

Rapport NF16

A2024 - TP 3 Listes Chainées

Groupe 2 A

Kristi TENEQEXHI et Yang XIANG

La liste des structures et des fonctions demandes et supplémentaires

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

// TYPE ELEMENT
typedef struct Element element;

// TYPE LISTE_LIGNE
typedef element *liste_ligne;

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

// FONCTIONS DEMANDEES
void remplirMatrice(matrice_creuse *m, int N, int M);
void afficherMatrice(matrice_creuse m);
void afficherMatriceListes(matrice_creuse m);
int rechercherValeur(matrice_creuse m, int i, int j);
void affecterValeur(matrice_creuse m, int i, int j, int val);
void additionnerMatrices(matrice_creuse m1, matrice_creuse m2);
int nombreOctetsGagnes(matrice_creuse m);

// FONCTIONS SUPPLEMENTAIRES
element *creerElement(int colonne, int valeur);
void viderBuffer();

//Notre Fonctions - SUPPLEMENTAIRES
matrice_creuse *cree_matrice();
void free_matrice(matrice_creuse *);
void free_contenuMatrice(matrice_creuse);
```

On a gardé la liste des structures suggère par l'annonce sans aucun changement.

Raison de choix des fonctions supplémentaires

- matrice_creuse *cree_matrice()

On a eu l'envie de ne pas écrire le saisir et les teste des saisit dans la main a fin de récupère une pointeur de l'espace mémoire biens allouer et puis retourner par une fonction isole de notre

matrice_creuse *matrice pour le stocker dans notre liste mémoire des matrices qui se trouve dans le main() matrice_creuse ** memory_matrice Dans cette fonction on teste le saisir d'utilisateur sur les dimension de la matrice creuse et une fois on a allouer l'espace mémoire on fait appelle à la fonction demande void remplirMatrice(matrice_creuse *m, int N, int M);

- element *creerElement(int colonne, int valeur);

Cette fonction est utilisée pour créer un nouvel élément. A cause du fait qu'on a besoin de le réutiliser plusieurs fois dans plusieurs fonctions, notamment en

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

void additionnerMatrices(matrice_creuse m1, matrice_creuse m2); et

void affecterValeur(matrice_creuse m, int i, int j, int val) on a le mis comme une

fonction propre qui prend les paramètre nécessaire et minimale (nb colonne et valeur) pour initialiser un élément de notre liste chainer simple. Il renvoi le pointeur de l'espace mémoire de notre nouvel élément.

- void free_contenuMatrice(matrice_creuse m)& void free_matrice(matrice_creuse*m)

Sont des fonctions qui prend respectivement une matrice ou un pointeur de matrice et le déréférencer pour libérer le mémoire qu'on a allouer dans la fonction matrice_creuse *cree_matrice() et element *creerElement(int colonne, int valeur). Ils parcourent tous les éléments pour trouver l'un à la fin de la liste d'une ligne, le libérer et reviennent à retraverser jusqu'à la ligne est vide. On suit cette procédure pour chaque ligne.

Complexité de chaque fonction implémentée.

Pour ce partie la, les notations (n,m) sont utilisé pour caractériser le taille de problème ou n-nombre de ligne et m-nombre de colonnes saisir pour chaque matrice par l'utilisateur:

element *creerElement(int colonne, int valeur)

Complexité en O(1) constant, juste 1 allocation mémoire, 3 affectations et une return.

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

Complexité en $O(n*m)$ il y a deux boucles imbriquées la première qui itère à chaque ligne et en dedans le deuxième qui itère chaque élément potentiel pour la lecture des données. Il y a un nombre constant des appels dans le corps des boucles et un appel (dans les pires des cas si l'élément n'est pas nul/0) à la fonction

`element *creerElement(int colonne, int valeur);` qui est de complexité linéaire

```
matrice_creuse *cree_matrice()
```

Complexité en $O(n*m)$ si l'utilisateur saisit des valeurs acceptables pour le nombre des lignes et colonnes. On a une boucle while qui répète à l'infini si l'utilisateur donne des valeurs non cohérentes. En dehors de cette boucle on a un nombre constant des opérations logiques, arithmétiques et d'affectation sauf qu'à cause de l'appel à la fonction

`void remplirMatrice(matrice_creuse *m, int N, int M);` notre fonction hérite sa complexité.

```
void afficherMatrice(matrice_creuse m)
```

Complexité en $O(n*m)$ En totalité avec toutes les optimisations, on a une boucle père qui itère chaque ligne et des boucles fils qui affiche le résultat nécessaire soit nul soit la valeur. En prend en compte les cas de ligne vide, de ligne avec quelques éléments avec des 0 avant, et les liste terminer par une liste de zéros mais sans faire des répétitions inutiles des boucles.

```
void afficherMatriceListes(matrice_creuse m)
```

Complexité en $O(n*m)$ Dans les pires des cas on a des lignes pleines des éléments (c.-à-d. une matrice totalement remplie) et on itère toujours dans chaque élément, mais ici ce nombre devient m . En réalité si la matrice est creuse la complexité devient plutôt $n \times$ le nombre des colonnes non nul de la ligne. Les instructions dans les corps des boucles sont constantes et liées avec le formatage d'affichage.

```
int rechercherValeur(matrice_creuse m, int i, int j)
```

Complexité en $O(m)$ On accède directement la ligne et itère chaque élément (valeur non nulle de matrice) dans la deuxième boucle jusqu'au moment qu'on dépasse la ligne désirée. Dans les pires des cas on a la ligne i avec pleines des éléments et la colonne dans laquelle nous avons besoin de récupérer l'élément c'est la dernière (indice $[m-1]$). Alors dans ce cas de figure il faut faire les tests logiques de la boucle m fois et exécuter le corps de la boucle, composé d'une seule affectation, $m-1$ fois.

```
void affecterValeur(matrice_creuse m, int i, int j, int val)
```

Complexité en $O(m)$ On accède directement la ligne. On a plusieurs cas : ligne vide, positionnement avant tous, positionnement au milieu de deux ou à la fin, positionnement dans un élément qui existe déjà et tous avec la variation du cas que l'utilisateur ajoute la valeur 0/NULL. On pire des cas on a une ligne pleine des éléments et cherche à affecter le tout dernier colonne (le dernier élément de notre liste chaînée) c.-à-d. on doit itère chaque élément, le nombre duquel correspond au nombre de colonnes m . Une fois on a trouvé la bonne position on fait des opérations logiques pour tester la valeur nulle et le positionnement en tête, et puis les créations d'élément nouveau, les affectations ou libération de mémoire nécessaire qui sont des opérations de nombre constant en fonction de la taille de problème.

```
void additionerMatrices(matrice_creuse m1, matrice_creuse m2)
```

Complexité en $O(n*m)$ On it n re tous les n -lignes dans les deux matrices en m me temp (d j  les matrices ont les m mes dimensions $[n \times m]$). Les pire cas sont les cas o  la ligne de la somme a m  l ments, c.- -d. ligne plein car chaque colonne apporte un  l ment, et le dernier  l ment (ici co incide avec la derni re colonne) est obtenu par la derni re colonne de la deuxi me matrice (soit on a fait une somme entre la derni re colonne de cette ligne des deux matrices, soit on  tait en train de parcourir la deuxi me matrice et on cr e un nouvel  l ment pour la colonne vide de la premi re matrice.) Le meilleur cas d'une somme avec une ligne pleine c'est si la deuxi me matrice a une ligne vide, est on laisse la ligne de la matrice 1 comme elle est.

```
int nombreOctetsGagnes(matrice_creuse m)
```

Complexit  en $O(n*m)$ M me raisonnement comme avant, deux boucles l'une dans l'autre, la premi re pour it n re les n lignes et la deuxi me pour it n re chaque  l ment et en pire des cas ce nombre co incide avec le nombre des colonnes m , c.- -d. toute la matrice est remplie avec des  l ments non nuls. (c'est plus une matrice creuse !) En le corps des boucles juste des op rations constantes d'affectations et une op ration incr ment (arithm tique) pour  num rer les  l ments. En dehors des op rations constantes.

```
void viderBuffer ()
```

Complexit  en $O(l)$ Ou l est la taille du probl me c.- -d. le nombre des caract res diff rents de '\n' et de EOF

```
void free_contenuMatrice(matrice_creuse m) & void free_matrice(matrice_creuse* m)
```

Complexit  en $O(n*m^2)$ Avec la boucle externe on parcourt chaque ligne. Chaque fois qu'on supprime le dernier  l ment d'une ligne, la fonction recommence au d but de la liste et parcourt jusqu'  la fin pour trouver et lib rer cet  l ment. Ce processus est r p t  pour chaque  l ment de la liste et en pire des cas cela co incide avec le nombre des colonnes n de cette ligne. La boucle interne rend cela possible avec une r p tition combin e de : $m + (m - 1) + (m - 2) + \dots + 1 = \frac{m(m+1)}{2} = O(m^2)$. Les op rations dedans sont constantes. En totalit  si toute la matrice est pleine (c'est plus creuse) on obtient : $n \frac{m(m+1)}{2} = O(nm^2)$

Changement apr s la s ance et pourquoi

Apr s la derni re s ance de TP d di e aux tests, le professeur a not  que nous devions am liorer la fonction `void affecterValeur(matrice_creuse m, int i, int j, int val)`, car nous n'avions pas pris en compte les entr es de valeur nulle provenant de l'utilisateur. Cela entra nerait le stockage de la valeur z ro dans un nouvel  l ment, ce qui va   l'encontre de l'objectif de nos listes cha n es, et ne permettrait pas de supprimer une valeur existante. Par cons quent, apr s consultation, nous avons d cid  de clarifier le code et de rendre les 6 grands cas plus lisibles et visibles :

- Le cas d'une ligne vide et  l ment non nul,
- Le cas d'une ligne vide et  l ment nul (ne fait rien),
- La mise   jour d'une valeur existante,
- La suppression d'une valeur existante si l' l ment est nul,
- Ne pas ajouter de z ro dans une liste non vide en tant qu' l ment de t te ou en tant qu' l ment du milieu/fin.
- Cr er un nouvel  l ment dans les deux derniers cas si la valeur n'est pas nulle.