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

```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 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 additionerMatrices(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 imbriquer la première qui itinère à chaque ligne et en dedans le deuxième qui itinéré chaque élément potentiel pour la lecture des données. Il y a une nombre constat des appelle dans le corps des boucles et une appelle (dans les pires des cas si le élément n'est pas nul/0) a la fonction `element *creerElement(int colonne, int valeur);` qui est de complexité linaire

`matrice_creuse *cree_matrice()`

Complexité en O(n*m) si l'utilisateur saisir des valeurs acceptables pour le nombre des ligne 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 logique, arithmétique et d'affectation sauf qu'à cause de l'appelle 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 itinère 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 remplit et on itinère toujours dans 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 nul de la ligne. Les instructions dans les corps des boucles sont constantes et lie avec le formatage d'affichage.

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

Complexité en O(m) On accède directement la ligne et itinère chaque élément (valeur non nulle de matrice) dans la deuxième boucle jusqu'au moment qu'on dépasse a la ligne désiré. Dans les pires des cas on a la ligne i avec pleines des éléments et la colonne dans lequel nous avons besoins de récupérer l'élément c'est le dernière (indice [m-1]). Alor 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 ajout le 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 chainer) c.-à-d. on doit itinère chaque élément, le nombre duquel correspond au nombre de colonnes ***m***. Une fois on a trouvé la bonne position on faire 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 itinère tous les n-lignes dans les deux matrices en même temp (déjà les matrice ont les mêmes dimensions [nxm]). Les pires cas sont les cas où la ligne de la somme a m éléments, c.-à-d. ligne pleine car chaque colonne apport un élément, et le dernière élément (ici coïncide avec dernière colonne) est obtenu par la dernière colonne de la deuxième matrice (soit on a fait une somme entre le dernière colonne de cette ligne des deux matrice, soit on était en train de parcourir le deuxième matrice e de crée une nouvelle élément pour le colonne vide de première matrice.) Le meilleur cas d'une somme avec une ligne pleine c'est si la deuxième matrice a une ligne vide, et on juste laisse le ligne de matrice 1 comme elle est.

`int nombreOctetsGagnes(matrice_creuse m)`

Complexité en O(n*m) Même raisonnement comme avant, deux boucle un dedans l'autre, le première pour itinère les ***n*** lignes et le deuxième pour itinère chaque élément et en pire des cas ce nombre coïncide avec le nombre des colonnes ***m***, c.-à-d. tous la matrice est remplie avec des éléments non nuls. (c'est plus une matrice creuse !) En le corps des boucles juste des opérations constantes d'affectations et un opération incrément (arithmétique) pour énumérer les éléments. En dehors des opérations constants.

`void viderBuffer ()`

Complexité en O(l) Ou ***l*** est le taille de problème c.-à-d. le nombre des caractères diffèrents 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 supprimes 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 en pire des cas cela coïncide a le nombre des colonnes **n** de cette ligne. Les boucle interne rend 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 (c'est plus creuse) on obtiens : $n\frac{m(m+1)}{2} = O(nm^2)$

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 et de rendre les 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.