UNIVERSITÉ DE TECHNOLOGIE DE COMPIÈGNE

Devoir 2

Pascal Quach, Victor Mignot

UV: *Algorithmique et structures de données (NF16)*
Date de rendu: *25 novembre 2019*
Semestre: *A19*

Modifications post-soutenance

Lors de la soutenance, un problème de duplication de bloc apparaissait après un changement de date. L'affichage des transactions était alors également touché, et il ne fonctionnait pas. Celui-ci a été corrigé, le programme fonctionne maintenant correctement.

Fonctions et structures utiles

```c
typedef struct Date Date;

struct Date {
    char Jour[3];
    char Mois[3];
    char Annee[5];
};

void afficher_infos_choix_menu(int choix_menu, Date
   CHOIX_DATE, const char* CHOIX_MENU_TEXTE[]);
 // Afficher les informations du sous-menu + date courante
void afficher_infos_transaction(int id_etu, float montant,
   char* descr);
 // Affiche les infos d'une transaction
void free_BlockChain(BlockChain bc);
 // Libere l'espace memoire alloue
void changement_date(Date * DateDuJour);
 // Change la date
int validation_entree_int(int cond, int inf, int sup);
 // Choix utilisateur conditionne
int nombreDeBlock(BlockChain bc);
 // Renvoie le nombre de bloc dans la blockchain
int bc_init(BlockChain bc);
 // Renvoie 1 si bc n'est pas initialise, sinon 0.
float validation_entree_float(float cond, float inf, float
   sup);
 // Choix utilisateur conditionne
T_Block * RechercherBlock(BlockChain bc, int id_block);
 // Retourne le bloc correspond a l'id donne
```

Listing 1 – Liste des structures et fonctions implémentées

1. Une structure `Date` a été rajoutée pour gérer la date. La structure telle qu'elle est définie permet de passer facilement d'une date à une autre en demandant à l'utilisateur.
2. La fonction `afficher_infos_choix_menu` est utilisée dans les sous-menus pour afficher le menu actuel et la date courante. Faire de cet affichage une fonction permet d'éviter du code redondant.
3. La fonction `afficher_infos_transaction` permet d'afficher uniformément les informations d'une transaction. Faire de cet affichage une fonction permet d'éviter du code redondant.
4. La fonction `free_BlockChain` libère l'espace mémoire alloué à la BlockChain. On essaye d'éviter de surcharger le `main()`.
5. La fonction `init_date` initialise la date en début de programme. On évite de surcharger le `main()` en appelant seulement la fonction.
6. La fonction `changement_date` demande à l'utilisateur une nouvelle date à laquelle passer. On évite de surcharger le `main()`

7. La fonction `validation_entree_int` et La fonction `_entree_float` vérifient l'entrée de l'utilisateur selon les conditions passées en paramètre, et redemandent l'entrée jusqu'à ce qu'elle soit valide. Comme il y a beaucoup d'entrées utilisateurs, on centralise par une fonction ce code redondant.
8. La fonction `nombreDeBlock` compte le nombre de blocs de la BlockChain. Cette fonction permet d'initialiser l'id de chaque bloc, et de dire à l'utilisateur combien de blocs il y a.
9. La fonction `bc_init` vérifie si la BlockChain a été initialisé. On centralise le code pour éviter la redondance.
10. La fonction `RechercherBlock` trouve le bloc correspond à l'id passé en paramètre. On l'utilise un peu partout, cela permet de centraliser le code et d'éviter les redondances.

Fonctions principales

Modifications apportées.

Listing 2 – Fonction principale modifiée

Le prototype a été modifiée pour prendre en compte la gestion de la date. En ajoutant le bloc, on ajoute également la date correspondante. Cela permet de centraliser le code et d'éviter de surcharger le `main()`.

Complexité.

Listing 3 – Ajout d'une transaction en tête d'une liste de transactions

1. La complexité est constante, on ne réalise que des opérations d'affectation.

$$O(1)$$

Listing 4 – Ajout d'un bloc en tête de la BlockChain

2. La complexité est constante, on ne réalise que des opérations d'affectation.

$$O(1)$$

Listing 5 – Calcul de la somme des EATCoin crédités et dépensés par un étudiant sur une journée

3. Si on note n_t, le nombre de transactions, alors la complexité est en $O(n_t)$. En effet, on parcourt toute la liste des transactions du bloc courant pour calculer la dépense d'un étudiant en particulier.

$$O(n_t)$$

Listing 6 – Calcul du solde total d'un étudiant

4. On note n_b, le nombre de blocs de la BlockChain ; $n_{t,i}$, $i = 0, 1, ..., n_b$, les nombres respectifs de transactions pour un bloc d'id i. On parcourt toutes les listes des transactions de tous les blocs existants. La complexité évolue donc proportionnellement à la somme des $n_{t,i}$, soit au max des $n_{t,i}$.

$$O(n_{t,i}$$

Listing 7 – Rechargement du compte d'un étudiant

5. `crediter` dépend de `ajouterTransaction`. La complexité est constante, car il n'y a que des opérations de comparaison et d'affectation.

$$O(1)$$

Listing 8 – Paiement d'un repas

6. `payer` dépend de `ajouterTransaction`. La complexité est constante, car il n'y a que des opérations de comparaison et d'affectation.

$$O(1)$$

Listing 9 – Historique d'un étudiant

7. On s'arrête au bout de 5 transactions ou moins, la complexité est donc constante.

$$O(1)$$

Listing 10 – Transfert de EATCoins entre deux étudiants

8. `transfert` dépend de `ajouterTransaction`. La complexité est constante, car il n'y a que des opérations de comparaison et d'affectation.