ANGELI Nicolas

1. *Représentation d'une matrice creuse*

Dans cette partie, on veut représenter une matrice sous forme d'un tableau de listes chaînées, de sorte que la i.ème liste chaînée contienne les éléments non nuls de la ligne i de la matrice.

Pour ce faire définissons d'abord les structures Element et Liste.

```
struct Element{
      int colonne; //servira pour afficher M(ligne,colonne)
      int nombre;
      Element *suivant;
};

struct Liste {
        Element *premier;
};
```

On crée ensuite une fonction initialisation

```
Liste *initialisation(){
        Liste *liste = malloc(sizeof(*liste));
        if(liste == NULL ) exit(EXIT_FAILURE);
        liste->premier = NULL;
        return liste;
}//crée une liste dont le premier élément est nul
```

Une fonction insertion

```
/*On crée un élément correspondant au nouveau nombre, on décale vers
al droite le premier élément de la liste puis on met le nouvel
élément en tête de liste. Ceci fonctionne car chaque élément possède
```

```
un pointeur vers le suivant et donc en décalant l'ancien premier
élément on décale aussi tout les éléments qui le suivait*/
void insertion(Liste *liste, int nb){
        Element *nouveau = malloc(sizeof(Element));
        if(liste == NULL || nouveau == NULL) exit(EXIT_FAILURE);
        nouveau->nombre = nb;
        nouveau->suivant = liste->premier;
        liste->premier = nouveau;
}
```

On crée une fonction pour afficher les listes :

```
/*L'élément actuel correspond initialement au premier élément de la
liste puis tant que l'élément actuel est non nul on l'affiche et le
remplace par l'élément qui suit*/
void afficherListe(Liste *liste){
        if(liste == NULL) exit(EXIT_FAILURE);
        Element *actuel = liste->premier;
        while(actuel != NULL){
                printf("%d, %d -> ", actuel->nombre, actuel-
>colonne);
                actuel = actuel->suivant;
        }
        printf("NULL\n");
}
```

Et enfin la fonction produisant le tableau de liste désiré :

```
/*La fonction prendra en paramètre le tableau 2d correspondant à la
matrice ainsi que son nombre de lignes et de colonnes.
*/
Liste** matCreuse(int *Matrice, int n, int m){
        int i, j;

        Liste **Tableau = malloc(n * sizeof(Liste*));

        for(i = 0; i < n; i++){
                Liste *L = initialisation();//On crée la liste
                for (j = m-1; j >= 0; j--){
//On instancie la liste à l'envers car l'insertion rajoute les
//éléments en début de liste. Les indices j serviront à pour
//afficher les M(ligne,colonne)
                        if (*((Matrice + i*m) + j) != 0) {
//Matrice est l'adresse du premier élément du tableau +i*m pour
//pointer vers le i-ème élément du tableau +j pour la j-ième colonne
//En effet le 3ème élément (colonne) du deuxième tableau (ligne)
//représenterai le (2m+3)-ième élément du tableau 2d
                                insertion(L, *((Matrice + i*m) + j));
```

```
                        L->premier->colonne = j;
                    }
                }
                Tableau[i] = L;
                printf("\n");
                afficherListe(Tableau[i]);
                printf("\n");
        }
        return Tableau;
}
```

On nous demande ensuite de coder une fonction pour afficher un élément de la matrice.

```
/*Prend comme paramètres la matrice et les coordonnées de l'élément
à afficher
*/
void afficheMij(Liste *T[], int k, int l) {
      if (T == NULL) {
            printf("0\n");
            exit(EXIT_FAILURE);
      }

      Liste *liste = T[k];
//On cherchera l'élément dans la k-ième liste
      Element *actuel = liste->premier;

      while (actuel != NULL && actuel->colonne != l) actuel = actuel-
>suivant;
//On trouve le l-ième élément de la k-ième liste
      if (actuel == NULL) printf("M(%d,%d)=0\n", k, l);
//S'il est NULL on affiche 0
      else printf("M(%d,%d)=%d\n", k, l, actuel->nombre);
}
```

2. *Manipulation d'une liste chaînée*

Commençons par créer une fonction taille :

```
/*i commence à 1 et la boucle for au premier élément de la liste
puis tant que l'élément suivant n'est pas nul, on passe à l'élément
suivant et on incrémente i*/
int taille(Liste *liste){
      int i=1;
      Element* e;
      for (e=liste->premier ; e->suivant != NULL ; e = e->suivant){
            i++;
      }
```

```
	return i;
}
```

On peut maintenant créer la fonction insertion en évitant les dépassements de capacité

```
/*Si la position est inférieure ou égale à zéro on crée l'élément
correspondant au nombre fourni en paramètre puis on le met en tête
de liste
Si la position est supérieure à la taille on la réajuste à
taille pour mettre l'élément en fin de liste
On parcourt la liste jusqu'à arriver à l'élément précédant la
position indiquée (où on veut insérer notre nombre).
Puis on copie le pointeur vers l'élément suivant (qui suivra ensuite
l'élément inséré) dans le nouvel élément que nous positionnons à la
suite de l'élément situé en pos-1.*/
void insertion(Liste *liste, int nb, int pos){
	Element *element = malloc(sizeof(Element));
	element->nombre = nb;
	int t = taille(liste);

	if (liste == NULL || element == NULL)
		exit(EXIT_FAILURE);

	if(pos<=0){
		Element *nouveau = malloc(sizeof(Element));
		if(liste == NULL || nouveau == NULL) exit(EXIT_FAILURE);
		nouveau->nombre = nb;
		nouveau->suivant = liste->premier;
		liste->premier = nouveau;

		return;
	}

	if(pos>=t){
		pos=t;
	}

	int i=0;

	Element *e = liste->premier;
	while (i < pos - 1) {
		e = e->suivant;
		i++;
	}
	element->suivant=e->suivant;
	e->suivant=element;
}
```

On crée maintenant une fonction qui calcule les occurrences d'un nombre ainsi qu'une fonction qui trouve et supprime la première occurrence du nombre :

```
int occurrences(Liste *liste, int nb){
	int i=0;
	Element* e;
	for (e=liste->premier ; e->suivant != NULL ; e = e->suivant){
		if(e->nombre == nb)  i++;
	}
	if(e->nombre == nb)   i++;
//pour le cas où le dernier élément serait une occurrence
	return i;
}



/*On crée une fonction qui supprime le premier élément dont le
nombre correspond à celui passé en paramètre*/
void del(Liste *liste, int nb){
	if(liste == NULL)
		exit(EXIT_FAILURE);
	else{
		Element* e; // élément qui parcourt
		e=liste->premier;
		while (e->suivant->nombre != nb){
			e = e->suivant;
		}

		Element *aSupprimer = e->suivant;

		e->suivant = aSupprimer->suivant;
		free(aSupprimer);
	}
}
```

Plus qu'à supprimer la première occurrence le nombre d'occurrence fois

```
void suppression(Liste *liste, int nb){
	int occ = occurrences(liste, nb);
	for(int i = 0; i<occ; i++) del(liste,nb);
}


/*On procède par échange en stockant 3 éléments dans des variables.
Avant que l'élément suivant ne devienne l'actuel l'élément qui le
précédait lui est affecté et l'élément qui le précédait a été
assigné à l'élément actuel.*/
void inversion(Liste *liste){
	if(liste == NULL)
```

```
            exit(EXIT_FAILURE);

      Element *precedent, *actuel, *suivant;

      precedent = NULL;
      actuel = liste->premier;
      while (actuel != NULL){
            suivant = actuel->suivant;
            actuel->suivant = precedent;
            precedent = actuel;
            actuel = suivant;
      }
      liste->premier = precedent;

}
```