NF16-Rapport de TP3 : Magasin
Mohamed TAHIRI et Soulaymane KEBLI

Sommaire :

---

1. Objectif du TP :

L'objectif de ce TP était de construire une interface de gestion d'un magasin, qui possède des rayons, et chaque rayon possède une liste de produits. Chaque produit possède un prix et une quantité.

---

2. Explications des fonctions/structures utilisées :

T_Rayon* chercher_rayon_par_nom(T_Magasin *magasin, char *nom) :

Permet de renvoyer un pointeur sur un rayon depuis notre magasin juste avec son nom. Cela est possible car les noms des rayons sont uniques d'après les questions précédentes.

int nombre_produits(T_Rayon *rayon) :

Prend un pointeur sur un rayon en paramètre et renvoie le nombre de produits de ce rayon sous la forme d'un entier.

int ajouter_produit_fin(T_Rayon *rayon, T_Produit *produit) :

Permet d'ajouter un produit à la fin d'un rayon, elle est différente de ajouterProduit car elle compare pas le prix du produit avec les valeurs déjà présentes dans le tableau car on sait déjà que le produit ira à la fin, on gagne donc du temps de calcul, retourne 1 si l'insertion s'est bien déroulée.

```
struct Produit_Recherche {
    T_Produit *produit;
    char* nom_rayon;
    struct Produit_Recherche *suivant;
```

} ;

Permet de répondre à la question 8 de l'énoncé. Il s'agit juste d'une liste chaînée de produits, tout comme T_Produit, cependant cette structure stocke chaque produit ET le nom de rayon dans lequel il se trouve. Cela nous permet donc l'affichage final de la fonction rechercheProduit.

T_Produit_Recherche* creer_produit_recherche(T_Produit *produit, char* nom_rayon) :

Permet de créer un produit recherché, fonction similaire à creerProduit, elle prend juste le nom du rayon dans lequel le produit se trouve en plus car c'est l'unique différence entre Produit et Produit_Recherche.

int ajouter_produit_recherche(T_Produit_Recherche *produit_recherche, T_Produit *produit, char* nom_rayon) :

Fonction similaire à ajouterProduit, elle trie aussi les produits recherchés par prix pour ensuite les stocker dans la liste chainées des produits recherchés.

---

3. Complexité des fonctions :

- **T_Produit *creerProduit(char *designation, float prix, int quantite) :**

On effectue dans cette fonction des opérations élémentaires à temps constant. Cependant la complexité de la fonction dépend aussi de l'utilisation de la fonction strlen et strcpy, qui ont chacune une complexité en O(n) avec n la longueur de la chaîne à copiée.
On a donc une complexité totale en O(2n) avec n = longueur de désignation.

==> Cela reste de l'ordre de O(n)

- **T_Rayon *creerRayon(char *nom) :**

Même raisonnement que pour la fonction précédente. La complexité de la fonction dépend des complexités des fonctions strlen et strcpy qui ont chacune une complexité en 0(n) avec n la longueur du nom de rayon.
On a donc une complexité totale en O(2n) n = longueur de nom.

==> Cela reste de l'ordre de O(n)

- **T_Magasin *creerMagasin(char *nom) :**

Même chose que pour le rayon.
Complexité totale en O(n) avec n = longueur de nom.

==> Cela reste de l'ordre de O(n)

- **int ajouterRayon(T_Magasin *magasin, char *nomRayon) :**

Cette fonction débute par un appel de la fonction creerRayon qui à une complexité en O(z) avec z la longueur du nom du rayon.
Elle effectue ensuite des opérations élémentaires qui ont une complexité à temps constant : O(1).
Pour la partie dans le else, on a un appel de la fonction strcmp à chaque boucle du while. Cette fonction compare deux chaines de caractères et à une complexité en O(m) avec m la longueur de la plus petite chaînes des 2 chaines comparées, or m est borné par S_MAX = 50.

Donc O(m) = O(50).

n : nombre de rayons dans le magasin.

Dans le pire des cas, on ajoute le rayon à la fin de la liste chainée. Donc on doit parcourir n éléments de la liste chainée et pour chaque élément (rayon) on effectue une comparaison des chaines caractère donc une opération en O(50).
On a donc une complexité pour cette boucle en O(n x 50).
Le reste des opérations effectuées sont élémentaires et ont donc une complexité à temps constant.

La complexité totale est donc en O(n x 50 + z) = O(50n).

==> Cela reste de l'ordre de O(n).

- **int ajouterProduit(T_Rayon *rayon, char *designation, float prix, int quantite)**

Cette fonction débute par un appel de la fonction creerProduit qui à une complexité en O(z) avec z la longueur de la désignation du produit.
On test si le produit existe déjà. Pour cela on parcours toute la liste des produit, on a donc une complexité en O(n) avec n la taille de liste chainée. De plus à chaque passage de la boucle, on fait un appel à la fonction strcmp qui a une complexité en O(m) avec m la longueur de la plus petite chaine. Pour cette boucle on a donc une complexité en O(n x m), or $m \leq S\_MAX = 50$.
On a ensuite une deuxième boucle qui parcours la liste en comparant les prix des produit et le prix du produit que l'on veut insérer. Dans le pire des cas, le prix du produit que l'on veut insérer est le plus grand de la liste et sera donc insérer à la fin. On a alors une complexité en O(n) avec n la taille de la liste.
Le reste des opérations effectuées sont élémentaires et ont donc une complexité à temps constant.
La complexité totale est donc en O(n x m + n) = O(50n) avec n = la taille de la liste des produits

==> Cela reste de l'ordre de O(n)

- **void afficherMagasin(T_Magasin *magasin) :**

La boucle while qui parcours tous les rayons à une complexité en O(n) avec n la taille de la liste des rayons. A chaque passage de la boucle on utilise une boucle for de complexité O(50) qui permet de rendre le tableau d'affichage plus homogène.

On a alors une complexité totale en O(50n).

==> Cela reste de l'ordre de O(n).

- **void afficherRayon(T_Rayon *rayon) :**

Même chose que pour afficher les rayons. Complexité en O(50n) avec n la taille de la liste des produits du rayon.

==> Cela reste de l'ordre de O(n).

- **int supprimerProduit(T_Rayon *rayon, char* designation_produit) :**

Dans le pire des cas le produit à supprimer se trouve au milieu ou à la fin de la liste.
On parcours grâce à la boule while toute la liste chainée, on a donc une complexité en O(n) avec n la longueur de la liste. A chaque appel de la boucle while on fait un appel de la fonction strcmp qui à une une complexité en O(m) avec m la longueur de la plus petite chaine.
La complexité totale dans le pire des cas est donc en O(n x m) avec :

n = longueur de la liste des produits
m = longueur de la désignation du produit, or m est borné par S_MAX = 50.

Donc O(n x m) = O(50n)

==> Cela reste de l'ordre de O(n)

- **int supprimerRayon(T_Magasin *magasin, char *nom_rayon) :**

La fonction débute par un appel de la fonction chercher_rayon_par_nom(magasin, nom_rayon). Cette fonction a une complexité en O(n) avec n la taille de la liste des rayons. La première boucle while a une complexité en O(n-1)=O(n) car elle parcourt toute la liste des rayons jusqu'à trouver le prédécesseur du rayon que l'on veut supprimer.

On libère ensuite l'espace mémoire alloué pour le rayon et ses produits. La complexité de la boucle while est en O(m) avec m la taille de la liste des produits du rayon.
Le reste des opérations sont des opérations élémentaires en temps constant.
La complexité totale est donc en O(n + n + m) = O(2n + m) avec :

n la taille de la liste des rayons,
m la taille de la liste des produits du rayon.

==> Cela reste de l'ordre de O(n)

- **void rechercheProduits(T_Magasin *magasin, float prix_min, float prix_max) :**

On a deux boucles while imbriquées l'une dans l'autre. On parcours la liste des rayons de taille n, et à chaque Rayon on parcours la liste des produits de taille m. A

chaque parcours d'un produit, on vérifie s'il est nul et donc on le crée à l'aide de la structure produit_recherche qui à une complexité constante en 0(1) ou alors dans le pire des cas on doit l'ajouter dans la liste des produits recherchés avec la fonction ajouter_produit_recherche qui a une complexité en O(z) avec z la taille de la liste des produits recherchés. Pour ces deux première boucles on a une complexité totale en O(n x m x z) avec :

n = taille de la liste des rayons
m = taille de la liste des produits
z = taille de la liste des produits recherchés.

La troisième boucle while permet de rendre le tableau homogène à l'affichage. Elle a une complexité en O(S_MAX x n) avec S_MAX = 50. Elle est donc en O(50n).

On a donc une complexité totale pour cette fonction en O(nmz + 50n) = O(nmz).

==> Cela reste de l'ordre de O(n^3)

- **void fusionnerRayons(T_Magasin *magasin)**

La fonction fait appel 3 fois séparément à la fonction chercher_rayon_par_nom qui à une complexité en 0(n), on a donc pour ces 3 appels une complexité en 0(3n).
On va ensuite parcourir premièrement tous les produits des 2 premiers rayon que l'on veut supprimer. Soit n la taille de de la liste des produits produit_9_1 et m la taille de la liste des produits produit_9_2. A chaque passage de la boucle on fait un appel à la fonction ajouter_produit_fin qui ajoute un produit à la fin d'une liste de produit et qui a une complexité en O(z), z étant la taille de la liste des produits produit_9_3 (du rayon fusionné).
On a donc une complexité en O(max{n,m} X z) n = taille liste produit du premier rayon
m = taille liste produit du deuxième rayon
z = taille de la liste des produits du rayon fusionné

On a ensuite deux nouvelles boucles séparées qui vont s'occuper d'ajouter les produits manquants. Pour ces deux boucles on a une complexité en 0(n+m).
Enfin, on fait appel deux fois à la fonction supprimerRayon qui a une complexité en O(2n + 2m + n1 + m1) avec :

n, m la taille de la liste des deux rayons Rayon_9_1 et Rayon_9_2.
n1, m1 la taille de la liste des produits de chaque rayon respectivement.

On a alors une complexité totale en O(max{n,m} x z + 2n + 2m + n1 + m1)

==> Cela reste de l'ordre de O(n^2).

La complexité des fonctions utilitaires utilisées tout au long du programme ont été définis précédemment dans les fonctions où elles ont été appelées.

— —

4. Conclusion :

Ce TP nous a entraîné à savoir gérer les listes chaînées simples et la gestion des chaînes de caractères en langage C. La gestion des chaînes de caractères n'est pas aussi simple qu'on ne le pensait car c'est plus compliqué à gérer que sur d'autres langages de programmation type Python, par exemple avec le \0, la taille maximale, les espaces, le buffer etc.

De plus, la question qui nous a demandé le plus de réflexion est la fonction rechercheProduits, car elle nous a demandé la création d'une nouvelle structure, et nous n'avions pas pensé à cela tout de suite, nous avons donc pris du temps pour trouver la solution.

Cependant, le plus compliqué était l'optimisation des complexités de fonctions car nous devions faire le moins d'opérations possibles pour qu'elles soient le plus efficace possible.