Eloïse Launay
Cléo Lesage

Rapport NF16
TP3

Langage : C
IDE : codeBlock
TP du mardi semaine A, 8h-10

Sommaire

Introduction

Une matrice est dite creuse lorsque le nombre d'éléments nuls y figurant est très supérieur à celui des éléments non nuls. Par souci d'économie, on peut représenter une telle matrice en ne tenant compte que des éléments non nuls : une matrice $N \times M$ est représentée par un tableau de N listes chaînées qui contiennent les éléments non nuls d'une ligne de la matrice coordonnés selon le rang de leur colonne. Chaque élément d'une liste contient l'indice de la colonne et la valeur de l'élément.

L'objectif de ce TP est de modéliser des matrices creuses à l'aide des listes chaînées en C. Nous effectuons des opérations simples sur celle-ci comme la recherche d'une valeur, l'addition de deux matrices ou encore leur affichage. Afin de réaliser ce TP, nous avons eu 2 semaines. Lors de la séance, nous avons pu mettre en place des algorithmes pour chacune des fonctions demandées. Cela nous a permis de se rendre compte du travail requis pour chaque question, réfléchir aux fonctions supplémentaires à ajouter et donc se partager le travail. Nous avons donc divisé chaque fonction entre nous, également les tests et la réalisation du main. Cela nous a permis de réaliser le travail efficacement tout en n'omettant rien de ce qui était demandé.

Fonctions ajoutées au code

Nous avons décidé d'ajouter deux fonctions supplémentaires : une pour l'initialisation d'une matrice creuse et une autre pour la libération de l'espace mémoire d'une matrice creuse. Cela n'était pas obligatoire, car nous ne les utilisons qu'une seule fois dans le code, mais par souci de maintenabilité nous avons décidé d'en faire des fonctions.

`matrice_creuse* constructeurMatriceCreuse()`
Cette fonction permet d'allouer de l'espace mémoire pour la structure d'une matrice creuse, puis elle la crée et l'initialise en intégrant son nombre de lignes, de colonnes, puis un tableau de pointeurs visant chaque ligne de la matrice.
Cette fonction est utilisée une fois lors du remplissage d'une matrice creuse. En plus de la praticité de son existence, cela permet également d'épurer le main.

`void deconstructeurMatriceCreuse(matrice_creuse m)`
Dans la consigne du TP, il était précisé que la mémoire allouée devait être libérée. C'est pour cela que cette fonction a été créée. En effet, cette fonction est appelée lorsque l'utilisateur quitte le programme. Alors, la mémoire dynamiquement allouée pour chaque élément des listes chaînées des matrices est libérée, puis celle pour ces listes.

Organisation des tests

Pour vérifier le bon fonctionnement de notre jeu, nous avons réalisé différents tests. Ceux-ci sont trouvables dans le dossier “test”. Ce sont des fichiers que nous donnons en entrée à notre programme avec la commande : `script\main.exe < tests\in-XX.txt`

`XX` sont deux chiffres. Le premier représente le numéro de la fonction que l’on souhaite tester et le second le numéro du test. Nous avons utilisé cette méthode pour simplifier et accélérer notre manière de tester.

Liste des tests passés :

- in-10 : remplir une matrice simple
- in-11 : remplir une matrice vide
- in-12 : remplir une matrice de 100 lignes
- in-13 : remplir une matrice à dimension négative
- in-14 : remplir une matrice à valeur négative
- in-15 : remplir une matrice avec des erreurs utilisateur sur le nombre de ligne
- in-16 : remplir une matrice avec des erreurs utilisateur sur la valeur
- in-20 : afficher une matrice sous forme d’un tableau
- in-21 : afficher une matrice vide sous forme d’un tableau
- in-30 : afficher une matrice sous forme d’une liste chainé
- in-31 : afficher une matrice sous forme d’une liste chainé vide
- in-40 : recherche d’une valeur existante
- in-41 : recherche d’une valeur nul
- in-42 : recherche d’une valeur inexistante
- in-50 : affectation d’une valeur existante
- in-51 : affectation d’une nouvelle valeur à la fin d’une ligne
- in-52 : affectation d’une nouvelle valeur avant une autre valeur
- in-53 : affectation d’une valeur inexistante
- in-60 : addition de deux matrices
- in-61 : addition avec une matrice vide
- in-70 : gain d'octet
- in-71 : aucun gain
- in-72 : gain d’octet d'une matrice vide

Complexités

➔ ***matrice_creuse* constructeurMatriceCreuse()***

Dans cette fonction, on alloue de la mémoire pour “tab_lignes”, le tableau de pointeurs visant chaque ligne de la matrice. Il faut donc réserver de la place pour N lignes. De plus, toutes les cases de ce tableau sont initialisées à NULL, donc encore une fois le tableau est parcouru pour N lignes. La complexité est donc ***O(N)***.

➔ ***void deconstructeurMatriceCreuse(matrice_creuse m)***

Dans cette fonction, le pire des cas correspond au fait que la matrice ne contient pas d’élément non nul. Il faut donc tout parcourir et libérer toutes les cellules. La fonction parcourt donc chaque ligne et supprime l’élément de chaque colonne. On a donc une complexité de ***O(NM)***.

➔ ***void remplirMatrice(matrice_creuse *m, int N, int M)***

Les lignes de code de cette fonction permettant de vérifier l’entrée de l’utilisateur, puis celles permettant l’ajout d’un nouvel élément (non nul) ont une complexité O(1). Dans tous les cas, la fonction va parcourir toutes les lignes et toutes les colonnes (via les boucles while) afin de demander à l’utilisateur d'entrer une valeur. La complexité est donc de ***O(NM)***.

➔ ***void afficherMatrice(matrice_creuse m)***

Pour cette fonction, dans tous les cas, il s’agit de parcourir toutes les colonnes de chaque ligne de la matrice. Pour une matrice de N lignes et M colonnes, on a donc une complexité de ***O(NM)***.

➔ ***void afficherMatriceListes(matrice_creuse m)***

Dans le pire des cas, chaque élément de la matrice est non nul. La fonction va donc parcourir chaque élément de chaque ligne. Ce qui correspond à parcourir toutes les colonnes M de toutes les lignes N. On a alors une complexité ***O(NM)***.

➔ ***int rechercherValeur(matrice_creuse m, int i, int j)***

Pour cette fonction, le pire des cas correspond au fait que dans la ligne i (où se trouve l’élément recherché), tous les éléments soient non nuls et la valeur à trouver n’existe pas ou se trouve en dernière colonne. Dans ce cas, la fonction va avoir une complexité **O(M)** où M est le nombre de colonne de la ligne.

➔ ***void affecterValeur(matrice_creuse m, int i, int j, int val)***

Dans cette fonction, le pire des cas revient à devoir insérer une valeur à la fin de la ligne i contenant que des éléments non nuls. Ainsi, toutes les colonnes de cette ligne seront vérifiées avant d’insérer la valeur au bout de la liste, la complexité est donc ***O(M)***, avec M le nombre de colonnes de la ligne (soit nombre max de colonnes de la matrice).

➔ ***void additionerMatrices(matrice_creuse m1, matrice_creuse m2)***

Dans le pire des cas, au moins la matrice m2 n’a pas d’élément nul. Chaque ligne N et chaque colonne de M de la matrice sont donc comparées puis sommées aux éléments de

m1 ou directement insérées dans m1. Toute la matrice m2 NxM étant parcouru, la fonction a pour complexité ***O(NM)***.

➔ ***int nombreOctetsGagnes(matrice_creuse m)***

Dans le pire des cas, la matrice ne contient pas d'élément non nul. Alors, la fonction va parcourir toutes les N lignes, et dans ces lignes, elle traverse toute la liste chaînées contenant donc le nombre de colonnes M de la matrice. La complexité est ***O(NM)***.

Ici, on prend bien en compte le pire des cas, avec tous les éléments d'une matrice qui sont non nul.

Problème rencontré

Lors de la démonstration de notre code, la fonction additionnerMatrices n'a pas fonctionnée correctement. Malgré un grand nombre de tests, nous n'avons pas prévu un cas. Celui où on devait ajouter un élément avant un autre. Notre programme s'arrêtait subitement lorsqu'on effectuait ce test, c'était un problème de pointeur. Finalement, nous n'avons pas réussi à trouver précisément qu'est ce qui causait ce bug donc nous avons préféré recoder la fonction entièrement. En plus, nous avons rajouté le test qui nous manquait dans notre liste.