Chapitre 5
Les Listes simplement chaînées

1. Gestion dynamique de la mémoire

Les fonctions de gestion dynamique de la mémoire offrent des moyens pour réserver et libérer de la mémoire dynamiquement et au fur et à mesure de l'exécution.

a. Opération d'allocation de mémoire

➢ Fonction malloc( )

La syntaxe de cette fonction est : **void * malloc(nb_octets)**
Cette fonction permet de réserver de l'espace mémoire . Elle retourne un pointeur pointant vers une zone mémoire de *nb_octets* octets.
Pour déterminer le nombre d'octets à allouer (*nb_octets*) on utilise la fonction **sizeof( )**.
Pour initialiser le pointeur en retour vers les objets désirés, il faut convertir le type de la sortie de la fonction.

Exemples :

```
int *p;
p = (int*)malloc(sizeof(int)); // allocation d'un espace mémoire de la taille d'un entier.


char * chaine ;
chaine = (char*)malloc(10 * sizeof(char)) ; // allocation d'un espace pour 10 caractères.
```

➢ Fonction calloc( )

La syntaxe de cette fonction est : **void * calloc(nb_objets, nb_octets)**
Elle a le même rôle que malloc( ) mais elle permet de réserver *nb_objets* objets de *nb_octets* octets et de les initialiser à 0.

Exemples :

```
p = (int*)calloc(N,sizeof(int)); // allocation d'un espace pour N entiers.
```

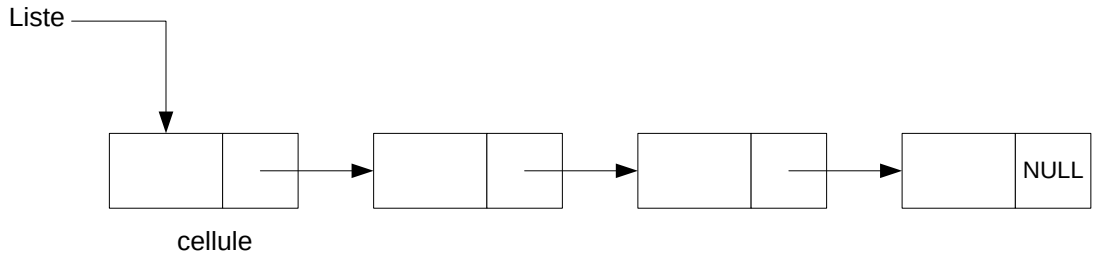
b. Opération de libération de mémoire

Lorsque l'on a plus besoin de l'espace mémoire alloué dynamiquement par les fonction malloc( ) ou calloc( ), il faut libérer ce bloc de mémoire. Ceci est fait par la fonction **free ( )** dont la syntaxe est :

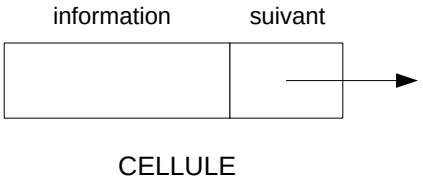
void free (nom_pointeur)

2. Définition d'une liste chaînée

Une liste chaînée est un ensemble ordonné de données homogènes. Ces éléments ne sont pas nécessairement contigus en mémoire à la différence des tableaux.
C'est un ensemble de cellules qui sont chaînées entre elles. La liste est déterminée par l'adresse de la première cellule.



Une cellule est composée de deux parties, une partie contenant la donnée utile et une deuxième partie contenant un pointeur pointant vers la cellule suivante. Une cellule est donc définie par une structure de deux champs ; un premier champ pouvant être de n'importe quel type et qui représente l'information et un deuxième champ qui est un pointeur contenant l'adresse de la cellule suivante.



Le pointeur de la dernière cellule d'une liste chaînée contient la valeur NULL.

Syntaxe

```
typedef … Element;

typedef  struct cellule {
      Element information ;
      struct cellule * suivant ;

} Cellule ;

typedef struct cellule  *Liste;
```

Exemple

On désire définir une liste chaînée dont les éléments sont des entiers.

```
typedef int Element;

typedef  struct cellule {
      Element information ;
      struct cellule * suivant ;

} Cellule ;

typedef struct cellule  *Liste;
```

3. Opérations sur les listes chaînées

a. Création d'une liste vide

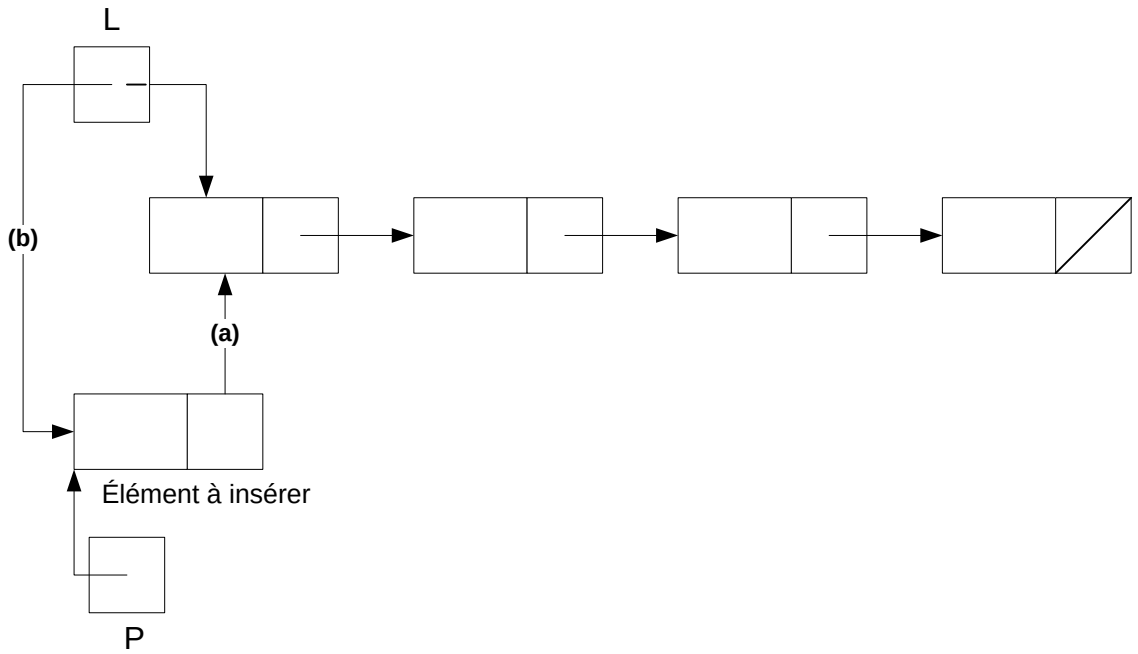
```
Liste FaireLvide()
{return NULL ; }
```

b. Vérifier si une liste est vide

```
int EstLvide(Liste L)
{return L == NULL ;}
```

c. Insertion d'éléments dans une liste

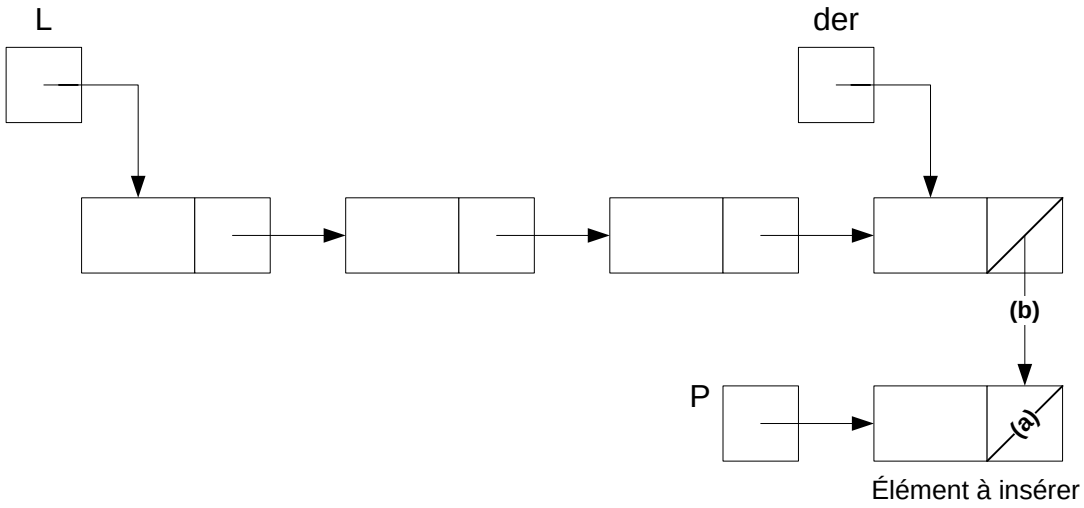
- ***Insertion en tête de liste***



Il faut commencer tout d'abord par créer la nouvelle cellule d'adresse P et dont le champ (*information*) reçoit la valeur de l'élément à insérer, et créer ensuite les deux liaisons (a) et (b) **dans cet ordre**.
Cette méthode est aussi correcte dans le cas où la liste de départ est vide.

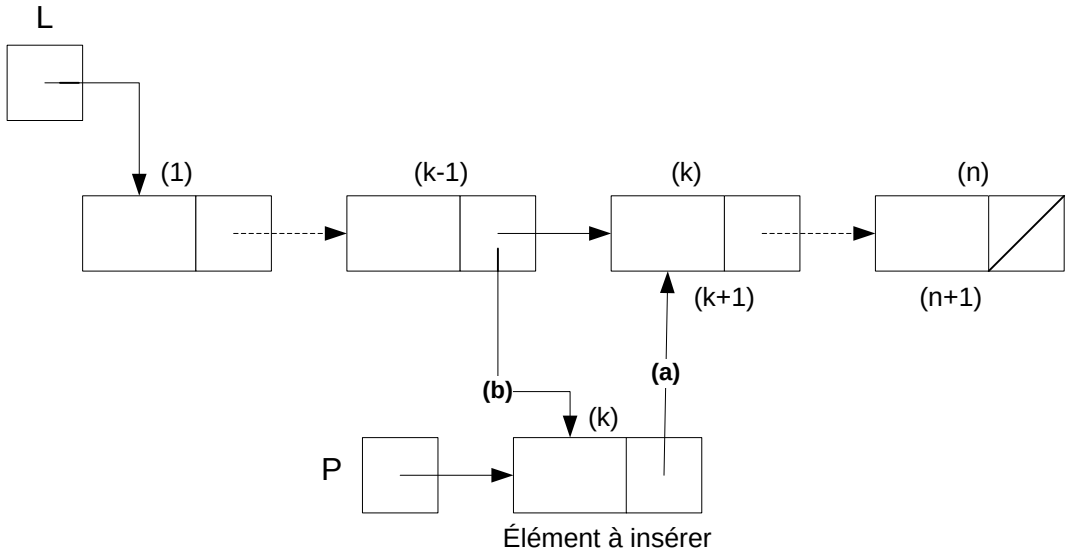
```
Liste Linsertdeb(Element x, Liste L)
{
Liste P;
P = (Liste)malloc (sizeof (Cellule));
P->information = x;
P->suivant = L;
return P;
}
```

- ***Insertion en fin de liste***



Après avoir créé une nouvelle cellule d'adresse P contenant l'élément à insérer, il faut effectuer les liaisons (a) et (b). Puisque la cellule à insérer sera la dernière cellule de la liste son champ (*suivant*) doit pointer vers la valeur NULL, ce qui est effectué par la liaison (a). Quant à la liaison (b) elle consiste à lier la dernière cellule avec l'avant dernière.
Pour effectuer la liaison (b) il faut connaître l'adresse (der) de la dernière cellule.

- ***Insertion à la K ième position***



L'insertion d'un élément à la K ième place consiste à créer les liaisons (a) et (b) dans cet ordre. Pour réaliser la liaison (b) il faut connaître l'adresse de la cellule précédente (à la position K-1). L'insertion n'est possible que si $k \in [1..n+1]$ où n représente le nombre d'éléments de la liste.

* ***Recherche d'une valeur à une position dans une liste :***

```
Liste rech_val(Liste l, element v)
/* rend l'adresse, NULL si non trouvé */
 {
  while(l!=NULL && l->val!=v) l=l->suiv;
  return(l);
 }
```

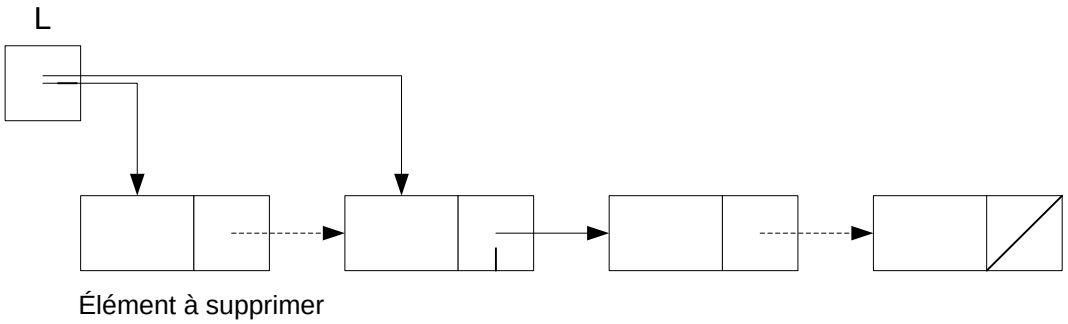
**** Insertion après la valeur trouvée***

Cette fonction doit être appelée après la fonction rech_val().
Prec est le résultat de la fonction rech_val().

```
void insert_après(Liste prec,Element val)
 {
  Liste P;
  P=(Liste)malloc(sizeof(element));
  P->information=val;
  P->suivant=prec->suivant;
  prec->suivant=P;
 }
```

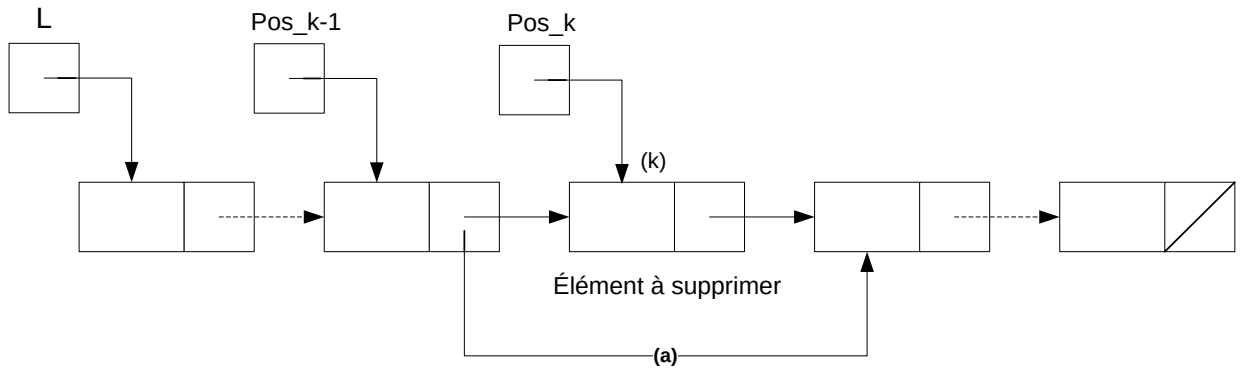
d. Suppression d'éléments d'une liste

- ***Suppression du premier élément***



- ***Suppression par position***

Il faut, comme pour l'insertion, déterminer l'adresse de la cellule qui précède celle que l'on veut supprimer ; c'est-à-dire la cellule de position (k-1), ensuite on modifie la valeur de l'adresse contenue dans le champ (*suivant*) de cette cellule comme le montre la figure.



Version itérative :

```
Liste LsupprimerIter(Element x, Liste L)
{
```

```
Liste b, c;
if (!EstLvide(L))/*si la liste n'est pas vide*/
if (L->information == x)/* Suppression du premier élément */
{
c = L;
L = L->suivant;
free (c);
}
else
   {/* Suppression par position*/
b = L ;
while (!EstLvide(b->suivant) && b->suivant->information != x)
b = b->suivant;
if (!EstLvide(b->suivant))
{
c = b->suivant;
b->suivant = b->suivant->suivant;
free (c);}
}
return a;}
```

Version Récursive :

```
Liste LsupprimerRec(Element x, Liste a)
{
Liste b;
if (!EstLvide(a))
if (a->information == x)
{
b = a;
a = a->suivant;
free (b);
}
else
a->suivant = LsupprimerRec(x,a->suivant);
return a;
}
```

Afficher une liste

```
void affiche_lst(adr_comp l)
 {
  while(l!=NULL)
   {
    printf("%6.1f ",l->val);
    l=l->suiv;
   }
  printf("\n");
 }
```

Calculer la longueur d'une liste

```
int longueur_lst(adr_comp l)
 {
  int n=0;
  while(l!=NULL)
   {
    l=l->suiv;
    n++;
   }
  return(n);
 }
```

Le code correspondant à la création de la liste :

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
typedef struct cellule
{
      int val ;
      struct cellule* suivant;
      }
      cellule;
      void creation(cellule* *pl)
      {
          *pl=NULL;
          cellule*p=(int*)malloc(sizeof(int));
          printf("creation de list en cours:\n");
          char rep;
          do{
                printf("donner le valeur a ajoute pour crée la liste \n");
                scanf("%d",&p->val);
                p->suivant=*pl;
                *pl=p;
                printf("voulez vous contimer la création(o/n):\n");
                scanf("%c",rep);
            }
            while( rep=='o' || rep=='n');
            }


           int main()
```

```
{cellule* tete;
creation(&tete);

getchar();
}
```