TP 8 : Allocation dynamique de mémoire

Objectif :

- Maîtriser l'allocation dynamique de mémoire et son utilisation

LES FONCTIONS D'ALLOCATION DYNAMIQUE DE MEMOIRE 1
1. NOTIONS DE BASE 1
EXERCICE 1
EXERCICE 1 : LISTE CHAINEE AVEC ALLOCATION DYNAMIQUE DE MEMOIRE 1
EXERCICE 2 : MATRICE CREUSE 3

Les fonctions d'allocation dynamique de mémoire

1. Notions de base

Le détail des fonctions utilisées dans cette partie (*malloc*, *free*) est donné dans la section 6.11 du document « Introduction au langage C ».

Exercice

Exercice 1 : Liste chaînée avec allocation dynamique de mémoire

Dans cet exercice nous réalisons les fonctions de gestion d'une liste chaînée de manière similaire au TP6, mais en utilisant l'allocation dynamique de mémoire (au lieu d'un tableau). Nous considérons les déclarations de type suivantes :

```
typedef struct element element ;
struct element {
      int valeur ;                  /* valeur de l'élément */
      element* suivant ;            /* adresse du successeur */
};

typedef element* liste ;
```

La création d'une liste vide, revient à faire la déclaration :

```
liste L = NULL ;  /* L contient l'adresse du premier élément de la liste */
```

La création d'un nouvel élément se réalise de la manière suivante :

```
element *e = (element* ) malloc(sizeof(element)) ;
```

Ensuite, les champs de *e* peuvent être consultés ou modifiés à l'aide de l'opérateur -> :

```
e-> valeur = 12 ;
e-> suivant = NULL ;
```

Ou, de manière équivalente (à l'aide de l'opérateur de déréférencement *) :

```
(*e).valeur = 12 ;
(*e).suivant = NULL ;
```

Réaliser les fonctions C permettant de gérer une telle liste et tester les (au moins) dans les cas suivants : insertion et suppression et tête, en fin et en milieu de liste.

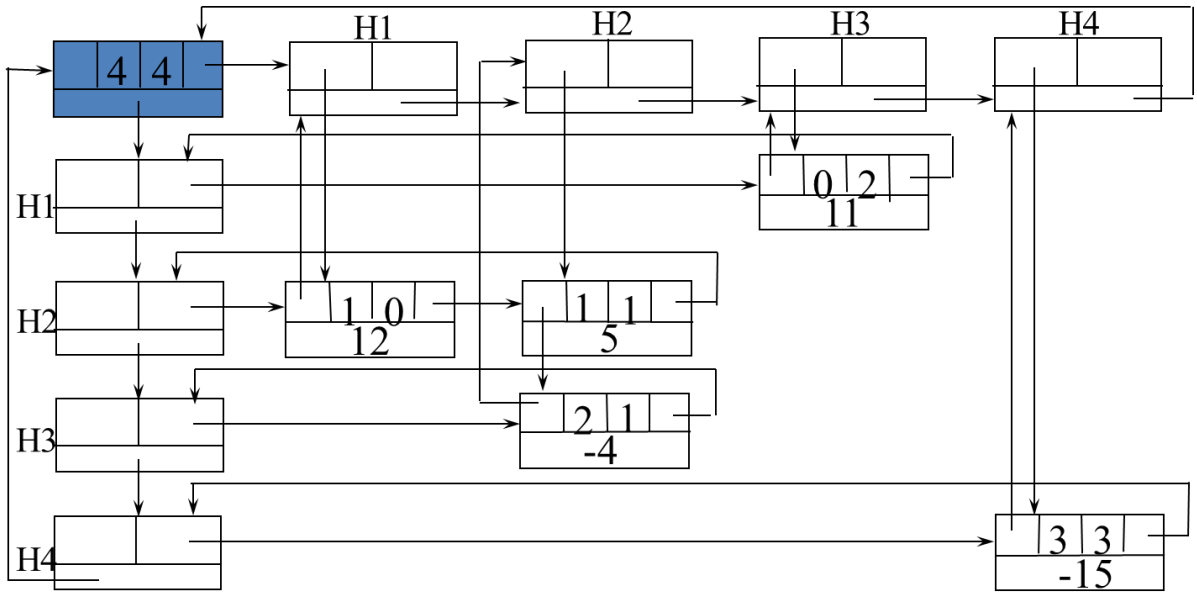
- `void afficherListe(liste *l)` : affiche tous les éléments de la liste dans l'ordre.
- `void insererElement(int x, liste *l)` : en supposant l triée, insère x dans l *en maintenant l triée*
- `void supprimerElement(int i, liste *l)` : supprime le i-ème élément de l.

NB : le type du paramètre *l* des fonctions ci-dessus est de type *liste**. Ceci est nécessaire dans le cas d'insertion/suppression en tête de liste. *Pourquoi ?*

Exercice 2 : Matrice creuse

Dans une *matrice creuse* N x M, la plupart des éléments sont nuls, si bien que la représenter en C par un tableau de N x M éléments a peu d'intérêt et termes d'occupation de la mémoire.
Une solution d'implémentation à l'aide de plusieurs listes chaînées est illustrée sur l'exemple suivant:

$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 5 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$



1. En vous inspirant de cette figure, proposez les types nécessaires et une implémentation associée permettant de représenter des matrices creuses et de réaliser deux opérations:
- Somme de deux matrices
- Produit de deux matrices
2. En supposant que chaque matrice creuse a, en moyenne, 10% d'éléments non nuls, évaluer le nombre de multiplications effectuées avec cette nouvelle représentation lors d'un produit matriciel. Comparer avec le nombre de multiplications nécessaires pour cette même opération avec une représentation des matrices avec des tableaux à deux dimensions.