ISIMA Première Année

Compte-Rendu de TP

Structures de Données

Gestion d'une matrice creuse de grande taille

Benjamin BARBESANGE
Pierre-Loup PISSAVY
Groupe G21

Enseignant :
Michelle CHABROL

juin 2015

ISIMA

Table des matières

1 | Présentation

Le but de ce TP est de pouvoir gérer une matrice creuse (contenant beaucoup de valeurs nulles) de grande taille. Il est demandé de pouvoir lire la matrice depuis un fichier et retourner la valeur d'un élément.

Les opérations suivantes sont permises avec la matrice :

- Lire depuis un fichier,
- Afficher la matrice,
- Afficher un élément connaissant sa position dans la matrice,
- Libérer la mémoire allouée pour la matrice.

1.1 Structures de données employées

1.1.1 Matrice

Les éléments non nuls de la matrice sont rangés dans des tables. Nous disposons d'une table principale réprésentant les lignes non-nulles. Il s'agit d'un tableau renseignant 2 éléments dans chaque case :

- Numéro de la ligne concernée,
- Pointeur sur la liste chaînée contenant les colonnes.

Comme l'indique le schéma de la figure 1.1, la structure de matrice contient également le nombre total de lignes non-nulles, ainsi que le nombre maximum de colonnes. Nous avons pris la liberté d'ajouter cette dernière valeur afin de faciliter l'affichage de la matrice, fonction qui n'était pas demandée, mais que nous fournissons tout de même. En effet, cela nous évite de parcourir chacune des listes chaînées à la recherche de l'indice de colonne le plus élevé.

Chaque matrice peut être stockée dans un fichier rédigé de la manière suivante (exemple en figure 1.2) :

- Le nom du fichier ne doit pas contenir d'espace,
- Les valeurs nulles ne sont pas inscrites,
- L'ordre entre les lignes n'a pas d'importance,
- A raison d'une ligne du fichier par élément non-nul de la matrice, et chaque information étant séparée par une espace au moins :
  `Ligne␣Colonne␣Valeur`

$$M = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 42 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Exemple de matrice creuse

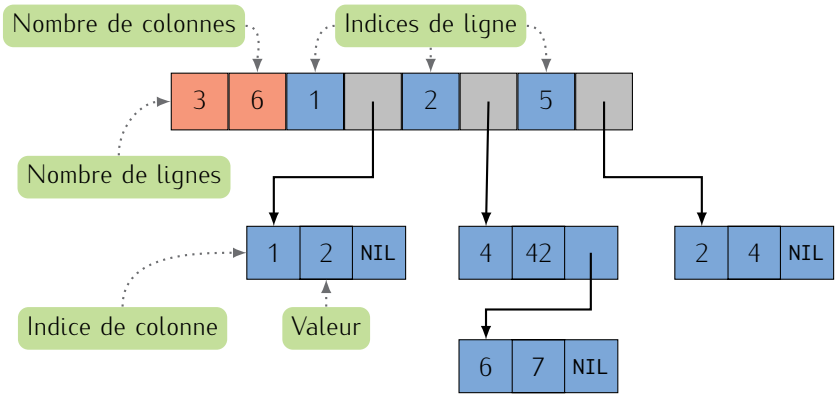

FIGURE 1.1 – Structure de matrice creuse

Matrice M

```
2 6 7
1 1 2
5 2 4
2 4 42
```

FIGURE 1.2 – Fichier descriptif équivalent

1.1.2 Fichier de commandes

Afin d'effectuer les tests, nous proposons l'interprétation d'un fichier qui peut être donné comme premier paramètre.

La structure de ce fichier doit respecter les règles suivantes :

- 80 caractères au maximum par ligne,
- Les caractères suivants sont acceptés en début de ligne :
  - **R :** Lecture de la matrice à partir d'un fichier, le nom de ce fichier doit être indiqué ensuite,
  - **E :** Affichage de la valeur d'un élément, nécessite les indices de ligne puis de colonne,
  - **A :** Afficher la matrice,
  - **L :** Libérer la matrice,
  - **#** : Provoque l'affichage du texte qui suit (commentaire affiché).
- Il est nécessaire d'indiquer des valeurs entières,
- Si l'on souhaite lire une matrice apres en avoir créé une première, il est nécessaire de libérer cette dernière,
- Tout autre caractère ou bien une ligne vide provoqueront l'affichage d'une ligne vide.

1.2 Organisation du code source

Nous avons décomposé le TP en 2 parties. La première partie est chargée de gérer les listes chaînées (utilisées dans la définition de la table secondaire) et la seconde est chargée de gérer la matrice.

1.2.1 Gestion des listes chaînées

- `src/list.h`
- `src/list.c`

1.2.2 Gestion de la matrice

- `src/mat.h`
- `src/mat.c`

1.2.3 Programme principal

- `src/main.c`

2 | Détails du programme

2.1 Gestion des listes chaînées

Code C

```c
/*  list.h
  Header

  -----| LISTE CHAINEE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#ifndef __LISTE_H__
#define __LISTE_H__

  #include <string.h>
  #include <ctype.h>

  typedef struct _node_t {
    int col;               /* Numero de colonne */
    int val;               /* Valeur */
    struct _node_t *next;  /* Colonne suivante */
  } node_t;

  typedef node_t cell_t;

  cell_t ** rech_prec(cell_t **, int, short int*);
  void supp_cell(cell_t **);
  void liberer_liste(cell_t **);
  int ins_cell(cell_t **, cell_t *);
  cell_t * creer_cell(int, int);

#endif
```

Code C

```c
/*  list.c
  Fonctions de gestion de la liste chainee

  -----| LISTE CHAINEE |-----

  BARBESANGE Benjamin,
```

```
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#include <stdio.h>
#include <stdlib.h>
#include "list.h"

/*  void adj_cell(cell_t **prec, cell_t *elt)
  Ajoute une cellule apres un element partir d'un pointeur sur l'element
  et d'un pointeur sur le pointeur de l'element apres lequel ajouter

  Entrees :
    cell_t **prec : pointeur sur le pointeur de l'element apres lequel ajouter
    cell_t *elt : pointeur sur l'element a ajouter a la liste chainee

  Sortie :
    Aucune
*/
void adj_cell(cell_t **prec, cell_t *elt) {
  elt->next = (*prec);
  (*prec) = elt;
}

/*  cell_t ** rech_prec(cell_t **liste, int col, short int *existe)
  Recherche le precedent d'un element dans la liste chainee triee selon les colonnes

  Entrees :
    cell_t **liste : pointeur sur le pointeur du premier element de la liste chainee
    int col : indice de colonne
    short int *existe : variable en entree/sortie indiquant si la colonne est presente ou non
      0 : absence
      1 : presence

  Sortie :
    cell_t ** : pointeur sur le pointeur de l'element precedent
*/
cell_t ** rech_prec(cell_t **liste, int col, short int *existe) {
  cell_t **prec = liste;
  while ((*prec) && (*prec)->col < col) {
    prec = &((*prec)->next);
  }
  /* Booleen de presence     */
  /* 1 : present             */
  /* 0 : absent              */
  *existe = (*prec && (*prec)->col == col)?1:0;
  return prec;
}

/*  void supp_cell(cell_t **prec)
  Permet de supprimer un element dans la liste chainee a partir
  de son precedent

  Entrees :
    cell_t **prec : pointeur sur le pointeur de l'element precedent l'element a supprimer
```

```
63
64    Sortie :
65      Aucune
66  */
67  void supp_cell(cell_t **prec) {
68    cell_t *elt = *prec;
69    *prec = elt->next;
70    free(elt);
71  }
72
73  /*  void liberer_liste(cell_t **liste)
74    Libere les allocations memoires de la liste
75
76    Entrees :
77      cell_t **liste : pointeur sur le pointeur du premier element de la liste chainee
78
79    Sortie :
80      Aucune
81  */
82  void liberer_liste(cell_t **liste) {
83    while (*liste) {
84      supp_cell(liste);
85    }
86    *liste = NULL;
87  }
88
89  /*  int ins_cell(cell_t **liste, cell_t *elt)
90    Permet d'inserer une cellule a la bonne place dans la liste chainee
91    Les colonnes sont triees par ordre croissant
92
93    Entrees :
94      cell_t **liste : pointeur sur le pointeur du premier element de la liste chainee
95      cell_t *elt : pointeur sur l'element a inserer dans la liste chainee
96
97    Sortie :
98      !existe: indique si la valeur a ete ajoutee (1: ajoutee, 0: pas ajoutee)
99  */
100 int ins_cell(cell_t **liste, cell_t *elt) {
101   short int existe;
102   cell_t **prec = rech_prec(liste,elt->col,&existe);
103   if (!existe) {
104     adj_cell(prec,elt);
105   }
106   return (int) !existe;
107 }
108
109 /*  node_t * creer_cell(int col, int val)
110   Permet de creer un element de la liste chainee a partir du numero
111   de colonne et de la valeur
112
113   Entrees :
114     int col : numero de colonne
115     int val : valeur
116
117   Sortie :
118     node_t* : pointeur sur l'element cree
```

```c
*/
node_t * creer_cell(int col, int val) {
  node_t *elt = (node_t*) malloc(sizeof(node_t));
  if (elt) {
    elt->val = val;
    elt->col = col;
  }
  return elt;
}
```

2.2 Gestion de la matrice

Code C

```c
/*  mat.h
  Header

  -----| MATRICE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#ifndef __MAT_H__
#define __MAT_H__
  #include <stdio.h>
  #include <stdlib.h>
  #include "list.h"

  #define MAX_ROWS 50

  typedef struct _row_t {
    int row;      /* Numero de ligne */
    cell_t *cols; /* Liste chainee des colonnes */
  } row_t;

  typedef struct _mat_t {
    int nbrow;   /* Nombre de lignes */
    int nbcol;   /* Nombre de colonnes */
    row_t *rows; /* Lignes */
  } mat_t;

  int init_mat(mat_t *);
  int lire_matrice(char *, mat_t*);
  void afficher_matrice(mat_t *);
  int element(mat_t *, int, int);
  void liberer_matrice(mat_t *);


#endif
```

Code C

```c
/*  mat.c
  Fonctions de gestion de matrice

  -----| MATRICE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#include "mat.h"
int inser_val(mat_t *m, int row, int col, int val);

/*  int init_mat(mat_t *m)
  Initialise la matrice en allouant le tableau de structures correspondant aux
  lignes

  Entrees :
    *m : adresse de la structure de matrice a initialiser

  Sortie :
    res : booleen indiquant la reussite de l'initialisation (1: OK, 0: erreur)
*/
int init_mat(mat_t *m) {
  int res = 0;  /* Retour */
  m->nbrow = 0; /* Nb de lignes */
  m->nbcol = 0; /* Nb de colonnes */
  m->rows = (row_t*) malloc((MAX_ROWS+1)*sizeof(row_t));
  if (m->rows) {  /* Allocation reussie */
    m->rows[0].row = MAX_ROWS+1; /* Initialisation de la valeur max pour la dichotomie */
    res = 1;    /* Mission accomplie */
  }
  return res;
}

/*  int lire_matrice(char *fichier, mat_t *m)
  Lit un fichier représentant une matrice et inscrit le contenu dans une structure
  de matrice

  Entrees :
    *fichier : chaine de caracteres: nom du fichier
    *m       : adresse de la structure de matrice a ecrire

  Sortie :
    res : booleen indiquant la reussite de la lecture (1: OK, 0: erreur)
*/
int lire_matrice(char *fichier, mat_t *m) {
  FILE *f = fopen(fichier,"r");
  int row, col, val;  /* Valeurs qui seront lues */
  int res = 1;        /* Valeur de retour */
  int read;           /* Indique le nombre de valeurs lues */
  if (init_mat(m) && f) { /* Initialisation et ouverture flux OK */
     while (res && !feof(f)) {  /* Fichier non fini et pas d'erreur */
       read = 0;      /* Cpt de valeurs car fscanf n'aime pas les fins de fichier */
```

```c
      read = fscanf(f,"%d %d %d", &row, &col, &val); /* Lecture */
      if (read >= 3 && !inser_val(m,row,col,val)) {  /* Lecture ou insertion non reussies */
        res = 0; /* Erreur */
      }
    }
    fclose(f);       /* Fermeture du flux */
  } else {           /* Erreur */
    res = 0;
  }
  return res;
}

/*  int rech_dich(mat_t *m, int row, short int *existe)
  Realise une recherche dichotomique sur les lignes pour trouver le rang d'insertion.

  Entrees :
    *m      : adresse de la matrice
    row     : indice de la ligne a rechercher
    *existe : booleen de presence a modifier

  Sortie :
    deb : rang d'insertion ou de position de la ligne row
*/
int rech_dich(mat_t *m, int row, short int *existe) {
  int deb, fin, mil; /* Indices locaux */
  deb = 0;           /* Recherche entre le debut */
  fin = m->nbrow;    /* Et la fin */
  while (deb != fin) {    /* Tant qu'il reste plus d'un élément dans le 'sous-tableau' */
    mil = (deb + fin)/2;  /* Rang du milieu */
    if (row <= m->rows[mil].row) {  /* Comparaison avec la valeur centrale */
      fin = mil;      /* Inférieure */
    } else {
      deb = mil + 1;  /* Supérieure */
    }
  }
  *existe = (m->rows[deb].row == row); /* Test existence */
  return deb;
}

/*  void decal_rows(mat_t *m, int r)
  Decale les lignes d'une matrice en vue d'une insertion.

  Entrees :
    *m : adresse de la matrice
    r  : rang a partir duquel decaler

  Sortie :
    Aucune
*/
void decal_rows(mat_t *m, int r) {
  int i;  /* Indice de boucle */
  for (i = m->nbrow; i >= r; i-=1) {
    m->rows[i+1] = m->rows[i]; /* Recopie */
  }
}

```

```c
/*  void inser_row(mat_t *m, int r, int row)
  Insere une ligne row dans une matrice m en position r

  Entrees :
    *m : adresse de la matrice
    r  : rang d'insertion
    row: indice de la ligne a inserer

  Sortie :
    Aucune
*/
void inser_row(mat_t *m, int r, int row) {
  decal_rows(m,r);  /* Decalage vers la droite */
  m->nbrow++;       /* Ajout d'une ligne */
  m->rows[r].row = row;   /* Insertion de la ligne */
  m->rows[r].cols = NULL; /* Initialisation pointeur de colonnes */
}

/*  int inser_val(mat_t *m, int row, int col, int val)
  Insere une valeur val dans une matrice m en ligne row et colonne col

  Entrees :
    *m : adresse de la matrice
    row: numero de ligne
    col: numero de colonne
    val: valeur a inserer

  Sortie :
    res: booleen de retour (1: reussite, 0: echec)
*/
int inser_val(mat_t *m, int row, int col, int val) {
  short int existe; /* Booleen pour les tests d'existence */
  int res = 1;      /* Valeur de retour */
  cell_t *c;
  int r = rech_dich(m,row,&existe);  /* Recherche ligne d'insertion */
  if (!existe) {
    inser_row(m,r,row);    /* Ajout d'une ligne si necessaire */
  }
  c = creer_cell(col,val); /* Creation de la cellule*/
  if (c) {  /* Creation reussie */
    if (!ins_cell(&(m->rows[r].cols),c)) {  /* Ajout de la valeur */
      printf("Valeur non ajoutee: %d\n",col);
      res = 0;
    } else {
      if (col > m->nbcol) {
        m->nbcol = col;     /* Actualisation du nombre maximal de colonnes */
      }
    }
  } else { /* Erreur memoire */
    res = 0;
  }
  return res;
}

/*  int element(mat_t *m, int i, int j)
  Retourne M(i,j)
```

```
168
169   Entrees :
170     *m : adresse de la matrice
171     i  : indice de ligne
172     j  : indice de colonne
173
174   Sortie :
175     elt : valeur de M(i,j)
176 */
177 int element(mat_t *m, int i, int j) {
178   int elt = 0;    /* Valeur par defaut (matrice creuse!) */
179   cell_t **prec;  /* Pointeur de recherche */
180   int r;          /* Rang de la ligne dans la matrice creuse */
181   short int existe;  /* Booleen d'existence */
182
183   if (i > 0 && j > 0 && i <= m->rows[m->nbrow].row && j <= m->nbcol) { /* Element dans la matrice */
184     r = rech_dich(m,i,&existe); /* Recherche de la ligne */
185     if (existe) {   /* Ligne presente */
186       prec = rech_prec(&(m->rows[r].cols),j,&existe);   /* Recherche colonne */
187       if (existe) { /* Colonne présente */
188         elt = (*prec)->val; /* Copie valeur */
189       }
190     }
191   }
192   return elt;
193 }
194
195 /*  void afficher_matrice(mat_t *m)
196   Affiche la matrice m
197
198   Entrees :
199     *m : adresse de la matrice (evite de recopier toute
200          la structure majeure lors de l'appel)
201
202   Sortie :
203     Aucune
204 */
205 void afficher_matrice(mat_t *m) {
206   int i, j;           /* Ligne reelle / colonne reelle */
207   int ligne_cour = 0; /* Ligne dans la structure */
208   cell_t *col_cour;   /* Pointeur de parcours des colonnes de la structure */
209   int max_col = m->nbcol; /* Nombre de colonnes reelles */
210   int max_row = (m->nbrow > 0) ? m->rows[m->nbrow-1].row : 0; /* Nombre de lignes reelles */
211
212   for (i = 1; i <= max_row; ++i) {  /* Pour chaque ligne de la matrice */
213     j = 1;                          /* Depart sur la colonne 1 */
214     if (m->rows[ligne_cour].row == i) {    /* Ligne presente dans la structure */
215       col_cour = m->rows[ligne_cour].cols; /* Premiere colonne */
216       while (col_cour) {   /* Pour toutes les colonnes jusqu'a la fin de la liste */
217         if (col_cour->col == j) {       /* Colonne presente dans la structure */
218           printf("%d\t",col_cour->val);
219           col_cour = col_cour->next;    /* Colonne suivante (struct) */
220         } else {                        /* Colonne absente (i.e. 0) */
221           printf("0\t");
222         }
```

```
        j++;                            /* Colonne suivante (reelle) */
      }
      ligne_cour++;                     /* Ligne de la structure traitee */
    }
    while (j <= max_col) {              /* Complete la ligne courante */
      printf("0\t");
      j++;                              /* Colonne suivante (reelle) */
    }
    printf("\n");                       /* Ligne i traitee */
  }
}

/*  void liberer_matrice(mat_t *m)
  Libère la mémoire allouée pour la matrice m

  Entrees :
    *m : adresse de la matrice

  Sortie :
    Aucune
*/
void liberer_matrice(mat_t *m){
  int i;  /* Lignes */
  for (i = 0; i < m->nbrow; i++) { /* Suppression de chaque ligne */
    liberer_liste(&(m->rows[i].cols)); /* Suppression des colonnes */
  }
  free(m->rows);    /* Suppression du tableau de lignes */
  m->rows = NULL;
  m->nbrow = 0;     /* Actualisation lignes */
  m->nbcol = 0;     /* Actualisation colonnes */
}
```

2.3 Programme principal

Code C

```
/*  main.c
  Fonction principale du programme, pour les tests

  -----| MATRICE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup
  ISIMA 1ere Annee, 2014-2015
*/

#include "mat.h"
#define COLUMNS 81 /* Nombre max de caracteres par ligne + 1 */

int main(int argc, char *argv[]) {
  int i, j;
  mat_t m;
  FILE *f;
  char buf[COLUMNS];
  char text[COLUMNS];
```

```
  if (argc > 1) {
    /* Lecture de fichier de commandes */
    f = fopen(argv[1],"r");
    if (f) {
      while (!feof(f)) {
        buf[0] = '\0';
        text[0] = '\0';
        fgets(buf,COLUMNS,f);
        switch (buf[0]) {
          case 'R': /* creer */
            sscanf(&buf[1],"%s",text);
            lire_matrice(text,&m);
            break;
          case 'E': /* element */
            sscanf(&buf[1],"%d %d",&i, &j);
            printf("M(%d,%d)=%d\n",i,j,element(&m,i,j));
            break;
          case 'A': /* afficher */
            afficher_matrice(&m);
            break;
          case 'L': /* liberer */
            liberer_matrice(&m);
            break;
          case '#': /* texte */
            printf("%s",&buf[1]);
            break;
          default:
            puts("\n");
        }
      }
      fclose(f);
    } else {
      fprintf(stderr, "Fichier invalide\n");
    }
  } else {
    fprintf(stderr, "Merci de donner un fichier de commandes en argument\n");
  }
  return 0;
}
```

3 | Principes et lexiques des fonctions

Dans cette partie, sont décrits les algorithmes de principe associés aux fonctions écrites en langage C, ainsi qu'un lexique concernant les variables intermédiaires des fonctions.

Les lexiques des variables d'entrée, sortie et entrée/sortie sont disponibles dans le code source directement.

3.1 Gestion des listes chaînées

Les fonctions de gestion des listes chaînées peuvent être trouvées dans les fichiers `list.c` et `list.h`. Les algorithmes de principe de ces fonctions ont déjà été fournis dans le TP 1. Ils ne seront donc pas inclus ici.

3.2 Gestion de la matrice

La gestion de la matrice s'effectue avec les fonctions contenues dans `mat.c` et définies dans `mat.h`.

3.2.1 lire_matrice

```
Algorithme (Principe)
Début
    Ouverture du fichier en lecture;
    Initialisation du code d'erreur à 1;
    Initialisation de la matrice;
    Si le fichier est correctement ouvert Et la matrice est initialisée Alors
        TantQue code d'erreur = 1 Et on n'a pas lu tout le fichier Faire
            On récupère les valeurs de la ligne du fichier;
            On insère la valeur à la bonne place dans la matrice;
            Si l'insertion n'a pas réussie Alors
                Le code d'erreur passe à 0;
            FinSi;
        FinTantQue;
    Sinon
        Le code d'erreur passe à 0;
    FinSi;
    Retourner Code d'erreur;
Fin
```

Lexique :

- *f : *descripteur sur le fichier contenant les informations sur la matrice*
- row : *entier pour récupérer le numéro de la ligne dans le fichier*
- col : *entier pour récupérer le numéro de la colonne dans le fichier*
- val : *valeur à insérer dans la matrice*
- res : *code d'erreur (1 aucune erreur, 0 sinon)*

3.2.2 init_mat

```
Algorithme init_mat (Principe)
Début
    Initialisation du code d'erreur à 1;
    Initialisation du nombre de lignes de la matrice à 0;
    Allocation du tableau de lignes de la matrice;
    Si le tableau de lignes est alloué Alors
        On place une valeur de ligne maximale pour la première ligne;
        [ Ceci permet d'avoir un bon résultat lors de la première recherche dichotomique ]
        Le code d'erreur passe à 1;
    FinSi;
    Retourner Code d'erreur;
Fin
```

Lexique :

res : *code d'erreur (1 si aucun problème, 0 sinon)*

3.2.3 rech_dich

```
Algorithme rech_dich (Principe)
Début
    Initialisation d'un entier deb à 0;
    Initialisation d'un entier fin au nombre de lignes de la matrice;
    Si ligne cherchée < numéro de la dernière ligne de la matrice Alors
        TantQue deb ≠ fin Faire
            mil prend le rang milieu entre deb et fin;
            Si ligne cherchée < numéro de la ligne à la place mil dans la matrice Alors
                fin prend la valeur mil;   [ Recherche sur la première moitié ]
            Sinon
                deb prend mil + 1;   [ Recherche sur la seconde moitié ]
            FinSi;
        FinTantQue;
    FinSi;
    Modification du booléen existe;
    Retourner deb;
Fin
```

Lexique :

deb : *entier représentant la position de départ de la recherche dichotomique, c'est également le retour de la fonction indiquant où se trouve l'élément cherché ou son adresse d'insertion*

fin : *entier représentant la position de fin de la recherche dichotomique*

mil : *entier représentant le milieu entre deb et fin*

*existe : *adresse du booléen indiquant si la ligne cherchée se trouve ou non déjà dans la matrice (1 si elle existe, 0 sinon)*

3.2.4 decal_rows

Cette procédure décale simplement les éléments de la table principale d'une case vers la droite jusqu'à la case spécifiée en paramètre, en partant de la dernière case.

3.2.5 inser_row

Cette procédure va dans un premier temps effectuer un décalage des éléments de la table vers la droite. Ceci va permettre d'avoir une place pour la nouvelle ligne. On incrémente le nombre de lignes de la matrice, on insère le numéro de ligne dans la table principale et on place le pointeur vers la liste chaînée représentant les colonnes de cette ligne sur `NIL`.

3.2.6 inser_val

```
Algorithme inser_val (Principe)
Début
    Initialisation du code d'erreur à 1;
    Recherche la ligne d'insertion par dichotomie;
    Si la ligne n'existe pas Alors
        On l'insère avec la fonction inser_val;
    FinSi;
    On crée la nouvelle cellule;
    Si la cellule est correctement crée Alors
        On ajoute la cellule dans la liste chaînée des colonnes, sur la ligne correspondante;
        Si l'indice de la colonne ajoutée < maximum de colonne courant Alors
            On actualise le maximum de colonnes;
        FinSi;
    Sinon
        Le code d'erreur passe à 0;
    FinSi;
    Retourner Code d'erreur;
Fin
```

Lexique :

- existe : *booléen indiquant si la ligne cherchée existe déjà ou non*
- res : *code d'erreur (1 si aucun problème, 0 sinon)*
- *c : *cellule créée, qui sera insérée dans la matrice*

3.2.7 element

```
Algorithme element (Principe)
Début
    Initialisation de la valeur de l'élément à 0;
    Si les indices sont dans la matrice Alors
        Recherche par dichotomie de la ligne;
        Si la ligne cherchée existe Alors
            Recherche de la colonne dans la liste chaînée des colonnes;
            Si la colonne existe Alors
                La valeur de l'élément prend la valeur de la cellule qui est trouvée;
            FinSi;
        FinSi;
    FinSi;
    Retourner valeur de l'élément;
Fin
```

Lexique :

- elt : *valeur de l'élément qui sera renvoyée*
- **prec : *pointeur sur l'élément trouvé dans la recherche de la colonne*
- r : *entier indiquant la position de la ligne cherchée dans la table majeure*
- existe : *booléen indiquant si la ligne cherchée existe ou non*

3.2.8 afficher_matrice

```
Algorithme afficher_matrice (Principe)
Début
    Initialise la ligne courante ligne_cour à 0;
    Pour chaque ligne de la matrice Faire
        On se place sur la première colonne;
        Si la ligne courante est présente dans la matrice Alors
            On stocke l'adresse de la première colonne non-nulle de la ligne courante;
            TantQue la liste chainée n'est pas terminée Faire
                Si la colonne courante est non-nulle Alors
                    Afficher la valeur de la cellule;
                    Stocker la prochaine colonne non-nulle;
                Sinon
                    Afficher 0;
                FinSi;
                Passer sur la colonne suivante;
            FinTantQue;
            Passer à la ligne suivante;
        FinSi;
        TantQue la ligne courante n'est pas complète Faire
            Ajouter un 0;
            Passer à la colonne suivante;
        FinTantQue;
        Afficher un retour à la ligne;   [ La ligne est complète ]
    FinPour;
Fin
```

Lexique :

- i : *indice de ligne*
- j : *indice de colonne*
- ligne_cour : *indice de la ligne courante, existante dans la matrice*
- *col_cour : *pointeur parcourant les cellules des colonnes des lignes de la matrice*
- max_col : *nombre de colonnes maximales dans la matrice*
- max_row : *nombre de lignes maximales dans la matrice*

3.2.9 liberer_matrice

Cette fonction permet de libérer la mémoire occupée par la matrice. On itère sur chaque ligne de la table majeure en libérant chaque liste avec la fonction *liberer_liste* (permettant de libérer une liste chaînée). Une fois les listes supprimées, on libère le tableau de la table majeure et on indique que la nombre maximal de lignes et de colonnes et de 0.

4 | Compte rendu d'exécution

4.1 Makefile

Makefile

```
#Compilateur et options de compilation
CC=gcc
CFLAGS=-Wall -ansi -pedantic -Wextra -g

#Fichiers du projet
SOURCES=main.c mat.c list.c
OBJECTS=$(SOURCES:.c=.o)

EXEC=prog

$(EXEC): $(OBJECTS)
	$(CC) $(CFLAGS) $^ -o $(EXEC)

.c.o:
	$(CC) -c $(CFLAGS) $*.c

clean:
	rm -rf $(OBJECTS) $(EXEC)
```

4.2 Jeux de tests

4.2.1 Matrice nulle

Fichier d'entrée (le fichier de matrice nulle est vide) :

test_nulle

```
#TEST MATRICE NULLE
R matrice_nulle
A
E 20 20
L
```

Résultat :

Résultat matrice nulle

```
TEST MATRICE NULLE
M(20,20)=0
```

4.2.2 Matrice quelconque

Fichier d'entrée :

test_1

```
#TEST MATRICE 1
#Lecture desordonnee sur les lignes et les colonnes
R matrice_1
#Affichage
A
#Element nul
E 2 3
#Element non-nul
E 12 5
#Liberation
L
```

Fichier matrice :

matrice_1

```
10 1 99
12 13 5
12 5 7
14 9 15
1 2 42
12 3 4
12 12 5
10 12 12
```

Résultat :

Résultat matrice quelconque

```
TEST MATRICE 1
Lecture desordonnee sur les lignes et les colonnes
Affichage
0    42   0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0
99   0    0    0    0    0    0    0    0    0    0    12   0
0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    4    0    7    0    0    0    0    0    0    5    5
0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    15   0    0    0    0
Element nul
M(2,3)=0
Element non-nul
M(12,5)=7
Liberation
```

4.2.3 Matrice ligne

Fichier d'entrée :

test_2

```
#TEST MATRICE 2
#Lecture matrice ligne
R matrice_2
#Affichage
A
#Element nul
E 1 1
#Element non-nul
E 1 7
#Liberation
L
```

Fichier matrice :

matrice_2

```
1 7 9
1 4 -6
1 50 -2
1 14 8
```

Résultat :

Résultat matrice ligne

```
TEST MATRICE 2
Lecture matrice ligne
Affichage
0  0  0  -6  0  0  9  0  0  0  0  0  0  8  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 ↪  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  -2
Element nul
M(1,1)=0
Element non-nul
M(1,7)=9
Liberation
```

4.2.4 Matrice très creuse (un seul élément non-nul)

Fichier d'entrée :

test_3

```
#TEST MATRICE 3
#Lecture matrice a 1 element non nul.
R matrice_3
#Affichage
A
#Element nul
E 2 3
#Element non-nul
E 20 24
#Liberation
L
```

Fichier matrice :

matrice_3

```
20 24 36
```

Résultat :

Résultat matrice très creuse

```
TEST MATRICE 3
Lecture matrice a 1 element non nul.
Affichage
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  36
Element nul
M(2,3)=0
Element non-nul
M(20,24)=36
Liberation
```

4.2.5 Bonne utilisation de la mémoire

Pour vérifier la bonne libération de la mémoire, nous avons utilisé `valgrind` avec le programme et un fichier de test. Aucun bloc de mémoire n'est perdu, le retour texte de valgrind est présenté en figure 4.1.

Resultat Valgrind

```
==17199== Memcheck, a memory error detector
==17199== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17199== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==17199== Command: ../src/prog test_1
==17199==
==17199==
==17199== HEAP SUMMARY:
==17199==     in use at exit: 0 bytes in 0 blocks
==17199==   total heap usage: 11 allocs, 11 frees, 2,080 bytes allocated
==17199==
==17199== All heap blocks were freed -- no leaks are possible
==17199==
==17199== For counts of detected and suppressed errors, rerun with: -v
==17199== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

FIGURE 4.1 – Passage dans l'outil `valgrind`

Notons que l'utilisation de cette structure de données est en effet intéressante pour les matrices creuses :

Par exemple pour le test de la matrice très creuse (page 23), `valgrind` nous indique que le programme a utilisé au total 1968 octets. Le seul stockage de cette matrice sous forme de tableau aurait nécessité $4 \times 20 \times 24 = 1920$ octets.

L'inconvénient de cette structure est que la table majeure occupe 8 à 12 octets par bloc de ligne, et que le nombre de ces lignes est fixé (ici 50+1, soit environ 400 octets mobilisés au minimum), on peut donc estimer qu'il faut près d'une centaine d'éléments nuls au minimum pour rendre cette structure de données intéressante du point de vue de la mémoire.

On pourrait éventuellement proposer d'autres tailles pour la table majeure, ou bien une longueur dynamique (mais au détriment des performances lorsqu'il s'agira de redimensionner l'espace, i.e. lors de l'insertion).