Alexandre EBERHARDT
Lola TRUPIN
A23

Rapport TP4 : Les ABR

L'objectif de ce TP est de développer un système pour gérer les inscriptions des étudiants aux UVs en utilisant des structures de données telles que les arbres binaires et les listes chaînées

1/ La liste des structures et des fonctions supplémentaires que vous avez choisi d'implémenter et les raisons de ces choix :

Nous avons de nombreuses fonctions supplémentaires, qui nous permettent de fractionner certaines actions.

T_Element * creerInscription(char* code) ; va créer l'élément de la liste chainé qui correspond au code entré en paramètre. Il renvoie un T_Element*, un pointeur vers un élément, qui pourra être utilisé dans la liste chainée des inscriptions. La fonction crée juste un élément donc elle est de complexité O(1).

T_Element* rechercherInscription (T_Element *liste, char* code) Cette fonction recherche dans la liste chainée si l'uv est déjà présente. Si oui elle renvoie NULL, sinon elle renvoie la chaine. Dans le pire des cas, si l'élément est à la fin de la chaine ou absent, nous parcourons toute la liste chainée, la complexité de la fonction est donc O(n).

T_Element*pred(T_Element *liste, char*code) la fonction pred sert à trouver le prédécesseur de d'un élément dans une liste chainée, afin de pouvoir l'ajouter correctement par la suite. Dans le pire des cas, l'élément se situe à la fin de la chaine, il faut donc la parcourir entièrement. La complexité est donc de l'ordre de O(n).

T_Arbre rechercherNoeud (T_Arbre abr, char *nom, char *prenom). La fonction recherche si un nœud est déjà dans l'arbre, l'arbre étant binaire et trié, sa complexité est ou O(h). En effet, dans le pire des cas, on parcourt la totalité de la hauteur de l'arbre pour aller chercher une feuille. Pour être encore plus précis, dans le pire des cas, l'arbre a une hauteur de n éléments. La complexité est donc de O(n).

T_Arbre creerNoeud(char *nom, char *prenom, char *code) La fonction crée un Nœud de l'arbre binaire en lui allouant de la mémoire et crée la première inscription. La première inscription ne nécessite pas de parcourir la liste car elle commence vide. La complexité de creerInscription est donc O(1). Toutes les autres fonctions utilisées sont O(1). La complexité de la fonction creerNoeud est donc O(1).

char* strupr_(char* s) La fonction met en majuscule une chaine de caractères, elle parcours la chaine et si le caractère est minuscule, alors on le passe en majuscule. A la fin, on renvoie la chaine après traitement. Il faut parcourir tous les caractères de la chaine donc la complexité de strupr_ est O(n) avec n le nombre de caractères de la chaine s.

Alexandre EBERHARDT
Lola TRUPIN

void viderBuffer() est une fonction qui parcours le buffer pour le vider, la complexité est donc O(n) avec n le nombre de caractères présents dans le buffer.

void SupprimerTout(T_Arbre abr) Nous avons implémenté cette fonction qui sert à libérer la mémoire de l'arbre à la fin du programme. Il parcourt tout l'arbre binaire, la complexité est donc O(n) avec n le nombre de nœuds.

void libererListeInscriptions(T_Element* liste) Sert à libérer la mémoire allouée à la liste des inscriptions. Il faut parcourir toute la liste pour tout supprimer donc la complexité est O(n) avec n le nombre d'éléments présents dans la liste.

T_Element* supprimerElementListe(T_Element* liste, char* code) : Cette fonction permet de supprimer un élément de la liste. Elle parcourt une liste jusqu'à trouver l'UV et la supprime. Dans le pire des cas, elle doit parcourir la totalité de la liste de m éléments. La complexité est de O(m).

T_Arbre trouverMinimum(T_Arbre abr) : permet de trouver une feuille à partir d'un nœud avec l'ordre alphabétique le plus proche du z. La boucle while parcourant les fils gauches peut être d'une complexité de O(n) dans le pire cas avec n le nombre de nœud.

T_Arbre supprimerNoeud(T_Arbre abr, char *nom, char *prenom) : Elle permet de supprimer un nœud de l'arbre. Cette fonction fait appel à la fonction trouverMinimum d'O(n) ainsi qu'un appel récursif dans certains cas du if d'une complexité de O(h) avec h la hauteur maximal qui peut être n, donc O(n). La complexité est donc de O(n+n)=O(n).

Pour le programme principal :

Dans le programme principal, le switch (les choix) peut tourner pour autant de choix que l'utilisateur le demande. Donc le switch a une complexité **d'O(n)** avec n le nombre de choix.

Choix 1 : une inscription d'un étudiant, avec un parcours de l'arbre (d'une complexité O(n) n étant le nombre d'étudiant de l'arbre, et un parcours de la liste des inscriptions, d'une complexité O(m), m étant le nombre d'uv. Les fonctions strupr_ sont de complexité O(p), p étant la longueur des chaines de caractères, mais ne sont utilisées que 3 fois, donc de l'ordre de O(n). On a donc O(n)+O(n)+O(n) => O(n).

Choix 2: Charger un fichier d'inscriptions. La complexité : la fonction parcourt chaque ligne du fichier, donc une complexité de O(m), m étant le nombre de lignes et insère les étudiants dans l'arbre (la recherche dans l'arbre est de complexité O(n), la complexité cumulée est donc O(m*n) avec m le nombre de lignes dans le fichier et n le nombre d'étudiants dans l'arbre.

Choix 3: Afficher tous les étudiants. La complexité : on fait un parcours de l'arbre, la complexité est donc O(n), n étant le nombre total d'étudiants.

Choix 4: Afficher les inscrits à une UV. Complexité : on doit parcourir l'arbre binaire pour voir tous les étudiants (complexité de O(n), avec n le nombre d'étudiants) puis on doit parcourir la

liste des inscriptions de chaque étudiants (complexité de O(m), m le nombre d'uv). La complexité est donc O(n*m).

Choix 5: Supprimer une inscription. Complexité : La suppression nécessite de trouver l'étudiant et de mettre à jour son inscription. Cela peut impliquer un parcours de l'arbre de complexité O(n) et la mise à jour de la liste des inscriptions (O(m)), avec m le nombre d'UVs. La complexité totale est donc O(m+n)

Choix 6: Quitter. Complexité : la fonction parcourt tout l'arbre et supprime chaque inscription, puis le nœud, il y a donc un parcours de l'arbre de complexité O(n) avec n le nombre d'étudiants puis on doit parcourir la liste des inscriptions de chaque étudiant pour les supprimer (complexité de O(m), m le nombre d'uv. La complexité est donc O(n*m).

Nos changements suite aux retours en TP :

Nous avons modifié la fonction libererListeInscriptions qui ne fonctionnait pas correctement. Ainsi la libération de la mémoire se fait correctement et le programme se termine en retournant 0.

De plus, nous avons modifié la fonction supprimerNoeud en rajoutant l'itération :

abr->listeInscriptions = temp->listeInscriptions;

Ainsi, les nœuds sont bien tous supprimés notamment ceux avec deux fils. Lors de la présentation du test de la fonction face au prof, nous avions une erreur d'affichage de la liste du fils droit ayant remplacé le nœud. Elle était notée vide. A présent, la liste d'inscription s'affiche correctement.

2/ Un exposé succinct de la complexité de chacune des fonctions implémentées :

T_Element *ajouterInscription(T_Element *liste, char* code) : Cette fonction fait appel à la fonction rechercherInscription d'une complexité de O(n). Les autres itérations sont de l'ordre de O(1). Dans la complexité d'ajouterInscription est de O(n) avec n le nombre d'UV de la liste.

T_Arbre inscrire(T_Arbre abr, char *nomx, char *prenomx, char *codex) : Cette fonction fait appel aux fonctions rechercherNoeud (d'une complexité de O(n)), ajouterInscription (O(m)) ou créerNoeud (O(1)). Dans le pire des cas, la fonction passe par le while et doit parcourir dans la boucle toute la hauteur de l'arbre qui peut être « n », le nombre de nœud de l'arbre : O(n). De plus, elle doit parcourir le nombre m d'inscription de la liste de l'étudiant. La complexité d'inscrire est donc de O(n+m).

T_Arbre chargerFichier(T_Arbre abr, char *filename) : cette fonction possède une boucle while qui parcourt la totalité du fichier d'une complexité de O(l) avec l le nombre de lignes du fichier. De plus, à chaque boucle il y a un nom && prenom && code_uv à inscrire grâce à la fonction du même nom et de complexité de O(n+m). LA complexité de chargerFichier est donc de O(l*(n+m)).

Alexandre EBERHARDT
Lola TRUPIN

void afficherInscriptions(T_Arbre abr) : cette fonction est récursive. On parcourt la totalité des nœuds de l'arbre et des listes d'UV : la complexité est de O(n*m) avec n le nombre de nœuds et m le nombre d'UV maximum des listes.

void afficherInscriptionsUV(T_Arbre abr, char *code) : cette fonction est récursive. On parcourt la totalité des nœuds de l'arbre et des listes d'UV : la complexité est de O(n*m) avec n le nombre de nœuds et m le nombre d'UV maximum des listes pour vérifier que tous les étudiants ne sont pas inscrits ou le sont à l'UV.

T_Arbre supprimerInscription(T_Arbre abr, char *nom, char *prenom, char *code) : cette fonction fait appel à rechercherNoeud (O(n)) et supprimerElementListe(O(m)). Donc on a une complexité de O(n+m) avec n le nombre de nœuds de l'arbre et m le nombre d'éléments de la liste.

Conclusion

Le programme est à présent opérationnel face aux demandes du sujet : il permet de saisir des inscriptions pour des étudiants, de les afficher, de charger un fichier, d'afficher les inscrits à une UV et de supprimer les inscriptions et la mémoire. De plus, nous pourrions améliorer grandement la mémoire utilisée et le nombre de fonction. Nous pourrions optimiser notre code pour le rendre plus succinct et moins couteux en mémoire.