LI Yue

FOREST Thomas

NF16 – TP3 : Listes linéaires chaînées

Objectif du TP :

Le but de ce TP est de se familiariser avec les listes chaînées et les différentes opérations nécessaires pour les manipuler. Pour ce faire, nous allons représenter les éléments non nuls de matrices creuses sous forme de listes chaînées afin de gagner de l'espace. Ainsi, chaque élément sera représenté par sa valeur et l'indice de la colonne à laquelle il se trouve. Différentes opérations pourront être réalisées sur les représentations de ces matrices telles que la recherche ou l'affectation d'une valeur, l'affichage d'une matrice ou bien l'addition de deux matrices.

Structure de données :

Afin de représenter correctement les différentes matrices, nous utiliserons une structure ***element*** composée de la valeur de l'élément et de l'indice de la colonne ainsi qu'un pointeur sur l'élément suivant. Ainsi, on nommera ***liste_ligne*** le type correspondant à un pointeur sur ***element***. La structure ***matrice_creuse*** contient le nombre de lignes et le nombre de colonnes de la matrice ainsi qu'un pointeur sur ***liste_ligne***. Ci-contre la représentation finale en **C** .

```c
// STRUCTURE ELEMENT
typedef struct Element {
    int col;
    int val;
    struct Element *suivant;
}element;

// TYPE LISTE_LIGNE
typedef element *liste_ligne;

// STRUCTURE MATRICE CREUSE
typedef struct MatriceCreuse {
    int Nlignes;
    int Ncolonnes;
    liste_ligne *tab_lignes;
}matrice_creuse;
```

Fonctions & Complexité:

1. *Fonctions ajoutées :*

- **isPositive(int *n)** : vérifie qu'un nombre entré est strictement positif.
- **isBetween(int *n, int a, int b)** : vérifie qu'un nombre est compris entre a et b.

Ces fonctions permettent simplement de contrôler les entrées de l'utilisateur afin d'éviter certains problèmes.

Complexité : **O(n)** car boucles *while* (n le nombre d'entrées de l'utilisateur).

- **afficherMenu()** : affiche le menu à l'utilisateur.
- **getChoix()** : récupère le choix de l'utilisateur pour le menu.
- **getValue()** : récupère la valeur à utiliser dans la fonction affecterValeur.
- **getNbMatrice()** : récupère le nombre de matrice à utiliser au début du programme.
- **getSizeMatrices(int *Nlignes, int *Ncolonnes)** : récupère la taille des matrices au début du programme.
- **getNumMatrice(int max)** : récupère le numéro de la matrice à manipuler.
- **getLigneColonne(int *i, int *j, int maxL, int maxC)** : récupère le numéro de la ligne et de la colonne de la matrice à manipuler.

Ces fonctions permettent simplement de récupérer les entrées de l'utilisateur. Nous avons décider de créer ces fonctions afin de simplifier la lecture du fichier *main.c*.

Complexité : **O(n)** car utilisations des fonctions de contrôles **isPositive** et **isBetween** (sauf pour les trois premières => **O(1)**).

- **inserer_tete(element *e, liste_ligne l)** : renvoie la liste_ligne l avec l'élément e en tête.

Complexité : **O(1)**

- **libererMatrice(matrice_creuse m)** : permet de libérer l'espace mémoire d'une matrice creuse.

Complexité : **O(N*M)** (N : nombre de lignes, M : nombre de colonnes). Boucle *for* parcourant la matrice ligne par ligne + boucle *while* parcourant chaque ligne.

2. *Fonctions du sujet :*

- **remplirMatrice(matrice_creuse *m, int N, int M)** :

Cette fonction permet de remplir une matrice creuse. Pour cela, on parcourt chaque ligne de la matrice en demandant les valeur de la matrice à l'utilisateur. Si la valeur entrée par l'utilisateur est différente de 0, on créer un nouvel élément et on l'ajoute à la liste chaînée *tab_lignes[i]*.

Complexité : **O(N*M)** (N : nombre de lignes, M : nombre de colonnes). Deux boucles *for* pour récupérer tous les éléments.

- **afficherMatrice(matrice_creuse m)** :

Cette fonction permet d'afficher une matrice creuse sous forme de tableau. Pour cela, on parcourt chaque ligne de la matrice à l'aide d'un pointeur sur chaque élément (***tmp***) et d'un indice (***j***) qui va de 0 au nombre de colonnes de la matrice. Ainsi, si ***j*** est égal à l'indice de la colonne de l'élément pointé par ***tmp*** alors on affiche la valeur de cet élément et on passe à l'élément suivant, sinon on affiche 0.

Complexité : **O(N*M)** (N : nombre de lignes, M : nombre de colonnes). Deux boucles *for* pour parcourir tous les éléments.

- **afficherMatriceListes(matrice_creuse m)** :

Cette fonction permet d'afficher une matrice creuse sous forme de liste. On représentera donc seulement les éléments significatifs (tous les éléments non nuls). Cette fonction ressemble à la précédente dans le fonctionnement. Or ici, nous n'avons pas besoin d'un indice de parcours étant donné que nous affichons seulement les éléments non nuls de la matrice. Ainsi, pour chaque ligne de la matrice, si celle-ci est vide on le signale, sinon on parcours tous les éléments de la ligne et on les affiches sous la forme **{col val}**.

Complexité : **O(N*M)** (N : nombre de lignes, M : nombre de colonnes) Boucle *for* parcourant la matrice ligne par ligne + boucle *while* parcourant chaque ligne.

- **rechercherValeur(matrice_creuse m, int i, int j)** :

Cette fonction permet de renvoyer la valeur qui se trouve à la ligne ***i*** et à la colonne ***j*** de la matrice ***m***. Pour cela, on parcourt simplement la ligne ***i*** de la matrice à l'aide d'un pointeur sur chaque élément (***tmp***). Ainsi, tant que ***tmp*** n'est pas NULL ET que la colonne de l'élément actuel est inférieure ou égale à ***j***, si la colonne de l'élément actuel est égale à ***j***, alors on renvoi la valeur de l'élément, sinon on passe à l'élément suivant. Si on sort de la boucle tant que, c'est que l'élément recherché est un 0.

Complexité : **O(M)** (M : nombre de colonnes). Une seule ligne à parcourir.

- **affecterValeur(matrice_creuse m, int i, int j, int val)** :

Cette fonction permet de changer une valeur d'une matrice. Trois cas se présentent alors à nous :

1. La valeur de la matrice est la même que la nouvelle valeur. Dans ce cas on ne fait rien.
2. La valeur de la matrice est égal à 0. Dans ce cas on doit insérer la nouvelle valeur au bon endroit dans la liste chaînée de la ligne en question.
3. La valeur de la matrice est différente de 0. Dans ce cas on parcourt la ligne jusqu'à la bonne colonne puis si la nouvelle valeur est 0, on doit supprimer l'ancien élément dans la liste chaînée, sinon on change simplement la valeur.

Complexité : Etant donné que l'on appelle la fonction **rechercherValeur** pour obtenir la valeur à remplacer dans la matrice, nous devons compte cette complexité. Ainsi nous avons :

- **Ω(M)** (M : nombre de colonnes) dans le cas n°1.
- **O(M+N)** (M : nombre de colonnes, N : nombre de lignes) dans les cas n°2 et 3.

La complexité ce cette fonction pourrait donc être amélioré en effectuant qu'un seul parcours.

- **additionerMatrices(matrice_cresue m1, matrice_creuse m2)** :

Cette fonction permet d'ajouter deux matrices en stockant le résultat dans la première. Pour cela, on parcourt chaque ligne des 2 matrices parallèlement. Tant qu'il y a des éléments dans les deux lignes, 3 cas se présentent :

1. La colonne de m1 est supérieure à la colonne de m2. Dans ce cas la valeur de m2 doit être ajoutée à celle de m1, puis on passe à la valeur suivante dans m2.
2. Les deux colonnes sont égales. Dans ce cas on somme les deux valeurs.
3. La colonne de m1 est inférieure à la colonne de m2. Dans ce cas on passe simplement à la valeur suivante dans m1.

Une fois que l'on est sortie de cette boucle, il faut ajouter, s'il en reste, tous les éléments de m2 à m1.

Complexité : **O(N*M)** (N : nombre de lignes, M : nombre de colonnes). Boucle *for* parcourant la matrice ligne par ligne + boucle *while* parcourant chaque ligne.

- **nombreOctetGagnes(matrice_creuse m)** :

Cette fonction permet de calculer le nombre d'octets gagnés grâce à cette représentation sous forme de listes chaînées comparer à si on avait simplement utilisé un tableau d'entiers. Pour cela, on doit parcourir toute la matrice pour y compter le nombre d'éléments non nuls. Ensuite on effectue de simples calculs pour compter le nombre d'octets de notre représentation.

Complexité : **O(N*M)** (N : nombre de lignes, M : nombre de colonnes). Boucle *for* parcourant la matrice ligne par ligne + boucle *while* parcourant chaque ligne.