Marianne KASSUBECK
Marion CHAN-RENOUS

RAPPORT NF16 -TP3

Listes chaînées

Introduction

Le but de ce TP était d'écrire un programme permettant la gestion d'un magasin et des rayons qui le constituent. Un magasin comporte plusieurs rayons, qui comportent eux-mêmes plusieurs produits. Les données des magasins, rayons, et produits, et le lien entre ces données ont été modélisées à l'aide de listes simplement chaînées.

Fonctions requises

Fonctions de création et d'initialisation des structures

- `T_Produit *creerProduit(char *marque, float prix, char qualite, int quantite)`
- `T_Rayon *creerRayon(char *nom)`
- `T_Magasin *creerMagasin(char *nom)`

Ces trois fonctions servent à créer et initialiser les différents types de structure. Les instructions dans chacune de ces trois fonctions sont des instructions élémentaires (affectation, allocation mémoire), leur complexité est donc la même dans le meilleur et dans le pire des cas. Ainsi, leur complexité est en $\Omega(1)$ et $O(1)$, donc en $\Theta(1)$.

Fonctions d'ajout

- `int ajouterRayon(T_Magasin *magasin, T_Rayon *rayon)`

Cette fonction permet l'ajout d'un rayon dans un magasin. Afin de respecter l'ajout dans l'ordre alphabétique du rayon, il est nécessaire de parcourir la liste des rayons du magasin. Dans le meilleur des cas, le rayon doit être ajouté en première place (si le magasin est vide ou si le rayon à ajouter est le premier dans l'ordre alphabétique). Il s'agit dans ce cas d'une simple affectation, donc la complexité dans le meilleur des cas est en $\Omega(1)$. Dans le pire des cas, le rayon doit être ajouté à la fin de la liste, donc la liste doit être parcourue en entier. La complexité dans le pire des cas de cette fonction est donc en $O(n)$.

- `int ajouterProduit(T_Rayon *rayon, T_Produit *produit)`

Cette fonction permet l'ajout d'un produit dans un rayon. La même marque de produit ne doit pas être ajoutée deux fois dans un même rayon, et les produits doivent être classés par ordre de prix croissant. Ainsi, dans le meilleur des cas, le rayon est vide et le produit est ajouté en tête de liste des produits du rayon. Il s'agit donc dans ce cas d'une simple affectation, donc la complexité dans le meilleur des cas est en $\Omega(1)$. Dans le pire des cas, il faut tout d'abord parcourir une première fois la liste de produits de rayons pour vérifier si la marque est déjà présente. Puis, dans le cas où la marque n'est pas déjà présente, il faut parcourir une deuxième fois la liste des produits afin de respecter l'ordre croissant de prix et insérer le produit au bon endroit dans la liste. Dans le pire des cas, le produit est inséré en fin de liste, la liste est donc parcourue en entière. La complexité dans le pire des cas de cette fonction est donc en $O(n)$.

Fonctions d'affichage

- `void afficherMagasin(T_Magasin *magasin)`
- `void afficherRayon(T_Rayon *rayon)`

Ces fonctions servent à afficher les rayons du magasin ou les produits d'un rayon du magasin. Dans les deux cas, la liste des rayons ou la liste des produits du rayon doit être parcourue entièrement. Ainsi, la complexité de ces deux fonctions est en O(n).

Fonctions de suppression

- `int supprimerProduit(T_Rayon *rayon, char* marque_produit)`
- `int supprimerRayon(T_Magasin *magasin, char *nom_rayon)`

Ces fonctions servent à supprimer un produit d'un rayon ou un rayon d'un magasin. Pour ces deux fonctions, dans le meilleur des cas (rayon/produit en tête de liste), il n'y aura que des instructions élémentaires, donc la complexité dans le meilleur des cas est en Ω(1). Pour ces deux fonctions, dans le pire des cas (rayon/produit en fin de liste ou non présent dans la liste), les listes doivent être parcourues entièrement, la complexité dans le pire des cas de ces deux fonctions est donc en O(n).

Fonction de recherche

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

Cette fonction permet de rechercher dans les différents rayons du magasins les produits dont le prix est compris dans une certaine fourchette de prix. Ainsi, ici il faut prendre en compte le nombre n de produits dans le magasin. Ainsi, pour chaque rayon, on parcourt tous les produits de celui-ci puis nous passons au suivant. Si le prix du produit est dans la fourchette, nous l'ajoutons à notre structure de données « T_Recherche » via la fonction ajouterProduitRecher (détaillée dans la suite de ce rapport). La complexité de cette fonction est O(l) avec l le nombre de produits dans la structure T_Recherche. Ainsi, dans le pire des cas, tous les produits du magasin sont dans compris dans la fourchette de prix, tous les produits doivent donc être ajoutés dans la structure T_Recherche (l=n). Ainsi, la complexité de cette fonction est $O(n^2)$.

Structures et fonctions supplémentaires

- `T_Rayon *rechercherRayon(char *nom, T_Magasin *magasin)`

Nous avons choisi de définir une fonction rechercherRayon. Celle-ci, permet de récupérer un pointeur sur un rayon (T_Rayon*) à partir du nom de celui-ci. Cette fonction permet ensuite de faire les traitements demandés sur le rayon. Si aucun rayon dont le nom est celui passé en paramètre n'existe dans le magasin, alors cette fonction renverra un pointeur nul. Sinon, la liste des rayons est parcourue jusqu'à atteindre le bon rayon. Ainsi, dans le meilleur des cas le rayon recherché est le premier de la liste, donc la complexité dans le meilleur des cas de cette fonction est en Ω(1). Dans le pire des cas (rayon non présent ou le rayon est le dernier de la liste), la liste des rayons est parcourue entièrement. Ainsi, la complexité dans le pire des cas est en O(n).

- `void supprimerProduitsRayon(T_Rayon* ray)`

Cette fonction de service est utile dans le cas de la suppression d'un rayon. Celle-ci, à partir du pointeur sur le rayon à supprimer, s'occupe de désallouer la mémoire allouée pour chacun des produits de ce rayon en question. Dans le meilleur des cas, la liste des produits est vide, la complexité dans le meilleur des cas est donc en Ω(1). Si la liste n'est pas vide, la liste des produits du rayon est ainsi parcourue entièrement, la complexité de cette fonction est donc en O(n).

- ```
  typedef struct Recherche{
        T_Produit* prod;
        char* nom_rayon;
        struct Recherche *suivant;
  }T_Recherche;
  ```

Afin de traiter la recherche des produits dans une fourchette de prix, nous avons choisi de définir une nouvelle structure de données. Celle-ci, possède un pointeur sur le produit, un champ « nom » permettant de stocker le nom du rayon dans lequel est le produit, et un pointeur sur la structure Recherche suivante. Il s'agit donc d'une liste simplement chaînée. Cette structure est « temporaire » pour chaque recherche. En effet, au début d'une recherche, un pointeur sur cette structure est alloué dynamiquement. Les pointeurs sur les produits sont eux aussi alloué dynamiquement. A la fin de la recherche, après l'affichage du contenu de la structure, la mémoire utilisée pour les produits et la structure recherche sera désallouée.

- ```
  T_Recherche*    ajouterProduitRecher(T_Recherche    *recherche,
  T_Produit *prod, char* nom_rayon);
  ```

Nous avons défini cette fonction afin d'ajouter les produits compris dans la fourchette de prix dans notre structure T_Recherche. Ainsi, cette fonction rempli les différents champs de la structure avec les données passées en argument (pointeur sur le produit concerné, et le nom du rayon de celui-ci). Cette fonction permet de classer les produits en ordre de prix croissant, sur le même modèle que celle permettant l'ajout d'un produit dans un rayon. Dans le meilleur des cas, le produit est à insérer en tête de liste, la complexité dans le meilleur des cas est donc en $\Omega(1)$. Dans le pire des cas, la liste doit être parcourue entièrement, la complexité de cette fonction est donc en $O(n)$.

Corrections suite à la démonstration

Plusieurs corrections ont été faites suite à la démonstration de notre programme pendant la séance de TP.

Ajout de la désallocation de la mémoire de la marque

En effet, nous nous sommes rendues compte que nous même si nous avions pensé par nous-mêmes à désallouer la mémoire utilisée pour les produits lors de la suppression d'un rayon, en revanche nous avions oublié de désallouer avant la mémoire utilisée pour le pointeur alloué dynamiquement sur une chaine de caractère permettant de stocker la marque du produit.

Ajouter la correction de recherche

Lors de la démonstration, notre fonction de recherche présentait un petit bug. En effet, l'ajout de deux produits ayant le même prix dans notre structure T_Recherche faisait planter le programme. En effet, nous avions simplement oublié de considérer ce cas. Le changement d'un « > » en un « >= » nous a permis de corriger ce problème.

Affichage des tableaux

Nous nous sommes également rendues compte pendant la démonstration que l'affichage des différents tableaux n'était pas idéale, et l'avons donc légèrement modifiée à l'aide du formatage possible du flux de sortie de la fonction printf.

Conclusion

Ce TP nous a permis de manipuler des listes simplement chaînées. Ainsi nous avons pu nous rendre compte de l'utilité de ce modèle à travers la programmation de différentes fonctions. La difficulté dans ce TP a été pour nous de choisir la structure de données la plus adéquate afin de traiter la recherche des produits. Nous avons longuement discuté et cherché afin de trouver la structure de données idéale pour traiter cette recherche. Nous n'avons à ce jour pas trouvé de solution plus « simple » que celle que nous avons trouvé, qui est de créer une liste simplement chainée pour stocker ces résultats de recherche.