A23

Rapport TP3 : Listes chaînées\*

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 :

1. **libererListeInscriptions** - Libère la mémoire allouée pour une liste d'inscriptions.

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

**Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquementT\_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).

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**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).

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**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 O(n).

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**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(log2(n)). C’est par ce qu’a chaque fois, on a au plus le choix entre le fils gauche et le fils droit.

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**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 necessite pas de parcourir la liste car elle commence vide. La complexité de creerInscription est donc O(1). Toutes les autres fonctions sont O(1). La complexité de la fonction creerNoeud est donc O(1).

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

**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)

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement

**void viderBuffer()** est une fonction qui parcours le buffer pour le vider, la complexité est donc O(n).

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**void SupprimerTout(T\_Arbre abr)** Nous avons implémenté cette fonction qui sert à libérer la mémoire de l’arbre a la fin du programme. Il parcours tout l’arbre binaire, la complexité est donc O(n).

Une image contenant texte, capture d’écran, Police, nombre

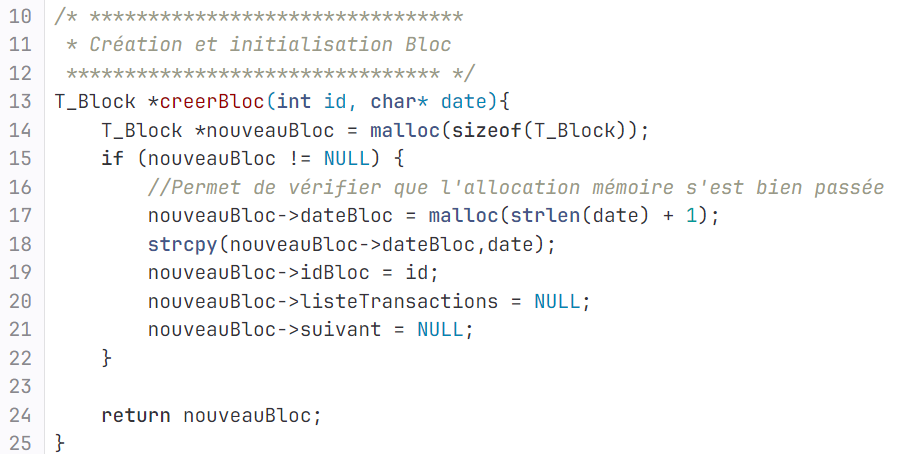
Description générée automatiquement

**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)

Nos changements suite aux retours en TP :

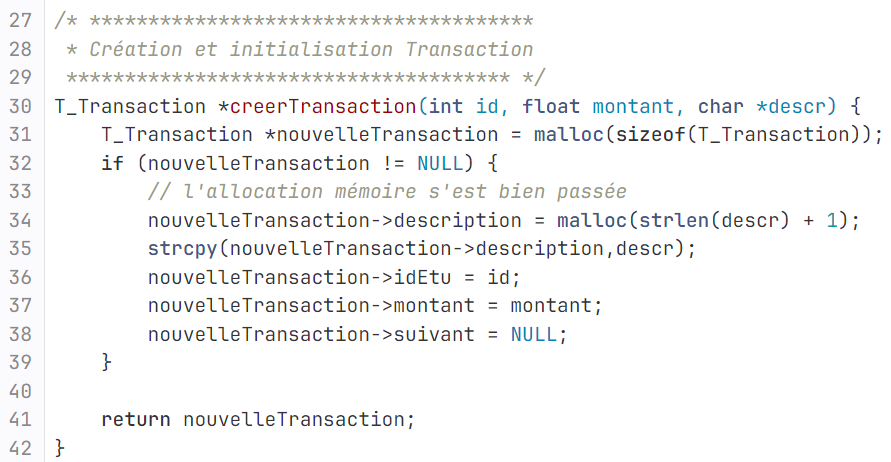
Nous avons modifié la fonction libererListeInscriptions qui ne fonctionnait pas correctement. Ainsi la libération de la mémoire se fait correctement et le programme se termine en retournant 0.

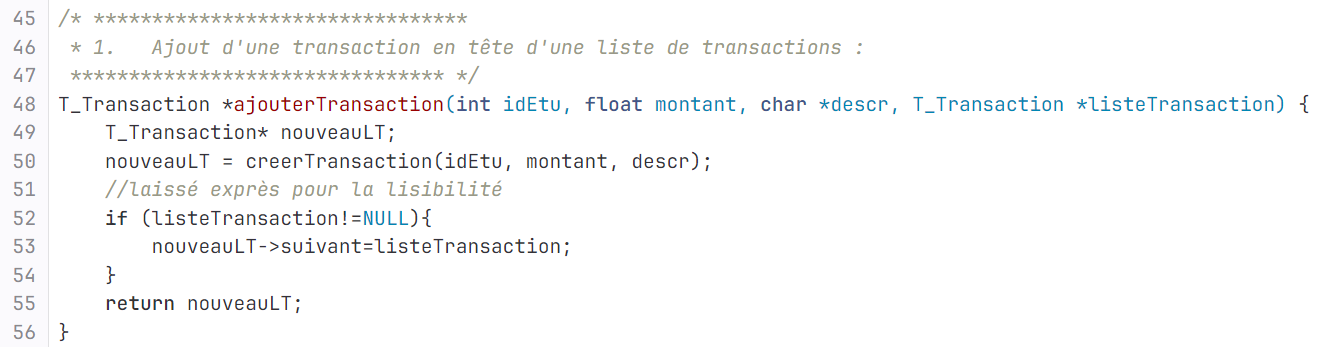
## 2/ Un exposé succinct de la complexité de chacune des fonctions implémentées :

Dans la fonction **creerBloc**:

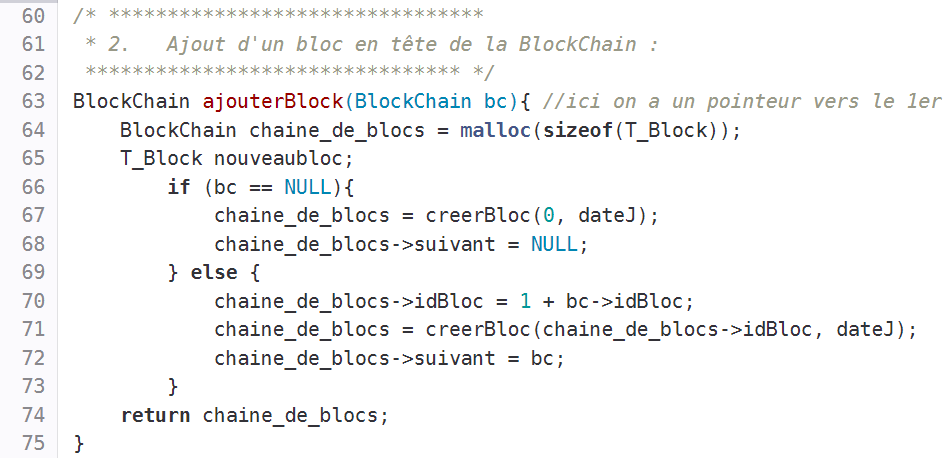
* la ligne 14 à une complexité O(1).
* les lignes 17 à 21 ont aussi une complexité d’O(1).
* la ligne 15 est aussi d’une complexité d’O(1).
* il en va de même pour la ligne 24.

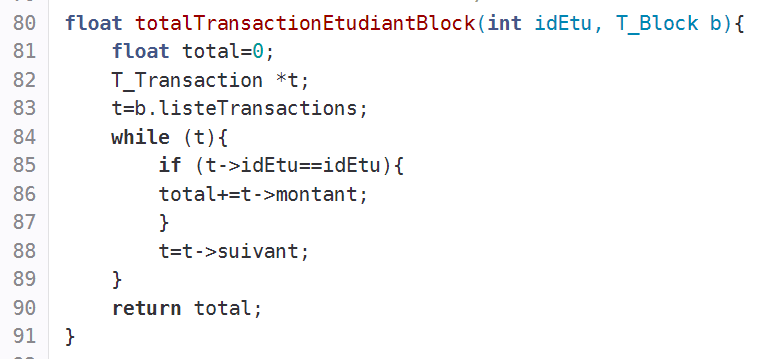
En conclusion, creerBloc est d’une complexité **d’O(1).** Il n’y a que de « simple » itération et pas de boucle ici.

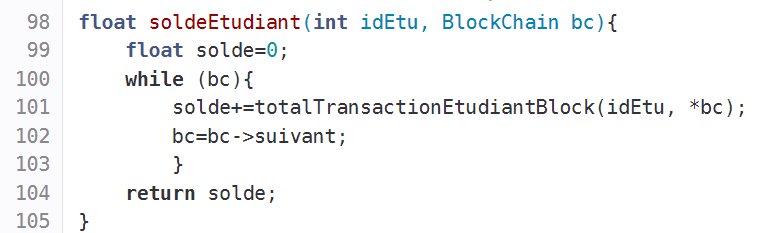
Dans la fonction **creerTransaction**, la première itération ligne 31 est d’une complexité d’O(1). Il en va de même pour les autres lignes de code. En effet, il n’y a pas de boucle ni de récursivité. La complexité de la fonction creerTransaction est de **O(1).**

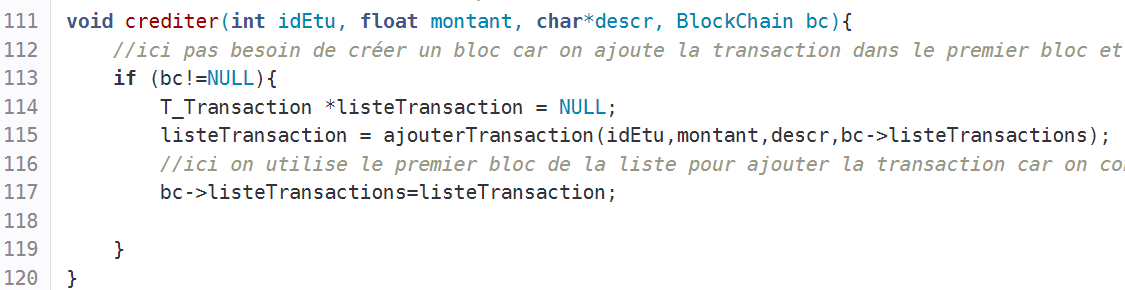


Dans la fonction **ajouterTransaction**, les lignes 51 à 57 ont une complexité d’O(1). En effet, il n’y a pas de boucle ni de récursivité. Et l’appel à la fonction creerTransaction est de O(1) car cette fonction est de O(1). Donc ajouterTransaction a une complexité **d’O(1).**

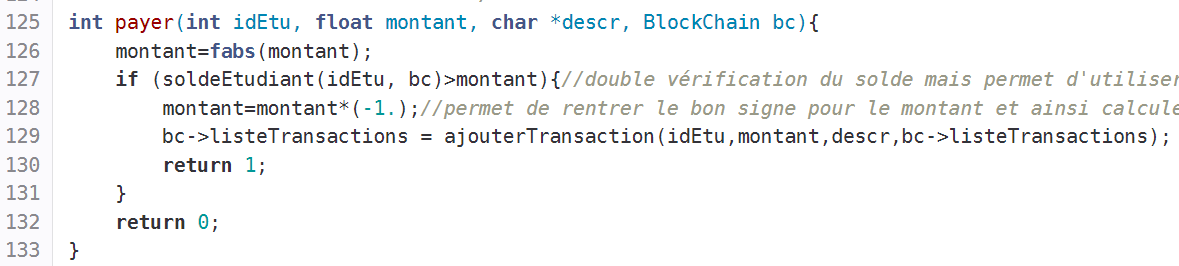
Dans la fonction **ajouterBlock**, il y a des if et des else et il est fait appel à la fonction creerBloc d’une complexité d’O(1). La fonction ajouterBlock est donc d’une complexité **d’O(1).**

Dans la fonction **totalTransactionEtudiantBlock**, il y a un parcours de la liste de transactions d’un block (de la journée en cours) à l’aide de la fonction while. Ainsi, si la taille de la liste est N, alors la complexité de la fonction totalTransactionEtudiantBlock est de **O(n).**

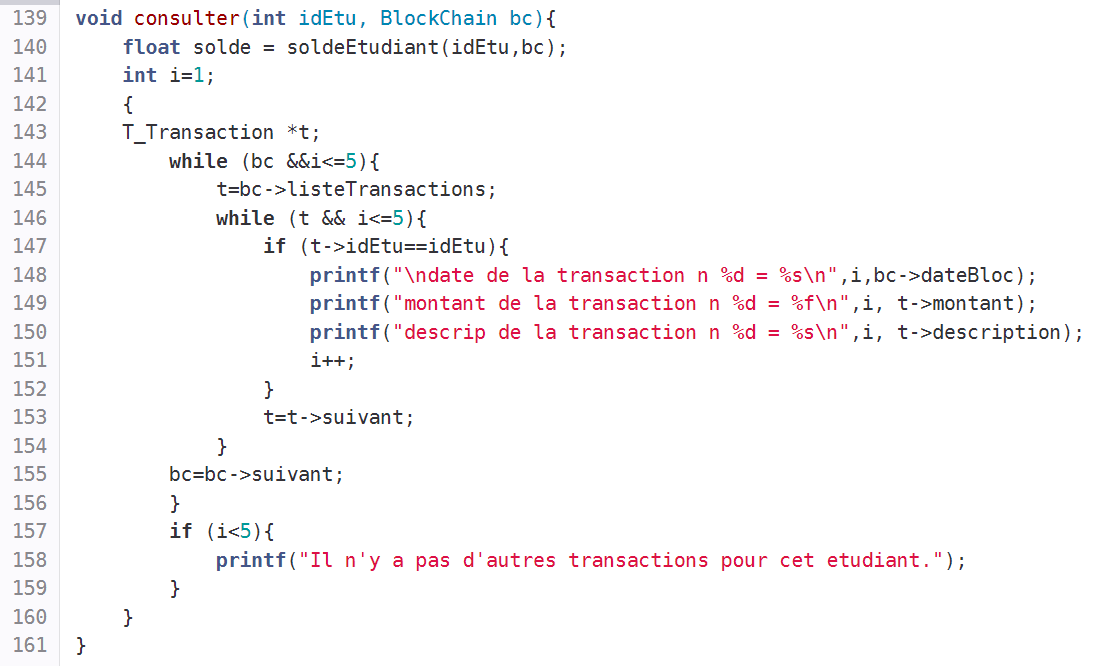
Dans la fonction **soldeEtudiant**, grâce au while, on parcourt la liste des blocs d’une taille M. De plus, à chaque bloc, on fait appel à la fonction totalTransactionEtudiantBlock d’une complexité d’O(n). Donc la complexité de soldeEtudiant est de **O(m\*n)** avec m le nombre de blocs et n le nombre de transactions maximal d’un bloc.

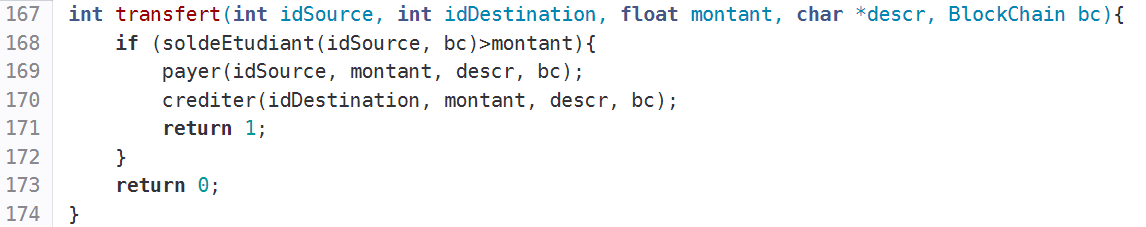


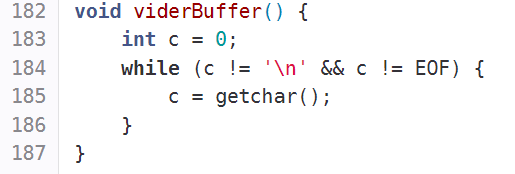
La fonction **crediter** possède un appel à la fonction ajouterTransaction d’une complexité d’O(1). La fonction crediter est **d’O(1)** car toutes ses itérations sont en O(1).



Dans la fonction **payer**, on fait appel à la fonction soldeEtudiant et à la fonction ajouterTransaction respectivement d’O(m\*n) et d’O(1). La fonction payer est donc d’une complexité **d’O(m\*n)** avec m le nombre de blocs et n le nombre max de transaction d’un bloc.

Dans la fonction **consulter**, on recherche 5 transactions maximums. Dans le pire des cas, on parcourt tous les blocs et toutes les transactions de chaque blocs pour en trouver 5 transactions ou moins. Donc la complexité est de **O(m\*n)** avec m le nombre de bloc et n le nombre de transactions maximal d’un bloc. C’est grâce aux boucles while que l’on parcourt les listes de blocs et de transactions.

Dans la fonction **transfert**, on fait appel aux fonctions payer (O(m\*n)) et crediter(O(1)). Donc la fonction transfert est d’une complexité **d’O(m\*n)** avec m le nombre de blocs et n le nombre de transaction max dans un bloc.

La fonction **viderBuffer** est d’une complexité **O(n)** avec n le nombre de caractères. En effet, il est fait appel à une boucle while pour parcourir une chaine de caractères stockée.

## 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.

Lorsque l’on rentre dans le détail des choix on a :

* Choix 1 : un parcours de la chaine de bloc donc une complexité **d’O(m)** avec m le nombre de blocs.
* Choix 2 : un parcours de la chaine de bloc et des transactions du bloc donc une complexité **d’O(m+n)** avec m le nombre de bloc et n le nombre de transaction.
* Choix 3 : un parcours des transactions d’un bloc, d’une complexité **d’O(n)** avec n transactions.
* Choix 4 : un appel au soldeEtudiant (O(m\*n)) et à consulter(m\*n) donc la complexité ici est de **O(m\*n)** avec m le nombre de blocs et n le nombre de transaction d’un bloc.
* Choix 5 : plusieurs appels à viderBuffer (O(c) avec c nombre de caractères), un appel à crediter (O(m\*n)). La complexité la plus importante(forte) étant O(m\*n) en comparaison de O(c), la complexité de la fonction est donc **O(m\*n)** avec m le nombre de blocs et n le nombre de transaction d‘un bloc.
* Choix 6 : il y a des appels à :
  + AjouterBlock (O(1))
  + soldeEtudiant (O(m\*n))
  + viderBuffer(O(c))
  + payer(O(m\*n))  
    Donc la complexité ici est de **O(m\*n)** car c’est le pire cas : avec m nombre de bloc et n nombre de transaction d’un bloc.
* Choix 7 : il y a des appels aux fonctions : viderBuffer, soldeEtudiant, transfert. Donc le pire cas étant pour la dernière transaction du dernier bloc (pour trouver le solde), on a une complexité de **O(m\*n)** avec m nombre de bloc et n nombre de transaction d’un bloc.
* Choix 8 : Ici, il va falloir ajouter un bloc pour chaque jour ajouter. Or, on peut changer la date comme on le souhaite. Il peut donc y avoir un nombre m de nouveaux blocs créés. La complexité est donc de **O(m)** avec m le nombre de blocs à ajouter, c’est-à-dire le nombre de jours entre la date précédente et la nouvelle date.
* Choix 9 : il y a un appel à la fonction libererEspace. La complexité est de **O(m\*n)** avec m le nombre de blocs et n le nombre de transaction d’un bloc.

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.

Introduction

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. Structure

Les structures de données utilisées dans ce TP sont les suivantes :

* Structure T\_Element : Représente une inscription individuelle à une UV. Elle est composée d'un code d'UV et d'un pointeur vers l'inscription suivante, formant ainsi une liste chaînée.
* Structure T\_Noeud : Représente un étudiant dans l'arbre binaire. Elle contient le nom et le prénom de l'étudiant, un pointeur vers la liste chaînée de ses inscriptions (T\_Element), et des pointeurs vers les fils gauche et droit, représentant d'autres étudiants dans l'arbre.
* Structure T\_Arbre : Représente l'arbre binaire dans son ensemble. Il s'agit d'un pointeur vers un T\_Noeud qui agit comme la racine de l'arbre.

2. Fonctions implémentées et complexité

* creerInscription : Crée une nouvelle inscription. Complexité en O(1).
* rechercherInscription : Recherche une inscription spécifique dans la liste. Complexité en O(n), où n est le nombre d'inscriptions.
* ajouterInscription : Ajoute une inscription à la liste d'un étudiant. Complexité en O(n).
* rechercherNoeud : Recherche un étudiant dans l'arbre binaire. Complexité en O(h), où h est la hauteur de l'arbre.
* creerNoeud : Crée un nouveau nœud pour un étudiant. Complexité en O(1).
* inscrire : Inscrit un étudiant à une UV. Complexité en O(h).
* afficherInscriptions : Affiche toutes les inscriptions d'un étudiant. Complexité en O(n), où n est le nombre total d'inscriptions.
* Autres fonctions : Incluent des fonctions auxiliaires pour la gestion des chaînes de caractères et le nettoyage du buffer d'entrée.

3. Conclusion

Grâce à ce TP, nous avons pu expérimenter l’implémentation de listes chaînées et d’arbres binaires de recherche en C, ainsi que leur gestion. Bien que le code soit fonctionnel, il existe des opportunités d'optimisation et d'amélioration, notamment en matière de gestion de la mémoire et de performance.