PARTIE V : Les listes chaînées

Définitions :

C'est un ensemble de nœuds reliés entre eux par des liens de chaînage (des adresses).

Chaque nœud est constitué de deux champs l'un pour l'information et l'autre pour l'adresse du nœud suivant.

Un nœud est appelé aussi élément ou maillon ou chaînant ou encore node c'est ce dernier qu'on va utiliser dans ce fascicule.

Définition d'une liste chaînée unidirectionnelle.

```
#include <stdlib.h>

Typedef struct   node   node;

Struct  node {

                Item    info;

                Struct  node   *suiv;

               };

Typedef   node*        liste;
```

NB : item c'est le type de l'information qui sera mise dans la liste. Si **item** est un type simple genre Int, float ou bien char … Au lieu d'utiliser le vocable « item », on utilisera le type simple. Par contre si item est de type complexe, il est utile d'utiliser le vocable item et le définir à l'aide de « typedef ».

Si vous voulez utiliser le type même complexe, à vous de choisir car ce n'est pas une règle !

Exemple1 :

```
Typedef Int  item;

Item info ; /* C'est comme si on déclare Int info;*/
```

Dans ce cas, il est préférable d'utiliser directement le type Int :

```
Int info ;
```

Exemple2 :

```
Typedef struct item   {        int num;
                               int  qte ;
                               float prix ;
                      } item;
Item info ;
```

/* On a déclaré info de type item qui est synonyme du type enregistrement*/

NB : Vous remarquez qu'avant de déclarer la liste L, on a inclus <stdlib.h> pour pouvoir utiliser la macro NULL qui veut dire 'pas d'adresse' ou 'Nulle part' tout simplement pour dire qu'il n'y a pas de suivant.

Après avoir définit le type 'liste', on peut déclarer une liste L.

Il existe trois (03) déclarations possibles, elles sont différentes mais équivalentes :

- Liste L = NULL ;
- Node *L = NULL ;
- Struct node *L = NULL ;

On remarque qu'on est en train de déclarer la liste L et en même temps l'initialiser avec NULL car pour le moment elle est vide.

Pour la suite, on va utiliser deux fonctions prédéfinies pour allouer de l'espace mémoire à l'aide de « malloc » et pour libérer de l'espace mémoire à l'aide de « free » et cela est possible grâce à l'inclusion de la < stdlib.h >.

Quand une liste est vide cela veut dire qu'elle est égale à NULL.

Quand une liste est constituée d'un seul nœud, on dit que la liste est un singleton.

OPERATIONS DE BASE SUR LES LISTES CHAINEES

Les plus importantes des opérations de bases sont les ajouts et les suppressions.

Ajout d'un nouveau nœud à la liste L. Trois cas de figure se présentent à savoir : En tête de la liste, en fin de la liste ou encore au milieu de la liste à une adresse donnée.

1. **Ajouter en tête de liste.**

```
/*Fonction qui réalise le travail*/
Liste    AjouterEnTete(liste    L , item  valeur)
{
   /*créer un nouveau nœud*/
        Node* nouveaunode  = malloc(sizeof(node)) ;
        Nouveaunode->info = valeur ;
        Nouveaunode->suiv = L ;
        Return nouveaunode ;
}
```

/* La fonction prédéfinie « malloc » permet de faire une allocation dynamique de la mémoire */

/*Si la liste L était vide, on retourne une liste singleton constituée du nouveau nœud qu'on vient d'ajouter*/

2. Ajout en fin de liste.

Le nouveau nœud qu'on va ajouter à la fin de la liste L sera le dernier de la liste donc son suivant (suiv) sera égal à NULL.

```
/*Fonction qui réalise le travail*/
Liste    AjouterEnFin(liste      L , item  valeur)
{
  /*créer un nouveau nœud*/
        Node* nouveaunode  = malloc(sizeof(node)) ;
        Nouveaunode->info = valeur ;
        Nouveaunode->suiv = NULL ;
        If (L == NULL)
          Return nouveaunode ;
        Else
            {    /*On doit parcourir la liste L jusqu'au dernier nœud*/
                 /*pour cela on aura besoin d'un pointeur */
                 Node* pt = L;
                 While (pt->suiv != NULL)
                 {
                          Pt = pt->suiv ;
                 }
                 Pt->suiv = nouveaunode ;
                 Return L ;
             }
}
```

3. Ajouter au milieu de la liste L et à l'adresse ADR.

Trois cas de figure se présentent :

- ADR = L donc l'ajout se fait en tête de la liste L.
- ADR n'existe pas, on retourne NULL.
- ADR est trouvée donc l'ajout se fait entre deux nœuds.

Pour ce dernier cas, on aura besoin de deux pointeurs, l'un pour pointer le nœud qui se trouve à l'adresse ADR et le deuxième pour pointer le nœud précédant.

Les deux pointeurs utilisés seront pt et ptprec.

```
        /*Fonction qui réalise le travail*/
Liste   AjouterAuMilieu(liste L , liste ADR , item  valeur)
{
  /*créer un nouveau nœud*/
        Node* nouveaunode  = malloc(sizeof(node)) ;
        Nouveaunode->info = valeur ;
        Node* pt = L ;
        Node* ptprec = L;
        While ((pt != ADR) && (pt != NULL))
        {
                Ptprec = pt;
                Pt = pt->suiv;
        }
        If (pt == NULL)          /* ADR n'existes pas*/
         {
           Return NULL ;
         }
        Else
                If (pt == L)      /*L = ADR*/
                {
                  Return AjouterEnTete(L, valeur) ;
                }
                Else              /* l'ajout se fait entre deux nœuds ptprec et pt*/
                    {
                        Nouveaunode->suiv = pt ;
                        Ptprec->suiv = nouveaunode ;
                    }
}
```

REFLEXION : Et si on utilisera des procédures au lieu des fonctions !

Suppression d'un nœud de la liste L. Trois cas de figure se présentent à savoir : En tête de la liste, en fin de la liste ou encore au milieu de la liste à une adresse donnée.

Attention : Dans tous les cas et avant de faire la suppression, il est nécessaire de vérifier si la liste n'est pas vide. Et dans ce cas soit faire retourner NULL, soit écrire un message d'erreur.

I. Suppression du nœud du début de la liste L.

```
/*Une fonction pour supprimer ce nœud*/
Liste suppressionEnTete(liste L)
{
        If (L == NULL)
           Return NULL ;
        Else
            {
                Node*  nodesuivant = L->suiv ;
                Free(L) ; /* Libérer la mémoire*/
                Return nodesuivant ;
             }
}
```

NB : D'une manière générale, il est prudent de faire suivre tout appel à la fonction « free » d'une mise à NULL du pointeur correspondant.

Dans notre cas free(L) ; sera free(L), L = NULL ;

Mais cela n'est pas obligatoire !

II. Suppression d'un nœud à la fin d'une liste.

On peut distinguer trois cas possibles :

- La liste est vide donc il n'y a rien à supprimer. On retourne NULL ou bien un message d'erreur est écrit.
- La liste est singleton donc elle contient un seul nœud qui va être supprimé et la liste devient vide! (Retourner NULL).
- La liste contient plus d'un nœud par conséquent on va parcourir la liste à l'aide de deux pointeurs pour qu'à la fin un pointeur est sur l'avant dernier nœud et le second sur le dernier.

```
/*La fonction qui fait ce travail*/
Liste suppressionEnFin(liste L)
{
        If (L == NULL)                  /*liste vide*/
          Return NULL ;
        else    /* La liste n'est pas vide*/
                if (L->suiv == NULL)    /* liste singleton*/
                {
                        Free(L), L=NULL;        /*libérer la mémoire*/
                        Return NULL;
                }
        Node* Pt = L;                           /* cas général*/
        Node* Ptprec = L;
        While (Pt->suiv != NULL)
        {
                Ptprec = Pt;
                Pt = Pt->suiv;
        }
        Ptprec->suiv = NULL;
        Free(Pt), Pt = NULL;    /* Libérer la mémoire*/
        Return L;
}
```

III. Supprimer un nœud au milieu d'une liste, ce nœud se trouve à l'adresse ADR par exemple.

Plusieurs cas peuvent se présenter :

- Liste vide donc rien à supprimer.
- ADR est l'adresse du premier nœud (L=ADR) donc la suppression se fait en tête de liste !
- ADR n'a pas été trouvée, retourner NULL ou retourner L (à choisir).
- ADR a été trouvée mais il n'a pas de suivant donc la suppression se fait en fin de liste !
- ADR a été trouvée et ce nœud a un suivant et un précédant donc il est bel et bien au milieu de la liste (cas général) ; on le supprime le plus normalement du monde !

```
/*La fonction qui fait ce travail*/
        Liste suppressionAuMilieu(liste L, liste ADR)
        {
                If (L == NULL)                  /*liste vide*/
                  Return NULL ;
                else
                        if (L == ADR)    /* Suppression en tête de liste*/
                            return SuppressionEnTete(L);
                        else /* Chercher ADR qui n'est pas en tête de liste*/
                        {
                                Node* Pt = L;
                                Node* Ptprec = L;
                                While ((Pt != NULL) && (Pt != ADR))
                                {
                                        Ptprec = Pt;
                                        Pt = Pt->suiv;
                                }
                                If Pt = NULL            /* ADR n'a pas été trouvée*/
                                  Return NULL; /* ou bien return L;*/
                                Else    /* ADR a été trouvée et  Pt = ADR*/
                                     If  Pt->suiv = NULL  /* dernier noeud*/
                                        Return SuppressionEnFin(L);
                                /* Le noeud est au milieu de la liste : cas général*/
                                Ptprec->suiv = Pt->suiv;
                                Free(Pt), Pt = NULL;
                                Return L;
        }
```

```
/*La fonction qui fait ce travail autre version*/
        Liste suppressionAuMilieu(liste L, liste ADR)
        {
                If (L == NULL)                  /*liste vide*/
                  Return NULL ;
                else
                        if (L == ADR)    /* Suppression en tête de liste*/
                          return SuppressionEnTete(L);
                        else /* ADR n'est pas en tête de liste il faut la chercher */
                        {
                                Node* Pt = L;
                                Node* Ptprec = L;
                                While ((Pt != NULL) && (Pt != ADR))
                                {
                                        Ptprec = Pt;
                                        Pt = Pt->suiv;
                                }
                                If Pt == NULL           /* ADR n'existe pas dans la liste*/
                                  Return L;
                                /* Le noeud est au milieu de la liste ou en fin de liste*/
                                /* ça marche dans les deux cas*/
                                Ptprec->suiv = Pt->suiv;
                                Free(Pt), Pt = NULL;
                                Return L;
        }
```

BIBLIOGRAPHIE

1- Langage C : Cours et références
Pierre NERZIC Mars 2003

2- Langage C : Support de cours
Patrick CORDE Mars 2006

3- Data structures and programming design
Robert L. Kruse

4- Algorithmique, structures de données et langage C

J.M. ENJALBERT Janvier 2005

5- An introduction to methodical programming

W. Findbay and D. Wattes

6- Apprendre à programmer

Algorithmes et conception objet.

C. Dabancourt