Alexis DESEURE
Matéo MARCILLAUD

Rapport TP3 - Listes chaînées

I. Choix d'implémentations

En plus de la fonction viderBuffer() permettant de vider le tampon d'entrée et des fonctions du sujet, nous avons choisi d'implémenter d'autres fonctions et structures décritent ci-dessous :

T_Rayon *obtenirRayon(T_Magasin *magasin, char *nom)

Cette fonction a été créée afin de rendre le code plus lisible et d'éviter la redondance. Elle permet, à partir d'une chaîne de caractères contenant le nom d'un rayon existant de renvoyer le pointeur vers le rayon associé. Si la chaîne de caractère ne correspond pas à un rayon existant, la fonction renvoie un pointeur pointant sur NULL.
La fonction est utilisée à plusieurs reprises dans le main.c afin d'obtenir un rayon associé au nom entré par l'utilisateur pris en paramètre par les fonctions : ajouterProduit, afficherRayon et supprimerProduit. Elle est aussi utilisée dans fusionnerRayons.

char* formatageNom(char* chaine, int n)

Nous avons choisi d'implémenter cette fonction avec un objectif purement graphique. Elle permet, à partir d'une chaîne de caractères et d'un entier n, de renvoyer une nouvelle chaîne contenant n caractères précisément. Si la chaîne initiale en a un nombre inférieur à n, alors la nouvelle chaîne contiendra des espaces à la fin pour atteindre une longueur n. Sinon, les caractères au-delà ne seront pas retenus et des points de suspension remplaceront les trois derniers caractères. Cette fonction est utilisée pour l'affichage de tableaux.

char* formatageChiffre(int nombre, int n)

Cette fonction utilise formatageNom et a le même objectif avec cette fois comme entrer un entier qui est converti en chaîne de caractères.

char* formatageFloat(float nombre, int n)

Cette fonction utilise formatageNom et a le même objectif avec cette fois comme entrer un flottant qui est converti en chaîne de caractères.

struct ProduitTrie

Nous avons décidé d'implémenter cette structure pour la question 8 et sa fonction rechercheProduits. Elle permet de stocker par ordre de prix croissant l'ensemble des produits situés dans l'intervalle sous la forme d'une liste chaînée simple pour ensuite les

afficher dans ce même ordre. Elle regroupe un pointeur vers un produit pour pouvoir y avoir accès sans copier ses informations, un pointeur vers un ProduitTrie suivant pour pouvoir respecter l'organisation en liste chainée et un pointeur rayon pour avoir accès au rayon dans lequel se situe le produit associé et ainsi, pouvoir l'afficher facilement dans le tableau demandé pour la question 8.

T_ProduitTrie *creerProduitTrie(T_Produit *produit, T_Rayon *rayon)

De la même manière que creerMagasin, creerRayon, creerProduit, cette fonction permet d'initialiser une structure ProduitTrie à partir d'un pointeur vers un produit et un autre vers un rayon.

II. Exposé de la complexité des fonctions implémentées

creerProduit, creerRayon, creerMagasin, creerProduitTrie

Ces fonctions d'initialisation pour les différentes structures comportent uniquement des opérations unitaires qui ne dépendent d'aucun paramètre, elles sont donc de complexité constante (O(1)).

viderBuffer

Cette fonction permettant de vider le tampon est de complexité linéaire (O(n)) car le nombre d'opérations dépend seulement du nombre de caractères différents de \n et EOF dans le tampon.

formatageNom

On considère par la suite tous les appels de fonctions issues de bibliothèques standards C comme des opérations élémentaires. Cette fonction est donc de complexité constante O(1).

formatageChiffre et formatageFloat

Ces fonctions sont de complexité linéaire O(n) car le nombre d'opérations varie en fonction du nombre de chiffres du nombre en paramètre. On considérera par la suite que ces fonctions sont de complexité constante dans les autres fonctions car elles ne sont pas nécessaires dans le sujet et ont un but de mise en forme uniquement.

obtenirRayon

Dans le pire des cas, le rayon n'est pas trouvé, la complexité est donc linéaire (O(n)).
On considérera cette fonction comme ayant une complexité constante pour le calcul des fonctions dans lequel elle est appelée.

ajouterRayon

Le nombre d'opérations varie en fonction de la position à laquelle on insère le rayon dans la liste chaînée. Dans le pire des cas, le rayon se place en dernière position, il a donc fallu parcourir toute la liste. On a une complexité linéaire O(n).

ajouterProduit

La complexité de cette fonction est linéaire O(n) car dans le pire des cas on a dû parcourir toute la liste chaînée avant d'ajouter le produit.

afficherMagasin

Cette fonction parcourt tous les rayons et tous les produits qu'il contient (pour calculer le nombre de produits dans chaque rayon. Or le nombre de produits et le nombre de rayons sont indépendants entre eux : on peut donc avoir un différent nombre de produits pour deux rayons différents. Dans tout magasin il y a n rayons et m produits en total.
Donc si on a plus de produits que de rayons (donc des rayons vides), on a une complexité de O(n), car on a 1 itération de la boucle pour chaque rayon du magasin, donc n itérations.
Dans le cas contraire, si on a plus de produits que de rayons, on a une complexité de O(m) car la boucle se répète m fois, une fois pour chaque produit.
Ainsi on a une complexité de O(max(n,m)).

afficherRayon

La fonction parcourt les éléments de la liste des produits du rayon. Son nombre d'opérations dépend du nombre de produits n dans le rayon. La complexité est linéaire (O(n)).

supprimerProduit

Dans le pire des cas, le produit n'a pas été trouvé, le programme a donc parcouru tous les produits du rayon, la complexité est linéaire en O(n) avec n le nombre de produits dans le rayon.

supprimerRayon

Dans le pire des cas, il a fallu parcourir tous les rayons avant de trouver celui à supprimer puis il a fallu parcourir tous les produits du rayon concernés afin de libérer leur espace mémoire. La complexité de la fonction est donc en O(n+m) avec n le nombre de rayons et me le nombre de produit dans le rayon à supprimer.

rechercheProduits

Dans le pire des cas, tous les produits du magasin rentrent dans l'intervalle de prix. Il faut donc parcourir tous les rayons puis pour chacun d'entre eux, tous les produits du rayon. Il faut ensuite ajouter tous ces produits dans une liste triée. Or le nombre de produits et le nombre de rayons sont indépendants entre eux : on peut donc avoir un différent nombre de produits pour deux rayons différents.
Cependant, tous les produits qui sont dans la fourchette de prix doivent être ajoutés dans une liste triée qui sera de longueur m. La boucle qui permet d'ajouter le produit à cette liste

est donc de complexité O(m), et la boucle qui vérifie que les produits puis les ajoute est de complexité O(m^2), vu que ces deux boucles sont imbriquées.
Ainsi, si on a plus de produits que de rayons (des rayons vides) que de nombre de produits au carré, on parcourt quand même ces rayons vides mais sans en tirer des valeurs. On parcourt aussi tous les produits mais il y en a moins que de rayons.
De même, si on a plus de produits m que de rayons n, et que tous les produits entrent dans l'intervalle, on a m itérations de la boucle.
De plus, il y a une dernière boucle à la fin de la fonction qui affiche tous les produits dans la fourchette. Si tous les m produits entrent dans la fourchette de prix, on a m itérations de la boucle.
Ainsi, la complexité est O(max(n,m^2)+m).

fusionnerRayons

La complexité de cette fonction est en O(n*m).
En effet, considérons n le nombre de produits dans le rayon 2 et m le nombre de produits dans le rayon 1, dans le pire des cas, à savoir le cas où les deux rayons sont entièrement différents, pour chaque produit de rayon2 on doit vérifier que les m produits de rayon1 n'y sont pas identiques puis ajouter le produit dans le rayon 1.
On vérifie m produits pour chaque produit n fois, donc on retrouve bien la complexité en O(n*m).