Ce document
appartient à :

M2 : listes chainées

Principes des listes chainées 1
1. Une chaine constituée de maillons 1
Définition 1
Le maillon d'une liste chainée 1
Représenter une liste chainée 1
Le début d'une liste chainée 2
Parcourir une liste chainée 2
Marquer la fin d'une liste chainée 3
2. Trois types de listes chainées 4
Simple 4
Symétrique ou doublement chainée 4
Circulaire simple, double 4
3. Deux implémentations possibles 4
Allocation dynamique de mémoire 4
Allocation contiguë de mémoire 4
4. Les actions sur une liste chainée 6
5. Généricité 6
6. Liste et "tas" 7

Implémenter une liste simple en dynamique 1
1. Structure de données 1
2. Liste vide 1
3. Ajouter 1
Initialiser un élément 1
Ajouter au début 2
Ajouter à la fin 3
4. Insérer 5
5. Parcourir 8
6. Supprimer 9
Supprimer début 9
Supprimer fin 10
Supprimer un élément correspondant à un critère 12
Supprimer tous les éléments correspondants à un critère 14
7. Extraire 16
Extraire début 17
Extraire fin 17
Extraire sur critère 17
8. Détruire une liste 18
9. Copier une liste 18
10. Sauvegarder une liste 19
Ecrire dans un fichier (save) 19
Lire dans un fichier (load) 20
11. Test dans le main() 21

Implémenter une liste circulaire (dynamique) 1
1. Principe 1
2. Liste circulaire dynamique simple 1

Ce document
appartient à :

Structure de données 1
Initialisation élément 2
Ajouter 2
Supprimer 2
Parcourir, afficher 3
Détruire une liste 3
Test dans le main() 4
3. Liste circulaire dynamique symétrique 5
Structure de données 5
Initialiser un élément 5
Ajouter 5
Supprimer 7
Parcourir, afficher 8
Détruire une liste 9
Test dans le main() 9

Principe de généricité en C, données void* 1
1. Principe 1
2. liste simple un seul type de données dans la chaine 1
Structure de données 1
Initialiser des données 2
Ajouter un élément 2
Extraire 2
Parcourir, afficher 2
Détruire la liste 3
Test dans le main() 3

EXERCICES 1
SEQUENCE DE TD TP 1

Principes des listes chainées

1. Une chaine constituée de maillons

Définition

Une liste chainée est simplement une liste d'objet de même type dans laquelle chaque élément contient :

- des informations relatives au fonctionnement de l'application, par exemple des noms et prénoms de personnes avec adresses et numéros de téléphone pour un carnet d'adresse,
- l'adresse de l'élément suivant ou une marque de fin s'il n'y a pas de suivant.
  C'est ce lien via l'adresse de l'élément suivant contenue dans l'élément précédent qui fait la "chaine" et permet de retrouver chaque élément de la liste.

L'adresse de l'objet suivant peut être :

- une adresse mémoire récupérée avec un pointeur.
- un indice de tableau récupéré avec un entier
- une position dans un fichier. Elle correspond au numéro d'ordre de l'objet dans le fichier multipliée par la taille en octet du type de l'objet. Elle est récupérée avec un entier.

Que la liste chainée soit bâtie avec des pointeurs ou des entiers, c'est toujours le terme de pointeur qui est utilisé : chaque élément "pointe" sur l'élément suivant, c'est à dire possède le moyen d'y accéder.

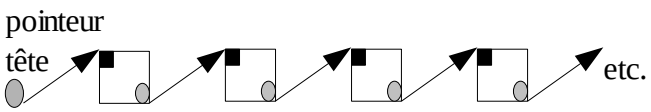
Le maillon d'une liste chainée

Typiquement un élément d'une liste chainée, appelé aussi un maillon, est défini par une structure. Cette structure contient les informations en relation avec les objectifs de l'application et un pointeur sur une structure de même type :

```
typedef stuct maillon{
  // 1 : les datas pour l'application courantes
  char nom[80];

  // 2 : le pointeur pour l'élément suivant
  struct maillon*suiv;
}t_maillon;
```

Représenter une liste chainée

Grâce au pointeur chaque élément peut contenir l'adresse d'un élément et une liste chainée peut se représenter ainsi :



Chaque carré correspond à un maillon de la chaine. Le petit carré noir en haut à gauche correspond à l'adresse en mémoire de l'élément. Le petit rond en bas à droite correspond au pointeur qui "pointe" sur le suivant, c'est à dire qui contient l'adresse de l'élément suivant. Au départ il y a le pointeur de tête qui contient l'adresse du premier élément c'est à dire l'adresse de la chaine.

Le début d'une liste chainée

Deux positions sont très importantes dans une liste chainée : le début et la fin, souvent désignées par "premier et dernier" ou "tête et queue". Sans le premier impossible de savoir où commence la chaine et sans le dernier impossible de savoir où elle s'arrête.

Le début est donné par l'adresse du premier maillon. En général une chaine prend ainsi le nom du premier maillon :

```
t_maillon*premier;
```

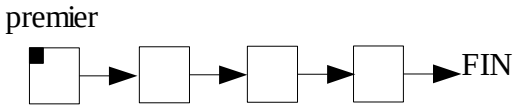
Lorsque la liste est vide le premier maillon prend la valeur NULL. Ensuite, quelque part dans le programme, en fonction d'une action de l'utilisateur, de la mémoire va être allouée à ce premier maillon et des valeurs vont être données aux champs de la structure du maillon, c'est à dire qu'il va être initialisé, par exemple :

```
premier=(t_maillon*)malloc(sizeof(t_maillon));
strcpy(premier->nom,"toto");
premier->suiv=NULL;
```

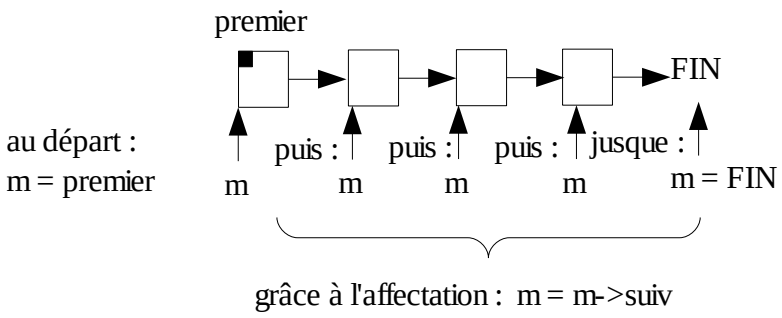
Le champ suiv prend obligatoirement une valeur : soit la valeur de l'élément suivant, soit, s'il n'y en a pas, la valeur de fin de chaine. Par défaut, dans la cas d'un maillon seul non inséré dans une chaine, il prendra la valeur NULL parce qu'elle est facilement repérable par la suite dans le programme.

Parcourir une liste chainée

Admettons que nous possédions une chaine déjà constituée de quatre éléments :



Pour parcourir la chaine nous allons faire avec les adresses de chaque maillon comme Tarzan dans la jungle avec des lianes : passer de l'une à l'autre grâce à un pointeur intermédiaire, ce qui donne par exemple :



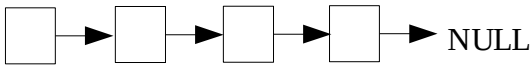
En code source nous avons :

```
t_maillon*m ;

  // au départ prendre l'adresse du premier maillon
  m=premier;

  // tant que m ne contient pas le marquage de fin de chaine
  while (m!=FIN){

    // téventuellement faire qq avec les datas du maillon courant, ici
    // affichage du nom
    printf("nom : %s\n",m->nom);

    // prendre ensuite l'adresse du maillon suivant
    m=m->suiv;
  }
```

Marquer la fin d'une liste chainée

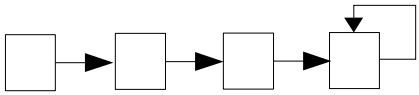
Il y a plusieurs possibilités pour marquer la fin de la chaine :

1) Fin de la chaine lorsque l'adresse de l'élément suivant vaut NULL :



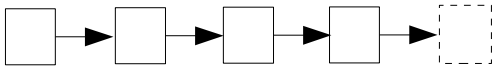
La fin est connue si :
m->suiv = = NULL

2) L'adresse de l'élément suivant est l'adresse de l'élément lui-même :



La fin est connue si :
m->suiv = = m

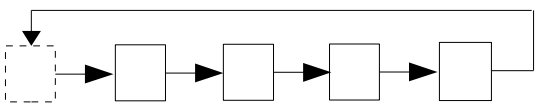
3) La liste est définie par un pointeur de début et un pointeur de fin (au contenu éventuellement indifférent). Le pointeur de fin désigne toujours le dernier élément de la liste et il sert de sentinelle :



La fin est connue si :
m->suiv = = dernier

Le dernier

4) La liste est circulaire ce qui veut dire que le dernier élément contient l'adresse du premier :



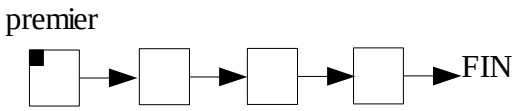
La fin est connue si :
m->suiv = = premier

Le premier

2. Trois types de listes chainées

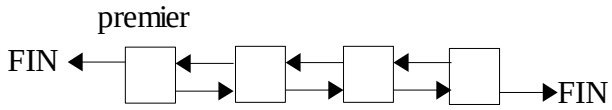
Simple

La liste chainée simple permet de circuler que dans un seul sens, c'est ce modèle :



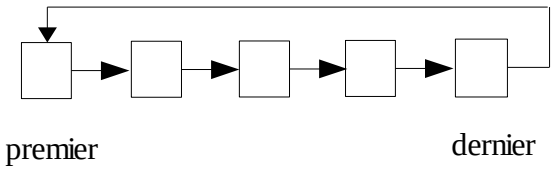
Symétrique ou doublement chainée

Le modèle double permet de circuler dans les deux sens :



Circulaire simple

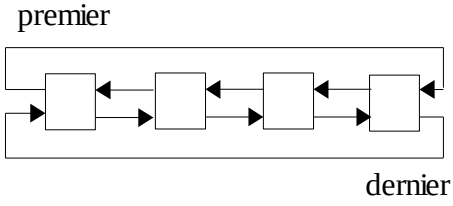
Nous avons déjà mentionné une liste circulaire simple, c'est le modèle ou le dernier prend l'adresse du premier :



Compte tenu de ce bouclage, premier et dernier n'ont plus la même importance, le premier peut être n'importe quel maillon de la chaine et le dernier celui qui le précéde.

Circulaire double

Même principe que précédemment mais avec une circulation possible dans les deux sens :



3. Deux types d'implémentation possibles

Allocation dynamique de mémoire

Les éléments de la liste sont dispersés en mémoire centrale, l'espace est réservé au fur et à mesure de la demande. Typiquement la structure de données pour l'allocation dynamique à la demande est celle que nous avons déjà présentée :

```
typedef struct maillon{
   // datas
   char nom[80];

   // pointeur sur suivant
   struct maillon * suiv;
}t_maillon;
```

Allocation contiguë de mémoire

Les éléments de la liste sont contiguës soit dans un tableau en mémoire centrale, soit sur fichier en accès direct c'est à dire un fichier dont la taille maximum est spécifiée et contrôlée dans le programme. Pour une allocation contiguë de mémoire les pointeurs sont des entiers qui correspondent à des indices de tableau ou des positions dans un fichier. La structure de données doit préciser la taille maximum de la liste et définir deux valeurs :

- une valeur de fin de liste
- une valeur qui indique si une position est libre ou pas

Dans les études proposées plus loin nous avons toujours choisi la taille du tableau comme fin de liste et une macro LIBRE à -1 pour indiquer qu'une position est libre, ce qui donne par exemple

```
#define NBMAX   10  // nombre d'éléments et indicateur fin de liste
#define LIBRE   -1  // indicateur position libre

typedef struct maillon{

  // datas
   char nom[80];

   // pointeur sur l'élément suivant
   int suiv;

}t_maillon;
```

Si la liste est en mémoire centrale, un tableau de NBMAX éléments

```
t_maillon LISTE[NBMAX];
```

Si la liste est sur fichier, un fichier ouvert en lecture-écriture et pouvant contenir NBMAX éléments :

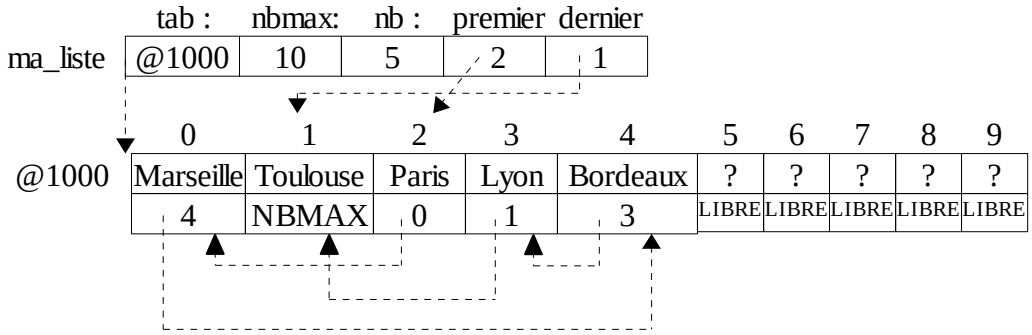
```
FILE*f;
```

La taille de la liste peut-être dynamique et faire l'objet d'une réallocation de mémoire au fur et à mesure des besoins. Dans ce cas la structure de données sera quelque chose comme :

```
struct liste{

   int nbmax, nb;
```

```
    int premier,dernier;
    t_maillon*tab;

}ma_liste;
```

Par exemple voici une liste de nom de ville :



La taille totale du tableau est 10 et il y a 5 éléments dans la liste.
le premier est à l'indice 2 : Paris
le second est à l'indice 0 : Marseille
le troisième est à l'indice 4 : Bordeaux
le quatrième est à l'indice 3 : Lyon
le cinquième est à l'indice 1 : Toulouse

Le parcours s'effectue ainsi :

```
int i;
   for (i=ma_liste.premier; i!=NBMAX; i=ma_liste.tab[i].suiv)
     puts(ma_liste.tab[i].nom);
```

4. Les actions sur une liste chainée

Les actions sont en général toujours les mêmes et toutes ne sont pas toujours nécessaires, en gros il s'agit d'écrire des fonctions pour :

Liste vide : savoir si une liste est vide ou pas.
Ajouter un maillon : ajouter un maillon à la liste, soit au début, soit à la fin.
Parcourir : passer chaque élément en revue dans l'ordre du début vers la fin
extraire : récupérer les données d'un maillon et l'enlever de la chaine
Supprimer : enlever un maillon de la liste sans récupérer ses données
Insérer : ajouter un maillon quelque part dans la chaine
Détruire une liste : désallouer tous les maillons de la liste
Copier une liste : cloner de la chaine
Sauvegarder une liste : copier la liste sur fichier ou récupérer la liste dans le programme

5. Généricité

Dans un programme il peut y avoir plusieurs listes chainées à gérer et chacune avec des données différentes. Dans ce cas il peut être intéressant de concevoir tout ce qui concerne la gestion de la liste chainée indépendamment des données manipulées. Pour ce faire nous pouvons utiliser le pointeur générique void* qui permet d'associer des données de n'importe

quel type avec toutes les fonctions nécessaires à la mise en œuvre des listes chainées.
La structure de données générique devient :

```
typedef struct maillon{

   void*data;               // champ de données générique
   struct maillon*suiv;
}t_maillon;
```

De cette façon il est possible d'associer à un maillon l'adresse de n'importe quel type de structure.

Attention toutefois, le fonctionnement du pointeur générique à des spécificités :
- affectation et cast fonctionnent parfaitement
- allocation malloc() et libération free() fonctionnent également
- indirection, addition et soustraction : *p, p+i et p-i ne fonctionnent qu'avec un cast

Du point de vue méthodologique, le pointeur générique n'est utilisé que comme "transporteur" d'adresses de n'importe quel type d'objet, en revanche il est déconseillé de l'utiliser à la place d'un pointeur typé pour gérer un objet. Accéder ou modifier un objet, allouer ou libérer la mémoire d'un objet doit se faire avec un pointeur déclaré dans le type de l'objet.

6. Listes et "tas"

Un ensemble d'informations et de données sans ordre est appelé un tas. Un tableau d'éléments quels qu'ils soient, dés lors que l'ordre dans lequel ils sont rassemblés est indifférent, peut être considéré comme un tas. Un tas peut aussi prendre la forme d'une liste chainée : des éléments nouveaux sont ajoutés les uns à la suite des autres sans ordre particulier et le tout constitue le tas.

Listes qui ordonnent les données

Toutefois la notion de liste se distingue dans la mesure où souvent une liste fait intervenir un ordre dans les données. Dans une liste les éléments sont ordonnés d'une manière ou d'une autre avec un premier et un dernier. De fait dans une liste chainée même si l'ordre est indifférent il y a toujours un premier et un dernier et il est impossible d'accéder aux éléments sans parcourir la chaîne. C'est une différence importante avec un tableau dans lequel chaque élément est accessible directement sans passer par les autres.

Par exemple, soit un tableau d'entiers :

```
int tab[100];
```

on peut écrire directement :

```
tab[10]=50;
```

Mais il n'y a pas d'équivalent à tab[10] dans une liste chainée. Pour accéder à l'élément 10 il faut partir du premier et passer par tous ceux qui précédent avant d'arriver à l'élément 10. Cette propriété d'ordonnancement d'une liste peut être utilisée pour parcourir un tas. Nous pouvons alors avoir plusieurs listes pour pouvoir parcourir un même tas de plusieurs façons.

Listes sous-ensembles dans un tas

Le fait de pouvoir avoir plusieurs listes pour un même tas permet également de diviser le tas en sous-ensembles, chaque sous ensemble correspondant alors à une liste.
Par exemple dans un carnet d'adresses, admettons que tous les contacts soient conservés dans un tableau. Ce tableau c'est le tas, il rassemble pèle-mêle tous les contacts. Nous pouvons avoir des listes superposées au tas pour que :

- les contacts soient présentées selon plusieurs classements : par ordre alphabétique, par date de rencontre, par adresse etc.
- les contacts soient répartis sur plusieurs listes différentes : liste des contacts professionnels, familiaux, amicaux etc.

La division d'un même tas en sous listes est abordée au chapitre 3 avec l'implémentation de listes simples en mémoire contiguë. Le fait d'avoir une multiplicité de parcours possibles pour un même tas est abordé au chapitre 4 avec des listes extériorisées en tableaux d'indices.

Implémenter une liste simple en dynamique

1. Structure de données

Chaque élément (maillon) de la liste est une structure. Pour l'ensemble des tests réalisés ici notre structure contient en guise de datas un nombre entier et une chaine de caractères. La liste chainée que nous allons réaliser est dynamique alors la structure contient un pointeur sur une structure de même type pour pouvoir construire cette liste à la demande :

```
typedef struct elem{

  int val;
  char s[80];

  struct elem* suiv;  // pointeur sur suivant
}t_elem;
```

L'adresse de la liste proprement dite est donnée par le premier élément, celui à partir duquel tous les autres sont accessibles. Ce premier élément est un pointeur que nous déclarerons dans le main()

```
t_elem*premier=NULL;
```

2. Liste vide

Nous utilisons comme fin de liste la valeur NULL. Si donc premier est NULL c'est que la liste est vide. Eventuellement nous pouvons avoir une petite fonction qui retourne 1 ou 0 selon que la liste est vide ou pas :

```
int liste_vide(t_elem*l)
{
   return (l==NULL);
}
```

3. Ajouter

Ajouter un élément suppose l'initialisation des datas de l'élément et de décider où il est ajouté : au début, à la fin ou inséré dans la liste à une position particulière selon un critère donné (ordre croissant, décroissant, alphabétique etc.).

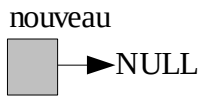
Initialiser un élément

Il est indispensable d'allouer la mémoire pour chaque nouvel élément de la liste. Le mieux est de le faire au moment de l'initialisation des datas, dans la fonction d'initialisation, lorsque les champs de l'élément prennent des valeurs. Pour les tests ici le champ val prend une valeur aléatoire entre 0 et 26 et le champ s prend une chaine de caractères composée d'une seule lettre de l'alphabet choisie au hasard. Par précaution le pointeur suiv est initialisé à NULL. En effet la manipulation des pointeurs est délicate. Pour éviter des bugs il est recommandé de toujours mettre à NULL un pointeur non alloué. A la fin, la fonction retourne l'adresse du nouvel élément alloué et initialisé :

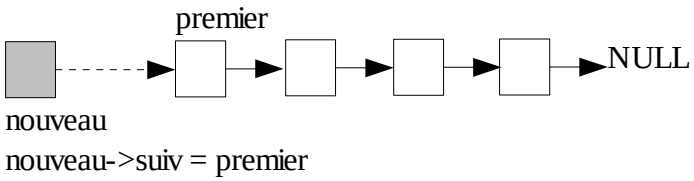
```
t_elem* init()
{
char* n[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","X","Y","Z"};

t_elem*e=malloc(sizeof(t_elem));
   e->val=rand()%26;
   strcpy(e->s,n[rand()%26]);
   e->suiv=NULL;
   return e;
}
```

Ajouter au début

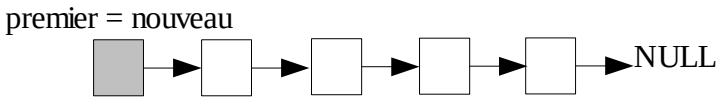
L'objectif de la fonction est d'ajouter un élément déjà initialisé avec la fonction init() :



Le nouvel élément est ajouté au début de la liste, le suivant c'est premier :



Le pointeur "premier" qui donne l'adresse de la liste, ne contient plus l'adresse du début qui est maintenant dans nouveau, premier doit donc prendre cette adresse :



Pour la fonction il y a deux possibilités d'écriture. Soit prendre en paramètre l'adresse de début et utiliser le mécanisme de retour pour renvoyer la nouvelle adresse, soit prendre en paramètre l'adresse "perso" de la variable premier, à savoir passer le pointeur de début par référence. Cette question est présente pour toute les fonctions qui peuvent modifier quelque chose à la liste. Nous présenterons à chaque fois les deux versions.

Première version. La fonction ajout() prend en argument l'adresse du début de la liste, l'adresse du nouvel élément et retourne la nouvelle adresse du début :

```
t_elem* ajout_debut1(t_elem*prem,t_elem*e)
{
   e->suiv=prem;
   prem=e;
   return prem;
}
```

Exemple d'appel, dans le programme ci-dessous une chaine de 10 éléments est fabriquée :

```
int main()
{
t_elem* premier=NULL;
t_elem*nouveau;
 int i;

   for (i=0; i<10; i++){
      nouveau=init();
      premier=ajout_debut1(premier,nouveau);
   }
   return 0;
}
```

Attention à bien initialiser premier sur NULL à la déclaration sinon la chaine n'aura pas de buttoir finale et il ne sera pas possible de repérer sa fin.

Deuxième version. Le mécanisme de retour n'est pas utilisé. Le premier paramètre est un pointeur de pointeur afin de pouvoir prendre comme valeur l'adresse de la variable pointeur qui contient l'adresse du premier élément : &premier

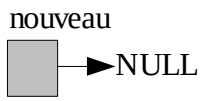
```
void ajout_debut2(t_elem**prem,t_elem*e)
{
   e->suiv=*prem;
   *prem=e;
}
```

Exemple d'appel :

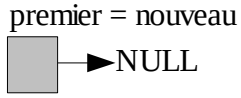
```
int main()
{
t_elem* premier=NULL;
t_elem*nouveau;
 int i;

   for (i=0; i<10; i++){
      nouveau=init();
      ajout_debut2(&premier,nouveau);
   }
   return 0;
}
```

Ajouter à la fin

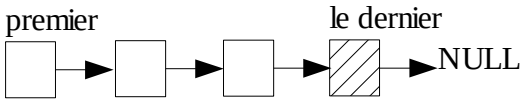
Pour ajouter à la fin d'abord créer un nouveau maillon :



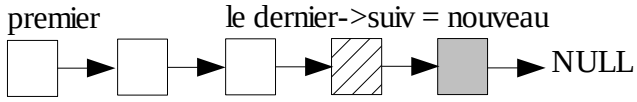
Ensuite deux cas se présentent : la liste est vide ou pas. Si la liste est vide le premier prend la valeur du nouveau :



Sinon se positionner sur le dernier élément de la liste :



et ajouter le nouveau à la suite :



De même que pour l'ajout en tête il y a deux écritures possibles de la fonction : avec ou non passage par référence du pointeur premier.

Première version sans passage par référence. Le paramètre p1 est destiné à recevoir l'adresse de la liste, c'est à dire la valeur du pointeur premier, au moment de l'appel. Le paramètre p2 reçoit l'adresse de l'élément nouveau. La fonction retourne l'élément premier éventuellement modifié, si au départ la liste est vide :

```
t_elem* ajout_fin1(t_elem*prem,t_elem*e)
{
t_elem*n;
   if(prem==NULL)
      prem=e;
   else{
      n=prem;
      while(n->suiv!=NULL)
         n=n->suiv;
      n->suiv=e;
   }
   return prem;
}
```

Le parcours pour trouver le dernier s'effectue avec une boucle while, par l'intermédiaire d'un pointeur n qui prend successivement les adresses de chaque maillon de la chaine depuis le premier au départ jusqu'au dernier à la fin de la boucle. Le dernier est reconnu lorsque le suivant est à NULL, test d'arrêt de la boucle. A la sortie de la boucle n contient l'adresse du dernier maillon de la chaine.

Deuxième version avec passage du pointeur premier par référence. Le paramètre p1 est un pointeur de pointeur afin de pouvoir recevoir l'adresse d'une variable de type pointeur. Le paramètre p2 reçoit l'adresse de l'élément nouveau. Du fait du passage par référence il n'y a pas de retour.

```
void ajout_fin2(t_elem**prem,t_elem*e)
{
t_elem*n;
   if(*prem==NULL)
      *prem=e;
   else{
      n=*prem;
      while(n->suiv!=NULL)
         n=n->suiv;
```

```
        n->suiv=e;
    }
}
```

Exemple d'appel pour les deux fonctions, chaque fonction est appelée une fois sur deux :

```
t_elem* premier=NULL;
t_elem*nouveau;
 int i;

   for (i=0; i<10; i++){
      nouveau=init();
      if (i%2)
         premier=ajout_debut1(premier,nouveau);
      else
          ajout_debut2(&premier,nouveau);
   }
```

Remarque :
Dans la perspective d'ajouter à la fin il serait avantageux d'avoir outre un pointeur premier pour la tête de liste, un second pointeur dernier pour la queue de liste. De ce fait nous aurions toujours ce pointeur sous le coude et il n'y aurait plus besoin de parcourir la liste pour accéder au dernier. L'ajout se fait alors très simplement :

```
   dernier->suiv=nouveau;
   dernier=nouveau;
```

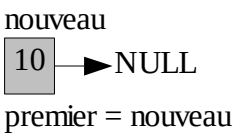
Un exemple de ce type de gestion de liste est étudié plus loin dans la partie sur les files.

4. Insérer

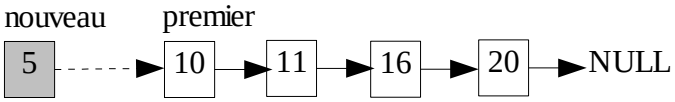
Le fait d'insérer plutôt qu'ajouter un élément dans la liste suppose un classement. Il faut un critère d'insertion, pourquoi ici et pas là ? Par exemple nous allons insérer nos éléments de façon à ce que les valeurs entières soient classées en ordre croissant.

Trois cas sont possibles : la liste est vide, insérer au début avant le premier, chercher la place précédente et insérer après.

1) Si la liste est vide, le nouveau à insérer devient le premier :



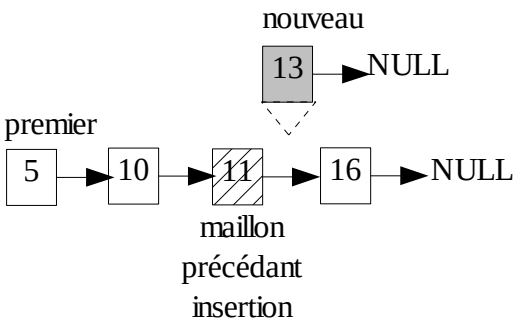
2) La liste n'est pas vide mais il faut insérer au début parce que la nouvelle valeur est inférieure à celle du premier maillon : si nouveau->val < premier->val



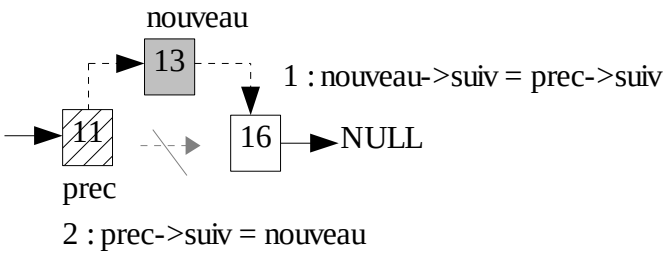
Soit les deux instructions :
nouveau->suiv = premier;

premier=nouveau;

3) Dernier cas, le maillon est à insérer quelque part dans la liste. Pour ce faire il faut chercher la bonne place et conserver l'adresse du maillon qui précède. Par exemple, insérer un nouveau maillon qui contient la valeur entière 13 dans la liste suivante :

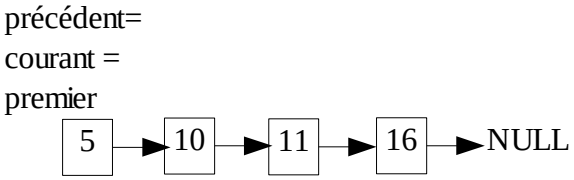


Lorsque le maillon précédent est trouvé, l'insertion est simple :



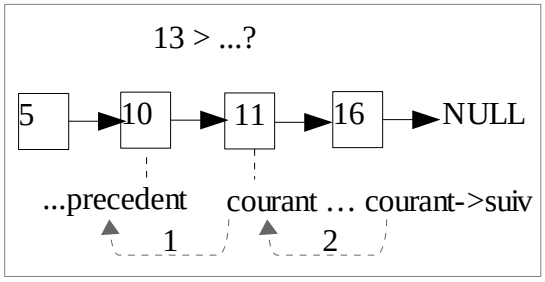
Pour trouver le maillon précédent la liste est parcourue tant que l'élément courant n'est pas NULL et que la valeur de l'élément à insérer est supérieure à la valeur de l'élément courant. Nous avons besoin de deux pointeurs : un pour l'élément courant et un pour l'élément précédent. A chaque tour, avant d'avancer l'élément courant de un maillon, l'adresse de l'élément précédent est sauvegardée :

au départ courant et précédent prennent la valeur de premier :



Ensuite si courant n'est pas NULL, comparaison entre les valeurs entières de courant et nouveau. Si la valeur entière de nouveau est supérieure à la valeur entière de courant, précédent prend l'adresse de courant et courant prend l'adresse du suivant, ainsi de suite jusqu'à trouver l'emplacement, cet emplacement peut être à la fin de la liste, lorsque courant vaut NULL :

Test :
courant != NULL et nouveau->val > courant ->val
si vrai alors :
precedent=courant
courant=courant->suiv



Ce qui donne :

```
courant != NULL et 13 >5 alors
    precedent=courant
    courant=courant->suiv

courant != NULL et 13 >10 alors
    precedent=courant
    courant=courant->suiv

courant != NULL et 13 >11 alors
    precedent=courant
    courant=courant->suiv

courant != NULL et 13 >16   FAUX : fin de la boucle
precedent contient l'adresse de l'élément 11
la liaison peut être effectuée.
```

Si l'élément à insérer rencontre un élément qui a la même valeur que lui, par exemple pour insérer 16, on a alors :

```
courant != NULL et 16 >16   FAUX : fin de la boucle
precedent contient l'adresse de l'élément précédent
la liaison est effectuée en insérant le nouvel arrivant avant.
```

Mais il y a problème si l'élément à insérer doit être insérée en premier soit parce que la liste est vide, soit parce que sa valeur est inférieure à celle du premier. Dans ces deux cas il faut insérer au début de la liste et modifier la tête de la liste, le pointeur premier. De même si la valeur à insérer est égale à celle du premier. L'algorithme se divise donc en deux possibilités :

```
SI   premier==NULL OU element->valeur <= premier->valeur ALORS
    insérer l'élément au début
SINON
    courant = precedent = premier
    TANT QUE  premier != NULL ET element->valeur < courant->valeur ALORS
        precedent=courant
        courant=courant->suiv
    FIN TANT QUE
    element->suiv = courant
    precedent->suiv = element
```

FINSI

Là encore il y a deux version de fonction sans ou avec passage par référence.
Première version sans passage par référence. Même mécanisme de retour et mêmes paramètres que pour les autres fonctions d'ajout, ce qui donne :

```
t_elem* inserer1(t_elem*prem,t_elem*e)
{
t_elem*n,*prec;
   // si liste vide ou ajouter en premier
   if(prem==NULL || e->val <= prem->val){  // attention <=
      e->suiv=prem;
      prem=e;
   }
   else{  // sinon chercher place précédente, insérer après
      n=prec=prem;
      while (n!=NULL && e->val > n->val){
        prec=n;
        n=n->suiv;
      }
      e->suiv=n;
      prec->suiv=e;
   }
   return prem;
}
```

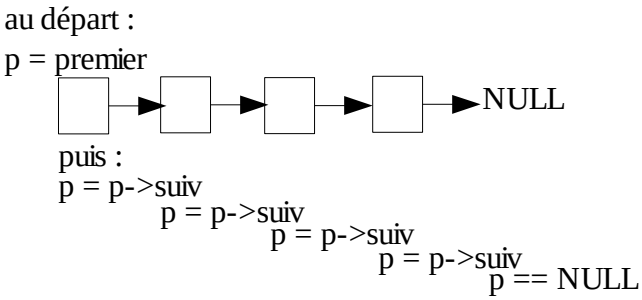
Deuxième version avec passage par référence, mêmes paramètres que pour les autres fonctions d'ajout.

```
void inserer2(t_elem**prem,t_elem*e)
{
t_elem*n,*prec;
   if(*prem==NULL)                     // si liste vide
      *prem=e;
   else if ( e->val<(*prem)->val){     // si premier, ajouter debut
      e->suiv=*prem;
      *prem=e;
   }
   else{  // sinon chercher place
      n=prec=*prem;
      while (n!=NULL && e->val>n->val){
        prec=n;
        n=n->suiv;
      }
      e->suiv=n;
      prec->suiv=e;
   }
}
```

Les appels sont identiques aux autres fonctions d'ajout.

5. Parcourir

Pour parcourir une liste chainée, nous l'avons déjà mentionné, il suffit avec un pointeur de prendre successivement l'adresse de chaque maillon suivant. Au départ le pointeur prend l'adresse du premier élément qui est l'adresse de la liste, ensuite il prend l'adresse du suivant et ainsi de suite :



Pour chaque élément il est possible d'accéder aux datas de l'élément. Dans la fonction ci-dessous les valeurs sont simplement affichées :

```
void parcourir(t_elem*prem)
{
   if (prem==NULL)
      printf("liste vide");
   else
      while(prem!=NULL){
         printf("%d%s--",prem->val,prem->s);
         prem=prem->suiv;
      }
   putchar('\n');
}
```

Remarque :
Attention au fait que le pointeur t_elem*prem déclaré en paramètre de fonction est une variable locale à la fonction. Elle ne dépend pas du contexte d'appel sauf pour la valeur qui lui est passée au moment de l'appel, exemple :

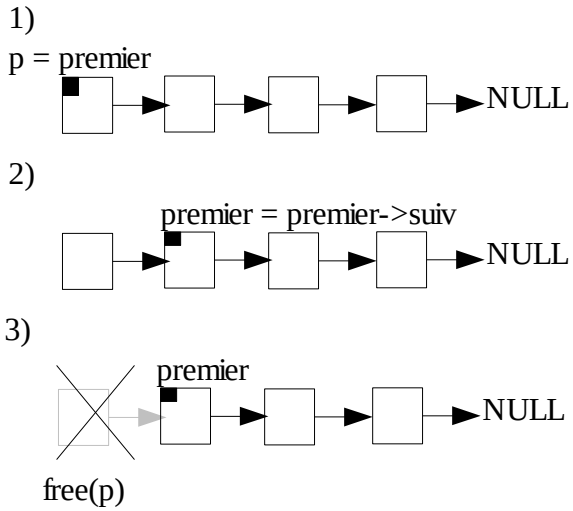
```
int main()
{
t_elem* premier=NULL;
t_elem*nouveau;
 int i;

   // fabriquer une liste de 10 éléments
   for (i=0; i<10; i++){
      nouveau=init();
      ajout_debut2(&premier,nouveau);
   }
   // afficher la liste :
   parcourir(premier);

   return 0;
}
```

6. Supprimer

Comme pour l'ajout d'éléments nous allons avoir trois cas : supprimer au début, à la fin ou dans la liste en fonction d'un critère de sélection. Il faut prendre en compte également que la liste peut être vide avec le premier pointeur à NULL

Supprimer début

La suppression du premier élément suppose de bien actualiser la valeur du pointeur premier qui indique toujours le début de la liste, ce qui donne si la liste n'est pas vide :



La modification du pointeur premier doit effectivement être répercutée sur le premier pointeur de la liste. De même que pour les fonctions d'ajout il faut retourner la nouvelle adresse de début ou utiliser un passage par référence afin de communiquer cette nouvelle adresse du début au contexte d'appel.

Version de la fonction avec mécanisme de retour :

```
t_elem* supprimer_debut1(t_elem*prem)
{
t_elem*n;
   if(prem!=NULL){
      n=prem;
      prem=prem->suiv;
      free(n);
   }
   return prem;
}
```

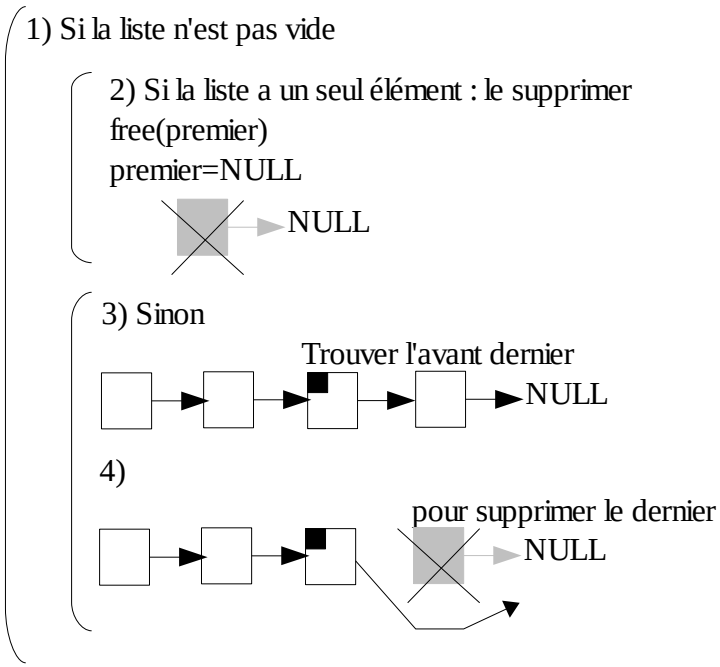
Version un passage par référence :

```
void supprimer_debut2(t_elem**prem)
{
t_elem*n;
   if(*prem!=NULL){
      n=*prem;
      *prem=(*prem)->suiv;
      free(n);
   }
}
```

Supprimer fin

La suppression à la fin suppose de se positionner sur l'avant dernier maillon puis de supprimer le

suivant et de mettre suivant à NULL. Il faut que la liste ne soit pas vide et traiter le cas ou il n'y a qu'un seul maillon dans la liste, le premier.



La première version avec mécanisme de retour donne :

```
t_elem* supprimer_fin1(t_elem*prem)
{
t_elem*n;
   if (prem!=NULL){   // si la liste n'est pas vide
      if(prem->suiv==NULL){ // si il n'y a qu'un seul maillon
         free(prem);
         prem=NULL;
      }
      else{  // si il y a au moins deux maillons
         n=prem; // à partir du premier,
          // regarder toujours le suivant du suivant pour accéder à
          // l'avant dernier
         while(n->suiv->suiv!=NULL)
            n=n->suiv;
         free(n->suiv); // une fois l'avant dernier trouvé,
         n->suiv=NULL;  // supprimer le suivant
      }
   }
   return prem;
}
```

Et la seconde avec passage par référence du pointeur de tête de liste :

```
void supprimer_fin2(t_elem**prem)
{
t_elem*n;
   if (*prem!=NULL){
      if((*prem)->suiv==NULL){
         free(*prem);
         *prem=NULL;
```

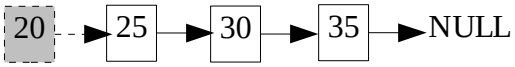
```
        }
        else{
           n=*prem;
           while(n->suiv->suiv!=NULL)
              n=n->suiv;
           free(n->suiv);
           n->suiv=NULL;
        }
     }
}
```

Supprimer un élément correspondant à un critère

L'objectif cette fois est de supprimer un maillon quelconque de la liste en fonction d'un critère donné. Le critère ici est que le champ val de l'élément soit égal à une valeur donnée. La recherche s'arrête si un élément répond au critère et nous supprimons uniquement le premier élément trouvé s'il y en a un. Comme pour les suppressions précédentes la liste ne doit pas être vide. Il faut prendre en compte le cas où l'élément à supprimer est le premier de la liste et pour tous les autre éléments il faut disposer de l'élément qui précéde celui à supprimer afin de pouvoir reconstruire la chaine.

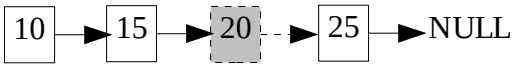
Prenons par exemple 20 comme valeur à rechercher. Si un maillon a 20 pour valeur entière il est supprimé et la recherche est terminée. Le principe est le suivant :

1) si c'est le premier élément :

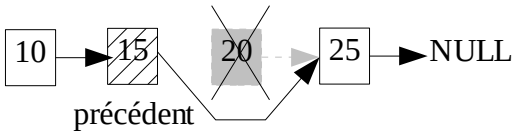


supprimer le premier comme indiqué plus haut

2) sinon trouver dans la liste un maillon où il y a 20 :



3) le supprimer, ce qui suppose d'avoir l'adresse du précédent afin de rétablir la chaine avec le suivant du maillon 20 :



La fonction reçoit en paramètre l'adresse du début de la liste et la valeur entière critère de sélection de l'élément à supprimer. Le premier élément trouvé qui a cette valeur dans son champ val est supprimé de la liste. Comme l'adresse de la liste est susceptible d'être modifiée, la fonction retourne cette adresse afin que cette modification puisse être récupérée dans le contexte d'appel. L'algorithme est le suivant :

- si la liste n'est pas vide : prem != NULL
  - s'il s'agit d'enlever le premier : prem->val==val
    - l'adresse du premier est gardée dans une variable n : n=prem
    - le premier devient le suivant du premier : prem=prem->suiv
    - l'adresse sauvée en n est déallouée : free(n)
  - si ce n'est pas le premier, c'est peut-être le suivant...
    - le précédent est le premier : prec=prem

- le maillon courant à regarder est le suivant du premier : n = prem->suiv
- tous les maillons de la liste vont ainsi être regardés et à chaque fois le précédent est sauvé, voici juste le parcours de la liste :

```
tant que n !=NULL faire
    prec=n
    n=n->suiv
fin tant que
```

- pendant le parcours, pour chaque maillon n regarder si la valeur val est trouvée :
  si (n->val == val) vrai
  - si vrai alors créer le lien entre le précédent et le suivant de n : prec->suiv=n->suiv
  - effacer l'élément n : free(n)
  - provoquer la fin de la boucle avec un break;
- à la fin retourner l'adresse de la tête de liste éventuellement modifiée

Ce qui donne la fonction suivante :

```
t_elem* critere_supp_un1(t_elem*prem,int val)
{
t_elem*n,*prec;
   if(prem!=NULL){
      if(prem->val==val){// si premier
         n=prem;
         prem=prem->suiv;
         free(n);
      }
      else{              // sinon voir les autres
         prec=prem;
         n=prem->suiv;
         while(n != NULL ){
            if (n->val==val){
               prec->suiv=n->suiv;
               free(n);
               break;
            }
            prec=n;
            n=n->suiv;
         }
      }
   }
   return prem;
}
```

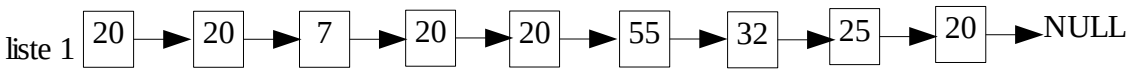
Une version avec passage par référence de la tête de liste, sans mécanisme de retour :

```
void critere_supp_un2(t_elem**prem,int val)
{
t_elem*n,*prec;
   if(*prem!=NULL){
      if((*prem)->val==val){// si premier
         n=*prem;
         *prem=(*prem)->suiv;
         free(n);
      }
      else{              // les autres
         prec=*prem;
         n=(*prem)->suiv;
         while(n != NULL ){
```

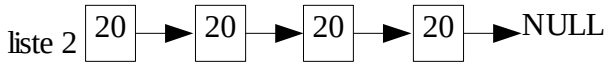
```
            if (n->val==val){
               prec->suiv=n->suiv;
               free(n);
               break;
            }
            prec=n;
            n=n->suiv;
         }
      }
   }
}
```

Supprimer tous les éléments correspondants à un critère

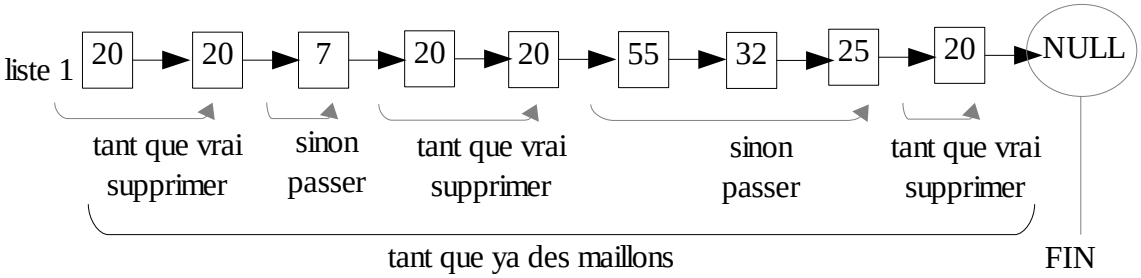
La fonction précédente ne peut supprimer qu'un élément, le premier rencontré qui répond au critère. L'objectif maintenant est de supprimer tous les éléments qui répondent au critère. L'algorithme est sensiblement le même avec deux différences toutefois. Si un élément est trouvé, il est supprimé et la boucle ne s'arrête pas, tous les éléments sont évalués et chaque élément qui répond au critère est supprimé. Cela pose un problème de plus : le cas ou plusieurs éléments à supprimer sont les uns à la suite des autres. Admettons que la valeur critère est 20 nous voulons supprimer tous les maillons à 20 de la liste :



Nous devons envisager qu'il peut y avoir des suites de maillons à supprimer. Nous pourrions même avoir une liste comme :



Il faut procéder en deux temps : supprimer tous les premiers et ensuite traiter le reste de la liste. Dans les deux cas, tant que l'élément courant répond au critère il est supprimé immédiatement et remplacé par l'élément suivant. C'est à dire que nous ne sommes obligés d'avancer dans la liste que pour passer les éléments qui ne sont pas à supprimer.

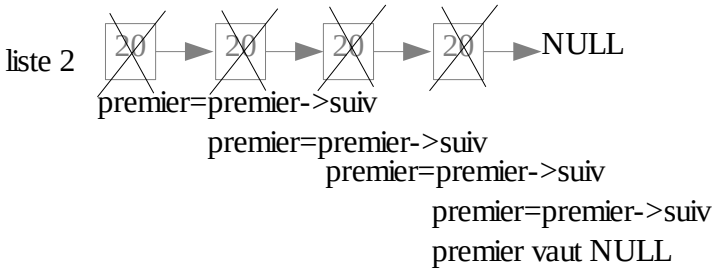


Pour supprimer tous les premiers s'il y a lieu, le pointeur premier prend successivement les adresses des maillons suivants : premier=premier->suiv tant que premier n'est pas NULL et que premier->val est égal à val, la valeur entière critère de sélection. A chaque fois l'adresse contenue dans premier est passée à un pointeur auxiliaire et après que premier ait pris la valeur de l'adresse suivante, cette adresse est libérée. Ce qui donne l'algorithme :

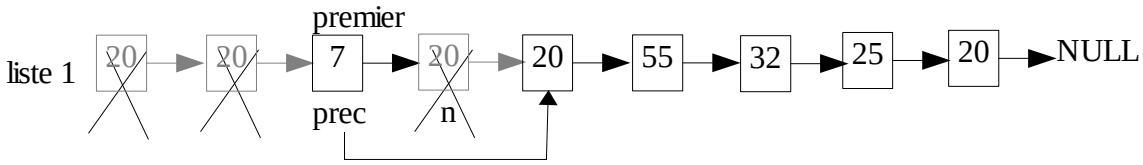
```
tant que premier!=NULL && premier->val==val faire
   ptr=premier
```

premier=premier->suiv
free(ptr)
fin tant que

Une fois cette étape terminée, c'est sûr, il n'y a plus de maillon répondant au critère au début de la liste. Éventuellement la liste peut être vide et premier avoir pris la valeur NULL.
C'est ce qui se passe dans le cas de la liste 2 :



La seconde étape commence donc par tester si la liste n'est pas vide c'est à dire si premier != NULL. Si oui, maintenant pour opérer une suppression nous avons besoin d'avoir toujours l'adresse du maillon précédent un maillon à supprimer. Par exemple dans le cas de la liste 1



Pour supprimer le maillon courant n il faut pouvoir établir le lien entre le précédent de n et le suivant de n. Un pointeur "prec" conserve toujours l'adresse du précédent de n au fur et à mesure de l'avancée de n dans la liste.

Au départ pour la seconde étape nous savons que le premier n'est pas à supprimer et nous initialisons le pointeur prec pour le maillon précédent et le pointeur n pour le maillon courant de cette façon : prec=premier et n=premier->suiv.
Le premier niveau de boucle concerne l'avancée jusqu'à la fin de la liste : tant que n!=NULL
Le second niveau de boucle concerne les cas de suppression. Nous supprimons tant que n != NULL et que le maillon courant répond au critère : n->val==val.
Si le maillon courant n'est pas à supprimer, alors passer au suivant. Le précédent prend la valeur du courant et le courant passe au suivant : prec=n, n=n->suiv

Cet algorithme suppose que l'adresse de la liste peut être modifiée et il est indispensable de le signifier au contexte d'appel de la fonction. La première version de la fonction utilise le mécanisme de retour :

```
t_elem*critere_supp_all1(t_elem*prem,int val)
{
t_elem*n,*sup,*prec;

   // supprimer au début
   while(prem!=NULL && prem->val==val){
      sup=prem;
      prem=prem->suiv;
      free(sup);
```

```
    }
    // les suivants
    if (prem!=NULL){
       prec=prem;
       n=prem->suiv;
       while (n!=NULL){
          while(n!=NULL && n->val==val){
             sup=n;
             n=n->suiv;
             prec->suiv=n;
             free(sup);
          }
          if (n!=NULL){
             prec=n;
             n=n->suiv;
          }
       }
    }
    return prem;
}
```

La seconde version un passage par référence :

```
void critere_supp_all2(t_elem**prem,int val)
{
t_elem*n,*sup,*prec;

   // supprimer au début
   while(*prem!=NULL && (*prem)->val==val){
      sup=*prem;
      *prem=(*prem)->suiv;
      free(sup);
   }
   // les suivants
   if (*prem!=NULL){
      prec=*prem;
      n=prec->suiv;
      while (n!=NULL){
         while(n!=NULL && n->val==val){
            sup=n;
            n=n->suiv;
            prec->suiv=n;
            free(sup);
         }
         if (n!=NULL){
            prec=n;
            n=n->suiv;
         }
      }
   }
}
```

7. Extraire un élément

Un élément extrait de la liste est un élément retiré de la liste mais non effacé. Ses données sont conservées en mémoire. L'élément extrait peut par exemple être inséré dans une autre liste. Comme pour les ajouts et suppressions, l'extraction peut se faire au début, à la fin ou n'importe où dans la

liste selon un critère donné. La différence c'est qu'il y a deux sorties à la fonction : le maillon extrait et éventuellement la tête de la liste qui peut s'en trouver modifiée. Le mécanisme de retour seul n'est pas suffisant et au moins un passage par référence est nécessaire afin d'éviter d'utiliser des variables globales qui ne seraient pas justifiées dans ce cas.

Extraire début

Extraire un élément au début revient à supprimer le premier élément de la liste et, sans le désallouer, retourner son adresse au contexte d'appel. Nous avons opté pour passer la tête de liste par référence et utiliser le mécanisme de retour pour le maillon extrait.

Le principe est le suivant :
- si la liste n'est pas vide
  - récupérer l'adresse du premier
  - le premier prend l'adresse du suivant
- retourner l'adresse du maillon extrait ou NULL si liste vide

Ce qui donne la fonction suivante :

```
t_elem* extraire_debut(t_elem**prem)
{
t_elem*n=NULL;
   if(*prem!=NULL){
      n=*prem;
      *prem=(*prem)->suiv;
   }
   return n;
}
```

Extraire fin

Pour extraire à la fin, si la liste n'est pas vide, deux cas se présentent. Le cas où la liste n'a qu'un maillon et le cas où il y en a plusieurs. S'il n'y en a qu'un, le premier, son adresse est retournée, puis la liste est vidée et premier passe à NULL. Sinon il faut se positionner sur l'avant dernier de la liste afin de pouvoir retourner l'adresse du dernier et de clore à la place la liste avec la valeur NULL. Pour ce faire nous utilisons un pointeur prec qui est amené à la position prec->suiv->suiv==NULL, le dernier est alors récupéré en n=prec->suiv et ensuite prec->suiv passe à NULL, ce qui donne :

```
t_elem* extraire_fin(t_elem**prem)
{
t_elem*n,*prec;
   n=*prem;
   if(n!=NULL){ // si liste non vide
      if (n->suiv==NULL)// si un seul élément dans la liste
         *prem=NULL;
      else{// si au moins deux éléments dans la liste
         prec=n;
         while(prec->suiv->suiv!=NULL)// chercher avant dernier
            prec=prec->suiv;
         n=prec->suiv;
         prec->suiv=NULL;
      }
   }
   return n;
}
```

Extraire sur critère

Pour extraire sur critère, reprendre l'algorithme de la suppression sur critère d'un seul élément de la liste. Retourner le maillon sélectionné sans libérer la mémoire.

8. Détruire une liste

Pour détruire une liste il faut parcourir la liste, récupérer l'adresse du maillon courant, passer au suivant et désallouer l'adresse précédente récupérée. Au départ bien vérifier que la liste n'est pas vide. La tête de liste est modifiée, elle doit à l'issue passer à NULL. Il y a les deux possibilités habituelles : retourner la valeur NULL à la fin et la récupérer avec le pointeur de tête premier dans le contexte d'appel ou passer la tête de liste, ce pointeur premier par référence.

La première version donne :

```
t_elem* detruire_liste1(t_elem*prem)
{
t_elem*n;
   while(prem!=NULL){
      n=prem;
      prem=prem->suiv;
      free(n);
   }
   return NULL;
}
```

La seconde avec passage par référence :

```
void detruire_liste2(t_elem**prem)
{
t_elem*n;
   while(*prem!=NULL){
      n=*prem;
      *prem=(*prem)->suiv;
      free(n);
   }
   *prem=NULL;
}
```

9. Copier une liste

Copier une liste suppose de parcourir la liste à copier et de construire en parallèle une autre liste. Pour chaque élément de la liste à copier il faut allouer un élément pour la copie et relier les éléments de la copie entre eux. Il faut également veiller à conserver l'adresse du début de la copie. La fonction copie ci-dessous reçoit en paramètre l'adresse d'une liste à copier, ensuite :

- vérifier si la liste est NULL, si oui la copie aussi
- si la liste à copier n'est pas NULL, allouer le premier élément, tête et adresse de la nouvelle liste. Conserver cette adresse jusqu'à la fin et la retourner.
- ensuite, tant que nous ne sommes pas à la fin de la liste : prem != NULL
  - copier le contenu du maillon courant de la liste à copier prem dans le maillon courant n de la copie : *n=*prem

- passer au maillon suivant de la liste à copier : prem=prem->suiv
- s'il n'est pas NULL et uniquement dans ce cas :
  - allouer le maillon suivant de la copie : n->suiv = malloc(...)
  - le maillon courant de la copie devient le suivant juste alloué : n=n->suiv

- Pour finir retourner l'adresse du début de la copie.

Ce qui donne la fonction :

```
t_elem* copier_liste(t_elem*prem)
{
t_elem*l2,*n;

   if(prem==NULL)
      l2=NULL;
   else{
      l2=malloc(sizeof(t_elem));
      n=l2;
      while(prem!=NULL){
         *n=*prem;
         prem=prem->suiv;
         if (prem!=NULL){
            n->suiv=malloc(sizeof(t_elem));
            n=n->suiv;
         }
      }
   }
   return l2;
}
```

10. Sauvegarder une liste

Ecrire dans un fichier (save)

L'objectif est de sauver une liste chainée dynamique dans un fichier binaire, c'est à dire en langage C en utilisant la fonction standart fwrite(). Le principe est simple : si la liste n'est pas vide, ouvrir un fichier en écriture et copier dedans dans l'ordre où il se présente chaque maillon de la liste, ce qui donne :

```
// en paramètre passer l'adresse d'une liste au moment de l'appel
void sauver_liste(t_elem*prem)
{
FILE*f;
   // si liste non vide
   if(prem!=NULL){

      //ouvrir un fichier binaire en écriture : suffixe b
      if((f=fopen("save liste.bin","wb"))!=NULL){

         // parcourir la liste jusque fin
         while(prem!=NULL){
            fwrite(prem,sizeof(t_elem),1,f); // copier chaque maillon
            prem=prem->suiv;                 // passer au maillon suivant
         }
         fclose(f);   // fermer le fichier
      }
      else
```

```
            printf("erreur création fichier\n");
      }
      else
         printf("pas de sauvegarde pour une liste vide\n");
   }
```

Pour sauver les données d'une liste chainée dynamique dans un fichier texte, il faut passer chaque maillon en revue et à chaque fois copier le contenu de chaque champ dans le fichier. Par exemple un maillon par ligne avec un séparateur qui délimite chaque champ. Le séparateur peut être un caractère de ponctuation comme le point virgule. Les maillons arrivent dans l'ordre de la liste et les données sont recopiées au fur et à mesure.

Lire dans un fichier (load)

Pour récupérer dans le programme une liste chainée dynamique préalablement sauvegardée dans un fichier binaire, il est nécessaire de reconstruire la liste au fur et à mesure. En effet les adresses contenues dans les pointeurs ne sont plus allouées et il faut réallouer toute la liste La fonction ci-dessous suppose qu'il y a au moins un maillon de sauvegardé dans le fichier et il ne doit pas y avoir de sauvegarde de liste vide (c'est à dire juste une création de fichier avec rien dedans). La fonction retourne l'adresse du début de la liste, c'est à dire l'adresse du premier élément, la tête de liste.

Tout d'abord, ouvrir le fichier dans lequel il y a la liste sauvegardée, ensuite :
- allouer la mémoire pour le premier maillon prem
- copier le premier maillon sauvé dans le fichier dans le premier maillon recréé prem avec la fonction fread()

Après cette première étape, tant qu'il y a des éléments à récupérer dans le fichier :
- récupérer dans une variable t_elem e les valeurs du maillon suivant : fread(&e, ...)
- allouer le suivant p->suiv du maillon courant p : p->suiv=malloc(...)
- le maillon suivant devient maillon courant : p=p->suiv
- copier les valeurs récupérée en e dans le maillon courant : *p=e;
- mettre systématiquement à NULL le maillon suivant : p->suiv=NULL, ceci afin d'avoir de façon simple NULL à la fin

Fermer le fichier
Retourner l'adresse de la tête de liste, adresse du premier élément.

Ce qui donne la fonction :

```
t_elem* load_liste()
{
FILE*f;
t_elem*prem=NULL,*p,e;
   if((f=fopen("save liste.bin","rb"))!=NULL){
      prem=malloc(sizeof(t_elem));
      fread(prem,sizeof(t_elem),1,f);
      p=prem;
      while(fread(&e,sizeof(t_elem),1,f)){
         p->suiv=malloc(sizeof(t_elem));
         p=p->suiv;
         *p=e;
         p->suiv=NULL;
      }
      fclose(f);
   }
   else
      printf("erreur ou fichier inexistant");
   return prem;
```

```
}
```

11. Test dans le main()

Menu

Toutes les actions présentées ci-dessus sont regroupées dans le menu suivant présenté à l'utilisateur :

```
int menu()
{
int res=-1;
   printf(  "1  : ajout debut\n"
            "2  : ajout fin\n"
            "3  : inserer en ordre croissant\n"
            "4  : supprimer debut\n"
            "5  : supprimer fin\n"
            "6  : supprimer critere un seul\n"
            "7  : supprimer critere all \n"
            "8  : extraire debut\n"
            "9  : extraire fin\n"
            "10 : tests save and load\n"
            "11 : parcourir\n"
            "12 : Detruire liste\n"
            "13 : copier liste\n"
            "0  : Creer une liste de n objets\n");
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

Appels correspondants

Selon le choix de l'utilisateur les actions sont déclenchées via un switch(). Lorsqu'il y a deux versions pour une fonction un petit mécanisme permet d'alterner l'appel de l'une ou de l'autre. (Un nombre est augmenté de un à chaque appel, s'il est impair la première fonction est appelée s'il est pair c'est la seconde). L'objectif est ici de pouvoir tester toutes les fonctions exposées plus haut :

```
int main()
{
t_elem*prem=NULL;
t_elem*e=NULL,*l2=NULL;
int fin=0,choix;
int flag=0;
   //srand(time(NULL));
   while (!fin){
      choix=menu();
      switch(choix){
         case 1 :// ajout début
            e=init();
            if(++flag%2)
               prem=ajout_debut1(prem,e);
            else
               ajout_debut2(&prem,e);
            parcourir(prem);
```

```
            break;

        case 2 :// ajout fin
            e=init();
            if(++flag%2)
               prem=ajout_fin1(prem,e);
            else
               ajout_fin2(&prem,e);
            parcourir(prem);
            break;

        case 3 :
            e=init();
            if(++flag%2)
               prem=inserer1(prem,e);
            else
               inserer2(&prem,e);
            parcourir(prem);
            break;

        case 4 :// supprimer debut
            if(++flag%2)
               prem=supprimer_debut1(prem);
            else
               supprimer_debut2(&prem);
            parcourir(prem);
            break;

        case 5 :// supprimer fin
            if(++flag%2)
               prem=supprimer_fin1(prem);
            else
               supprimer_fin2(&prem);
            parcourir(prem);
            break;

        case 6 :// supprimer un seul élément selon un critère
            prem=supp_critere_un(&prem);
            parcourir(prem);
            break;

        case 7 :// supprimer tous les éléments selon un critère
            prem=supp_critere_all(&prem);
            parcourir(prem);
            break;
        case 8 :// extraire début (tête)
            if((e = extraire_debut(&prem))!=NULL){
               printf("extrait : %d%s\n",e->val,e->s);
               free(e);
            }
            else
               printf("liste vide\n");
            parcourir(prem);
            break;
        case 9 : //extraire à la fin
            if((e = extraire_fin(&prem))!=NULL){
               printf("extrait : %d%s\n",e->val,e->s);
               free(e);
            }
            else
               printf("liste vide\n");
```

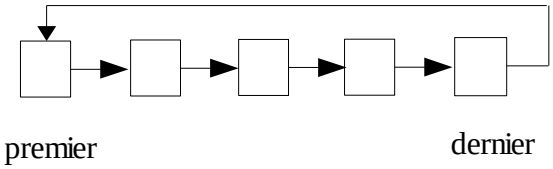
```
            parcourir(prem);
            break;
         case 10 : //test save et load
            sauver_liste(prem); // sauver liste courante
            detruire_liste2(&prem);// détruire la liste
            prem=load_liste(); // loader la liste
            parcourir(prem);
            break;
         case 11 : //parcourir
            parcourir(prem);
            break;
         case 12 : // détruire liste
            if(++flag%2)
               prem=detruire_liste1(prem);
            else
               detruire_liste2(&prem);
            parcourir(prem);
            break;
         case 13 : // copier une liste
            l2= copier_liste(prem);
            parcourir(prem);
            parcourir(l2);
            break;
         case 0 :
            creer_liste(&prem);
            parcourir(prem);
            break;
         default : fin=1;
      }

   }
   detruire_liste2(&prem);
   detruire_liste2(&l2);
   return 0;
}
```

Implémenter une liste circulaire

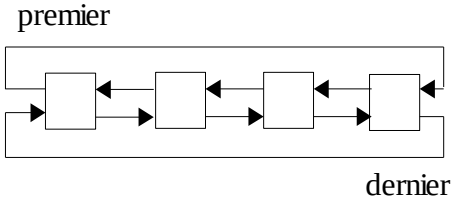
1. Principe

Nous l'avons déjà indiqué au chapitre 1, une liste chainée circulaire est une liste dans laquelle la fin pointe sur le début :

Liste circulaire simple :



Liste circulaire symétrique, chainée dans les deux sens :



Il y a deux différences du fait de ce bouclage avec des listes non circulaires :
- le premier peut être n'importe quel maillon de la liste, c'est l'adresse contenue dans un pointeur qui peut changer l'adresse du maillon qu'il contient, le pointeur courant
- la fin de la chaine c'est en partant de l'adresse contenue dans pointeur courant lorsque se retrouve cette adresse.

La valeur NULL n'est plus un indicateur de fin. C'est vrai pour les deux sens, pour une liste simple et pour une liste symétrique. C'est à partir de ça qu'il faut construire une chaine circulaire.

2. liste circulaire dynamique simple

Les différences avec une liste simple non circulaire sont :
- L'initialisation du pointeur suiv. Par défaut chaque élément pointe sur lui-même.
- L'ajout se fait après l'adresse contenue dans le pointeur courant et non au début ou à la fin
- La suppression également.
- Le parcours se fait à partir de n'importe quel adresse de maillon contenue dans le pointeur courant jusqu'à retrouver cette même adresse.

Structure de données

L'élément est le même que pour une liste dynamique non circulaire :

```
// structure de données
typedef struct elem{
  // les données
```

```
  int val;
  char s[80];
  // pour construire la liste
  struct elem*suiv;
}t_elem;
```

Nous utilisons dans le test le tableau global S de chaines de caractères qui donne l'alphabet :

```
char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

Initialisation élément

Premier changement avec une liste non circulaire, l'élément par défaut pointe sur lui-même :

```
t_elem* init_elem(int val, char s[])
{
t_elem*e;
   e=(t_elem*)malloc(sizeof(t_elem));
   e->val=val;
   strcpy(e->s,s);
   e->suiv=e;     // pas de NULL pointe sur lui-même
   return e;
}
```

Ajouter

Pour ajouter nous avons besoin du pointeur courant qui fait figure de premier. La valeur de sont suivant va changer c'est pourquoi il est passé par référence afin de ne pas perdre la modification. L'algorithme est le suivant :

- Si courant est NULL c'est que la liste est vide dans ce cas le nouvel élément donne son adresse à courant.
- Sinon, il y a au moins un élément dans la liste, peut-être plusieurs. Alors le nouvel élément prend pour suivant le suivant de l'élément courant et l'élément courant prend pour suivant le nouvel arrivant.

Ce qui donne :

```
void ajout_suiv(t_elem**cour,t_elem*e)
{
   if (*cour==NULL)
      *cour=e;
   else{
      e->suiv=(*cour)->suiv;
      (*cour)->suiv=e;
   }
}
```

Supprimer

De même que pour ajouter, pour supprimer il nous faut le pointeur courant qui fait figure de premier. Sa valeur de suivant va changer et il est passé par référence afin de conserver la modification dans le contexte d'appel. Algorithme :

- Tout d'abord, vérifier que la liste n'est pas vide
- Ensuite récupérer l'adresse du suivant à supprimer

- Modifier le suivant du courant qui devient le suivant de l'élément à supprimer
- Si l'élément à supprimer est le courant, c'est à dire s'il n'y a qu'un élément dans la liste, alors mettre courant à NULL pour indiquer liste vide.
- Dans tous les cas libérer la mémoire de l'élément supprimé.

Ce qui fait :

```
void supp_suiv(t_elem**cour)
{
t_elem*e;
   if (*cour!=NULL){
      e=(*cour)->suiv;
      (*cour)->suiv=e->suiv;
      if (*cour==e)
         *cour=NULL;
      free(e);
   }
}
```

Parcourir, afficher

Si la liste n'est pas vide, pour parcourir la liste à partir du maillon courant, nous avons besoin de conserver l'adresse contenue dans ce maillon courant afin de pouvoir la reconnaître et savoir que le parcours est terminé. Nous n'utilisons pas le pointeur cour pour ce parcours mais un autre pointeur p qui prend la valeur de cour au départ. La valeur de cour ne bouge donc pas. Ici nous avons opté pour une boucle infinie :

- Premièrement nous affichons les données,
- ensuite nous regardons si la fin de la liste est arrivée, c'est à dire si p->suiv == cour,
  - si oui un break provoque une sortie immédiate de la boucle,
- si non nous passons à l'élément suivant.

Ce qui donne :

```
void affiche(t_elem*cour)
{
t_elem*p;
   if (cour==NULL)
      printf("liste vide\n");
   else{
      p=cour;
      while(1){
         printf("%d%s--",p->val,p->s);
         if (p->suiv==cour)
            break;
         p=p->suiv;
      }
      putchar('\n');
   }
}
```

Détruire une liste

Pour détruire la liste nous allons parcourir autrement la liste. Si la liste n'est pas vide, nous partons du second élément jusqu'à retrouver le premier. L'adresse de chaque élément est supprimée via un pointeur intermédiaire sup. A l'issue de la boucle il ne reste plus que le pointeur courant et l'adresse qu'il contient est à son tour libérée :

```
void detruire_liste(t_elem**cour)
{
t_elem*p,*sup;
   if (*cour!=NULL){
      p=(*cour)->suiv;
      while (p!=*cour){
         sup=p;
         p=p->suiv;
         free(p);
      }
      free(*cour);
      *cour=NULL;
   }
}
```

Test dans le main()

Il y a deux actions possibles, ajouter ou supprimer :

```
int menu()
{
int res=-1;
   printf(  "1  : ajouter\n"
            "2  : supprimer\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

Selon le choix de l'utilisateur les actions sont exécutées. A chaque fois la liste est affichée. La fonction pour détruire une liste est appelée à la sortie du programme (elle pourrait aussi être utilisée dans le switch) :

```
int main()
{
int fin=0;
t_elem*cour=NULL,*e;

   while(!fin){
      switch(menu()){
         case 1 :
            e=init_elem(rand()%26,S[rand()%26]);
            ajout_suiv(&cour,e);
            affiche(cour);
            break;
         case 2 :
            supp_suiv(&cour);
            affiche(cour);
            break;
         default : fin = 1;
      }
   }
   detruire_liste(&cour);
   return 0;
}
```

3. liste circulaire dynamique symétrique

Nous allons maintenant ajouter à l'étude précédente la possibilité d'une circulation dans les deux sens pour notre liste.

Structure de données

Seule modification par rapport à l'étude précédente, il faut deux pointeurs l'un pour l'élément suivant et l'autre pour l'élément précédent :

```
typedef struct elem{
  // les données
  int val;
  char s[80];
  // pour construire la liste
  struct elem*suiv;
  struct elem*prec;
}t_elem;
```

Nous utilisons toujours ce tableau de chaines de caractères pour initialiser les éléments :

```
char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

Initialiser un élément

Même principe que pour une liste circulaire simple, les deux pointeurs par défaut pour sur l'élément lui-même :

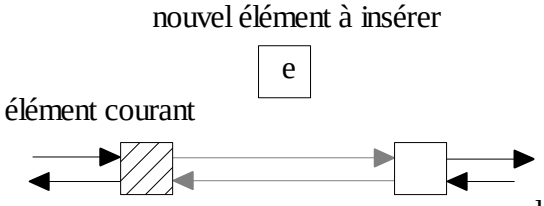
```
t_elem* init_elem(int val, char s[])
{
t_elem*e;
   e=(t_elem*)malloc(sizeof(t_elem));
   e->val=val;
   strcpy(e->s,s);
   e->suiv=e;     // pas de NULL pointe sur lui-même
   e->prec=e;
   return e;
}
```

Ajouter

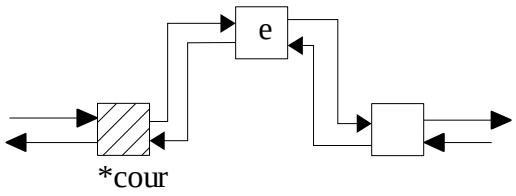
Nous avons deux cas soit ajouter vers la droite (suivant), soit ajouter vers la gauche (précédent).

Ajouter vers suivant

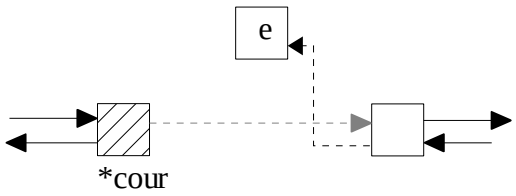
Commençons par le cas général, la situation de départ est la suivante :



La situation à l'arrivée est la suivante :

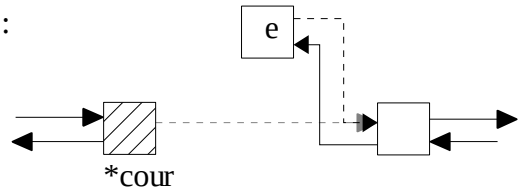


Pour construire l'insertion, première étape :



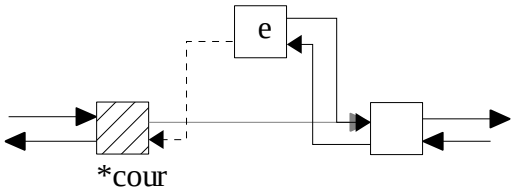
(*cour)->suiv->prec=e;

Deuxième étape :



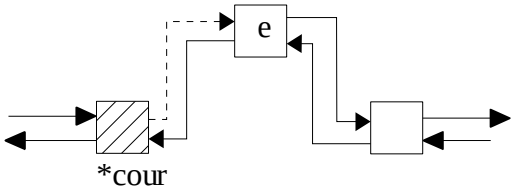
e->suiv=(*cour)->suiv;

Troisième étape :



e->prec=*cour;

Quatrième étape :



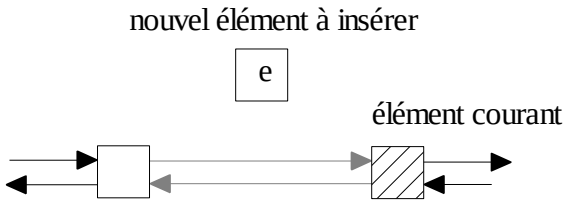
(*cour)->suiv=e;

Reste le cas particulier de la liste vide, dans ce cas le pointeur courant prend la valeur de e : *cour=e;
C'est pour cette raison que le pointeur courant qui permet d'accéder à la liste est passé par référence, parce que sa valeur peut être modifiée dans la fonction. La fonction ajout prend ainsi en paramètre l'adresse du pointeur courant passé par référence et l'adresse du nouvel élément à ajouter à la liste, ce qui donne :

```
void ajout_suiv(t_elem**cour,t_elem*e)
{
   if(*cour==NULL)
      *cour=e;
   else{
      (*cour)->suiv->prec=e;
      e->suiv=(*cour)->suiv;
      e->prec=*cour;
      (*cour)->suiv=e;
   }
}
```

Ajouter vers précédent

Le principe est exactement le même sauf que l'ajout se fait sur la gauche du pointeur courant :



L'algorithme est identique, il y a juste à inverser certaines affectations, ce qui donne la fonction :

```
void ajout_prec(t_elem**cour,t_elem*e)
{
   if(*cour==NULL)
      *cour=e;
   else{
     (*cour)->prec->suiv=e;
     e->suiv=*cour;
     e->prec=(*cour)->prec;
     (*cour)->prec=e;
   }
}
```

Supprimer

L'élément courant peut être supprimé. Le nouvel élément courant sera ensuite soit l'élément de gauche soit l'élément de droite, au choix. Puisqu'il va être modifié un passage par référence est nécessaire. A partir de l'élément courant nous avons accès à celui de gauche et à celui de droite : il y a juste à établir entre eux la bonne connexion. Ça marche sauf le cas particulier où il n'y a qu'un seul élément dans la liste. Dans ce cas libérer la mémoire de l'élément courant et le mettre à NULL, la liste est maintenant vide. L'algorithme est le suivant :

Si la liste n'est pas vide avec courant différent de NULL

- Premier cas si un seul élément avec suivant de courant égal à courant
  alors, libérer adresse de courant et mettre courant à NULL
- Dans tous les autres cas
  - récupérer l'adresse de courant : p=*cour;
  - le suivant du précédent de courant devient le suivant de courant :
    p->prec->suiv=p->suiv;
  - le précédent du suivant de courant devient le précédent de courant :
    p->suiv->prec=p->prec;
  - là faut choisir pour courant s'il part à droite ou à gauche :
    *cour=p->suiv; // pour droite
    *cour=p->prec;// pour gauche
  - maintenant l'adresse sauvée en p peut être libérée : free(p);

L'élément courant est modifié ce qui nécessite un passage par référence, ce qui donne la fonction :

```
void suppression(t_elem**cour)
```

```
{
t_elem*p;
   if(*cour!=NULL){
      if ((*cour)->suiv==*cour){ // si un seul
         free(*cour);
         *cour=NULL;
      }
      else{
         p=*cour;
         p->prec->suiv=p->suiv;
         p->suiv->prec=p->prec;
         *cour=p->suiv; // ou dans l'autre sens *cour=p->prec, au choix
         free(p);
      }
   }
}
```

Parcourir, afficher

Nous pouvons parcourir la liste à partir de l'élément courant soit en partant dans un sens soit en partant dans l'autre sens ce qui donne les deux fonctions suivantes.

En partant vers la droite (suivant)

Si la liste est vide le signifier.
sinon, afficher le premier, se positionner sur le suivant et tant qu'il est différent du premier, l'afficher et passer au suivant. Ceci donne la fonction :

```
void affiche_gauche(t_elem*cour)
{
t_elem*p;
   if (cour==NULL)
      printf("liste vide\n");
   else{
      printf("%d%s--",cour->val,cour->s);
      p=cour->prec;
      while(p!=cour){
         printf("%d%s--",p->val,p->s);
         p=p->prec;
      }
      putchar('\n');
   }
}
```

En partant vers la gauche (précédent)

Si la liste est vide le signifier.
sinon, afficher le premier, se positionner sur le précédent et tant qu'il est différent du premier, l'afficher et passer au précédent. Ceci donne la fonction :

```
void affiche_droite(t_elem*cour)
{
t_elem*p;
   if (cour==NULL)
      printf("liste vide\n");
   else{
      printf("%d%s--",cour->val,cour->s);
```

```
      p=cour->suiv;
      while(p!=cour){
         printf("%d%s--",p->val,p->s);
         p=p->suiv;
      }
      putchar('\n');
   }
}
```

Détruire une liste

Tant que courant est différent de NULL appeler la fonction de suppression de courant :

```
void detruire_liste(t_elem**cour)
{
   while(*cour!=NULL)
      suppression(cour);
}
```

Test dans le main()

Le menu est le suivant :

```
int menu()
{
int res=-1;
   printf(  "1  : ajout suiv, affiche droite\n"
            "2  : ajout prec, affiche gauche\n"
            "3  : suppression, affiche droite\n"
            "4  : detruire liste\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

Et voici les appels correspondants :

```
int main()
{
int fin=0;
t_elem*cour=NULL,*e;

   while(!fin){
      switch(menu()){
         case 1 :
            e=init_elem(rand()%26,S[rand()%26]);
            ajout_suiv(&cour,e);
            affiche_droite(cour);
            break;
         case 2 :
            e=init_elem(rand()%26,S[rand()%26]);
            ajout_prec(&cour,e);
            affiche_gauche(cour);
            break;
         case 3 :
            suppression(&cour);
            affiche_droite(cour);
```

```
            break;
         case 4 :
            detruire_liste(&cour);
            affiche_gauche(cour);
            break;
         default : fin = 1;
      }
   }
   detruire_liste(&cour);
   return 0;
}
```

Principe de Généricité en C, données void*

1. Principe

L'idée est de pouvoir désolidariser la mécanique de la chaine d'avec les données qui y sont stockées. Le princip est alors d'écrire une fois pour toutes les fonctions associées à la chaine (ajouter, supprimer etc.) et de les utiliser avec n'importe quel type de données. C'est possible en utilisant des pointeurs génériques void*. Ainsi un maillon de la chaine peut recevoir l'adresse d'une donnée quelque soit son type. Il est même envisageable de faire des listes d'objets différents. Mais le problème en C est qu'à partir d'une adresse mémoire void* il n'est plus possible à la machine de reconnaître l'objet concerné et donc il n'est plus possible de l'utiliser, si c'est une structure d'accéder à ses champs, de modifier les valeurs etc.. Nous sommes alors obliger de spécifier explicitement de quel objet il s'agit, c'est à dire de caster le void* dans le type de donnée vers lequel il pointe effectivement.

2. liste simple un seul type de données dans la chaine

Dans cette première étude la liste contient des données void*, c'est à dire qu'elle peut fonctionner avec n'importe quel type de données. Toutefois si nous changeons les données t_data que nous avons utilisées pour tester, il y aura quelques modifications à faire dans le programme : ce qui concerne l'initialisation et l'utilisation des données y compris l'affichage des valeurs. Ce qui concerne la liste, ajouter, extraire, détruire.., fonctionnera en revanche très bien. Des problèmes se posent pour avoir différents types de données avec la même structure de liste simultanément, c'est l'objet de la seconde étude de ce chapitre qui met en scène une liste constituée d'objets de types différents

Structure de données

Dans la structure de données du programme les données sont séparées de la structure de liste.
Pour les données nous avons toujours notre structure de test composée d'un entier et d'une chaine de caractères :

```
// les données
typedef struct data{
  int val;
  char s[80];
}t_data;
```

Nous avons également toujours ce tableau en global pour l'initialisation des chaines de caractères :

```
char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

La structure de liste est un type à part composé d'un void* générique et d'un pointeur sur la structure suivante, pour une liste simplement chainée :

```
// la chaine avec les données
typedef struct elem{
   void*dat;
```

```
   struct elem*suiv;
}t_elem;
```

Initialiser des données

Les données sont initialisées en fonction de leur type, t_data via la fonction suivante qui retourne l'adresse d'une structure t_data initialisée avec les valeur passées en paramètre :

```
t_data* init_data(int val, char s[])
{
t_data*d;
   d=(t_data*)malloc(sizeof(t_data));
   d->val=val;
   strcpy(d->s,s);
   return d;
}
```

Ajouter un élément

Les données à ajouter à la liste sont transmises en void*, elles sont donc indifférentes. La fonction crée un nouveau maillon et lui affecte l'adresse des données transmises, ensuite ce maillon est ajouté au début de la liste :

```
void ajout_debut(t_elem**prem,void*d)
{
t_elem*e;
   // allouer un nouvel élément
   e=(t_elem*)malloc(sizeof(t_elem));
   e->dat=d;      // lui affecter les datas
   e->suiv=*prem; // le lier à la chaine
   *prem=e;       // devient premier
}
```

Extraire un élément

La fonction d'extraction récupère les datas du premier maillon, supprime ce maillon et retourne ces datas ensuite :

```
void* extraire_debut(t_elem**prem)
{
t_elem*sup;
void*d=NULL;
   if (*prem){
      d=(*prem)->dat;
      sup=*prem;
      *prem=(*prem)->suiv;
      free(sup);
   }
   return d;
}
```

Parcourir, afficher

Parcourir la liste se fait comme d'habitude. En revanche il est nécessaire de connaître le type des données pour pouvoir les afficher. C'est le rôle de la fonction affiche_data() :

```
void affiche_liste(t_elem*prem)
{
   if (prem==NULL)
      printf("liste vide");
   while(prem!=NULL){
    // attention cast obligatoire pour affichage données void*
    //printf("%d%s--",((t_data*)prem->dat)->val,((t_data*)prem->dat)->s);
      affiche_data(prem->dat);
      prem=prem->suiv;
   }
   putchar('\n');
}
```

Le paramètre de cette fonction n'est pas un void* c'est un t_data* de la sorte l'adresse void* qui lui est transmise est implicitement castée en t_data*, ce qui permet de l'utiliser comme telle :

```
void affiche_data(t_data*d)
{
   printf("%d%s--",d->val,d->s);
}
```

Détruire la liste

Détruire la liste ne pose pas de problème particulier, chaque élément est supprimé au fur et à mesure et pour chaque élément les datas qui lui sont associé :

```
void detruire_liste(t_elem**prem)
{
t_elem*sup;
   while(*prem){
     sup=*prem;
     *prem=(*prem)->suiv;
     free(sup->dat);
     free(sup);
   }
}
```

Test dans le main()

Le menu propose :

```
int menu()
{
int res=-1;
   printf(  "1  : ajout debut et affiche\n"
            "2  : extraire debut et affiche\n"
            "3  : detruire liste\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

et voici les appels correspondants dans le main() en fonction des choix de l'utilisateur :

```
int main()
{
int fin=0;
t_elem*prem=NULL;
void*d;
   srand(time(NULL));
   while(!fin){
      switch(menu()){
         case 1 :
            d=init_data(rand()%26,S[rand()%26]);
            ajout_debut(&prem,d);
            affiche_liste(prem);
            break;
         case 2 :
            if ((d=extraire_debut(&prem))!=NULL){
               affiche_data(d);
               putchar('\n');
               free(d);
            }
            affiche_liste(prem);
            break;
         case 3 :
            detruire_liste(&prem);
            affiche_liste(prem);
            break;
         default : fin = 1;
      }
   }
   detruire_liste(&prem);
   return 0;
}
```

Exercices listes chainées

Les exercices sont faits en liste chainée dynamique et/ou contiguë

Exercice 1

L'objectif est de faire une suite de nombre entiers sous la forme d'une liste chainée. Le programme à faire :

- initialise chaque entier avec une valeur aléatoire comprise entre 0 et 1000
- affiche une liste de nb entiers, nb entré par l'utilisateur
- peut détruire la liste afin d'en faire une nouvelle.
- calcule la somme des entiers de la liste
- met à -1 le maillon nb/2 de la liste
- passe en négatif tous les maillons inférieurs à un seuil déterminé par l'utilisateur. Afficher le résultat.
- Efface tous les maillons dont la valeur est comprise entre un seuil haut et un seuil bas entrés par l'utilisateur. Afficher le résultat.
- Duplique les maillons qui ont la même valeur qu'une valeur entrée par l'utilisateur. Afficher le résultat.

Exercice 2

L'objectif est d'écrire les deux fonctions suivantes : la première permet de transformer en liste chainée un tableau dynamique de nb éléments. La seconde transforme à l'inverse une liste chainée de nb éléments en un tableau dynamique. Faire un programme de test.

Exercice 3

Ecrire une fonction qui prend en paramètre une liste chainée et renvoie une autre liste ayant les mêmes éléments mais dans l'ordre inverse. Tester dans un programme.

Exercice 4

Ecrire une fonction de concaténation de deux listes chainées. Il y a deux versions de la fonction : une destructrice des deux listes données au départ et l'autre qui préserve ces deux listes. Tester dans un programme.

Exercice 5

Ecrire une fonction qui détermine si une liste chainée d'entiers est triée ou non en ordre croissant.
Ecrire une fonction qui insère une élément dans une liste chainée triée.
Tester dans un programme.

Exercice 6

Ecrire une fonction qui permet de fusionner deux listes chainées. La fusion se faite en alternant un élément d'une liste avec un de l'autre liste. Tous les éléments des deux listes doivent trouver leur place dans la liste résultante même en cas de différence de taille. Faire deux versions, une qui conserve les deux listes de départ et une autre qui les détruit. Tester dans un programme.

Exercice 7

Ecrire une fonction qui prend en entrée une liste chainée d'entiers et qui ressort deux listes chainées, une pour les nombres pairs et une autre pour les nombres impairs. La liste initiale est détruite. Tester dans un programme.

Exercice 8

Ecrire les instructions qui saisissent puis affichent une liste de chevaux de course. Chaque cheval est entré séparément par l'utilisateur. Un cheval est défini par un nom, un dossard, un temps réalisé dans la course, un classement et la liste est une liste chainée. Le programme permet d'ordonner la liste selon le classement des chevaux et de supprimer des chevaux de la liste.

Exercice 9

Ecrire un programme permettant de lire un texte à partir d'un fichier. Chaque mot du texte est récupéré dans une liste chainée qui regroupe tous les mots du texte. Les mots sont dans l'ordre du texte sur le fichier et il n'y a pas de répétition de mot.

Exercice 10

Ecrire un programme qui permet de distribuer les cartes d'un jeu de 52 cartes entre nb joueurs entré par l'utilisateur. Le jeu de chaque joueur est constitué par une liste chainée. Tester dans un programme.

Exercice 11

Le problème de Joséphus Flavius : Dans un bureau de recrutement, n personnes numérotées de 1 à n sont disposées en cercle suivant l'ordre de leur numéro. Le chef de bureau est au centre, puis se dirige vers la personne n°1. Sa stratégie est d'éliminer chaque deuxième personne qu'il rencontre en tournant sur le cercle. la dernière personne restante est embauchée. Par exemple s'il y a 10 personnes, n=10, les personne 2, 4, 6, 8, 10, 3, 7, 1, 9 sont éliminées et la personne restante est le n°5. Faire un programme de simulation :

1) Pour n entré par l'utilisateur donner le numéro de la personne restante.

2) au lieu de prendre chaque deuxième personne généraliser en prenant la k-éme personne, k entré par l'utilisateur.

Il s'agit de faire une liste circulaire. Chaque élément est une personne (nom, numéro)

Exercice 12

Division cellulaire de l'algue Anabaena Catenula. Au départ nous avons une cellule de taille 4 symbolisée par un sens, par exemple une flèche ← . Elle grandit de 1 à chaque étape de son évolution. A l'étape 5 elle a la taille 9 qui est le maximum qu'une cellule peut atteindre. Alors elle se divise en deux cellules, une de 4, sens ←, et une de 5 symbolisée par le sens → et l'évolution reprend. Dés qu'une cellule arrive à la taille 9 elle se divise en deux cellules de 4 et 5. Programmer une simulation. Là encore il s'agit de faire une liste circulaire. Définir pour commencer une structure cellule avec une représentation du sens. Trouver un mode d'affichage et pouvoir afficher une étape quelconque de la progression.

Séquence TD-TP liste chainée, fichiers, tris

Une vidéothèque gère un ensemble de films. Chaque film regroupe des informations : le titre, le nom du réalisateur, l'année de réalisation et la durée en minutes.

Les films de la vidéothèque sont stockés dans un fichier texte :

- la première ligne donne le nombre de films
- les informations d'un film sont séparées par un caractère de séparation

5
Mar adentro:Amenabar:2004:118
Lost in translation:Coppola:2003:111
Million baby dollar:Eastwood:2005:126
Apocalypse Now:Coppola:1974:187
Seven:Fincher:1995:130

Questions 1.1:

a) Quelles sont les structures de données les mieux adaptées pour mémoriser :
- un film,
- Les films stockés dans le fichier.

Justifiez vos choix. Puis définissez ces structures de données en C.

b) Ecrire une fonction qui lit les films du fichier pour les mémoriser dans vos structures de données.

La fonction **strtok** peut être utile si vous souhaitez extraire une chaîne d'une chaîne initiale *strToken* à partir d'une chaîne contenant des caractères délimiteurs *strDelimit* :

char *strtok(char * *strToken* **, const char *** *strDelimit* **);**

Questions 1.2: On souhaite trier par ordre croissant selon 2 critères : nom du réalisateur, puis titre du film.

a) Ecrire une fonction C qui trie les films mémorisés à la question précédente, en justifiant votre choix de tri (complexité temporelle et stabilité).

b) Un tableau d'indices contenant les positions des films mémorisés peut s'avérer intéressant pour votre tri : il s'agit de trier non pas les films mémorisés mais de réorganiser leurs positions dans le tableau d'indices. Ces positions donnent l'ordre des films avant et après le tri.
- Donnez les avantages et les inconvénients de ce tableau d'indices.
- Modifiez votre fonction de tri en utilisant ce tableau d'indices.

Questions 1.3: Après le tri de la question précédente, écrire une fonction de recherche dichotomique d'un nom de réalisateur. Que se passe-t-il s'il y a des homonymes ? Une fois cette recherche effectuée, quelle est votre méthode pour afficher tous les titres de films qu'il a réalisés ?

Questions 1.4: On ne souhaite plus indiquer le nombre de films dans la première ligne du fichier.
a) Quel est l'intérêt d'enlever cette information du fichier ? En déduire et justifier la structure de données la mieux adaptée pour mémoriser les films du fichier. Définir cette structure en C.
b) Ecrire une nouvelle fonction qui lit les films du fichier et les mémorise dans cette nouvelle structure de données. Puis réécrire votre fonction de tri de la question 2 adaptée à cette nouvelle structure de données.
c) La recherche dichotomique est-elle adaptée avec cette nouvelle structure de données ? Pourquoi ? d) En déduire les avantages et les inconvénients entre les 2 structures de données.