Chapitre I

Structure de données
Les listes chaînées

1. Introduction : Objectif et Motivation

- Rappelons qu'un programme informatique est un ensemble d'instructions et de données qui permettent à l'ordinateur de traiter des informations.
- Ces informations sont organisées dans la mémoire principale et dans les mémoires auxiliaires.
- Par ailleurs, notons que l'ordinateur définit au niveau matériel :
  - Un jeu d'instructions élémentaires (additionner, soustraire, multiplier...),
  - Quelques représentations de données élémentaires (bits).
- Mais, il faut signaler que le matériel ne supporte que des données élémentaires, donc non structurées.
- On peut ainsi définir une donnée structurée comme étant une collection organisée de données élémentaires.
- En informatique, on appelle une telle collection, une ***structure de données***.
- Une ***structure de données*** est donc une manière d'organiser et de représenter des informations en mémoire centrale et dans les mémoires auxiliaires.
- Une structure de données contient deux types de renseignement :
  - les informations à stocker,
  - les liens entre ces informations.
- On peut distinguer deux types de liens :
  - Liens implicites : l'adresse de l'information est calculée en fonction de l'emplacement physique de l'information en mémoire.
  - liens explicites : les pointeurs.

1. Introduction : Objectif et Motivation

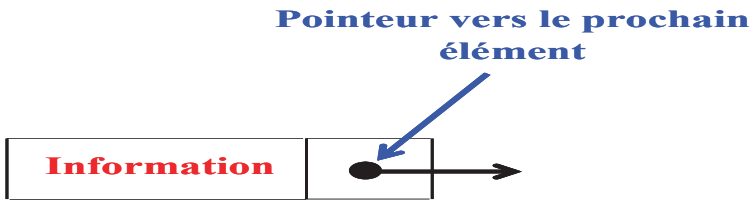
- En langage C, il existe un grand nombre de structure de données :
  Exemple : les tableaux les structures, les listes chaînées, les piles, les files, les arbres binaires...
- Le choix d'une structure dépend, en général, des opérations à réaliser sur les informations. Certaines structures sont plus adaptés que d'autres à certaines opérations.
- ***L'objectif*** est la gestion d'un ensemble fini d'éléments dont le nombre n'est pas fixé.
- Les éléments de cet ensemble peuvent être de différentes types : nombres entiers ou flottants, chaînes de caractères,...
- On ne s'intéressera pas aux éléments de l'ensemble en question mais aux opérations que l'on effectue sur cet ensemble, indépendamment de la nature de ses éléments.
- En programmation informatique, ces ensembles sont considérés comme des objets dynamiques.
- En effet, le nombre de leurs éléments varie au cours de l'exécution du programme, puisqu'on peut y ajouter et supprimer des éléments en cours de traitement.
- Plus précisément les opérations que l'on s'autorise sur les ensembles sont les suivantes :
  - tester si un ensemble est vide.
  - ajouter un élément à un ensemble.
  - supprimer un élément à un ensemble.
  - rechercher un élément dans un ensemble.
- Cette gestion des ensembles doit, pour être efficace, répondre à deux critères :
  - un minimum de place mémoire utilisée,
  - et un minimum d'instructions élémentaires pour réaliser une opération.

1. Introduction : Objectif et Motivation

- Notons que la structure de données la plus utilisée pour manipuler des données sont les tableaux.
- Mais, dans certains cas, les tableaux ne constituent pas la meilleure solution pour stocker et manipuler les données.
- En effet, Si on veut supprimer un élément au milieu d'un tableau, il faut recopier les éléments temporairement, réallouer le tableau, puis le remplir à partir de l'élément supprimé.
- Ces manipulations sont coûteuses en ressources.
- En langage C, les ***listes chaînées*** (ou tout simplement les listes) peuvent être considérées comme des structures de données très pratiques pour manipuler des éléments d'un ensemble d'objets dynamiques.
- En effet, une liste chaînée est différente par rapport à un tableau dans le sens où ses éléments sont répartis dans la mémoire et reliés entre eux par des pointeurs.
- Ceci permet d'ajouter et de supprimer des éléments d'une liste à n'importe quel endroit, sans modifier la liste entière.

2. Les listes chaînées

- Une *liste chaînée* est une structure de base de la programmation informatique.
- C'est une structure de données dans laquelle les objets sont stockés de manière ordonnée (l'ordre signifie que chaque objet possède une position dans la liste).
- Cependant, contrairement au tableau, pour lequel l'ordre est déterminé par les indices, l'ordre d'une liste chaînée est déterminé par un pointeur dans chaque objet.
- Les listes chaînées sont des structures dynamiques capables,
  - d'accueillir un nombre infini d'éléments,
  - et de supporter toutes les opérations sur les ensembles dynamiques (insertion, suppression, recherche,...).
- *Dans la suite du chapitre*, on utilise la liste chaînée pour représenter un ensemble d'éléments.
- Chaque élément est contenu dans une *cellule* (appelée aussi *maillon*), et chaque cellule est composée d'une partie contenant l'information et d'au moins un pointeur.



Représentation d'une cellule avec un seul pointeur.

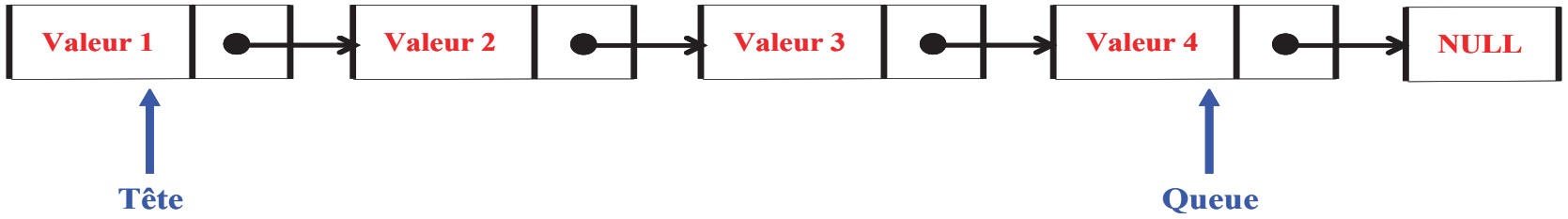
2. Les listes chaînées

- Chaque élément est accessible par celui qui le précède grâce au pointeur.
- Une liste chaînée est donc une succession de cellules, dont le dernier porte une adresse de fin (le pointeur `NULL`). La première cellule de la liste est appelée tête et la dernière cellule est appelée queue.
- L'accès à la liste chaînée se fait par un pointeur vers le premier élément.
- Pour atteindre un des éléments de la liste,

  on est obligé de parcourir la liste depuis le début.
- A partir d'un élément quelconque,

  on ne peut atteindre que le suivant.
- Une liste chaînée peut prendre différentes formes :
  - Elle peut être ***simple*** (dite aussi ***simplement chaînée***) : chaque cellule est composé de l'information et d'un seul pointeur (pointe sur l'adresse de la cellule suivante).
  - Ou ***doublement chaînée*** : chaque cellule est composé de l'information et de deux pointeurs (l'un pointe sur l'adresse de la cellule suivante et l'autre sur celle de la cellule précédente).
  - Ou ***circulaire*** : la tête pointe sur la queue de la liste, et la queue pointe sur la tête de la liste.

- Rappelons qu'une liste simplement chaînée est une liste où chaque cellules contient des informations accessibles et modifiables par l'utilisateur ainsi qu'un pointeur vers la cellule suivante.



- Comme les listes simplement chaînées sont une structure de données importante en programmation informatique,
- il est fondamental de s'intéresser à la manière de les implanter en vue de leurs manipulations informatique.
- Deux modes de d'implantation peuvent être envisagés :
  - Implantation par pointeur.
  - Implantation par tableau.

2.1 Les listes simplement chaînées : Implantation par pointeur

Déclaration

- Le langage C permet de réaliser des listes chaînées à l'aide des structures (le type structure en C qui est créée par le mot clé `struct`).
- En effet, les cellules sont des objets constitués :
  - d'un champ de données,
  - d'un pointeur vers la cellule suivante.
- La déclaration correspondante est la suivante, où l'on suppose ici que les valeurs stockées dans chaque cellule sont de type entier.

Code en C

```
typedef int element;
struct cellule
 {
    element valeur;
    struct cellule * suivant;
 };
typedef struct cellule cellule, * liste;
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Remarque 1:

Notons qu'on peut créer des listes chaînées contenant des valeurs qui sont :

- de différents type, c-à-d des entiers, des flottants, des caractères,
- des tableaux, des structures,
- une combinaison de plusieurs types,
- ou des listes chaînées.

Exemple :

```
typedef double element;
struct cellule
 {
    element valeur;
    struct cellule * suivant;
 };
typedef struct cellule cellule, * liste;
```

Remarque 2:

Les instructions suivantes permettent de déclarer, de façons différentes mais équivalentes, une liste chaînée :

```
struct cellule *l;
cellule *l;
liste l;
```

Création et initialisation

- Pour créer une liste chaînée, on doit l'initialiser à NULL (macro de stdlib.h), cela va nous permettre d'allouer la première cellule.
- L'initialisation d'une liste chaînée, ou la création d'une liste vide, est généralement faite par la fonction suivante :

```
liste creerListeVide( )
  {
    return NULL;
  }
```

- Pour tester si une liste est vide, on utilise la fonction suivante :

```
 int ListeVide(liste l)
  {
    return l==NULL;
  }
```

Insertion d'une cellule (Ajout d'une valeur dans une liste) :

- Il est toujours possible d'insérer des cellules (ajouter des valeurs) à une liste chaînée donnée.
- En effet, il y a deux façons d'ajouter des cellules dans des listes chaînées :
  - ajouter une nouvelle cellule en tête de la liste,
  - ajouter une nouvelle cellule à la fin de la liste.
  - Ainsi, les fonctions `ajouterElementTeteListe()` et `ajouterElementFinListe()` permettent respectivement l'ajout d'un élément en tête et à la fin d'une liste.

Ajout d'un élément en tête de la liste

```
liste ajouterElementTeteListe(element val, liste l)
  {
    liste ln;
    ln=malloc(sizeof(cellule));
    ln->valeur=val;
    ln->suivant=l;
    return ln;
  }
```

Insertion d'une cellule (Ajout d'une valeur dans une liste) :

Ajout d'un élément à la fin de la liste

```
liste ajouterElementFinListe(element val, liste l)
  {
    liste ln,lp;
    ln=malloc(sizeof(cellule));
    ln->valeur=val;
    ln->suivant=NULL;
    if(l==NULL)
      return ln;
    else
    {
      lp=l;
      while(lp->suivant != NULL)
       {
        lp= lp->suivant;
       }
      lp->suivant =ln;
      return l;
    }
  }
```

Suppression d'une cellule (supprimer une valeur dans une liste) :

- Il est possible également de supprimer des cellules (des valeurs) dans une liste chaînée donnée.
- En effet, il y a trois façons de supprimer des cellules dans des listes chaînées :
  - supprimer une cellule en tête de la liste,
  - supprimer une cellule à la fin de la liste,
  - supprimer une cellule quelconque dans une liste.

Suppression d'un élément en tête de la liste

```
liste supprimerElementTeteListe(liste l)
  {
    liste lp;
    if(l==NULL)
     return NULL;
    else
     {
      lp=l->suivant;
      free(l);
      return lp;
     }
  }
```

Suppression d'une cellule (supprimer une valeur dans une liste) :

Suppression d'un élément à la fin de la liste

```
liste supprimerElementFinListe(liste l)
  {
    liste lp1,lp2;
    if(l==NULL)
     return NULL;
    if(l->suivant==NULL)
     {
      free(l);
      return NULL;
     }
    lp1=l;
    while(lp1->suivant->suivant!=NULL)
     {
      lp1=lp1->suivant;
     }
    lp2=lp1->suivant;
    lp1->suivant=NULL;
    free(lp2);
    return l;
  }
```

Suppression d'une cellule (supprimer une valeur dans une liste) :

Suppression d'un élément quelconque dans la liste (Méthode itérative)

```
liste supprimerItertiveElementListe(element val, liste l)
  {
    liste lp1,lp2;
    if(l!=NULL)
      if(l->valeur==val)
        {
          lp1=l;
          l=l->suivant;
          free(lp1);
        }
      else
        {
          lp2=l;
          while(lp2->suivant!=NULL && lp2->suivant->valeur!=val)
            lp2=lp2->suivant;
          if(lp2->suivant!=NULL)
            {
              lp1=lp2->suivant;
              lp2->suivant=lp2->suivant->suivant;
              free(lp1);
            }
        }
        return l;
  }
```

Suppression d'une cellule (supprimer une valeur dans une liste) :

Suppression d'un élément quelconque dans la liste (Méthode récurrente)

```
liste supprimerRecurrenteElementListe(element val, liste l)
  {
    liste lp;
    if(l!=NULL)
      if(l->valeur==val)
        {
          lp=l;
          l=l->suivant;
          free(lp);
        }
      else
        l->suivant=supprimerRecurrenteElementListe(val, l->suivant);
      return l;
 }
```

Recherche dans une liste :

- Étant donnée une liste chaînée `liste` et une valeur `val`, l'objectif ici est d'effectue un parcours de liste pour rechercher la valeur `val` dans `liste`. Pour ceci, on peut envisager deux types de fonction :
  - une fonction qui teste si la valeur recherchée `val` figure dans la liste `liste`.
  - une fonction qui renvoie l'adresse de la cellule contenant la valeur recherchée `val`.

fonctions qui testent si `val` figure dans `liste`

Recherche dans une liste (Méthode récurrente)

```
int rechercherRecListe(element val, liste L)
  {
  if(L==NULL)
   return 0;
  else if(L->valeur==val)
   return 1;
  else
   return rechercherRecListe(val, L->suivant);
}
```

Recherche dans une liste (Méthode itérative)

```
int rechercherIterListe(element val, liste L)
 {
 while(L!=NULL)
  {
  if(L->valeur==val)
  return 1;
  L=L->suivant;
  }
     return 0;
 }
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Recherche dans une liste :

fonctions qui renvoient l'adresse de la cellule contenant `val`

Recherche dans une liste (Méthode récurrente)

```
liste rechercherLRec(element val, liste L)
  {
  if(L==NULL)
   return NULL;
  else if(L->valeur==val)
   return L;
  else
   return rechercherLRec(val, L->suivant);
}
```

Recherche dans une liste (Méthode itérative)

```
liste rechercherLIter(element val, liste L)
  {
  while(L!=NULL)
   {
   if(L->valeur==val)
   return L;
   L=L->suivant;
   }
      return NULL;
  }
```

Affichage d'une liste :

- La fonction suivante permet d'afficher les valeurs stockée d'une liste.

Affichage d'une liste

```
void afficherListe(liste l)
  {
    while(l!=NULL)
      {
        printf("%d ",l->valeur);
        l=l->suivant;
      }
    printf("\n");
  }
```

Exercice :

Écrire un programme complet en langage C qui permet de

- construire une liste chaînée,
- afficher une liste chaînée,
- calculer la longueur d'une liste chaînée,
- rechercher une valeur dans la liste et de retourner sa position dans la liste.
- effectuer les manipulations suivantes :
  - supprimer une valeur quelconque de la liste,
  - supprimer une valeur en début de liste,
  - supprimer une valeur en fin de liste,
  - supprimer une valeur à une position donnée dans la liste,
  - supprimer une valeur avant ou après une position donnée dans la liste.

2.1 Les listes simplement chaînées : Implantation par tableau

- Une liste peut être aussi implantée à l'aide d'un tableau.
- Les éléments de la liste seront donc stockés dans les cases contiguës du tableau.
- Comme les listes sont des structures de données qui gèrent des ensembles dynamiques, une taille maximum pour le tableau doit être donc définie lors des déclarations.
- C-à-d, on ne peut pas modifier la taille maximum du tableau lors de l'exécution d'un programme.
- Et si on désire modifier la taille maximum du tableau, il faut modifier aussi le programme.
- Par consequent, il faut définir un tableau dont la taille sera suffisante pour accueillir la plus grande liste pouvant être utilisée!!!
- Mais cette démarche peut encombrer la mémoire de l'ordinateur.
- C'est pourquoi, il est en général préférable d'utiliser l'implantation des listes par des pointeurs.

2.1 Les listes simplement chaînées : Implantation par tableau

Déclaration

- En langage C, pour implanter une liste chaînée à l'aide d'un tableau, on utilise les structures.
- En effet, la liste est un objet constitués :
  - d'un tableau, de taille suffisante pour accueillir la plus grande liste à traiter.
  - d'un entier contenant l'indice de la fin de la liste. Cet entier contient l'indice+1 du dernier élément de la liste, ou la taille de la liste.

- La déclaration correspondante est la suivante, où l'on suppose ici que les valeurs stockées dans la liste sont de type entier.

Code en C

```
typedef int element;
struct listeTab
 {
    element valeur[taille_MAX];
    int fin;
 };
typedef struct listeTab listeTab, liste;
```

Création et initialisation

- Pour créer une liste chaînée, on doit l'initialiser à un tableau de taille nulle.
- L'initialisation d'une liste chaînée, ou la création d'une liste vide, est généralement faite par la fonction suivante :

```
int creerListeVide( )
  {
    return 0;
  }
```

- Pour tester si une liste est vide, on utilise la fonction suivante :

```
 int ListeVide(liste l)
  {
    return l.fin==0;
  }
```

Ajout d'une valeur à une liste :

- Il est toujours possible d'ajouter des valeurs à une liste chaînée donnée.
- En effet, comme l'implantation par les pointeurs, il y a différentes façons d'ajouter des valeurs à une liste chaînée en particulier on peut
  - ajouter une nouvelle valeur en tête de la liste,
  - ajouter une nouvelle valeur à la fin de la liste.
  - ajouter une nouvelle valeur à une position quelconque de la liste.

Ajout d'un élément en tête de la liste

```
liste ajouterTeteLTab(element val, liste l)
  {
    int i;
    l.fin=l.fin+1;
    for(i=l.fin-1;i>=1;i--)
      l.valeur[i]=l.valeur[i-1];
    l.valeur[0]=val;
    return l;
  }
```

Ajout d'un élément à la fin de la liste

```
liste ajouterFinLTab(element val, liste l)
  {
    l.valeur[l.fin]=val;
    l.fin=l.fin+1;
    return l;
  }
```

Ajout d'une valeur à une liste :

Ajout d'un élément après une position donnée dans la liste

```
liste  ajouterApresPosLTab(element val, int pos, liste l)
  {
    int i;
    l.fin=l.fin+1;
    for(i=l.fin-1;i>=pos+1;i--)
      l.valeur[i]=l.valeur[i-1];
    l.valeur[pos]=val;
    return l;
  }
```

Ajout d'un élément après un élément donnée dans la liste

```
liste ajouterApresElementLTab(element val, element elem, liste l)
  {
    int i;
    for(i=0;i<l.fin;i++)
      if(l.valeur[i]==elem)
        l=ajouterApresPosLTab(val, i+1, l);
      return l;
  }
```

Suppression d'une valeur dans une liste :

- Il est possible également de supprimer des valeurs dans une liste donnée.
- En effet, il y a différentes façons de supprimer des valeurs dans des listes :
  - supprimer une valeur en tête de la liste,
  - supprimer une valeur à la fin de la liste,
  - supprimer une valeur quelconque dans une liste.

Suppression d'un élément en tête de la liste

```
liste supprimerTeteLTab(liste l)
  {
    int i;
    l.fin=l.fin-1;
    for(i=0;i<l.fin;i++)
      l.valeur[i]=l.valeur[i+1];
      return l;
  }
```

Suppression d'un élément en fin de la liste

```
liste supprimerFinLTab(liste l)
  {
    l.fin=l.fin-1;
    return l;
  }
```

Suppression d'une valeur dans une liste :

Suppression d'un élément quelconque dans une liste

```
liste supprimerElemLTab(element val, liste l)
  {
    int i,j;
    for(i=0;i<l.fin;i++)
      if(l.valeur[i]==val)
        j=i;
    l.fin=l.fin-1;
    for(i=j;i<l.fin;i++)
      l.valeur[i]=l.valeur[i+1];
    return l;
  }
```

Suppression d'un élément quelconque à une position donnée d'une liste

```
liste supprimerPosLTab(int pos, liste l)
  {
    int i;
    l.fin=l.fin-1;
    for(i=pos-1;i<l.fin;i++)
      l.valeur[i]=l.valeur[i+1];
    return l;
  }
```

Recherche dans une liste :

- L'objectif ici est de rechercher une valeur `val` dans `liste`. Pour ceci, on peut envisager deux types de fonction :
  - une fonction qui teste si la valeur recherchée `val` figure dans la liste `liste`.
  - une fonction qui renvoie la position dans la liste de la valeur contenant la valeur recherchée `val`.

Tester si la valeur recherchée existe dans la liste

```
int rechercherLTab(element val, liste L)
  {
    int i;
    for(i=0;i<l.fin;i++)
      if(l.valeur[i]==val)
        return 1;
    return 0;
  }
```

Rechercher une position d'un élément d'une liste

```
int rechercherPosLTab(element val, liste L)
  {
    int i;
    if(l.fin==0)
      return -1;
    else
      for(i=0;i<l.fin;i++)
        if(l.valeur[i]==val)
          return i+1;
    return 0;
  }
```

Affichage d'une liste :

- La fonction suivante permet d'afficher les valeurs stockées d'une liste.

Affichage d'une liste

```
void afficherLTab(liste l)
  {
    int i;
    if(l.fin<=0)
      printf("votre liste est vide!!!\n");
    else
     {
      for(i=0;i<l.fin;i++)
        printf("%d ",l.valeur[i]);
      printf("\n\n");
     }
  }
```

Exercice :

Écrire un programme complet en langage C qui permet, en utilisant l'implantation par tableau, de

- saisir une liste chaînée,
- afficher une liste chaînée,
- calculer la longueur d'une liste chaînée,
- rechercher une valeur dans la liste et de retourner sa (ou ses) position(s) dans la liste.
- effectuer les manipulations suivantes :
  - supprimer une valeur quelconque de la liste,
  - supprimer une valeur en début de liste,
  - supprimer une valeur en fin de liste,
  - supprimer une valeur à une position donnée dans la liste,
  - supprimer une valeur avant ou après une position donnée dans la liste.

Chapitre II

Structure de données
Les Piles et les Files

1. Introduction

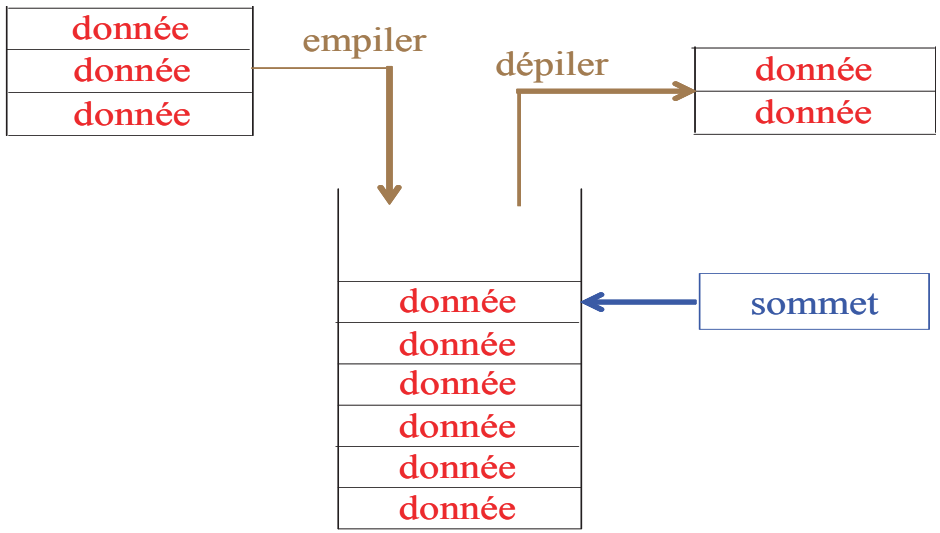
- Les *piles* (en anglais ***stack***) et les *files* (en anglais ***queue***) sont des types de listes (ou de structures de données) particuliers qui servent à stocker et manipuler des données dans des ensembles dynamiques.
- Leur particularité résident dans leur gestion des opérations d'insertion et de suppression des éléments dans les ensembles qui manipulent. En effet,
  - dans une pile, l'élément supprimé est le dernier inséré,
  - dans une file, l'élément supprimé est toujours le plus ancien.
- Par conséquent, dans une pile ou une file on ne peut accéder qu'à un seul élément de l'ensemble traité.
- Les piles et les files ont beaucoup d'applications dans différentes domaines notamment en informatique.
- Il existe plusieurs manières efficaces d'implantation, en langage C, des piles et des files.
- Dans ce chapitre, nous allons présenter comment les implanter à l'aide des pointeurs, des tableaux et des listes chaînées (seulement pour les files) .

2. Piles : 2.1 présentation générale

- Une pile est une structure de donnée où il n'est possible d'ajouter ou de supprimer des éléments qu'à une seule extrémité appelée ***sommet***. En effet,
- Le dernier élément ajouté devient le sommet de la pile et il sera le seul élément accessible.
- Lorsque on désire supprimer un élément de la pile, on doit supprimer le sommet, et le dernier élément ajouté avant lui devient alors le sommet de la pile.
- C-à-d, *le premier élément supprimé est le dernier élément ajouté dans la pile*. Cette structure est connue sous le nom de ***LIFO*** (***Last In, First Out***).
- Pour résumer, une pile permet la mis en attente d'informations ou de données dans le but de les récupérer plus tard, dans l'ordre inverse d'arrivée.
- La notion de pile est une structure très utilisée en informatique, surtout dans les langages de programmation, en effet
  - elle sert à implémenter les appels de fonctions (sous-programmes),
  - gère la récursivité,
  - permet le calcul des expressions arithmétiques,
  - etc.
- Généralement, il y a deux façons d'implanter une pile en langage C :
  - soit en utilise un pointeur indiquant le sommet de la pile,
  - soit on utilise un tableau.

Opérations sur les piles :

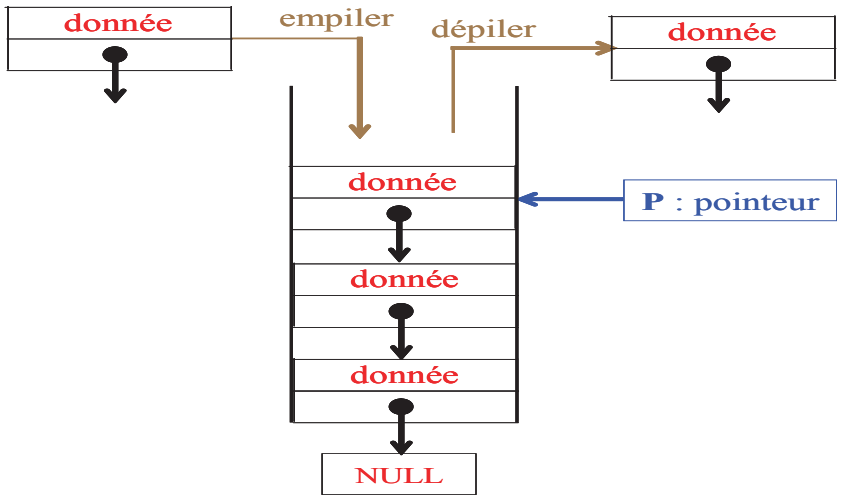
- Dans une pile, on ne peut autoriser que 5 opérations :
  1. *tester* si la pile est vide,
  2. *empiler* un élément, c-à-d l'ajouter au sommet de la pile,
  3. *dépiler* un élément, c-à-d le supprimer du sommet de la pile,
  4. *récupérer* le dernier élément de la pile, c-à-d acceder au sommet de la pile,
  5. *vider* le contenue de la pile.



Pile (LIFO)

Déclaration d'une pile :

- Comme le cas des listes, le langage C permet de réaliser des piles à l'aide des structures.
- En effet, chaque cellule de la pile est un objet constitué :
  - d'un champ de données,
  - d'un pointeur vers la cellule précédente.
- La déclaration correspondante est la suivante, où l'on suppose ici que les valeurs stockées dans la pile sont de type entier.



Pile (LIFO)

Code en C

```
typedef int element;
struct cellulePile
 {
    element valeur;
    struct cellulePile * precedent;
 };
typedef struct cellulePile cellulePile, * Pile;
```

Création et initialisation d'une pile :

- Pour créer une pile, on doit l'initialiser à NULL, cela va nous permettre d'allouer la première cellule.
- L'initialisation d'une pile, ou la création d'une pile vide, est faite par la fonction suivante :

```
Pile pileVide( )
  {
   return NULL;
  }
```

- Pour tester si une pile est vide, on utilise la fonction suivante :

```
 int testerPileVide(Pile p)
 {
   return p==NULL;
 }
```

Ajouter un élément dans la pile (empiler) :

- Il n'est possible d'insérer qu'une seule valeur en tête (ou sommet) de la pile.
- La valeur, une fois insérée, devient le sommet de la pile.

Code en C

```
Pile empiler(element val, Pile p)
  {
    Pile pn;
    pn=malloc(sizeof(cellulePile));
    pn->valeur=val;
    if(p==NULL)
     {
        pn->precedent=NULL;
        return pn;
     }
    else
     {
        pn->precedent=p;
        p=pn;
        return p;
     }
  }
```

Supprimer un élément de la pile (dépiler) :

- Il n'est possible de supprimer qu'une seule valeur dans une pile.
- Cette valeur n'est que la dernière ajoutée dans la pile, c-à-d le sommet de la pile.
- Ainsi, après cette suppression, le dernier élément inséré devient le sommet de la pile.

Code en C

```
Pile depiler(Pile p)
  {
   Pile pPile;
   if(p==NULL)
    return NULL;
   else
    {
     pPile=p->precedent;
     free(p);// ou p=NULL;
     return pPile;
    }
  }
```

Retourner l'élément supprimer de la pile :

- Puisque une pile permet la mise en attente d'informations dans le but de les récupérer après, la fonction suivante permet de retourner l'élément (ou le sommet) retiré de la pile.

Code en C

```
element valeurSommet(Pile *p)
  {
    element vdep;
    Pile pPile;
    if(*p==NULL)
     return -1;
    else
     {
      vdep=(*p)->valeur;
      pPile=(*p)->precedent;
      free(*p);
      *p=pPile;
      return vdep;
     }
  }
```

Code en C

```
Pile valeurSommet2(Pile p,element *vdep)
  {
    Pile pPile;
    if(p==NULL)
     return NULL;
    else
     {
      *vdep=p->valeur;
      pPile=p->precedent;
      free(p);// ou p=NULL;
      return pPile;
     }
  }
```

récupérer le sommet de la pile :

- Puisque chaque pile est définie par son sommet, la fonction suivante permet de retourner la valeur du sommet d'une pile.

Code en C

```
element sommetPile(Pile p)
  {
   return p->valeur;
  }
```

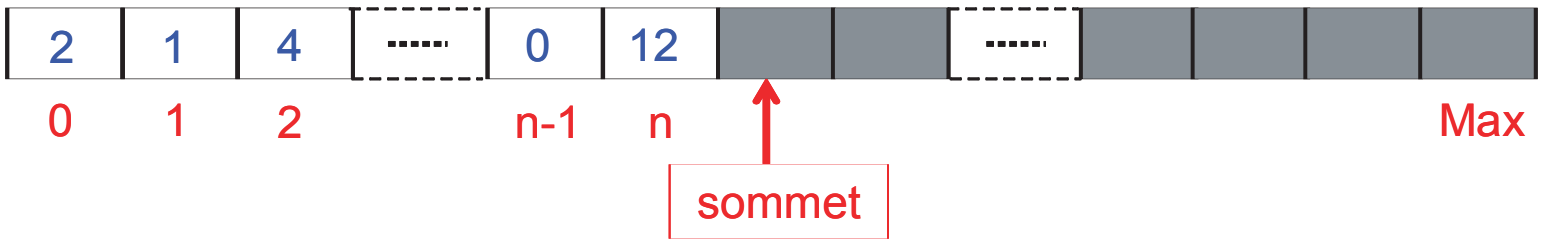
Vider la pile :

- La fonction suivante permet de vider le contenue d'une pile.

Code en C

```
Pile viderPile(Pile p)
  {
   while(p!=NULL)
     p=depiler(p);
   return p;
  }
```

- Comme le cas des listes, une pile peut être aussi implantée à l'aide d'un tableau.
- Les valeurs de la pile seront donc stockés dans les cases du tableau.
- Comme les piles gèrent des ensembles dynamiques, une taille maximum pour le tableau doit être donc définie lors des déclarations.
- C-à-d, on ne peut pas modifier la taille maximum du tableau lors de l'exécution d'un programme.
- Et si on désire modifier la taille maximum du tableau, il faut modifier aussi le programme.
- Par consequent, il faut définir un tableau dont la taille sera suffisante pour accueillir la plus grande pile pouvant être utilisée.



Déclaration d'une pile :

- En langage C, pour implanter une pile à l'aide d'un tableau, on utilise une structure constituée de deux champs,
  - un champs contenant un tableau, de taille suffisante pour accueillir la plus grande pile à traiter.
  - un autre champs contenant un entier représentant le sommet de la pile. En fait, cet entier contient l'indice+1 du sommet de la pile.

Code en C

```
typedef int element;
struct pileTab
 {
   element valeur[Max];
   int sommet;
 };
typedef struct pileTab pileTab;
```

Création et initialisation d'une pile :

- Pour créer une pile, on doit l'initialiser à un tableau de taille nulle.
- L'initialisation d'une pile, ou la création d'une pile vide, est généralement faite par la fonction suivante :

```
void pileVide(pileTab *p)
  {
   p->sommet=0;
  }
```

- Pour tester si une pile est vide, on utilise la fonction suivante :

```
 int testerPileVide(pileTab p)
 {
   return p.sommet==0;
 }
```

Ajouter un élément dans la pile (empiler) :

- Comme le cas d'implantation avec les pointeurs, il n'est possible d'insérer qu'une seule valeur en tête (ou sommet) de la pile.
- La valeur, une fois insérée, devient le sommet de la pile.

Code en C

```
pileTab empiler(element val, pileTab p)
  {
   p.valeur[p.sommet]=val;
   (p.sommet)++;
   return p;
  }
```

Supprimer un élément de la pile (dépiler) :

- De même que l'implantation avec les pointeurs, Il n'est possible de supprimer qu'une seule valeur dans une pile.
- Cette valeur n'est que la dernière ajoutée dans la pile, c-à-d le sommet de la pile.
- Ainsi, après cette suppression, le dernier élément inséré devient le sommet de la pile.

Code en C

```
pileTab depiler(pileTab p)
  {
   p.sommet=p.sommet-1;
   return p;
  }
```

Retourner l'élément supprimer de la pile :

- La fonction suivante permet de retourner le dernier élément (ou le sommet) retiré de la pile.

Code en C

```
int valeurSommet(pileTab *p)
  {
    int vdep;
    vdep=p->valeur[p->sommet-1];
    p->sommet=p->sommet-1;
    return vdep;
  }
```

récupérer le sommet de la pile :

- La fonction suivante permet de renvoyer la valeur du sommet d'une pile.

Code en C

```
element sommetPile(pileTab p)
  {
   return p.valeur[p.sommet-1];
  }
```

Vider la pile :

- La fonction suivante permet de vider le contenue d'une pile.

Code en C

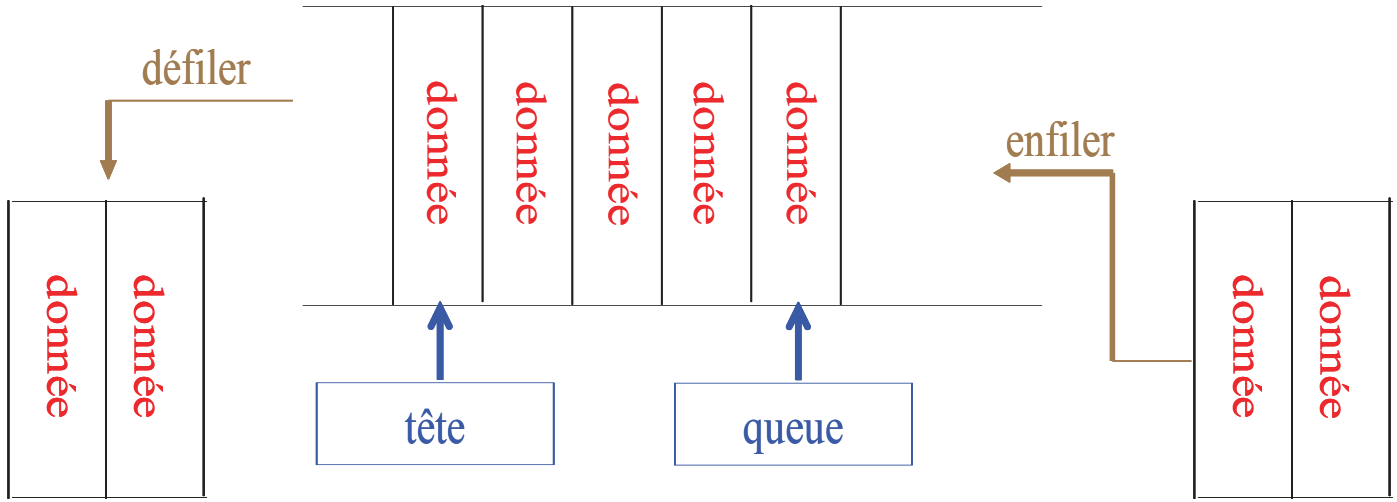
```
pileTab viderPile(pileTab p)
  {
   while(p.sommet!=0)
     p=depiler(p);
   return p;
  }
```

3. Files : 3.1 présentation générale

- Une file (connue aussi sous le nom de ***file d'attente***) est une structure de donnée identique à une pile sauf,
- les insertions se font à une extrémité appelée ***queue*** (fin de la file), les suppressions à l'autre appelée ***tête*** (debut de la file).
- Notons que dans une file, on a l'accès qu'à un seul élément, la ***tête*** de la file.
- Le dernier élément ajouté sera le dernier élément accessible.
- Lorsque on retire un élément de la file, on doit retirer toujours la tête, et cet élément étant le premier qui a été inséré dans la file.
- C-à-d, *le premier élément ajouté dans la file est le premier élément à en être supprimé*. Cette structure est connue sous le nom de ***FIFO*** (***First In, First Out***).
- Pour résumer, une file permet la mis en attente d'informations ou de données dans le but de les récupérer plus tard, dans l'ordre d'arrivée.
- La notion de file est également une structure utilisée en informatique :
  - la mise en attente de programme pour exécution,
  - l'accès à une périphérique, par exemple une imprimante.
  - etc.
- Généralement, il y a trois façons d'implanter une file en langage C :
  - soit en utilise un pointeur indiquant la queue de la file,
  - soit on utilise un tableau.
  - ou on peut utiliser des listes chaînées.

Opérations sur les files :

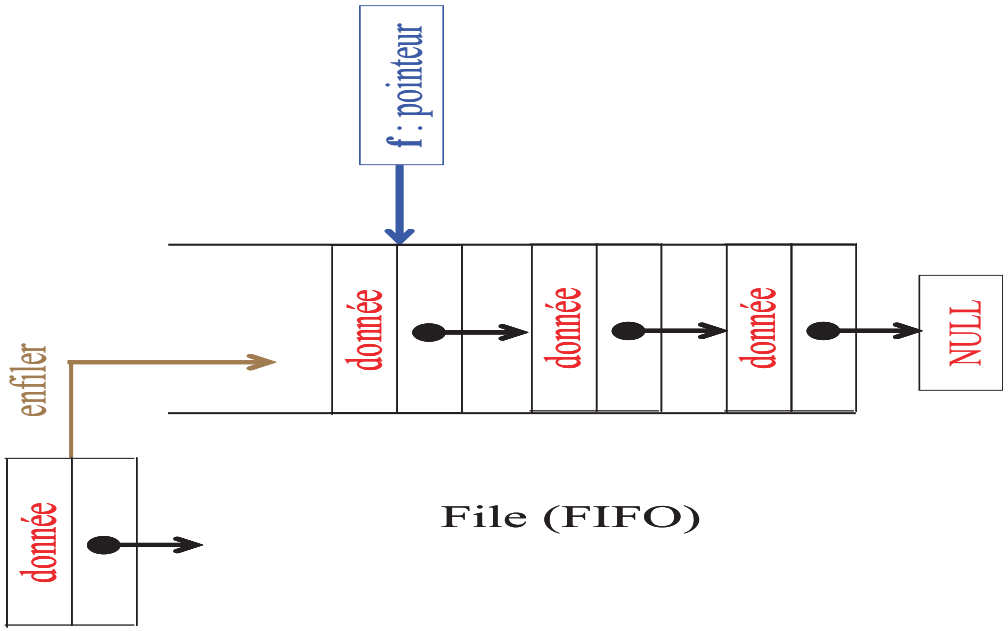
- Comme le cas des piles, dans une file, on ne peut autoriser que 5 opérations :
  1. *tester* si la file est vide,
  2. *enfiler* un élément, c-à-d l'ajouter à la queue de la file,
  3. *défiler* un élément, c-à-d le supprimer de la tête de la file,
  4. *récupérer* le premier élément de la file, c-à-d acceder à la tête de la file,
  5. *vider* le contenue de la file.



File (FIFO)

Déclaration d'une file :

- Comme le cas des piles t les listes, le langage C permet de réaliser des files à l'aide des structures.
- En effet, chaque cellule de la file est un objet constitué :
  - d'un champ de données,
  - d'un pointeur vers la cellule suivante.
- La déclaration correspondante est la suivante, où l'on suppose ici que les valeurs stockées dans la file sont de type entier.



File (FIFO)

Code en C

```
typedef int element;
struct celluleFile
 {
    element valeur;
    struct celluleFile * suivant;
 };
typedef struct celluleFile celluleFile, * File;
```

Création et initialisation d'une file :

- Pour créer une file, on doit l'initialiser à NULL, cela va nous permettre d'allouer la première cellule.
- L'initialisation d'une file, ou la création d'une file vide, est faite par la fonction suivante :

```
File fileVide( )
  {
   return NULL;
  }
```

- Pour tester si une file est vide, on utilise la fonction suivante :

```
 int testerFileVide(File f)
 {
  return f==NULL;
 }
```

Ajouter un élément dans la file (enfiler) :

- Il n'est possible d'insérer qu'une seule valeur en queue de la file.
- La valeur, une fois insérée, devient la queue de la file.

Code en C

```
File enfiler(element val, File f)
  {
    File fn;
    fn=malloc(sizeof(celluleFile));
    fn->valeur=val;
    if(f==NULL)
     {
        fn->suivant=NULL;
        return fn;
     }
    else
     {
        fn->suivant=f;
        f=fn;
        return f;
     }
  }
```

Supprimer un élément de la file (défiler) :

- Il n'est possible de supprimer qu'une seule valeur dans une file.
- Cette valeur n'est que la première entrée dans la file, c-à-d la tête de la file.

Code en C

```
File defiler(File f)
  {
    File pfile1, pfile2;
    if(f==NULL)
     return NULL;
    if(f->suivant==NULL)
     {
       free(f);
       return NULL;
     }
    pfile1=f;
    while(pfile1->suivant->suivant!=NULL)
      pfile1=pfile1->suivant;
    pfile2=pfile1->suivant;
    pfile1->suivant=NULL;
    free(pfile2);
    return f;
  }
```

Retourner l'élément supprimer de la pile :

- Puisque une pile permet la mise en attente d'informations dans le but de les récupérer après, la fonction suivante permet de retourner l'élément (ou le sommet) retiré de la pile.

Code en C

```
int valeurTete(File *f)
  {
    int vdef;
    File pfile1, pfile2;
    if((*f)->suivant==NULL)
     {
       vdef=(*f)->valeur;
       free(*f);
       return vdef;
     }
    pfile1=*f;
    while(pfile1->suivant->suivant!=NULL)
      pfile1=pfile1->suivant;
    pfile2=pfile1->suivant;
    pfile1->suivant=NULL;
    vdef=pfile2->valeur;
    free(pfile2);
    return vdef;
  }
```

Code en C

```
File valeurTete(File f, int *vdef)
  {
    File pfile1, pfile2;
    if(f->suivant==NULL)
     {
       *vdef=f->valeur;
       free(f);
       return NULL;
     }
    pfile1=f;
    while(pfile1->suivant->suivant!=NULL)
      pfile1=pfile1->suivant;
    pfile2=pfile1->suivant;
    *vdef=pfile2->valeur;
    pfile1->suivant=NULL;
    free(pfile2);
    return f;
  }
```

récupérer la queue de la file :

- Puisque chaque file peut être définie par sa queue, la fonction suivante permet de retourner la valeur de la queue d'une file.

Code en C

```
element queueFile(File f)
  {
   return f->valeur;
  }
```

récupérer la tête de la file :

- Puisque chaque file peut être définie aussi par sa tête, la fonction suivante permet de retourner la valeur de la tête d'une file.

Code en C

```
element teteFile(File f)
  {
   while(f->suivant!=NULL)
     f=f->suivant;
   return f->valeur;
  }
```

Vider la file :

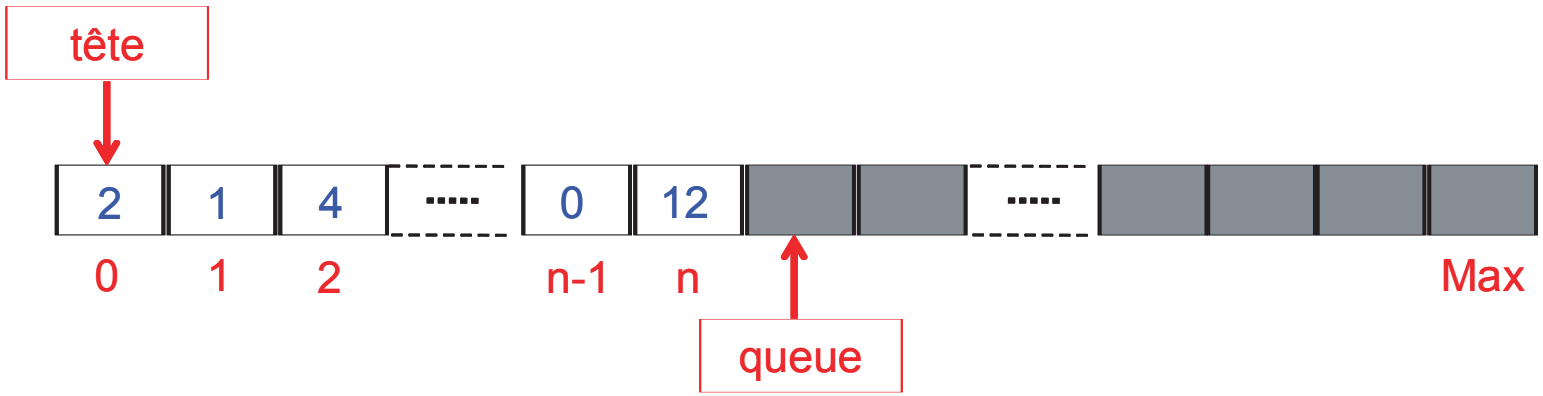
- La fonction suivante permet de vider le contenue d'une file.

Code en C

```
File viderFile(File f)
  {
   while(f!=NULL)
     f=defiler(f);
   return f;
  }
```

3. Files : 3.3 Implantation par tableau

- Comme le cas des piles et des listes, une file peut être aussi implantée à l'aide d'un tableau.
- Les valeurs de la file seront donc stockés dans les cases du tableau.
- Comme les files gèrent des ensembles dynamiques, une taille maximum pour le tableau doit être donc définie lors des déclarations.
- C-à-d, on ne peut pas modifier la taille maximum du tableau lors de l'exécution d'un programme.
- Et si on désire modifier la taille maximum du tableau, il faut modifier aussi le programme.
- Par consequent, il faut définir un tableau dont la taille sera suffisante pour accueillir la plus grande pile pouvant être utilisée.



Déclaration d'une file :

- En langage C, pour implanter une file à l'aide d'un tableau, on utilise une structure constituée de deux champs,
  - un champs contenant un tableau, de taille suffisante pour accueillir la plus grande file à traiter.
  - un autre champs contenant un entier représentant la queue de la file. En fait, cet entier contient l'indice+1 de la queue de la file.

Code en C

```
typedef int element;
struct fileTab
 {
   element valeur[Max];
   int queue;
 };
typedef struct fileTab fileTab;
```

Création et initialisation d'une file :

- Pour créer une file, on doit l'initialiser à un tableau de taille nulle.
- L'initialisation d'une file, ou la création d'une file vide, est généralement faite par la fonction suivante :

```
void fileVide(fileTab *f)
  {
   f->queue=0;
  }
```

- Pour tester si une file est vide, on utilise la fonction suivante :

```
 int testerFileVide(fileTab f)
 {
  return f.queue==0;
 }
```

Ajouter un élément dans la file (enfiler) :

- Comme l'implantation avec les pointeurs, il n'est possible d'insérer qu'une seule valeur en queue de la file.
- La valeur, une fois insérée, devient la queue de la file.

Code en C

```
fileTab enfiler(element val, fileTab f)
  {
   f.valeur[f.queue]=val;
   (p.queue)++;
   return f;
  }
```

Supprimer un élément de la file (défiler) :

- Comme l'implantation avec les pointeurs, il n'est possible de supprimer qu'une seule valeur dans une file.
- Cette valeur n'est que la première entrée dans la file, c-à-d la tête de la file.

Code en C

```
fileTab defiler(fileTab f)
  {
   int i;
   for(i=0;i<=f.queue-2;i++)
    f.valeur[i]=f.valeur[i+1];
   f.queue=f.queue-1;
   return f;
  }
```

Retourner l'élément supprimer de la file :

- La fonction suivante permet de retourner le dernier élément (ou la tête) retiré de la file.

Code en C

```
int queueFile(fileTab *f)
  {
   int vdef,i;
   vdef=f->valeur[0];
   for(i=0;i<=f.queue-2;i++)
     f.valeur[i]=f.valeur[i+1];
   f.queue=f.queue-1;
   return vdef;
  }
```

récupérer la queue de la file :

- La fonction suivante permet de renvoyer la valeur de la queue d'une file.

Code en C

```
element queueFile(fileTab f)
  {
    return f.valeur[f.queue-1];
  }
```

récupérer la tête de la file :

- Puisque chaque file peut être définie aussi par sa tête, la fonction suivante permet de retourner la valeur de la tête d'une file.

Code en C

```
element teteFile(fileTab f)
  {
   return f.valeur[0];
  }
```

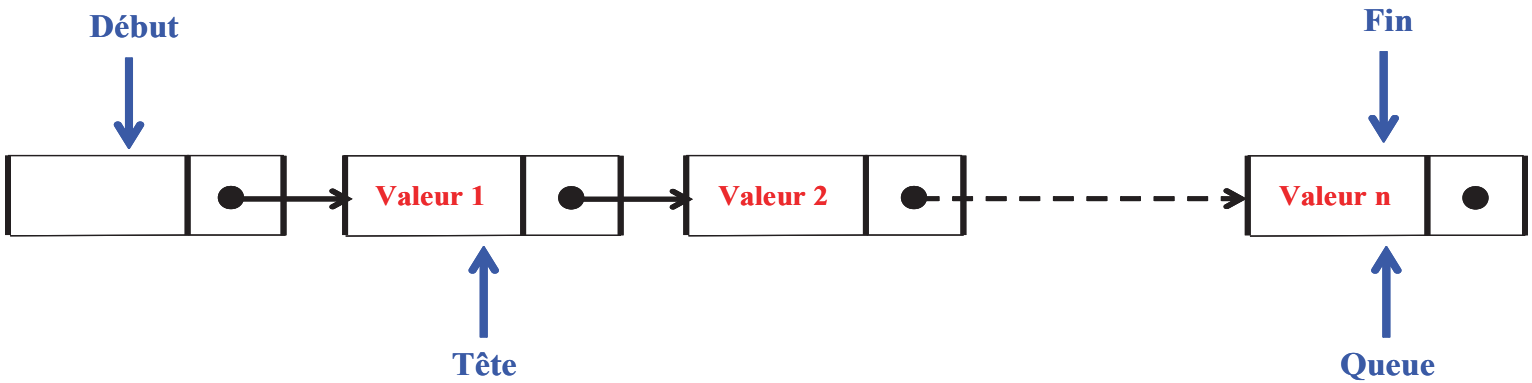
Vider la file :

- La fonction suivante permet de vider le contenue d'une file.

Code en C

```
fileTab viderFile(fileTab f)
  {
   while(f.queue!=0)
    f=defiler(f);
   return f;
  }
```

- Une autre façon de gérer des files consiste à utiliser des listes chaînées.
- En effet, comme une file est un cas particulier de liste chaînée, chaque file sera définie par deux cellules : une cellule qui pointe vers le début (la tête) de la file et une autre qui pointe sur sa fin (queue).



File représenté par une liste chaînée

Déclaration d'une file :

- En langage C, pour implanter une file à l'aide d'une liste chaînée, on utilise une structure constituée de deux champs contenant des pointeurs de type liste,
  - le premier pointe vers la cellule représentant la tête de la file.
  - le deuxième pointe sur la cellule représentant la queue de la file.

Code en C

```
struct fileListe
{
  liste deb;
  liste fin;
};
typedef struct fileListe fileListe;
```

Création et initialisation d'une file :

- Pour créer une file, la cellule qui pointe sur la queue de la file doit pointer sur la même adresse du pointeur qui pointe vers la cellule début.
- À noter que cette adresse ne contient aucun valeur de type `element`.
- L'initialisation d'une file, ou la création d'une file vide à l'aide d'une liste est faite par la fonction suivante :

```
fileListe fileVide()
  {
   fileListe f;
   f.deb=(liste)malloc (sizeof (cellule));
   f.fin=f.deb;
   return f;
  }
```

- Pour tester si une file est vide, on utilise la fonction suivante :

```
 int testerFileVide(fileListe f)
 {
  return f.deb==f.fin;
 }
```

Exercice :

Écrire les fonctions suivantes en utilisant une implantation des files par les listes chaînées :

- *enfiler* un élément,
- *défiler* un élément,
- *récupérer* la tête de la file,
- *vider* le contenue de la file.

Chapitre III

Structure de données
Les listes chaînées

1. Introduction : Objectif et Motivation

- Rappelons qu'un programme informatique est un ensemble d'instructions et de données qui permettent à l'ordinateur de traiter des informations.
- Ces informations sont organisées dans la mémoire principale et dans les mémoires auxiliaires.
- Par ailleurs, notons que l'ordinateur définit au niveau matériel :
  - Un jeu d'instructions élémentaires (additionner, soustraire, multiplier...),
  - Quelques représentations de données élémentaires (bits).
- Mais, il faut signaler que le matériel ne supporte que des données élémentaires, donc non structurées.
- On peut ainsi définir une donnée structurée comme étant une collection organisée de données élémentaires.
- En informatique, on appelle une telle collection, une ***structure de données***.
- Une ***structure de données*** est donc une manière d'organiser et de représenter des informations en mémoire centrale et dans les mémoires auxiliaires.
- Une structure de données contient deux types de renseignement :
  - les informations à stocker,
  - les liens entre ces informations.
- On peut distinguer deux types de liens :
  - Liens implicites : l'adresse de l'information est calculée en fonction de l'emplacement physique de l'information en mémoire.
  - liens explicites : les pointeurs.

1. Introduction : Objectif et Motivation

- En langage C, il existe un grand nombre de structure de données :
  Exemple : les tableaux les structures, les listes chaînées, les piles, les files, les arbres binaires...
- Le choix d'une structure dépend, en général, des opérations à réaliser sur les informations. Certaines structures sont plus adaptés que d'autres à certaines opérations.
- ***L'objectif*** est la gestion d'un ensemble fini d'éléments dont le nombre n'est pas fixé.
- Les éléments de cet ensemble peuvent être de différentes types : nombres entiers ou flottants, chaînes de caractères,...
- On ne s'intéressera pas aux éléments de l'ensemble en question mais aux opérations que l'on effectue sur cet ensemble, indépendamment de la nature de ses éléments.
- En programmation informatique, ces ensembles sont considérés comme des objets dynamiques.
- En effet, le nombre de leurs éléments varie au cours de l'exécution du programme, puisqu'on peut y ajouter et supprimer des éléments en cours de traitement.

- Plus précisément les opérations que l'on s'autorise sur les ensembles sont les suivantes :
  - tester si un ensemble est vide.
  - ajouter un élément à un ensemble.
  - supprimer un élément à un ensemble.
  - rechercher un élément dans un ensemble.

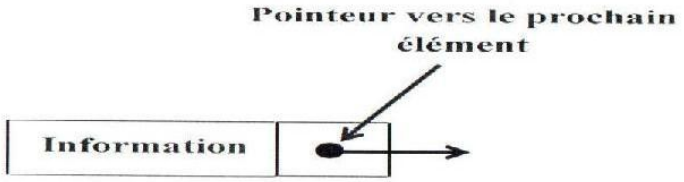
- Cette gestion des ensembles doit, pour être efficace, répondre à deux critères :
  - un minimum de place mémoire utilisée,
  - et un minimum d'instructions élémentaires pour réaliser une opération.

1. Introduction : Objectif et Motivation

- Notons que la structure de données la plus utilisée pour manipuler des données sont les tableaux.
- Mais, dans certains cas, les tableaux ne constituent pas la meilleure solution pour stocker et manipuler les données.
- En effet, Si on veut supprimer un élément au milieu d'un tableau, il faut recopier les éléments temporairement, réallouer le tableau, puis le remplir à partir de l'élément supprimé.
- Ces manipulations sont coûteuses en ressources.
- En langage C, les ***listes chaînées*** (ou tout simplement les listes) peuvent être considérées comme des structures de données très pratiques pour manipuler des éléments d'un ensemble d'objets dynamiques.
- En effet, une liste chaînée est différente par rapport à un tableau dans le sens où ses éléments sont répartis dans la mémoire et reliés entre eux par des pointeurs.
- Ceci permet d'ajouter et de supprimer des éléments d'une liste à n'importe quel endroit, sans modifier la liste entière.

2. Les listes chaînées

- Une *liste chaînée* est une structure de base de la programmation informatique.
- C'est une structure de données dans laquelle les objets sont stockés de manière ordonnée (l'ordre signifie que chaque objet possède une position dans la liste).
- Cependant, contrairement au tableau, pour lequel l'ordre est déterminé par les indices, l'ordre d'une liste chaînée est déterminé par un pointeur dans chaque objet.
- Les listes chaînées sont des structures dynamiques capables,
  - d'accueillir un nombre infini d'éléments,
  - et de supporter toutes les opérations sur les ensembles dynamiques (insertion, suppression, recherche,...).
- *Dans la suite du chapitre*, on utilise la liste chaînée pour représenter un ensemble d'éléments.
- Chaque élément est contenu dans une *cellule* (appelée aussi *maillon*), et chaque cellule est composée d'une partie contenant l'information et d'au moins un pointeur.



Représentation d'une cellule avec un seul pointeur.

2. Les listes chaînées

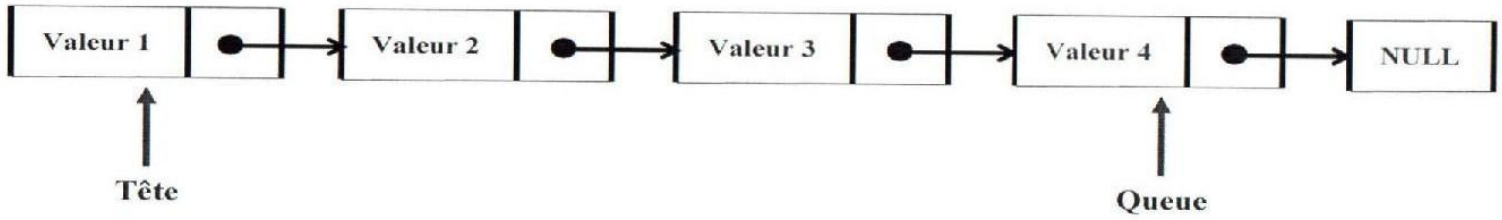
- Chaque élément est accessible par celui qui le précède grâce au pointeur.
- Une liste chaînée est donc une succession de cellules, dont le dernier porte une adresse de fin (le pointeur `NULL`). La première cellule de la liste est appelée tête et la dernière cellule est appelée queue.
- L'accès à la liste chaînée se fait par un pointeur vers le premier élément.
- Pour atteindre un des éléments de la liste,

  on est obligé de parcourir la liste depuis le début.
- A partir d'un élément quelconque,

  on ne peut atteindre que le suivant.
- Une liste chaînée peut prendre différentes formes :
  - Elle peut être ***simple*** (dite aussi ***simplement chaînée***) : chaque cellule est composé de l'information et d'un seul pointeur (pointe sur l'adresse de la cellule suivante).
  - Ou ***doublement chaînée*** : chaque cellule est composé de l'information et de deux pointeurs (l'un pointe sur l'adresse de la cellule suivante et l'autre sur celle de la cellule précédente).
  - Ou ***circulaire*** : la tête pointe sur la queue de la liste, et la queue pointe sur la tête de la liste.

2. Les listes chaînées : 2.1 Les listes simplement chaînées

- Rappelons qu'une liste simplement chaînée est une liste où chaque cellules contient des informations accessibles et modifiables par l'utilisateur ainsi qu'un pointeur vers la cellule suivante.



- Comme les listes simplement chaînées sont une structure de données importante en programmation informatique,
- il est fondamental de s'intéresser à la manière de les implanter en vue de leurs manipulations informatique.
- Deux modes d'implantation peuvent être envisagés :
  - Implantation par pointeur.
  - Implantation par tableau.

2.1 Les listes simplement chaînées : Implantation par pointeur

Déclaration

- Le langage C permet de réaliser des listes chaînées à l'aide des structures.
- En effet, les cellules sont des objets constitués :
  - d'un champ de données,
  - d'un pointeur vers la cellule suivante.
- La déclaration correspondante est la suivante, où l'on suppose ici que les valeurs stockées dans chaque cellule sont de type entier.

Code en C

```
typedef int element;
struct cellule
 {
    element valeur;
    struct cellule * suivant;
 };
typedef struct cellule cellule, * liste;
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Remarque 1:

Notons qu'on peut créer des listes chaînées contenant des valeurs qui sont :

- de différents type, c-à-d des entiers, des flottants, des caractères,
- des tableaux, des structures,
- une combinaison de plusieurs types,
- ou des listes chaînées.

Exemple :

```
typedef double element;
struct cellule
 {
    element valeur;
    struct cellule * suivant;
 };
typedef struct cellule cellule, * liste;
```

Remarque 2:

Les instructions suivantes permettent de déclarer, de façons différentes mais équivalentes, une liste chaînée :

```
struct cellule *l;
cellule *l;
liste l;
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Création et initialisation

- Pour créer une liste chaînée, on doit l'initialiser à NULL (macro de stdlib.h), cela va nous permettre d'allouer la première cellule.
- L'initialisation d'une liste chaînée, ou la création d'une liste vide, est généralement faite par la fonction suivante :

```
liste creerListeVide( )
 {
   return NULL;
 }
```

- Pour tester si une liste est vide, on utilise la fonction suivante :

```
int ListeVide(liste l)
 {
   return l==NULL;
 }
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Insertion d'une cellule (Ajout d'une valeur dans une liste) :

- Il est toujours possible d'insérer des cellules (ajouter des valeurs) à une liste chaînée donnée.
- En effet, il y a plusieurs façons d'ajouter des cellules dans des listes chaînées :
  - ajouter une nouvelle cellule en tête de la liste,
  - ajouter une nouvelle cellule à la fin de la liste.
  - ajouter une nouvelle cellule à un emplacement quelconque dans la liste.
  - Exemple : Les fonctions `ajouterElementTeteListe()` et `ajouterElementFinListe()` permettent respectivement l'ajout d'un élément en tête et à la fin d'une liste.

Ajout d'un élément en tête de la liste

```
liste ajouterElementTeteListe(element val, liste l)
  {
    liste ln;
    ln=malloc(sizeof(cellule));
    ln->valeur=val;
    ln->suivant=l;
    return ln;
  }
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Insertion d'une cellule (Ajout d'une valeur dans une liste) :

Ajout d'un élément à la fin de la liste

```
liste ajouterElementFinListe(element val, liste l)
  {
    liste ln,lp;
    ln=malloc(sizeof(cellule));
    ln->valeur=val;
    ln->suivant=NULL;
    if(l==NULL)
      return ln;
    lp=l;
    while(lp->suivant != NULL)
      lp= lp->suivant;
    lp->suivant =ln;
    return l;
  }
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Suppression d'une cellule (supprimer une valeur dans une liste) :

- Il est possible également de supprimer des cellules (des valeurs) dans une liste chaînée donnée.
- En effet, il y a plusieurs façons de supprimer des cellules dans des listes chaînées :
  - supprimer une cellule en tête de la liste,
  - supprimer une cellule à la fin de la liste,
  - supprimer une cellule quelconque dans une liste.

Suppression d'un élément en tête de la liste

```
liste supprimerElementTeteListe(liste l)
  {
    liste lp;
    if(l==NULL)
     return NULL;
    lp=l->suivant;
    free(l);
    return lp;
  }
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Suppression d'une cellule (supprimer une valeur dans une liste) :

Suppression d'un élément à la fin de la liste

```
liste supprimerElementFinListe(liste l)
  {
    liste lp1,lp2;
    if(l==NULL)
     return NULL;
    if(l->suivant==NULL)
     {
      free(l);
      return NULL;
     }
    lp1=l;
    while(lp1->suivant->suivant!=NULL)
     lp1=lp1->suivant;
    lp2=lp1->suivant;
    lp1->suivant=NULL;
    free(lp2);
    return l;
  }
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Suppression d'une cellule (supprimer une valeur dans une liste) :

Suppression d'un élément quelconque dans la liste (Méthode itérative)

```
liste supprimerItertiveElementListe(element val, liste l)
  {
    liste lp1,lp2;
    if(l!=NULL)
      if(l->valeur==val)
        {
          lp1=l;
          l=l->suivant;
          free(lp1);
        }
      else
        {
          lp2=l;
          while(lp2->suivant!=NULL && lp2->suivant->valeur!=val)
            lp2=lp2->suivant;
          if(lp2->suivant!=NULL)
            {
              lp1=lp2->suivant;
              lp2->suivant=lp2->suivant->suivant;
              free(lp1);
            }
        }
    return l;
  }
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Suppression d'une cellule (supprimer une valeur dans une liste) :

Suppression d'un élément quelconque dans la liste (Méthode récurrente)

```
liste supprimerRecurrenteElementListe(element val, liste l)
 {
   liste lp;
   if(l!=NULL)
     if(l->valeur==val)
       {
         lp=l;
         l=l->suivant;
         free(lp);
       }
     else
       l->suivant=supprimerRecurrenteElementListe(val, l->suivant);
   return l;
}
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Recherche dans une liste :

- Étant donnée une liste chaînée `liste` et une valeur `val`, l'objectif ici est d'effectue un parcours de liste pour rechercher la valeur `val` dans `liste`. Pour ceci, on peut envisager deux types de fonction :
  - une fonction qui teste si la valeur recherchée `val` figure dans la liste `liste`.
  - une fonction qui renvoie l'adresse de la cellule contenant la valeur recherchée `val`.

fonctions qui testent si val figure dans liste

Recherche dans une liste (Méthode récurrente)

```
int rechercherRecListe(element val, liste l)
 {
 if(l==NULL)
  return 0;
 if(l->valeur==val)
  return 1;
 return rechercherRecListe(val, l->suivant);
}
```

Recherche dans une liste (Méthode itérative)

```
int rechercherIterListe(element val, liste l)
{
 while(l!=NULL)
 {
  if(l->valeur==val)
   return 1;
  l=l->suivant;
 }
 return 0;
}
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Recherche dans une liste :

fonctions qui renvoient l'adresse de la cellule contenant val

Recherche dans une liste (Méthode récurrente)

```
liste rechercherLRec(element val, liste l)
 {
  if(l==NULL)
   return NULL;
  if(l->valeur==val)
   return l;
  return rechercherLRec(val, l->suivant);
}
```

Recherche dans une liste (Méthode itérative)

```
liste rechercherLIter(element val, liste L)
 {
  while(l!=NULL)
  {
    if(l->valeur==val)
     return l;
    l=l->suivant;
  }
 return NULL;
 }
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Affichage d'une liste :

- La fonction suivante permet d'afficher les valeurs stockée d'une liste.

Affichage d'une liste

```
void afficherListe(liste l)
 {
   while(l!=NULL)
     {
       printf("%d ",l->valeur);
       l=l->suivant;
     }
   printf("\n");
 }
```

2.1 Les listes simplement chaînées : Implantation par pointeur

Exercice :

Écrire un programme complet en langage C qui permet de

- construire une liste chaînée,
- afficher une liste chaînée,
- calculer la longueur d'une liste chaînée,
- rechercher une valeur dans la liste et de retourner sa position dans la liste.
- effectuer les manipulations suivantes :
  - supprimer une valeur quelconque de la liste,
  - supprimer une valeur en début de liste,
  - supprimer une valeur en fin de liste,
  - supprimer une valeur à une position donnée dans la liste,
  - supprimer une valeur avant ou après une position donnée dans la liste.

2.1 Les listes simplement chaînées : Implantation par tableau

- Une liste peut être aussi implantée à l'aide d'un tableau.
- Les éléments de la liste seront donc stockés dans les cases contiguës du tableau.
- Comme les listes sont des structures de données qui gèrent des ensembles dynamiques, une taille maximum pour le tableau doit être donc définie lors des déclarations.
- C-à-d, on ne peut pas modifier la taille maximum du tableau lors de l'exécution d'un programme.
- Et si on désire modifier la taille maximum du tableau, il faut modifier aussi le programme.
- Par consequent, il faut définir un tableau dont la taille sera suffisante pour accueillir la plus grande liste pouvant être utilisée!!!
- Mais cette démarche peut encombrer la mémoire de l'ordinateur.
- C'est pourquoi, il est en général préférable d'utiliser l'implantation des listes par des pointeurs.

2.1 Les listes simplement chaînées : Implantation par tableau

Déclaration

- En langage C, pour implanter une liste chaînée à l'aide d'un tableau, on utilise les structures.
- En effet, la liste est un objet constitués :
  - d'un tableau, de taille suffisante pour accueillir la plus grande liste à traiter.
  - d'un entier contenant l'indice de la fin de la liste. Cet entier contient l'indice+1 du dernier élément de la liste, ou la taille de la liste.

- La déclaration correspondante est la suivante, où l'on suppose ici que les valeurs stockées dans la liste sont de type entier.

Code en C

```
typedef int element;
struct listeTab
 {
    element valeur[taille_MAX];
    int fin;
 };
typedef struct listeTab listeTab, liste;
```

2.1 Les listes simplement chaînées : Implantation par tableau

Création et initialisation

- Pour créer une liste chaînée, on doit l'initialiser à un tableau de taille nulle.
- L'initialisation d'une liste chaînée, ou la création d'une liste vide, est généralement faite par la fonction suivante :

```
int creerListeVide( )
 {
   return 0;
 }
```

- Pour tester si une liste est vide, on utilise la fonction suivante :

```
int ListeVide(liste l)
 {
   return l.fin==0;
 }
```

2.1 Les listes simplement chaînées : Implantation par tableau

Ajout d'une valeur à une liste :

- Il est toujours possible d'ajouter des valeurs à une liste chaînée donnée.
- En effet, comme l'implantation par les pointeurs, il y a différentes façons d'ajouter des valeurs à une liste chaînée en particulier on peut
  - ajouter une nouvelle valeur en tête de la liste,
  - ajouter une nouvelle valeur à la fin de la liste.
  - ajouter une nouvelle valeur à une position quelconque de la liste.

Ajout d'un élément en tête de la liste

```
liste ajouterTeteLTab(element val, liste l)
 {
   int i;
   l.fin=l.fin+1;
   for(i=l.fin-1;i>=1;i--)
     l.valeur[i]=l.valeur[i-1];
   l.valeur[0]=val;
   return l;
 }
```

Ajout d'un élément à la fin de la liste

```
liste ajouterFinLTab(element val, liste l)
 {
   l.valeur[l.fin]=val;
   l.fin=l.fin+1;
   return l;
 }
```

2.1 Les listes simplement chaînées : Implantation par tableau

Ajout d'une valeur à une liste :

Ajout d'un élément après une position donnée dans la liste

```
liste  ajouterApresPosLTab(element val, int pos, liste l)
 {
   int i;
   l.fin=l.fin+1;
   for(i=l.fin-1;i>=pos+1;i--)
     l.valeur[i]=l.valeur[i-1];
   l.valeur[pos]=val;
   return l;
 }
```

Ajout d'un élément après un élément donnée dans la liste

```
liste ajouterApresElementLTab(element val, element elem, liste l)
 {
   int i;
   for(i=0;i<l.fin;i++)
     if(l.valeur[i]==elem)
       l=ajouterApresPosLTab(val, i+1, l);
     return l;
 }
```

2.1 Les listes simplement chaînées : Implantation par tableau

Suppression d'une valeur dans une liste :

- Il est possible également de supprimer des valeurs dans une liste donnée.
- En effet, il y a différentes façons de supprimer des valeurs dans des listes :
  - supprimer une valeur en tête de la liste,
  - supprimer une valeur à la fin de la liste,
  - supprimer une valeur quelconque dans une liste.

Suppression d'un élément en tête de la liste

```
liste supprimerTeteLTab(liste l)
  {
    int i;
    l.fin=l.fin-1;
    for(i=0;i<l.fin;i++)
      l.valeur[i]=l.valeur[i+1];
      return l;
  }
```

Suppression d'un élément en fin de la liste

```
liste supprimerFinLTab(liste l)
  {
    l.fin=l.fin-1;
    return l;
  }
```

2.1 Les listes simplement chaînées : Implantation par tableau

Suppression d'une valeur dans une liste :

Suppression d'un élément quelconque dans une liste

```
liste supprimerElemLTab(element val, liste l)
 {
   int i,j;
   for(i=0;i<l.fin;i++)
     if(l.valeur[i]==val)
       j=i;
   l.fin=l.fin-1;
   for(i=j;i<l.fin;i++)
     l.valeur[i]=l.valeur[i+1];
   return l;
 }
```

Suppression d'un élément quelconque à une position donnée d'une liste

```
liste supprimerPosLTab(int pos, liste l)
 {
   int i;
   l.fin=l.fin-1;
   for(i=pos-1;i<l.fin;i++)
     l.valeur[i]=l.valeur[i+1];
   return l;
 }
```

2.1 Les listes simplement chaînées : Implantation par tableau

Recherche dans une liste :

- L'objectif ici est de rechercher une valeur `val` dans `liste`. Pour ceci, on peut envisager deux types de fonction :
  - une fonction qui teste si la valeur recherchée `val` figure dans la liste `liste`.
  - une fonction qui renvoie la position dans la liste de la valeur contenant la valeur recherchée `val`.

Tester si la valeur recherchée existe dans la liste

```
int rechercherLTab(element val, liste L)
 {
   int i;
   for(i=0;i<l.fin;i++)
     if(l.valeur[i]==val)
       return 1;
   return 0;
 }
```

Rechercher une position d'un élément d'une liste

```
int rechercherPosLTab(element val, liste L)
 {
   int i;
   if(l.fin==0)
     return -1;
   else
     for(i=0;i<l.fin;i++)
       if(l.valeur[i]==val)
         return i+1;
   return 0;
 }
```

2.1 Les listes simplement chaînées : Implantation par tableau

Affichage d'une liste :

- La fonction suivante permet d'afficher les valeurs stockées d'une liste.

Affichage d'une liste

```
void afficherLTab(liste l)
  {
    int i;
    if(l.fin<=0)
      printf("votre liste est vide!!!\n");
    else
     {
      for(i=0;i<l.fin;i++)
        printf("%d ",l.valeur[i]);
      printf("\n\n");
     }
  }
```

2.1 Les listes simplement chaînées : Implantation par tableau

Exercice :

Écrire un programme complet en langage C qui permet, en utilisant l'implantation par tableau, de

- saisir une liste chaînée,
- afficher une liste chaînée,
- calculer la longueur d'une liste chaînée,
- rechercher une valeur dans la liste et de retourner sa (ou ses) position(s) dans la liste.
- effectuer les manipulations suivantes :
  - supprimer une valeur quelconque de la liste,
  - supprimer une valeur en début de liste,
  - supprimer une valeur en fin de liste,
  - supprimer une valeur à une position donnée dans la liste,
  - supprimer une valeur avant ou après une position donnée dans la liste.