du bénévole et son année de naissance. La **sortie** de cette fonction est un pointeur sur le bénévole recherché. Si le bénévole recherché n'est pas dans l'ABR, cette fonction renvoie le pointeur nul. Cette fonction appelle la fonction supplémentaire « chercherTranche » dont la complexité est O(h). Dans le pire des cas, la liste des bénévoles devra être parcourue totalement afin de trouver le bénévole recherché. Les autres instructions sont des instructions élémentaires. Le nombre n de bénévoles étant toujours supérieur ou égal à la hauteur de l'arbre, la **complexité** de cette fonction est O(n).

Fonctions de suppression

- `int supprimerBen(Tranche **racine , int CIN, int annee)`

Cette fonction permet de supprimer un bénévole dans l'ABR. Les **entrées** de cette fonction sont la racine de l'ABR, et le CIN et l'année de naissance du bénévole à supprimer. La **sortie** de cette fonction est 0 si tout s'est bien passé, 1 sinon. Cette fonction appelle la fonction supplémentaire « chercherTranche » dont la complexité est O(h). Comme pour la fonction précédente, dans le pire des cas la liste des bénévoles devra être parcourue totalement. Le nombre n de bénévoles étant toujours supérieur ou égal à la hauteur de l'arbre, la **complexité** de cette fonction est O(n).

- `int supprimerTranche(Tranche **racine , int borneSup)`

Cette fonction permet de supprimer un bénévole dans l'ABR. Les **entrées** de cette fonction est un double pointeur sur la racine de l'ABR (afin de pouvoir modifier ce pointeur) et la borne supérieure de la tranche à supprimer. La **sortie** de cette fonction est 0 si tout s'est bien passé, 1 sinon. Tout d'abord, il est nécessaire de trouver la tranche à supprimer, et son père. Cette procédure est en O(h), h étant la hauteur de l'arbre. Si la tranche est trouvée, il est ensuite nécessaire de distinguer 3 cas :

- Si la tranche à supprimer n'a pas de fils : il suffit de supprimer la feuille.
- Si la tranche à supprimer à un fils : on remplace le nœud par son fils.
- Si la tranche à supprimer à deux fils : nous avons choisi de remplacer le nœud par son prédécesseur dans l'ABR (le nœud maximum de son sous arbre gauche).

Dans les 3 cas il est nécessaire de désallouer la mémoire allouée à tous les bénévoles présents dans la liste de bénévoles de la tranche, puis de desallouer la mémoire allouée à la liste en elle-même. Nous avons choisi d'implémenter une fonction supplémentaire « supprimerListBen », dont la complexité est O(n) avec n le nombre de bénévoles dans la liste, pour effectuer cela.
Le nombre n de bénévoles étant toujours supérieur ou égal à la hauteur de l'arbre, la **complexité** de cette fonction est O(n).

Autres fonctions

- `ListBenevoles *BenDhonneur(Tranche *racine)`

Cette fonction permet de créer une liste avec les bénévoles les plus âgés de l'association. L'**entrée** de cette fonction est donc la racine de l'ABR et la **sortie** est un pointeur sur la liste des bénévoles. Afin de trouver les bénévoles les plus âgés il est nécessaire de trouver le nœud de l'ABR ayant la borne supérieure la plus élevée, grâce à la fonction maximum, dont la complexité est en O(h). Ensuite, à partir de cette tranche il est nécessaire de connaitre l'âge maximal des bénévoles de cette tranche. On parcourt donc la liste des bénévoles jusqu'au dernier (complexité en O(n), n étant le nombre de bénévoles dans la liste). On récupère ainsi l'âge maximal. Ensuite, on parcourt une nouvelle fois la liste des bénévoles jusqu'à atteindre le premier ayant l'âge maximal. On affecte fait pointer le premier élément de la liste sur ce bénévole, les suivants seront ainsi toujours chaînés à lui jusqu'au dernier qui pointera sur NULL ce qui permettra leur affichage. On compte le nombre de bénévoles d'honneur afin de mettre à jour le champ « nombre d'éléments » de la liste

en parcourant la liste entièrement : complexité en O(n). Le nombre n de bénévoles étant toujours supérieur ou égal à la hauteur de l'arbre, la **complexité** de cette fonction est O(n).

Nous avons programmé deux versions de la fonction actualiser

- `int actualiser(Tranche **current, Tranche **racine)`

Cette fonction permet d'actualiser l'ABR en déplaçant les bénévoles lorsqu'il ne font plus partie d'une tranche d'âge, lors d'un changement d'année. L'**entrée** de cette fonction est la racine de l'ABR et un double pointeur sur la tranche en cours de traitement et la **sortie** est le nombre de personnes déplacées. Nous avons défini cette fonction comme devant être appelée à chaque changement d'année. Ainsi, si l'on change d'année, les bénévoles à déplacer seront les bénévoles ayant précédemment l'âge correspondant à la borne supérieure de la tranche dont ils font partie. Ainsi, par exemple, les bénévoles à déplacer de la tranche dont la borne supérieure est 50 si l'on change d'année sont ceux qui ont 51 ans l'année précédente. Cette fonction parcourt l'arbre récursivement. Ainsi pour chaque tranche de l'ABR parcourue, nous allons :

- Trouver l'année de naissance correspondant à la borne supérieure de la tranche
- Trouver s'il y a des bénévoles à déplacer dans la liste de bénévoles
- S'il y a des bénévoles à déplacer : on cherche si la tranche où ils doivent être dorénavant existe :
    - Si elle n'existe pas on la crée
    - Sinon on ajoute les bénévoles à déplacer en tête de liste de la tranche (pour respecter le tri par ordre croissant d'âge car ils seront forcément les moins âgés de la tranche)
- Si la tranche est vide suite au déplacement : on la supprime
- On passe à la tranche suivante

Le nombre n de bénévoles étant toujours supérieur ou égal à la hauteur de l'arbre, la **complexité** de cette fonction est O(n).

- `int totalBenTranche(Tranche *racine , int borneSup)`

Cette fonction permet de calculer le total des bénévoles d'une tranche d'âge. Les **entrées** de cette fonction sont la racine de l'ABR et la borne supérieure de la tranche en question. La **sortie** de cette fonction est le nombre de bénévoles. Cette fonction renvoie -1 dans le cas où la tranche n'est pas présente dans l'ABR. Cette fonction fait appel à la fonction « chercherTranche » dont la complexité est O(h). Les autres instructions sont des instructions élémentaires. Ainsi, la **complexité** de cette fonction est O(h).

- `int totalBen(Tranche *racine)`

Cette fonction permet de calculer le nombre total de bénévoles de l'ABR. L'**entrée** de cette fonction est la racine de l'ABR et la **sortie** est le nombre de bénévoles. Cette fonction s'appelle récursivement, jusqu'à ce que l'on atteigne la fin de l'arbre (feuilles). Ainsi, la **complexité** de la fonction est O(n) avec n le nombre de nœuds de l'ABR.

- `float pourcentageTranche(Tranche *racine , int borneSup)`

Cette fonction permet de calculer le pourcentage de bénévoles dans une tranche d'âge. Les **entrées** de cette fonction sont la racine de l'ABR et la borne supérieure de la tranche en question. La **sortie** de cette fonction est le pourcentage de bénévoles. Cette fonction appelle la fonction « chercherTranche », dont la complexité est O(h), afin de trouver la tranche en question. Puis la fonction appelle la fonction « totalBen », dont la complexité est en O(n), afin de calculer le

pourcentage. Le nombre n de bénévoles étant toujours supérieur ou égal à la hauteur de l'arbre, la **complexité** de cette fonction est O(n).

Fonctions d'affichage

- `void afficherTranche(Tranche *racine , int borneSup)`

Cette fonction permet d'afficher une tranche de l'ABR. Les **entrées** de cette fonction sont la racine de l'ABR et la borne supérieure de celle-ci. Cette fonction appelle la fonction « chercherTranche », dont la complexité est O(h), afin de trouver la tranche en question. Puis, la fonction parcourt la totalité de la liste de bénévoles afin d'en afficher le contenu, la complexité du parcourt est donc O(n) avec n le nombre de bénévoles de la liste. Le nombre n de bénévoles étant toujours supérieur ou égal à la hauteur de l'arbre, la **complexité** de cette fonction est O(n).

- `void afficherArbre(Tranche *racine)`

Cette fonction permet d'afficher l'ABR. L'**entrée** de cette fonction est donc la racine de celui-ci. Afin d'afficher l'ABR, nous avons utilisé un parcours infixe afin d'afficher les tranches d'âge dans l'ordre croissant. Ainsi la **complexité** de cette fonction est O(n) avec n le nombre de nœuds de l'ABR.

Fonctions supplémentaires

Nous avons choisi d'implémenter plusieurs fonctions supplémentaires à celles imposées, dans un souci d'efficacité.

- `int calculBorne(int annee)`

Cette fonction calcule et renvoie la valeur de la borne supérieure de la tranche d'âge associée à une année de naissance fournie en paramètre. Cette fonction ne comporte que des instructions élémentaires, sa complexité est dont O(1).

- `Tranche *chercherTranche(Tranche *racine, int borneSup)`

Cette fonction permet de chercher une tranche dans l'ABR dont la valeur de la borne supérieure est celle fournie en paramètre. Cette fonction renvoie un pointeur sur la tranche si celle-ci est présente dans l'ABR, et le pointeur nul si la tranche n'existe pas. Sa complexité est en O(h), avec h la hauteur de l'arbre.

- `int insererListBen(Tranche *tranche, Benevole *benevole)`

Cette fonction permet d'ajouter le bénévole passé en paramètre dans la liste de bénévoles de la tranche passée en paramètre. Cette fonction renvoie 0 si tout s'est bien passé, 1 sinon. Afin de respecter l'ajout du bénévole selon l'ordre croissant de l'âge, dans le pire des cas il sera nécessaire de parcourir toute la liste afin d'ajouter le bénévole. Ainsi, la complexité de cette fonction est O(n), avec n le nombre de bénévoles dans la liste.

- `int supprimerListBen(Tranche *t);`

Cette fonction supprime la liste des bénévoles associée à la tranche passée en paramètre. La fonction s'occupe de désallouer la mémoire allouée à chaque bénévole, puis de désallouer la mémoire allouée à la liste en elle-même. Cette fonction renvoie 0 si tout s'est bien passé, 0 sinon. La fonction parcourt toute la liste de bénévoles, ainsi, la complexité de cette fonction est O(n) avec n le nombre de bénévoles dans la liste.

- `Tranche* maximum(Tranche *racine)`

Cette fonction sert à trouver la tranche dont la valeur de la borne supérieure est maximale à partir d'une tranche racine. La complexité de cette fonction est donc en O(h) avec h la hauteur de l'ABR.

- `void afficherBenDhonneur(ListBenevoles* l)`

Cette fonction permet l'affichage de la liste des bénévoles d'honneur passée en paramètre. Cette fonction s'occupe également de désallouer la mémoire allouée pour la liste à la fin de la fonction d'affichage. La complexité de cette fonction est donc O(n) avec n le nombre de bénévoles dans la liste.

- `void destructionArbre(Tranche** racine)`

Cette fonction permet de détruire l'arbre. Cette fonction parcourt l'arbre de façon récursive grâce à un parcours postfixe. Ainsi, pour chaque tranche, on supprime la liste de bénévoles de celles-ci grâce à la fonction « SupprimerListBen », et on désalloue le pointeur sur la Tranche. Le « flag » du main « arbreCree » est remi à 0 afin de forcer l'utilisateur du programme à initialiser un nouvel arbre suite à la destruction de l'ancien. Ainsi, la **complexité** de cette fonction est en O(n) avec n le nombre de tranches de l'ABR.

- `ListBenevoles* sauvegarderListe(ListBenevoles *l)`

Cette fonction permet de sauvegarder la liste de bénévoles d'un nœud de l'ABR dans le cas de la suppression d'un nœud ayant deux fils. En effet, nous avons choisi de remplacer le nœud à supprimer par le maximum du sous-arbre gauche du nœud. Ainsi, on remplace les données du nœud à supprimer par le maximum, il est donc nécessaire de sauvegarder la liste des bénévoles avant la suppression du maximum dans l'arbre. Cette fonction parcourt entièrement la liste de bénévoles à sauvegarder, la **complexité** est donc en O(n) avec n le nombre de bénévoles dans la liste.

Conclusion

Ce TP nous a aidé à comprendre la structure et le fonctionnement des ABR. Nous avons pu comprendre et programmer les fonctions utiles à leur utilisation.