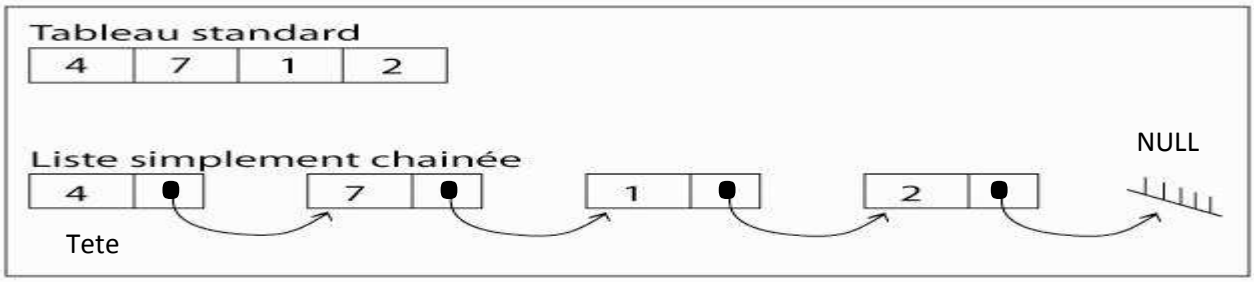
Les listes chainées

1. Généralité

Lorsqu'on crée un algorithme utilisant des conteneurs, il existe différentes manières de les implémenter, la façon la plus courante étant les tableaux. Lorsqu'on crée un tableau, les éléments de celui-ci sont placés de façon contiguë en mémoire. Pour pouvoir le créer, il nous faut connaître sa taille. Si on veut supprimer un élément au milieu du tableau, il nous faut recopier les éléments temporairement, ré-allouer de la mémoire pour le tableau, puis le remplir à partir de l'élément supprimé. En bref, ce sont beaucoup de manipulations coûteuses en ressources.

Une liste chaînée est différente dans le sens où les éléments de notre liste sont répartis dans la mémoire et reliés entre eux par des pointeurs. On peut ajouter et enlever des éléments d'une liste chaînée à n'importe quel endroit, à n'importe quel instant, sans devoir recréer la liste entière.
On va essayer de voir ceci plus en détail sur ces schémas :



Vous avez sur ce schéma la représentation que l'on pourrait faire d'un tableau et d'une liste chaînée. Chacune de ces représentations possède ses avantages et inconvénients. C'est lors de l'écriture de votre programme que vous devez vous poser la question de savoir laquelle des deux méthodes est la plus intéressante.

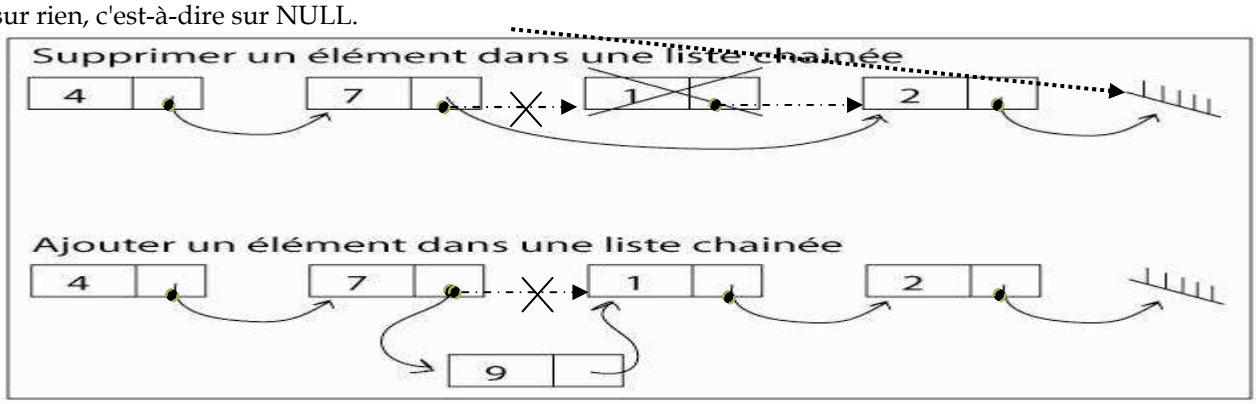
Tableau	***Liste***
✓ *Dans un tableau, la taille est connue, l'adresse du premier élément aussi. Lorsque vous déclarez un tableau, la variable contiendra l'adresse du premier élément de votre tableau.* *Comme le stockage est contigu, et la taille de chacun des éléments connue, il est possible d'atteindre directement la case i d'un tableau.* ✓ *Pour déclarer un tableau, il faut connaître sa taille.* ✓ *Pour supprimer ou ajouter un élément à un tableau, il faut créer un nouveau tableau et supprimer l'ancien. Ce n'est en général pas visible par l'utilisateur, mais c'est ce que* ***realloc*** *va souvent faire. L'adresse du premier élément d'un tableau peut changer après un* ***realloc****, ce qui est tout à fait logique puisque* ***realloc*** *n'aura pas forcement la possibilité de trouver en mémoire la place nécessaire et contiguë pour allouer votre nouveau tableau.* ***realloc*** *va donc chercher une place suffisante, recopier votre tableau, et supprimer l'ancien.*	✓ *Dans une liste chaînée, la taille est inconnue au départ, la liste peut avoir autant d'éléments que votre mémoire le permet.* ✓ *Il est en revanche impossible d'accéder directement à l'élément i de la liste chainée.* *Pour ce faire, il vous faudra traverser les i-1 éléments précédents de la liste.* ✓ *Pour déclarer une liste chaînée, il suffit de créer le pointeur qui va pointer sur le premier élément de votre liste chaînée, aucune taille n'est donc à spécifier.* ✓ *Il est possible d'ajouter, de supprimer, d'intervertir des éléments d'une liste chaînée sans avoir à recréer la liste en entier, mais en manipulant simplement leurs pointeurs.*

2. Caractéristiques d'une liste

Chaque élément d'une liste chaînée est composé de deux parties :

- ***la valeur qu'on veut stocker,***
- ***l'adresse de l'élément suivant, s'il existe.***
  S'il n'y a plus d'élément suivant, alors l'adresse sera NULL, et désignera le bout de la chaîne.

Voilà deux schémas pour expliquer comment se passent l'ajout et la suppression d'un élément d'une liste chaînée. Remarquons le symbole en bout de chaîne qui signifie que l'adresse de l'élément suivant ne pointe sur rien, c'est-à-dire sur NULL.



Comme vous vous en doutez certainement maintenant, la liste chaînée est un type structuré. Nous en avons terminé avec ces quelques généralités, nous allons pouvoir passer à la définition d'une structure de données nous permettant de créer cette fameuse liste !

3. Déclaration d'une liste chainée en C

On doit obligatoirement spécifier le type de l'élément de la liste chaînée. En effet, on peut créer des listes chaînées de n'importe quel type d'éléments : entiers, caractères, structures, tableaux, voir même d'autres listes chaînées. On peut même combiner plusieurs types dans une même liste.

Exemple :

```
typedef struct element {
    int val;
    struct element *suivant;
} element;
element * Liste=NULL;
```

ou bien

```
struct element {
    int val;
    struct element * suivant;
} element;
 struct element * Liste=NULL;
```

On crée le type element qui est une structure contenant un entier (val) et un pointeur sur élément (suivant), qui contiendra l'adresse de l'élément suivant. Ensuite, il nous faut créer la variable ***Liste*** qui est en fait un pointeur sur le type element. Lorsque nous allons déclarer la liste chaînée, nous devrons déclarer un pointeur sur element, l'initialiser à NULL, pour pouvoir ensuite allouer le premier élément. N'oubliez pas d'inclure **stdlib.h** afin de pouvoir utiliser la macro NULL.
Il est important de toujours initialiser la liste chaînée à NULL. Le cas échéant, elle sera considérée comme contenant au moins un élément. C'est une erreur fréquente.

4. Manipulation des listes chainées

Maintenant que nous savons comment déclarer une liste chaînée, il serait intéressant d'apprendre à ajouter des éléments dans cette liste, à supprimer des éléments de cette liste, rechercher un élément dans cette liste, de compter le nombre d'éléments que contient cette liste, ainsi que de lire et d'afficher ce qu'elle contient.

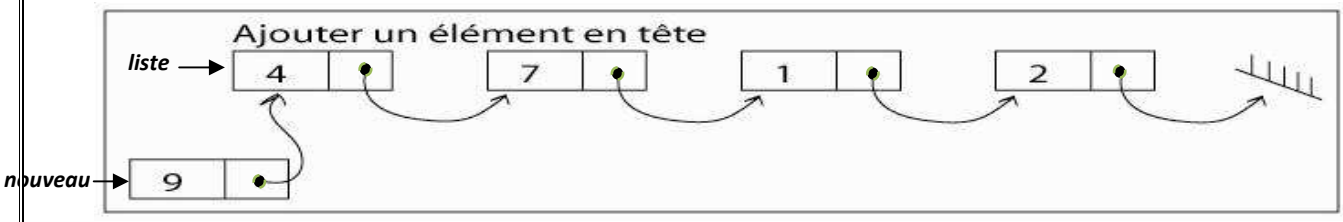
4.1 Ajouter un élément

Lorsque nous voulons ajouter un élément dans une liste chaînée, il faut savoir où l'insérer. Les deux ajouts génériques des listes chaînées sont les ajouts en tête, et les ajouts en fin de liste.

a. Ajouter en tête

Lors d'un ajout en tête, nous allons créer un élément, lui assigner la valeur que l'on veut ajouter, puis pour terminer, raccorder cet élément à la liste passée en paramètre. Lors d'un ajout en tête, on devra donc assigner à

suivant l'adresse du premier élément de la liste passé en paramètre. Visualisons tout ceci sur un schéma :

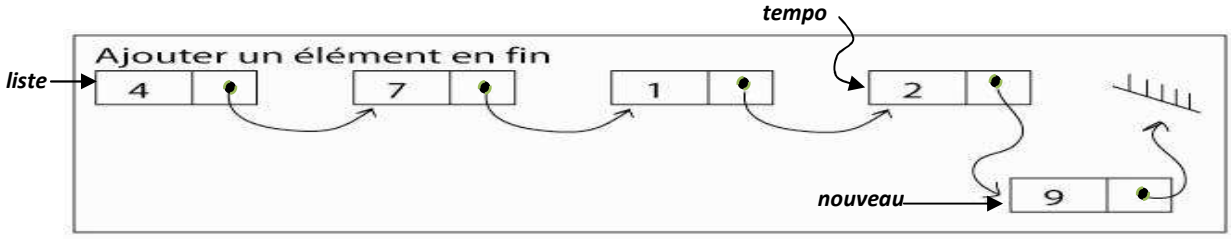



```
element*  ajouterDebut ( element* liste, int valeur  )  {
   element* nouveau = malloc(sizeof(element));                  /* On crée un nouvel élément */
      nouveau ->val = valeur ;                                  /* On assigne la valeur au nouvel élément */
     nouveau ->suivant = liste ;                          /* On assigne l'adresse de l'élément suivant au nouvel élément */
     return nouveau ;                          /* On retourne la nouvelle liste, i.e. le pointeur sur le premier élément */
}
```

On crée un nouvel élément puis on le relie au début de la liste originale. Si l'original est (vide) c'est **NULL** qui sera assigné au champ suivant du nouvel element. La liste contiendra dans ce cas-là un seul élément.

b. Ajouter en fin de liste

Cette fois-ci, c'est un peu plus compliqué. Il nous faut tout d'abord créer un nouvel élément, lui assigner sa valeur, et mettre l'adresse de l'élément suivant à NULL. En effet, comme cet élément va terminer la liste nous devons signaler qu'il n'y a plus d'élément suivant. Ensuite, il faut faire pointer le dernier élément de liste originale sur le nouvel élément que nous venons de créer. Pour ce faire, il faut créer un pointeur temporaire sur element qui va se déplacer d'élément en élément, et regarder si cet élément est le dernier de la liste. Un élément sera forcément le dernier de la liste si NULL est assigné à son champ ***suivant***.




```
element* ajouterFin(element*  liste, int valeur){
 element* nouveau = malloc(sizeof(element));/*On crée un nouvel élément */
  nouveau->val = valeur; /* On assigne la valeur au nouvel élément */
  nouveau ->suivant = NULL;/* On ajoute en fin, donc aucun élément ne va suivre */
    if(liste == NULL)    {
   return nouveau; /* Si la liste est vide il suffit de renvoyer l'élément crée */
    }     else     {
element* tempo=liste;  /* Sinon, on parcourt la liste à l'aide d'un pointeur temporaire et on indique que le
                                  dernier élément de la liste est relié au nouvel élément */
   while(tempo->suivant != NULL)
        {
            tempo = tempo->suivant;
        }
        tempo->suivant = nouveau;
        return liste;     }    }
```

Comme vous remarquez, nous nous déplaçons le long de la liste chaînée grâce au pointeur tempo. Si l'élément pointé par tempo n'est pas le dernier (tempo->suivant != NULL), on avance d'un pas (tempo = tempo->suivant) en assignant à tempo l'adresse de l'élément suivant. Une fois que l'on est au dernier élément, il ne reste plus qu'à le relier au nouvel élément crée.

4.2 Affichage du contenu de la liste

Le programme suivant va nous servir pour afficher le contenu de la liste.

```
void afficherListe(element* liste)
{
    element *tmp = liste;
    while(tmp != NULL)    /* Tant que l'on n'est pas au bout de la liste */
    {
        printf("%d ", tmp->val);      /* On affiche */
        tmp = tmp->suivant;           /* On avance d'une case */
    }           }
```

4.3 Exploitation des trois fonctions crées :

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
typedef struct element
{
   int val;
   struct element *suivant;
} element;
element* ajouterDebut( element* liste, int valeur){
   element* nouveau = malloc(sizeof(element));
   nouveau ->val = valeur;
   nouveau ->suivant = liste;
   return nouveau ;
    }
 element*  ajouterFin( element* liste, int valeur){
  element* nouveau = malloc(sizeof(element));
   nouveau->val = valeur;
   nouveau->suivant = NULL;
   if(liste == NULL)
   {
    return nouveau;
   }
  else    {
      element* temp=liste;
      while(temp->suivant != NULL)
      {
         temp = temp->suivant;
      }
      temp->suivant = nouveau;
      return  liste;
  }          }
void afficherListe( element* liste){
    element *tmp = liste;
  while(tmp != NULL)
{
        printf("%d ", tmp->val);
      tmp = tmp->suivant;
   }           }
int main(){
   element* liste=NULL;
   int i, valeur;
   for (i=0;i<5;i++){
   printf(" entrer une valeur ");
   scanf("%d",&valeur);
   liste= ajouterDebut(liste, valeur);
}
printf(" Affichage des données ajoutées au debut
de la liste\n");
 afficherListe(liste);
printf("\n");
    for (i=0;i<5;i++){
   printf(" entrer une valeur ");
   scanf("%d",&valeur);
   liste= ajouterFin(liste, valeur);
}
printf(" Affichage des données ajoutées au debut et
en fin de la liste\n");
 afficherListe(liste);
 getch();
return 0;      }
```

Exemple d'exécution

```
C:\Users\Moi\Desktop\liste1.exe
 entrer une valeur 11
 entrer une valeur 12
 entrer une valeur 14
 entrer une valeur 15
 entrer une valeur 16
 Affichage des donnúes ajoutúes au debut de la liste
16 15 14 12 11
 entrer une valeur 21
 entrer une valeur 22
 entrer une valeur 23
 entrer une valeur 24
 entrer une valeur 25
 Affichage des donnúes ajoutúes au debut et en fin de la liste
16 15 14 12 11 21 22 23 24 25 _
```

Exercice1

Écrivez une fonction qui renvoie 1 si la liste est vide, et 0 si elle contient au moins un élément. Son prototype est le suivant : int estVide(element * liste);

L'intérêt d'une telle fonction est le suivant :

```
if(estVide(ma_liste))
  printf("La liste est vide");
  else   afficherListe(ma_liste);
```

Exercice2 : Refaire les fonctions définies dans le cours mais cette fois les données sont les informations d'un élève (il s'agit du code de l'élève (int) , son nom (cha[10]) et sa moyenne (float))

Dans ce qui suit, nous allons voir un ensemble de fonctions : Supprimer des éléments, rechercher un élément, compter le nombre des éléments… etc.

4.4 Suppression d'un élément de la liste

a. Supprimer un élément en tête

Il s'agit là de supprimer le premier élément de la liste. Pour ce faire, il nous faudra utiliser la fonction free que vous connaissez certainement. Si la liste n'est pas vide, on stocke l'adresse du premier élément de la liste après suppression (i.e. l'adresse du 2ème élément de la liste originale), on supprime le premier élément, et on renvoie la nouvelle liste. Attention quand même à ne pas libérer le premier élément avant d'avoir stocké l'adresse du second, sans quoi il sera impossible de la récupérer.

```
element* supprimerEnTete(element* liste){
    if(liste != NULL)  // Si la liste est non vide, on se prépare à renvoyer
    {                          // l'adresse de l'élément en 2ème position
        element* aRenvoyer = liste->suivant;
        free(liste);          /* On libère le premier élément */
        return aRenvoyer;     /* On retourne le nouveau début de la liste */
    }
    else         return NULL;    }
```

b. Supprimer un élément en fin de liste

Cette fois-ci, il va falloir parcourir la liste jusqu'à son dernier élément, indiquer que l'avant-dernier élément va devenir le dernier de la liste et libérer le dernier élément pour enfin retourner le pointeur sur le premier élément de la liste d'origine.

```
element* supprimerEnFin(element* liste)   {
    if (liste == NULL)    /* Si la liste est vide, on retourne NULL */
        return NULL;
    if(liste->suivant == NULL) {   /* Si la liste contient un seul élément */
      free(liste);      /*On le libère et on retourne NULL (la liste est maintenant vide)*/
        return NULL;
    }
      element * courant=liste, *tempo ;
      Courant=malloc(sizeof(element)) ;
      While(courant->suivant !=NULL){
      tempo=courant ;
      Courant=courant->suivant ;
}
tempo->suivant=courant->suivant ;
free(courant) ;
return liste ;}
```

4.5 Rechercher un élément dans une liste

Le but de cette fonction est de renvoyer l'adresse du premier élément trouvé ayant une certaine valeur. Si aucun élément n'est trouvé, on renverra NULL. L'intérêt est de pouvoir, une fois le premier élément trouvé, chercher la prochaine occurrence en recherchant à partir de ***element Trouve->suivant***. On parcourt donc la liste jusqu'au bout, et dès qu'on trouve un élément qui correspond à ce que l'on recherche, on renvoie son adresse.

```
element* rechercherElement(element* liste, int valeur){
    element *tempo=liste ;
```

```
        while(tempo != NULL){   /* Tant que l'on n'est pas au bout de la liste */
  if(tempo->val == valeur){ // Si l'élément a la valeur recherchée, on renvoie son adresse
            break;
        }
      //  tempo = tempo->suivant;  // pour retourner l'adresse de l'élément suivant
    }
    return tempo;}
```

4.6 Recherche du i-ème élément

Exercice : ecrire une fonction qui retournera le ième élément de la liste. (On se déplace i fois à l'aide du pointeur tmp le long de la liste chaînée et de renvoyer l'élément à l'indice i. Si la liste contient moins de i élément(s), alors on renverra NULL).

```
element* element_i(element* liste, int indice)
{
    int i;
for(i=0; i<indice && liste != NULL; i++)/* On se déplace de i cases, tant que c'est possible */
    {
        liste = liste->suivant;
    }
    if(liste == NULL) // Si l'élément est NULL, c'est que la liste contient moins de i éléments

    {
        return NULL;
    }
    else      /* Sinon on renvoie l'adresse de l'élément i */
    {
       return liste;
    }      }
```

4.7 Compter le nombre d'occurrences d'une valeur

Pour ce faire, nous allons utiliser la fonction précédente permettant de rechercher un élément. On cherche une première occurrence : si on la trouve, alors on continue la recherche à partir de l'élément suivant, et ce tant qu'il reste des occurrences de la valeur recherchée. Il est aussi possible d'écrire cette fonction sans utiliser la précédente bien entendu, en parcourant l'ensemble de la liste avec un compteur que l'on incrémente à chaque fois que l'on passe sur un élément ayant la valeur recherchée. Cette fonction n'est pas beaucoup plus compliquée, mais il est intéressant d'un point de vue algorithmique de réutiliser des fonctions pour simplifier nos codes.

```
int nombreOccurences(element* liste, int valeur)   {
    int i = 0;
     if(liste == NULL)  /* Si la liste est vide, on renvoie 0 */
        return 0;  /* Sinon, tant qu'il y a encore un élément ayant la val = valeur */
    while((liste = rechercherElement(liste, valeur)) != NULL)
    {
        liste = liste->suivant;          /* On incrémente */
        i++;
    }
    return i;   } /* Et on retourne le nombre d'occurrences */
```

4.7 Récupérer la valeur d'un élément

C'est une fonction du même style que la fonction ***estVide***. Elle sert à "masquer" le fonctionnement interne à l'utilisateur. Il suffit simplement de renvoyer à l'utilisateur la valeur d'un élément. Il faudra renvoyer un code d'erreur entier si l'élément n'existe pas (la liste est vide),. Dans ce code, je considère qu'on ne travaille qu'avec des nombres entiers positifs, on renverra donc -1 pour une erreur. Vous pouvez mettre ici n'importe quelle valeur que vous êtes sûrs de ne pas utiliser dans votre liste. Une autre solution consiste à renvoyer un pointeur sur int au lieu d'un int, vous laissant donc la possibilité de renvoyer NULL.

```
#define ERREUR -1
int valeur(element* liste)  {
    return ((liste == NULL)? ERREUR:(liste->val));
}
```

4.8 Compter le nombre d'éléments d'une liste chaîné

C'est un algorithme un peu vraiment simple. Vous parcourez la liste de bout en bout et incrémentez d'un (1) pour chaque nouvel élément que vous trouvez. Jusqu'à maintenant, nous n'avons utilisé que des algorithmes itératifs qui consistent à boucler tant que l'on n'est pas au bout. Nous allons voir qu'il existe des algorithmes que l'on appelle récursifs et qui consistent en fait à demander à une fonction de s'appeler elle-même. Pour créer un algorithme récursif, il faut connaître la condition d'arrêt (ou condition de sortie) et la condition de récurrence. Il faut en fait vous imaginer que votre fonction a fait son office pour les n-1 éléments suivants, et qu'il ne reste plus qu'à traiter le dernier élément.

```
int nombreElements(element* liste)  {
    if(liste == NULL)    /* Si la liste est vide, il y a 0 élément */
        return 0;
  /* Sinon, il y a un élément (celui que l'on est en train de traiter)
    plus le nombre d'éléments contenus dans le reste de la liste */
  return nombreElements(liste->suivant)+1;      }
```

4.9 Effacer tous les éléments ayant une certaine valeur

Pour cette dernière fonction, nous allons encore une fois utiliser un algorithme récursif.

```
element* supprimerElement(element* liste, int valeur){
    if (liste == NULL)    /* Liste vide, il n'y a plus rien à supprimer */
    return NULL;
 if(liste->val == valeur){ //Si l'élément en cours de traitement doit être supprimé
    element* tmp = liste->suivant;      /* On le supprime en prenant soin de
     free(liste);                  mémoriser l'adresse de l'élément suivant */
/*L'élément est supprimé, la liste commencera à l'élément suivant pointant sur une liste qui
          ne contient plus aucun élément ayant la valeur recherchée */
        tmp = supprimerElement(tmp, valeur);
        return tmp;}
        else   {
    /* Si l'élement en cours de traitement ne doit pas être supprimé,
 alors la liste finale commencera par cet élément et suivra une liste ne contenant
 plus d'élément ayant la valeur recherchée */
  liste->suivant = supprimerElement(liste->suivant, valeur);
        return liste;    }
   }
```

4.10 Effacer une liste

Nous allons maintenant écrire une fonction permettant d'effacer complètement une liste chaînée de la mémoire. On va écrire avec un algorithme itératif dans un premier temps, puis une seconde fois grâce à un algorithme récursif. Dans le premier cas, on doit parcourir la liste, stocker l'élément suivant, effacer l'élément courant et avancer d'une case. A la fin la liste est vide, nous retournerons NULL.

Itératif

```
element* effacerListe(element* liste){
    element* tmp = liste;
    element* tmpsuiv;
//Tant que l'on n'est pas au bout de la liste
    while(tmp != NULL)    {
/* On stocke l'élément suivant pour
pouvoir ensuite avancer */
  tmpsuiv = tmp->suivant;
   /* On efface l'élément courant */
   free(tmp);
   /* On avance d'une case */
        tmp = tmpsuiv;
    } // La liste est vide : on retourne NULL
    return NULL;  }
```

Récursif

```
element* effacerListe(element* liste){
    if(liste == NULL)    {
return NULL; /* Si la liste est vide,
il n'y a rien à effacer, on retourne
une liste vide i.e. NULL */
    }
    else    {
/* Sinon, on efface le premier élément
et on retourne le reste de la
      liste effacée */
  element *tmp;
       tmp = liste->suivant;
       free(liste);
       return effacerListe(tmp);
   }     }
```

Solution de l'exercice2 de la page 5:

```
#include <stdlib.h>                    #include <stdio.h>
```

```
#include <conio.h>
typedef struct {
int code;
char nom[10];
float moy;
} donnee;
typedef struct element    {
   donnee eleves;
   struct element *suivant;
}element;
 element* ajouterDebut( element* liste, donnee
eleve){
      /* On crée un nouvel élément */
   element* nouveau = malloc(sizeof(element));
    /* On assigne la valeur au nouvel élément */
   nouveau->eleves = eleve;
/* On assigne l'adresse de l'élément suivant au
nouvel élément */
   nouveau->suivant = liste;
 /* On retourne la nouvelle liste, i.e. le pointeur sur le
premier élément */
   return nouveau;
}
 element* ajouterFin( element* liste, donnee eleve){
       /* On crée un nouvel élément */
  element* nouveau = malloc(sizeof(element));
   /* On assigne la valeur au nouvel élément */
   nouveau->eleves = eleve;
/* On ajoute en fin, donc aucun élément ne va suivre
*/
  nouveau->suivant = NULL;
   if (liste == NULL)    {
 /* Si la liste est videé il suffit de renvoyer l'élément
créé */
    return nouveau;
   } else  {
/* Sinon, on parcourt la liste à l'aide d'un pointeur
temporaire et on  indique que le dernier élément de
la liste est relié au nouvel élément */
      element* temp=liste;
      while(temp->suivant != NULL)
      {
         temp = temp->suivant;
      }
      temp->suivant = nouveau;
      return liste;
   }
}
void afficherListe( element* liste){
    element *tmp = liste;
   /* Tant que l'on n'est pas au bout de la liste */
  while(tmp != NULL)
   {
      /* On affiche */
      printf("%d   %s   %f \n", tmp->eleves.code,tmp-
>eleves.nom,tmp->eleves.moy);
      /* On avance d'une case */
      tmp = tmp->suivant;
   }            }
void main(){
   element* liste=NULL;
   int i;
   donnee valeur;
   for (i=0;i<5;i++){
   printf(" entrer le code de l'eleve ");
   scanf("%d",&valeur.code);
   printf(" entrer le nom de l'eleve ");
   scanf("%s",valeur.nom);
   printf(" entrer le la moyenne de l'eleve ");
   scanf("%f",&valeur.moy);
   liste= ajouterDebut(liste, valeur);
}
printf(" Affichage des données ajoutées au debut de
la liste\n");
 afficherListe(liste);
printf("\n");
    for (i=0;i<5;i++){
   printf(" entrer le code de l'eleve ");
   scanf("%d",&valeur.code);
   printf(" entrer le nom de l'eleve ");
   scanf("%s",valeur.nom);
   printf(" entrer le la moyenne de l'eleve ");
   scanf("%f",&valeur.moy);
   liste= ajouterFin(liste, valeur);
}
printf(" Affichage des données ajoutées au debut et
en fin de la liste\n");
 afficherListe(liste);    getch();   return 0;  }
```