

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



# Table des matières

<b>1</b>	<b>Présentation</b>	<b>2</b>
1.1	Structures de données employées	2
1.1.1	Matrice	2
1.1.2	Fichier de commandes	3
1.2	Organisation du code source	4
1.2.1	Gestion des listes chaînées	4
1.2.2	Gestion de la matrice	4
1.2.3	Programme principal	4
<b>2</b>	<b>Détails du programme</b>	<b>5</b>
2.1	Gestion des listes chaînées	5
2.2	Gestion de la matrice	9
2.3	Programme principal	15
<b>3</b>	<b>Principes et lexiques des fonctions</b>	<b>17</b>
3.1	Gestion des listes chaînées	17
3.2	Gestion de la matrice	17
3.2.1	lire_matrice	18
3.2.2	init_mat	18
3.2.3	rech_dich	19
3.2.4	decal_rows	20
3.2.5	inser_row	20
3.2.6	inser_val	20
3.2.7	element	21
3.2.8	afficher_matrice	22
3.2.9	liberer_matrice	22
<b>4</b>	<b>Compte rendu d'exécution</b>	<b>23</b>
4.1	Makefile	23
4.2	Jeux de tests	23

# 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 représentant 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`

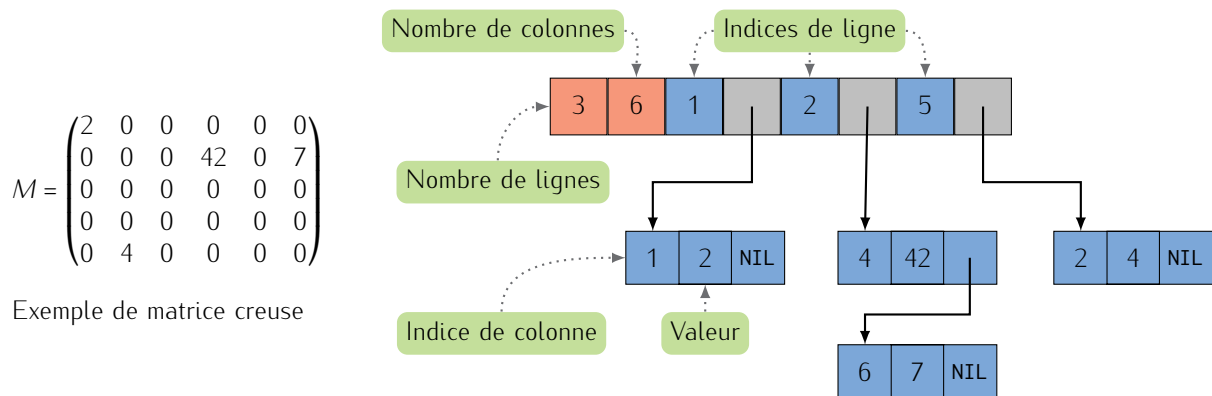


FIGURE 1.1 – Structure de matrice creuse

---

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 une fonction basique effectuant des tests sommaires, ainsi que l'interprétation d'un fichier qui peut être donné comme premier paramètre.

Le cas échéant, 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 après 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

```
1  /* list.h
2  Header
3
4  -----| LISTE CHAINEE |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11
12 #ifndef __LISTE_H__
13 #define __LISTE_H__
14
15 #include <string.h>
16 #include <ctype.h>
17
18 typedef struct _node_t {
19     int col;          /* Numero de colonne */
20     int val;          /* Valeur */
21     struct _node_t *next; /* Colonne suivante */
22 } node_t;
23
24 typedef node_t cell_t;
25
26 cell_t ** rech_prec(cell_t **, int, short int*);
27 void supp_cell(cell_t **);
28 void liberer_liste(cell_t **);
29 int ins_cell(cell_t **, cell_t *);
30 cell_t * creer_cell(int, int);
31
32 #endif
```

Code C

```
1  /* list.c
2  Fonctions de gestion de la liste chainee
3
4  -----| LISTE CHAINEE |-----
5
6  BARBESANGE Benjamin,
```

```

7      PISSAVY Pierre-Loup
8
9      ISIMA 1ere Annee, 2014-2015
10     */
11
12     #include <stdio.h>
13     #include <stdlib.h>
14     #include "list.h"
15
16     /* void adj_cell(cell_t **prec, cell_t *elt)
17      Ajoute une cellule apres un element partir d'un pointeur sur l'element
18      et d'un pointeur sur le pointeur de l'element apres lequel ajouter
19
20      Entrees :
21          cell_t **prec : pointeur sur le pointeur de l'element apres lequel ajouter
22          cell_t *elt : pointeur sur l'element a ajouter a la liste chainee
23
24      Sortie :
25          Aucune
26     */
27     void adj_cell(cell_t **prec, cell_t *elt) {
28         elt->next = (*prec);
29         (*prec) = elt;
30     }
31
32     /* cell_t ** rech_prec(cell_t **liste, int col, short int *existe)
33      Recherche le precedent d'un element dans la liste chainee trie selon les colonnes
34
35      Entrees :
36          cell_t **liste : pointeur sur le pointeur du premier element de la liste chainee
37          int col : indice de colonne
38          short int *existe : variable en entree/sortie indiquant si la colonne est presente ou non
39              0 : absence
40              1 : presence
41
42      Sortie :
43          cell_t ** : pointeur sur le pointeur de l'element precedent
44     */
45     cell_t ** rech_prec(cell_t **liste, int col, short int *existe) {
46         cell_t **prec = liste;
47         while ((*prec) && (*prec)->col < col) {
48             prec = &((*prec)->next);
49         }
50         /* Booleen de presence */
51         /* 1 : present */
52         /* 0 : absent */
53         *existe = (*prec && (*prec)->col == col)?1:0;
54         return prec;
55     }
56
57     /* void supp_cell(cell_t **prec)
58      Permet de supprimer un element dans la liste chainee a partir
59      de son precedent
60
61      Entrees :
62          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 /* void ins_cell(cell_t **liste, cell_t *elt)
90     Permet d'insérer une cellule a la bonne place dans la liste chainee
91     Les colonnes sont trieées 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 insérer dans la liste chainee
96
97     Sortie :
98     Aucune
99 */
100 int ins_cell(cell_t **liste, cell_t *elt) {
101     short int existe;
102     cell_t **prec = rech_prec(liste, elt->val, &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 créer 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

```



```
119 */
120 node_t * creer_cell(int col, int val) {
121     node_t *elt = (node_t*) malloc(sizeof(node_t));
122     if (elt) {
123         elt->val = val;
124         elt->col = col;
125     }
126     return elt;
127 }
```

## 2.2 Gestion de la matrice

Code C

```
1  #ifndef __MAT_H__
2  #define __MAT_H__
3      #include <stdio.h>
4      #include <stdlib.h>
5      #include "list.h"
6
7      #define MAX_ROWS 50
8
9      typedef struct _row_t {
10         int row; /* Numero de ligne */
11         cell_t *cols; /* Liste chaine des colonnes */
12     } row_t;
13
14     typedef struct _mat_t {
15         int nbrow; /* Nombre de lignes */
16         int nbcol; /* Nombre de colonnes */
17         row_t *rows; /* Lignes */
18     } mat_t;
19
20     int init_mat(mat_t *);
21     int lire_matrice(char *, mat_t*);
22     void afficher_matrice(mat_t *);
23     int element(mat_t *, int, int);
24     void liberer_matrice(mat_t *);
25
26
27 #endif
```

Code C

```
1  /* mat.c
2     Fonctions de gestion de matrice
3
4     -----| MATRICE |-----
5
6     BARBESANGE Benjamin,
7     PISSAVY Pierre-Loup
8
9     ISIMA 1ere Annee, 2014-2015
10 */
11
12 #include "mat.h"
13 int inser_val(mat_t *m, int row, int col, int val);
14
15 /* int init_mat(mat_t *m)
16    Initialise la matrice en allouant le tableau de structures correspondant aux
17    lignes
18
19    Entrees :
20        *m : adresse de la structure de matrice a initialiser
21
22    Sortie :
23        res : boolean indiquant la reussite de l'initialisation (1: OK, 0: erreur)
24 */
```

```

25 int init_mat(mat_t *m) {
26     int res = 0; /* Retour */
27     m->nbrow = 0; /* Nb de lignes */
28     m->nbcou = 0; /* Nb de colonnes */
29     m->rows = (row_t*) malloc((MAX_ROWS+1)*sizeof(row_t));
30     if (m->rows) { /* Allocation reussie */
31         m->rows[0].row = MAX_ROWS+1; /* Initialisation de la valeur max pour la dichotomie */
32         res = 1; /* Mission accomplie */
33     }
34     return res;
35 }
36
37 /* int lire_matrice(char *fichier, mat_t *m)
38  Lit un fichier representant une matrice et inscrit le contenu dans une structure
39  de matrice
40
41  Entrees :
42      *fichier : chaine de caracteres: nom du fichier
43      *m       : adresse de la structure de matrice a ecrire
44
45  Sortie :
46      res : booleen indiquant la reussite de la lecture (1: OK, 0: erreur)
47  */
48 int lire_matrice(char *fichier, mat_t *m) {
49     FILE *f = fopen(fichier, "r");
50     int row, col, val; /* Valeurs qui seront lues */
51     int res = 1; /* Valeur de retour */
52     int read; /* Indique le nombre de valeurs lues */
53     if (init_mat(m) && f) { /* Initialisation et ouverture flux OK */
54         while (res && !feof(f)) { /* Fichier non fini et pas d'erreur */
55             read = 0; /* Cpt de valeurs car fscanf n'aime pas les fins de fichier */
56             read = fscanf(f, "%d %d %d", &row, &col, &val); /* Lecture */
57             if (read >= 3 && !inser_val(m, row, col, val)) { /* Lecture ou insertion non reussies */
58                 res = 0; /* Erreur */
59             }
60         }
61         fclose(f); /* Fermeture du flux */
62     } else { /* Erreur */
63         res = 0;
64     }
65     return res;
66 }
67
68 /* int rech_dich(mat_t *m, int row, short int *existe)
69  Realise une recherche dichotomique sur les lignes pour trouver le rang d'insertion.
70
71  Entrees :
72      *m       : adresse de la matrice
73      row      : indice de la ligne a rechercher
74      *existe  : booleen de presence a modifier
75
76  Sortie :
77      deb : rang d'insertion ou de position de la ligne row
78  */
79 int rech_dich(mat_t *m, int row, short int *existe) {
80     int deb, fin, mil; /* Indices locaux */

```

```

81     deb = 0;          /* Recherche entre le debut */
82     fin = m->nbrow;    /* Et la fin */
83     while (deb != fin) { /* Tant qu'il reste plus d'un élément dans le 'sous-tableau' */
84         mil = (deb + fin)/2; /* Rang du milieu */
85         if (row <= m->rows[mil].row) { /* Comparaison avec la valeur centrale */
86             fin = mil; /* Inférieure */
87         } else {
88             deb = mil + 1; /* Supérieure */
89         }
90     }
91     *existe = (m->rows[deb].row == row); /* Test existence */
92     return deb;
93 }
94
95 /* void decal_rows(mat_t *m, int r)
96    Decale les lignes d'une matrice en vue d'une insertion.
97
98    Entrees :
99        *m : adresse de la matrice
100        r  : rang a partir duquel decaler
101
102    Sortie :
103        Aucune
104 */
105 void decal_rows(mat_t *m, int r) {
106     int i; /* Indice de boucle */
107     for (i = m->nbrow; i >= r; i--) {
108         m->rows[i+1] = m->rows[i]; /* Recopie */
109     }
110 }
111
112 /* void inser_row(mat_t *m, int r, int row)
113    Insere une ligne row dans une matrice m en position r
114
115    Entrees :
116        *m : adresse de la matrice
117        r  : rang d'insertion
118        row: indice de la ligne a inserer
119
120    Sortie :
121        Aucune
122 */
123 void inser_row(mat_t *m, int r, int row) {
124     decal_rows(m,r); /* Decalage vers la droite */
125     m->nbrow++; /* Ajout d'une ligne */
126     m->rows[r].row = row; /* Insertion de la ligne */
127     m->rows[r].cols = NULL; /* Initialisation pointeur de colonnes */
128 }
129
130 /* int inser_val(mat_t *m, int row, int col, int val)
131    Insere une valeur val dans une matrice m en ligne row et colonne col
132
133    Entrees :
134        *m : adresse de la matrice
135        row: numero de ligne
136        col: numero de colonne

```

```

137     val: valeur a inserer
138
139     Sortie :
140     res: booléen de retour (1: réussite, 0: échec)
141 */
142 int inser_val(mat_t *m, int row, int col, int val) {
143     short int existe; /* Booléen pour les tests d'existence */
144     int res = 1;      /* Valeur de retour */
145     cell_t *c;
146     int r = rech_dich(m,row,&existe); /* Recherche ligne d'insertion */
147     if (!existe) {
148         inser_row(m,r,row); /* Ajout d'une ligne si nécessaire */
149     }
150     c = creer_cell(col,val); /* Creation de la cellule*/
151     if (c) { /* Creation réussie */
152         ins_cell (&(m->rows[r].cols),c); /* Ajout de la valeur */
153         if (col > m->nbcoll) {
154             m->nbcoll = col; /* Actualisation du nombre maximal de colonnes */
155         }
156     } else { /* Erreur memoire */
157         res = 0;
158     }
159     return res;
160 }
161
162 /* int element(mat_t *m, int i, int j)
163     Retourne M(i,j)
164
165     Entrees :
166     *m : adresse de la matrice
167     i  : indice de ligne
168     j  : indice de colonne
169
170     Sortie :
171     elt : valeur de M(i,j)
172 */
173 int element(mat_t *m, int i, int j) {
174     int elt = 0; /* Valeur par défaut (matrice creuse!) */
175     cell_t **prec; /* Pointeur de recherche */
176     int r; /* Rang de la ligne dans la matrice creuse */
177     short int existe; /* Booléen d'existence */
178
179     if (i < m->rows[m->nbrow].row && j < m->nbcoll) { /* Element dans la matrice */
180         r = rech_dich(m,i,&existe); /* Recherche de la ligne */
181         if (existe) { /* Ligne présente */
182             prec = rech_prec(&(m->rows[r].cols),j,&existe); /* Recherche colonne */
183             if (existe) { /* Colonne présente */
184                 elt = (*prec)->val; /* Copie valeur */
185             }
186         }
187     }
188     return elt;
189 }
190
191 /* void afficher_matrice(mat_t *m)
192     Affiche la matrice m

```

```

193
194 Entrees :
195     *m : adresse de la matrice (evite de recopier toute
196         la structure majeure lors de l'appel)
197
198 Sortie :
199     Aucune
200 */
201 void afficher_matrice(mat_t *m) {
202     int i, j;          /* Ligne reellev / colonne reellev */
203     int ligne_cour = 0; /* Ligne dans la structure */
204     cell_t *col_cour;   /* Pointeur de parcours des colonnes de la structure */
205     int max_col = m->nbcoll; /* Nombre de colonnes reellev */
206     int max_row = m->rows[m->nbrow-1].row; /* Nombre de lignes reellev */
207
208     for (i = 1; i <= max_row; ++i) { /* Pour chaque ligne de la matrice */
209         j = 1;                      /* Depart sur la colonne 1 */
210         if (m->rows[ligne_cour].row == i) { /* Ligne presente dans la structure */
211             col_cour = m->rows[ligne_cour].cols; /* Premiere colonne */
212             while (col_cour && j <= max_col) { /* Pour toutes les colonnes jusqu'a la fin de la
213                 ↪ liste */
214                 if (col_cour->col == j) { /* Colonne presente dans la structure */
215                     printf("%d\t", col_cour->val);
216                     col_cour = col_cour->next; /* Colonne suivante (struct) */
217                 } else { /* Colonne absente (i.e. 0) */
218                     printf("0\t");
219                 }
220                 j++; /* Colonne suivante (reellev) */
221             }
222             ligne_cour++; /* Ligne de la structure traitee */
223         }
224         while (j <= max_col) { /* Complete la ligne courante */
225             printf("0\t");
226             j++; /* Colonne suivante (reellev) */
227         }
228         printf("\n"); /* Ligne i traitee */
229     }
230 }
231
232 /* void liberer_matrice(mat_t *m)
233     Libere la memoire allouee pour la matrice m
234
235 Entrees :
236     *m : adresse de la matrice
237
238 Sortie :
239     Aucune
240 */
241 void liberer_matrice(mat_t *m){
242     int i; /* Lignes */
243     for (i = 0; i < m->nbrow; i++) { /* Suppression de chaque ligne */
244         liberer_liste(&(m->rows[i].cols)); /* Suppression des colonnes */
245     }
246     free(m->rows); /* Suppression du tableau de lignes */
247     m->rows = NULL;
248     m->nbrow = 0; /* Actualisation lignes */

```

```
248 | m->nbc1 = 0;    /* Actualisation colonnes */  
249 | }
```

## 2.3 Programme principal

Code C

```
1  /* main.c
2  Fonction principale du programme, pour les tests
3
4  -----| ARBRES |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11
12 #include "mat.h"
13 #define COLUMNS 81 /* Nombre max de caracteres par ligne + 1 */
14
15 int main(int argc, char *argv[]) {
16     int i, j;
17     mat_t m;
18     FILE *f;
19     char buf[COLUMNS];
20     char text[COLUMNS];
21     if (argc > 1) {
22         /* Lecture de fichier de commandes */
23         f = fopen(argv[1], "r");
24         if (f) {
25             while (!feof(f)) {
26                 buf[0] = '\0';
27                 text[0] = '\0';
28                 fgets(buf, COLUMNS, f);
29                 switch (buf[0]) {
30                     case 'R': /* creer */
31                         sscanf(&buf[1], "%s", text);
32                         lire_matrice(text, &m);
33                         break;
34                     case 'E': /* element */
35                         sscanf(&buf[1], "%d %d", &i, &j);
36                         element(&m, i, j);
37                         break;
38                     case 'A': /* afficher */
39                         afficher_matrice(&m);
40                         break;
41                     case 'L': /* liberer */
42                         liberer_matrice(&m);
43                         break;
44                     case '#': /* texte */
45                         printf("%s", &buf[1]);
46                         break;
47                     default:
48                         puts("\n");
49                 }
50             }
51             fclose(f);
52         } else {
53             fprintf(stderr, "Fichier invalide\n");
```



```
54     }  
55 } else {  
56     fprintf(stderr, "Merci de donner un fichier de commandes en argument\n");  
57 }  
58 return 0;  
59 }
```

## 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 seule;
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 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;
Initialise le nombre de lignes de la matrice à 0;
Allocation du tableau de lignes de la matrice;
Si le tableau de ligne 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 la valeur (deb + fin) / 2;

| | Si ligne cherchée < numéro de la ligne à la place mil dans la matrice Alors

| | | fin prend la valeur mil;

| | Sinon

| | | deb prend mil + 1;

| | 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 NULL.

### 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 créée 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 pour chaque ligne de la matrice Faire Faire
    L'indice j de colonne est à 1;
    Si la ligne courante est présente dans la matrice Alors
        Stocke l'adresse de la première colonne col_cour de la ligne courante;
        TantQue col_cour ≠ NULL Et j ≤ nombre maximum de colonnes Faire
            Si col_cour = j Alors
                Affiche la valeur de la cellule;
                col_cour passe sur la cellule suivante;
            Sinon
                Affiche 0;
            FinSi;
            Incrémente j;
        FinTantQue;
        Incrémentation de la ligne;
    FinSi;
    TantQue on est pas au bout de la ligne courante Faire
        On complète avec des 0;
        Incrémente j;
    FinTantQue;
    Afficher un retour à la ligne;
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 fait une boucle for sur chaque élément de la table majeure et dans cette boucle on libère chaque listes avec la fonction *liberer\_liste* (permettant de libérer une liste chaînée). Une fois sortis de la boucle, on libère le tableau de la table majeure et on indique que le nombre maximal de lignes et de colonnes est de 0.

## 4 | Compte rendu d'exécution

### 4.1 Makefile

```
1  #Compilateur et options de compilation
2  CC=gcc
3  CFLAGS=-Wall -ansi -pedantic -Wextra -g
4
5  #Fichiers du projet
6  SOURCES=main.c mat.c list.c
7  OBJECTS=$(SOURCES:.c=.o)
8
9  EXEC=prog
10
11 $(EXEC): $(OBJECTS)
12     $(CC) $(CFLAGS) $^ -o $(EXEC)
13
14 .c.o:
15     $(CC) -c $(CFLAGS) $.c
16
17 clean:
18     rm -rf $(OBJECTS) $(EXEC)
```

### 4.2 Jeux de tests