# [NF16] TP4

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

La compléxité est en O(1) car il y a un nombre fini d'instructions. On initialise juste toutes les données dans le nouveau bénévole. Il faut bien allouer de la mémoire.

***Tranche *nouvelleTranche(int borneSup);***

La compléxité est en O(1) car il y a un nombre fini d'instructions. Idem on initialise la nouvelle tranche et on lui alloue de la mémoire.

***ListBenevoles *nouvelleListe();***

La compléxité est en O(1) car il y a un nombre fini d'instructions. Idem que précédemment.

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

La complexité dans le meilleur des cas est en O(1) car on ajoute la racine et on sort du programme. Sinon, on est dans le pire des cas en O(h) car on parcourt l'arbre jusqu'à trouver un embranchement vide : au maximum on parcourt la branche la plus longue qui est de hauteur h. Si la borne sup de la tranche à ajouter est plus grande que la tranche courante, alors on passe dans le fils droit de la tranche courante ; sinon on passe dans le fils gauche.
**Entrées :** pour modifier l'arbre, il faut que racine soit un pointeur sur tranche sinon les données ne seront pas modifiées. On n'a pas besoin de pointeur pour borneSup car c'est une valeur rentrée par l'utilisateur qu'on ne modifie pas.
**Sorties :** La sortie est un pointeur sur la tranche qui vient d'être créée.

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

Pour vérifier que le bénévole n'est pas encore enregistré, il faut juste vérifier pendant le parcours de la liste des bénévoles de la tranche si l'année du bénévole et le CIN du bénévole ne correspondent pas à ceux parcourus. Il n'y a pas besoin de vérifier toute la liste vu que l'on ajoute le bénévole en fonction de son âge grâce à la fonction AnneeActuelle().

La complexité dans le meilleur des cas est en O(1) dans le cas où l'âge calculé est inférieur à 0 ou que la racine n'est pas initialisée. Sinon, tant qu'il y a des tranches à parcourir et qu'on a pas encore trouvé la tranche dans laquelle insérer le bénévole alors on continue dans une boucle. Si on a trouvé la tranche alors dans le pire des cas il faut ajouter le bénévole à la fin de la liste, donc parcourir tous les bénévoles de la liste avant. La complexité max du parcours de tranche est en O(h) et soit m le nombre de bénévole dans la liste, la complexité du parcours de tous les bénévoles est en O(m). Vu qu'on ne parcourt pas la tranche tant qu'on n'a pas trouvé la bonne tranche, la complexité des deux boucles sont donc additionnées : la complexité est en O(h+m).

**Entrées :** La racine et un pointeur sur la structure bénévole qu'on veut ajouter à l'arbre.
**Sorties :** La sortie est un pointeur sur le bénévole qui vient d'être ajouté.

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

Pareil que dans insererBen, on parcourt les différentes tranches et si on la bonne tranche on parcourt tous les bénévoles de la liste correspondant à la tranche. Dans le pire des cas la complexité est en O(h+m) et dans le meilleur des cas en O(1) car on n'entre pas dans la boucle. (Le parcours est quasi identique à la fonction insererBen).
**Entrées :** La racine et les informations pour effectuer la recherche dans l'arbre : le CIN et l'année.
**Sorties :** La sortie est un pointeur sur le bénévole trouvé.

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

Si la racine n'existe pas on est en O(1). Sinon on se déplace dans l'arbre jusqu'à trouver la bonne tranche, le premier while est donc en O(h). Le second while parcourt la liste des bénévoles pour la tranche donnée : O(m) avec m le nombre de bénévoles dans la tranche. Dans le cas où il n'y a pas besoin de supprimer la tranche, la complexité totale est donc en O(h+m) car il suffit de raccorder le fils gauche ou le fils droit au père de la tranche courante en fonction de la position de la tranche par rapport à son père. Par contre, s'il faut supprimer la tranche, étant donné qu'on fait appel à supprimerTranche() qui est O(h+m), pour la complexité finale on additionne les deux complexités, la complexité finale est alors en O(h+m).
**Entrées :** Idem que pour la recherche.
**Sorties :** Un booléen indiquant ou non la réussite de la suppression.

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

Si la racine n'existe pas on est en O(1). Sinon on se déplace dans l'arbre jusqu'à trouver la bonne tranche, le premier while est donc en O(h). Le second while parcourt la liste des bénévoles pour la tranche donnée afin de les supprimer : O(nbBénévoles). On utilise un pointeur qui pointe sur le bénévole précédent à celui courant afin de faciliter la suppression. Il ne faut pas oublier de libérer la mémoire du bénévole ainsi que de toute la liste. Une fois que c'est fait, on peut enfin supprimer la tranche. Si elle n'a pas de fils ou bien seulement un gauche ou un droit, cette opération est en O(1). Si elle a un fils gauche ou un fils droit, il faut relier le père de la tranche à ce fils. Sinon, il faut trouver le successeur. Pour cela, on fait appel à la fonction utilisateur successeur() en O(h) (cf. explication fonction successeur). Enfin, on remplace la tranche par son successeur et on appelle la fonction récursivement. La complexité finale dans le pire des cas est donc en O(h+m) car même si on appelle la fonction récursivement, on l'appelle au pire 1 fois et la complexité additionnée donnera quand même O(h+m). Nous avons choisi de mettre une condition dans le cas où l'utilisateur veut supprimer la racine : supprimer la racine n'est donc pas possible ; cependant, s'il on avait voulu le faire, il aurait fallu mettre un double pointeur pour racine pour pouvoir modifier et sauvegarder la racine en sortant de la fonction. Il aurait aussi fallu mettre des doubles pointeurs de racine dans le fichier main.
**Entrées :** La racine et la clé de la tranche à supprimer.
**Sorties :** Un booléen indiquant ou non la réussite de la suppression.

***ListBenevoles *BenDhonneur(Tranche *racine);***

On commence ici par chercher la tranche la plus vieille, on a donc une boucle en O(h). S'il n'y a pas de bénévole, on remonte dans l'ABR, au pire des cas cette boucle sera elle aussi en O(h). Donc la complexité des deux est en O(h+h) = O(h). On parcourt à l'aide d'un while en O(m) la tranche où se trouve le bénévole d'honneur. On parcourt à nouveau la liste afin de mettre le pointeur du début de la

nouvelle liste au premier élément correspondant ; cette partie est en O(m) car on parcourt encore dans le pire des cas tous les bénévoles de la liste jusqu'au dernier. On additionne donc les complexités obtenues : O(h+h+m+m) et on obtient donc une complexité en O(h+m).
**Entrées :** La racine.
**Sorties :** Une liste du ou des bénévoles d'honneurs.

***int actualiser(Tranche *racine);***

On parcourt l'arbre à l'aide d'un parcours préfixe utilisant le principe des piles. On va donc parcourir toutes les tranches et voir tous les bénévoles de la tranche correspondent à la borne supérieure de leur tranche. Soit N le nombre total de bénévoles de tout l'arbre. Si chaque bénévole doit être déplacé, alors pour chaque bénévole il faut le "supprimer" O(1) et ensuite on utilise insererBen pour le remettre au bon endroit dans l'arbre, qui est en O(h+m). On va donc faire N fois supprimer et inserer (supprimer + insérer donnent une complexité de O(1+h+m) = O(h+m)) ce qui donne en complexité O(N*(h+m)).
**Entrées :** La racine.
**Sorties :** Un booléen indiquant ou non la réussite de l'actualisation.

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

On parcourt l'arbre comme précédemment et on lorsqu'on trouve la tranche correspondante, alors on retourne le nombre de bénévole affecté à la liste de la tranche. Dans le pire des cas on parcourt tout l'arbre donc la complexité dans le pire des cas est en O(h). Dans le meilleur des cas elle est en O(1) car on rentre pas dans la boucle si la racine est nulle.
**Entrées :** La racine et la clé de la tranche dont on veut connaître le nombre de bénévoles.
**Sorties :** Le nombre de bénévoles si ça a marché, -1 sinon.

***int totalBen(Tranche *racine);***

On fait un parcours préfixe à l'aide des fonctions de pile pour avoir le nombre total de bénévoles dans l'arbre. Pour chaque tranche parcourue, on donne le nombre de bénévole correspondante. Soit n le nombre de tranches dans l'arbre, on va donc parcourir toutes les tranches pour renvoyer le nombre ; la complexité sera donc O(n).
**Entrées :** La racine et la clé de la tranche à supprimer.
**Sorties :** Le nombre de bénévoles total.

***float pourcentageTranche(Tranche *racine, int borneSup);***

On fait ici appel à totalBenTranche() et totalBen() dans le calcul. La complexité est donc en O(n+h) car on additionne les complexités des deux fonctions.
**Entrées :** La racine et la clé de la tranche à supprimer.
**Sorties :** Le nombre total de bénévoles.

***void afficherTranche(Tranche *racine, int borneSup);***

Pour afficher la tranche correspondante à la borne supérieure donnée, lorsque l'on part de la racine, on va chercher à trouver la tranche correspondante. Si la borne supérieure est plus grande que la borne courante pointée par le pointeur parcours, alors parcours va prendre la valeur de son fils droit. Sinon celle de son fils gauche. On fera cette opération au maximum h fois qui est égal à la hauteur de l'arbre.

Lorsque l'on trouve la bonne tranche, on affiche tous les bénévoles de la tranche. Soit m le nombre de bénévoles de la tranche correspondante, on fera un affichage m fois. Donc dans le pire des cas, on aura une complexité de O(h+m).
**Entrées :** La racine et la clé de la tranche à afficher.

***void afficherArbre(Tranche *racine);***

Pour afficher les tranches de l'arbre, on utilise un parcours infixe qui permet de parcours les tranches dans l'ordre croissant de leur borne supérieure. On parcourt une fois chaque tranche donc soit n le nombre de tranches dans l'ABR, alors la complexité dans le pire des cas sera en O(n). Sinon, dans le meilleur des cas elle est en O(1) car une seule instruction lorsque la racine est nulle.
**Entrées :** La racine.

***void supprimerArbre(Tranche *racine);***

Pour supprimer l'arbre, il va falloir supprimer toutes les tranches de de l'arbre. On fait un parcours préfixe à l'aide d'une pile pour accéder à toutes les tranches. Pour chaque tranche, on va la supprimer donc soit n le nombre de tranches dans l'arbre, on va donc faire n*supprimerTranche.
SupprimerTranche est en O(h+m) donc supprimerArbre est en O(n*(h+m)).
**Entrées :** La racine.

## Fonctions utilisateurs :

***Tranche *successeur(Tranche *trch);***

Fonction recherchant la tranche qui succède à celle passée en paramètre. La complexité est en O(h) car on parcourt au maximum une branche entière dont la hauteur est la plus élevée ; on prend le minimum du sous-arbre droit de la tranche.

**Entrées :** Un pointeur sur la tranche dont on cherche le successeur.
**Sorties :** Un pointeur sur la tranche successeuse.

## Fonctions concernant la gestion des piles :

***Pile *creerPile();***
La compléxité est en O(1) car il y a un nombre fini d'instructions. On crée une pile en créant un pointeur qui serait mis à la tranche en haut de pile.

***void empiler(Tranche *trch, Pile *p);***
La compléxité est en O(1) car il y a un nombre fini d'instructions. S'il n'y a rien dans la pile, on met le pointeur tête sur cette tranche sinon s'il y a déjà une tranche alors on met le pointeur du suivant sur la tranche à empiler et la tête est mise sur cette nouvelle tranche.

***Tranche *depiler(Pile *p);***
La compléxité est en O(1) car il y a un nombre fini d'instructions.

***int pileVide(Pile *p);***
La compléxité est en O(1) car il y a un nombre fini d'instructions.