Rapport NF16

A2024 - TP 3 Listes Chainées

Groupe 2 A

Kristi TENEQEXHI et Yang XIANG

La liste des structures et des fonctions demandées et supplémentaires

```c
// SRUCTURE 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;
```

```c
// 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 additionerMatrices(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 suggérées par l'annonce sans aucun changement.

Raisons du choix des fonctions supplémentaires

- `matrice_creuse *cree_matrice()`

Nous avons choisi de ne pas écrire la saisie et les tests de saisie dans le main, afin de récupérer un pointeur de l'espace mémoire bien alloué, qui sera ensuite retourné par une fonction isolée `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 alloué l'espace mémoire on fait appelle à la fonction demandée `void remplirMatrice(matrice_creuse *m, int N, int M);`.

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

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

`void remplirMatrice(matrice_creuse *m, int N, int M);`
`void additionerMatrices(matrice_creuse m1, matrice_creuse m2);` **et**
`void affecterValeur(matrice_creuse m, int i, int j, int val)` on a définie comme une fonction distincte qui prend les paramètres nécessaires et minimaux (numéro de colonne et valeur) pour initialiser un élément de notre liste chaînée simple. Elle renvoie le pointeur de l'espace mémoire de notre nouvel élément.

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

Ce sont des fonctions qui prennent respectivement une matrice, ou un pointeur de matrice et le déréférencent, pour libérer la mémoire allouée dans la fonction `matrice_creuse *cree_matrice()`**et** `element *creerElement(int colonne, int valeur)`. Elles parcourent tous les éléments pour trouver celui en fin de liste d'une ligne, le libèrent, puis retraversent de début la liste jusqu'à ce que la ligne soit vide. Cette procédure est répétée pour chaque ligne.

Complexité de chaque fonction implémentée

Pour cette partie, les notations (n, m) sont utilisées pour caractériser la taille du problème, où **n** - nombre de lignes et **m** - nombre de colonnes saisies 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 itère sur chaque ligne, et dedans la deuxième itère sur chaque élément potentiel pour la lecture des données. Il y a un nombre constant d'appels dans le corps des boucles et un appel (dans le pire 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 de lignes et de colonnes. On a une boucle while s'exécute indéfiniment si l'utilisateur fournit des valeurs incohérentes. En dehors de cette boucle, il y a un nombre constant d'opérations logiques, arithmétiques et d'affectation, sauf qu'en raison de l'appel à la fonction `void remplirMatrice(matrice_creuse *m, int N, int M);` notre fonction hérite de sa complexité.

`void afficherMatrice(matrice_creuse m)`

Complexité en O(n*m) Au total, avec toutes les optimisations, on a une boucle père qui itère chaque ligne et des boucles fils qui affichent le résultat nécessaire, soit nul, soit la valeur. On prend en compte les cas de lignes vides, de lignes contenant quelques éléments précédés de 0, et de listes se terminant par une série de zéros mais sans faire des répétitions inutiles des boucles.

`void afficherMatriceListes(matrice_creuse m)`

Complexité en O(n*m) Dans le pire des cas, on a des lignes pleines des éléments (c.-à-d. une matrice totalement remplit) et on itère toujours sur chaque élément, mais ici ce nombre deviens ***m***. En réalité, si le matrice est creuse, la complexité deviens plutôt n*le nombre des colonnes non nulles de la ligne. Les instructions dans le corps des boucles sont constantes et liées au formatage d'affichage.

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

Complexité en O(m) On accède directement à la ligne et itère sur chaque élément (valeur non nul de la matrice) dans la deuxième boucle jusqu'à dépasser la ligne désirée. Dans les pires des cas on a la ligne i avec pleines des éléments et la colonne dans laquelle nous devons récupérer l'élément est la dernière (indice [m-1]). Alors, dans ce cas de figure il faut effectuer 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 à la ligne. Plusieurs cas sont possibles : ligne vide, positionnement avant tous les éléments, positionnement entre deux éléments ou à la fin, positionnement dans un élément existant et tous avec la variation du cas que l'utilisateur ajout le valeur 0/NULL. Dans le pire des cas, la ligne est pleine d'éléments et on cherche à affecter la toute dernier colonne (le dernier élément de notre liste chaînée) c.-à-d. on doit itérer sur chaque élément, dont le nombre correspond au nombre de colonnes ***m***. Une fois la bonne position trouvée, des opérations logiques sont effectuées pour tester la valeur nulle et le positionnement en tête, suivies de la création de nouveaux éléments, des affectations, ou des libérations de mémoire nécessaires, qui sont des opérations en nombre constant par rapport à la taille du problème.

`void additionerMatrices(matrice_creuse m1, matrice_creuse m2)`

Complexité en O(n*m) On itère sur les n lignes des deux matrices en même temps (les matrices ayant déjà les mêmes dimensions [nxm]). Le pire cas survient lorsque la ligne de la somme contient ***m*** éléments, c.-à-d. une ligne pleine, car chaque colonne apporte un élément, et le dernier élément (qui correspond ici à la dernière colonne) est obtenu par la dernière colonne de la deuxième matrice. Soit une somme entre les dernières colonnes de cette ligne dans les deux matrices, soit on parcourt la deuxième matrice pour créer un nouvel élément pour la colonne vide de la première matrice. Le meilleur cas d'une somme avec une ligne pleine se présente si la deuxième matrice a une ligne vide, auquel cas on laisse simplement la ligne de la matrice 1 telle quelle.

`int nombreOctetsGagnes(matrice_creuse m)`

Complexité en O(n*m) Même raisonnement qu'auparavant : deux boucles imbriquées, la première pour itérer sur les ***n*** lignes et la seconde pour itérer sur chaque élément. Dans le pire des cas, ce nombre coïncide avec le nombre de colonnes ***m***, c.-à-d. que toute la matrice est remplie d'éléments non nuls (ce n'est plus une matrice creuse !). Dans le corps des boucles, seules des opérations constantes d'affectation et une opération d'incrémentation (arithmétique) pour énumérer les éléments sont effectuées. En dehors, il n'y a que des opérations constantes.

`void viderBuffer ()`

Complexité en O(l) Ou ***l*** est la taille de problème c.-à-d. le nombre de caractères différents de '\n' et de EOF.

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

Complexité en O(n*m²) Avec la boucle externe, on parcourt chaque ligne. Chaque fois qu'on supprime le dernier élément d'une ligne, la fonction recommence au début de la liste et parcourt jusqu'à la fin pour trouver et libérer cet élément. Ce processus est répété pour chaque élément de la liste et dans le pire des cas cela coïncide à le nombre des colonnes **m** de cette ligne. Les boucles internes rendent cela possible avec une répétition combinais de : $m + (m-1) + (m-2) + \cdots + 1 = \frac{m(m+1)}{2} = O(m^2)$. Les opérations dedans sont constant. En totalité si tout la matrice est pleine (ce n'est plus creuse) on obtient : $n\frac{m(m+1)}{2} = O(nm^2)$

Changement après la séance et pourquoi

Après la dernière séance de TP dédiée aux tests, le professeur a noté que nous devions améliorer la fonction `void affecterValeur(matrice_creuse m, int i, int j, int val)` , car nous n'avions pas pris en compte les entrées de valeur nulle provenant de l'utilisateur. Cela entraînerait le stockage de la valeur zéro dans un nouvel élément, ce qui va à l'encontre de l'objectif de nos listes chaînées, et ne permettrait pas de supprimer une valeur existante. Par conséquent, après consultation, nous avons décidé de clarifier le code, faire des changements et de rendre les tous 6 grand cas plus lisibles et visibles :

- Le cas d'une ligne vide et élément non nul,
- Le cas d'une ligne vide et élément nul (ne fait rien),
- La mise à jour d'une valeur existante,
- La suppression d'une valeur existante si l'élément est nul,
- Ne pas ajouter de zéro dans une liste non vide en tant qu'élément de tête ou en tant qu'élément du milieu/fin.
- Créer un nouvel élément dans les deux derniers cas si la valeur n'est pas nulle.