Rapport TP03 NF16

Antoine COLLAS et Emmanuelle LEJEAIL

1. Justification des fonctions outils et des structures supplémentaires

```
typedef struct ABR{
  T_Produit* produit;
  char nom_rayon[50];
  struct ABR *gauche, *droite;
}ABR;
```

Nous avons jugé intéressant la création d'une structure ABR (arbre binaire de recherche) notamment pour la fonction rechercheProduits afin d'avoir une structure dans laquelle on peut facilement afficher les produits des différents rayons dans l'ordre croissant des prix.

En lien avec cette structure, nous avons réalisé plusieurs fonctions qui permettent de la manipuler :

ABR_inserer(ABR* tete, T_Produit* produit, char* rayon) et affichageABR(ABR* x).

ABR_inserer est une fonction qui permet d'insérer une structure de type T_Produit dans notre arbre binaire de recherche. On fait aussi passer le nom du rayon en argument pour permettre la création d'un nouveau nœud portant ce nom (ce qui permettra lors de l'affichage de garder le nom des rayons de provenance des différents produits).

AffichageABR est une fonction qui va réaliser l'affichage infixe des nœuds de l'arbre binaire de recherche que l'on a construit. Elle prend un simple pointeur vers la racine de l'ABR puis par des appels récursifs, affiche d'abord l'élément le plus à gauche (qui a le prix le plus faible), puis on affiche l'élément à sa droite et ainsi de suite jusqu'à avoir afficher tous les noeuds de notre arbre (la dernière feuille est celle dont le prix est le plus élevé).

Ainsi notre fonction rechercheProduit parcourt les produits, de chaque rayon du magasin, dont le prix est inférieur au prix maximum indiqué par l'utilisateur. La fonction utilise ABR_inserer pour ajouter dans notre ABR tous les produits compris dans la fourchette de prix. Puis on utilise affichageABR pour afficher tous les produits trouvés dans l'ordre croissant de prix.

Nous avons également créé d'autres fonctions outils telles que viderRayon, predRayon et predProduit :

PredRayon est une fonction qui nous permet de récupérer un pointeur vers le rayon précédant celui passé en argument, dans le magasin (lui aussi passé en argument). Cette fonction nous est utile notamment dans supprimerRayon car il permet de récupérer directement un pointeur vers le rayon précédant celui que l'on désire supprimer et on peut ainsi mettre a jour son champ suivant en lui donnant la valeur du champ suivant du rayon que l'on désire supprimer.

ViderRayon est une fonction outils qui nous permet, lors de la suppression d'un rayon, de supprimer tous les produits contenus dans ce rayon. Cette fonction désalloue toute la mémoire des produits contenus dans le rayon, afin de ne pas perdre de l'espace mémoire.

PredProduit fonctionne sur le même modèle que predRayon sauf qu'elle permet de retourner un pointeur vers le produits précédant un produit donné (passé en argument) dans un rayon (passé en argument). On se sert de cette fonction dans supprimerProduit afin de récupérer un pointeur vers le produit qui précède celui que l'on supprime afin de modifier directement ses champs.

Nous avons aussi choisi de créer trouverRayon et trouverProduits ainsi que demanderAction et demanderChaine :

trouverRayon permet de récupérer un pointeur vers un rayon quelconque dans un magasin à partir de son nom. Nous avons utilisé cette fonction dans la construction de notre main afin de récupérer des pointeurs qui servent ensuite d'arguments dans certaines de nos fonctions (argument de ajouterProduit, afficherRayon et supprimerProduit).

trouverProduit permet de récupérer un pointeur vers un produit quelconque en connaissant sa marque. Nous l'utilisons dans la fonction ajouterProduit afin de vérifier si il n'existe pas déjà un produit qui porte le même nom de marque dans le rayon passé en argument.

demanderAction est une fonction que nous utilisons dans le main. Son objectif est simple, afficher le menu puis lire le choix entré au clavier par l'utilisateur. Si jamais le choix ne correspond pas à l'un des numéros du menu, elle réitère sa demande à l'utilisateur.

demanderChaine est utilisée en argument beaucoup de nos fonctions (notamment créerMagasin, creerRayon, creerProduit, trouverRayon, supprimerProduit, supprimerRayon). Elle est aussi utilisée au sein même de fusionnerRayon. Elle retourne une chaîne de caractères entrée par l'utilisateur après avoir affiché ce à quoi servira la chaîne de caractères.

2. La complexité des fonctions implémentées

Dans cette partie nous considérons les tailles suivantes du problème :

-**N** : le nombre de rayons.

-**M** : le nombre de produits du rayon comptant le plus de produits.

Dans tout la partie qui suit nous n'avons pas tenu compte de la complexité lors des opérations sur les chaines de caractères comme la copie et la comparaison. Il faudrait en toute rigueur en tenir compte.

CreerProduit, creerRayon, creerMagasin : Pas de boucles, on effectue un nombre constant d'instructions donc la complexité est en O(1).

DemanderAction : l'affichage du menu se fait en une série d'instructions donc O(1) puis on entre dans une boucle TANT QUE mais le nombre d'itérations de cette boucle ne dépend pas de la taille des données, elle se répète tant que le choix entré par l'utilisateur n'est pas valide donc cette boucle est aussi en O(1). Complexité totale en O(1).

TrouverRayon : il s'agit d'une fonction construite de manière récursive qui effectue trois comparaison (instructions IF) et un appel récursif si aucune des conditions n'est satisfaite. On effectue les appels sur le rayon suivant c'est pourquoi notre problème est de taille N et la complexité est en O(N).

AjouterRayon : cette fonction se compose de trois tests IF principaux dans lesquels on effectue un nombre constant d'instructions (donc O(1)) mais dans le dernier cas (cas général), on réalise une boucle TANT QUE dont la condition d'arrêt est d'arrivé à la fin du magasin et d'avoir parcouru tous les rayons (donc O(N)).

A l'intérieur de cette boucle TANT QUE on effectue encore des tests en comparant le nom du rayon à ajouter avec le nom du rayon sur lequel on pointe actuellement. Ces tests effectuent un nombre constant d'opérations (complexité en O(1)). La complexité totale de notre WHILE est de O(N) et il s'agit de la complexité globale de la fonction.

TrouverProduit : on réalise une boucle TANT QUE qui a pour conditions de sortie, soit on a parcouru tout le rayon et on n'a pas trouvé le produit, soit on a bien trouvé le produit que l'on cherchait. Dans le meilleur cas, notre complexité est $\Omega(1)$ mais dans le pire cas (parcours de toute la liste chaînée des produits du rayon) elle est en O(M).

AjouterProduit : la fonction est construite de manière semblable à ajouterRayon. Trois tests IF sont effectués et leur complexité est de O(1). Dans les autres cas, on exécute la boucle TANT QUE qui ne s'arrête qu'en cas de RETURN ou lorsque l'on arrive au dernier produit du rayon.

Dans cette boucle WHILE, on effectue un test IF pour vérifier le prix du produit à ajouter avec celui de chaque produit (jusqu'à ce que le prix soit supérieur). Dans ce cas on effectue des instructions en O(1). Sachant que la complexité de la boucle TANT QUE dépend du nombre M de produits, alors la complexité de la fonction est O(M).

demanderChaine : cette fonction récupère simplement une chaîne de caractère, elle effectue donc un nombre d'instruction constant donc sa complexité est en O(1).

afficherMagasin : le test IF est en O(1) puis il est suivi par une boucle TANT QUE qui parcourt les rayons du magasin jusqu'au dernier. La taille du problème est donc le nombre N de rayons dans le magasin. Donc notre complexité est en O(N) puisqu'on va répéter notre boucle N fois.

AfficherRayon : cette fonction consiste simplement en parcourt de tout le rayon avec un affichage de chaque produit. Pour passer d'un produit à l'autre la procédure est en O(1). L'affichage est aussi en O(1). Comme le rayon comporte au maximum M produits, l'affichage du rayon se fait en O(M).

predProduit : Cette fonction permet de trouver le prédécesseur d'un produit dont l'adresse est fournie en paramètre. La fonction parcourt tout le rayon jusqu'à trouver le prédécesseur du produit. Dans le pire des cas le produit se trouve en fin de chaine. Pour passer d'un produit à l'autre, il y a un nombre constant d'opérations, donc O(1). Donc la complexité est O(M).

supprimerProduit : Pour supprimer un produit nous commençons par appeler trouverProduit pour obtenir l'adresse du produit. Cette fonction est O(M). Puis il faut chercher le prédécesseur de ce produit à l'aide predProduit qui est aussi en O(M). Enfin nous supprimons le produit et libérons l'espace mémoire en O(1). Donc la supression d'un produit se fait en O(M).

predRayon : Cette fonction parcourt les rayons jusqu'à trouver le rayon passé en paramètre. Le pire cas est quand il faut parcourir tous les rayons. Donc l'algorithme est en O(N).

viderRayon : Cette fonction permet de libérer l'espace mémoire alloué à un rayon. Elle va donc parcourir tout le rayon et libérer un à un les espaces mémoire de chaque produit. La libération de mémoire est en O(1). Comme il est nécessaire de parcourir tout le rayon , la complexité est en O(M).

supprimerRayon : La fonction commence par chercher le rayon passé en paramètre. Pour cela elle utilise trouverRayon qui est en O(N). Puis il faut chercher le prédécesseur de ce rayon à l'aide de predRayon : O(N). La suppression du rayon est O(1). Enfin il faut libérer l'espace mémoire alloué aux produits du rayon à l'aide de viderRayon : O(M). Donc la complexité globale est en O(N+N+1+M)=O(N+M)=O(max(N,M))=O(M) car dans la plupart des magasins il y a plus de produits que de rayons.

ABR_inserer : Supposons que l'arbre contienne déjà p produits. Le pire cas est quand les produits ont été insérés par ordre croissant (ou décroissant de prix). Dans ce cas de figure, l'arbre est une chaine de p éléments. Si le produit qu'on ajoute a un prix plus élevé (respectivement plus faible) que les autres produits alors il faut entièrement parcourir avant d'ajouter le produit. La fonction est donc en O(p).

affichageABR : affichageABR est une fonction récursive. La fonction a au plus N*M produits à afficher. A chaque appel nous appelons deux fois affichageABR : pour le sous-arbre binaire de recherche gauche et pour le sous-arbre binaire de recherche droit ; et un nœud (i.e un produit) est affiché. Ainsi, à chaque appel nous réduisons de un la taille du problème.

Par exemple au premier appel, nous transformons le problème de taille N*M en un problème de taille N*M-1. L'algorithme s'arrête quand le pointeur de l'arbre a la valeur NULL c'est à dire quand il n'y a plus rien à afficher. Comme chaque appel est en O(1), affichageABR est en O(N*M).

rechercheProduits : Le pire des cas est quand tous les produits appartiennent à la fourchette de prix et que chaque rayon contient exactement M produits. Dans ce cas, le magasin contient N*M produits à afficher. La fonction commence par la construction de l'arbre : nous parcourons tous les produits et les insérons dans l'arbre.

Si tous les produits sont dans l'ordre croissant des prix, i.e si le prix du dernier produit de chaque rayon est inférieur au prix du premier produit du rayon suivant, alors la construction de l'arbre revient à la construction d'une chaine avec ajout en fin de chaine. Le nombre d'itérations est donc $\sum_{(i=1)}^{(N*M)} i = N*M\frac{1+N*M}{2} \approx O\left(\left(N*M\right)^2\right)$. Puis l'arbre est affiché en O(N*M). La complexité est donc $O\left(\left(N*M\right)^2\right)$.

fusionnerRayons : La fonction commence par chercher les rayons demandés par l'utilisateur. trouverRayon est en O(N). Puis la fonction fusionne les deux rayons dans le premier rayon. Pour cela nous parcourons les deux rayons en comparant les prix. La comparaison d'un produit est en O(1), tout comme l'ajout d'un produit du rayon deux dans le rayon un.

Comme nous parcourons les deux rayons en entier, la fusion est en O(M). Puis nous supprimons le rayon deux qui est vide, cela consiste simplement à le retirer de la liste chainée des rayons : O(1). Comme l'utilisateur donne un nouveau nom au rayon fusionné il reste à le décaler à la bonne place pour maintenir l'ordre croissant alphabétique.

Nous vérifions que le nom du rayon fusionné n'est pas déjà pris (ce qui permet par la même occasion de trouver où placer ce rayon dans la liste chainée) : O(N). Enfin, nous déplaçons la tête du rayon : O(1). Donc la complexité globale est O(N+M+N)=O(max(N,M)). Comme dans la plupart des magasins M>N, fusionnerRayons est en O(M).