# Algorithmique et Structures de Données

L2 MPI-MI

Dr Ousmane DIALLO

# Chap13: Les structures de données élémentaires



## 13.1. Introduction

- De manière générale, résoudre un problème avec un ordinateur revient à déterminer un algorithme opérant sur une ***structure de données*** bien choisie.
- En d'autres termes, il faut structurer le codage de l'information de façon adéquate.
- Ce chapitre présente quelques exemples élémentaires, notamment les ***listes***, les ***piles***, les ***files*** et les ***arbres***, tous représentant la notion d'ensemble.
- En informatique il existe plusieurs manières de représenter la notion mathématique d'ensemble.
- *L'objectif principal est de gérer un ensemble fini d'éléments dont le nombre n'est pas fixé à priori.*
- Les éléments de cet ensemble peuvent être des entiers ou des réels, des chaînes de caractères ou même des objets informatiques plus complexes.

## 13.1. Introduction

- il est possible d'appliquer un certain nombre d'opérations sur cet ensemble indépendamment de la nature des élément le constituant.
- La série d'opération potentiellement supportée est la suivante :
  - ***vide(E)*** permet de tester si l'ensemble ***E*** est vide ou pas
  - ***recherche(E,x)*** vérifie l'appartenance de l'élément ***x*** à l'ensemble ***E***
  - ***insertion(E, x)*** ajoute ***x*** à l'ensemble ***E***
  - ***suppression(E,x)*** supprime l'élément ***x*** de l'ensemble ***E***

## 13.2. Listes chaînées
### 13.2.1. Définition

- Les listes chaînées sont, comme les tableaux, des structures de données linéaires, c'est à dire qui permettent de relier des données en séquence (on peut numéroter les éléments).

- *Une liste est une représentation de données par un ensemble L de valeurs qui possède les propriétés suivantes :*
  - *l'existence d'un premier élément $L_1$, que nous appellerons **tête** de L (TETE(L))*
  - *l'existence d'un dernier élément $L_n$ appelé **queue** de liste*
  - *l'existence pour tout $L_i$ de L, à l'exception de la queue d'un élément $L_{i+1}$ que nous appellerons **successeur** de $L_i$*

- Contrairement aux tableaux, Les éléments d'une liste ne sont pas forcément contigüs en mémoire et ils sont reliés par des ***pointeurs***.
- Ces éléments sont ajoutés au fur et à mesure par allocation dynamique lors de l'exécution.
  - Les listes chaînées sont ainsi qualifiées de ***structures de données dynamiques***.

## 13.2. Listes chaînées
## 13.2.2. Fonctions d'accès à une liste

- Les fonctions d'accès à une liste sont les suivantes :
  - $Premier(L)=L_1$
  - $Dernier(L) = L_n$
  - $Successeur(L_i) = L_{i+1}$
  - $Predeceseur(L_i)=L_{i-1}$
  - $Successeur(L_n) = nil$
  - $Predeceseur(L_1)=nil$

## 13.2. Listes chaînées
## 13.2.3. Opérations sur une liste

- Les opérations suivantes sont admises sur les listes :
  - ***Lecture d'un élément*** : il s'agit d'accéder à un élément de rang quelconque et d'en déduire toutes les informations le concernant
  - ***Ecriture d'un élément*** : il s'agit de modifier l'information contenue dans un élément
  - ***Suppression d'un élément*** : Pour que la structure reste cohérente, il faut mettre à jour la fonction Successeur à chaque fois qu'on ôte un élément
    - Si on veut supprimer $L_i$ pour $1 \leq i \leq n$
      - $Successeur(L_{i-1}) = L_{i+1}$
    - Si celui-ci est le premier de la liste ($L_1$) alors :
      - $Premier(L) = Successeur(L_1)$
    - Si celui-ci est le dernier élément de la liste alors :
      - $Dernier(L) = L_{n-1}$ et par conséquent $Successeur(L_{n-1}) = nil$

## 13.2. Listes chaînées
## 13.2.3. Opérations sur une liste

- Les opérations suivantes sont admises sur les listes :
  - ***Adjonction d'un nouvel élément $L_s$***
    - ajout au milieu entre $L_i$ et $L_{i+1}$ , on fait dans l'ordre :
      - $Successeur(L_s) = L_{i+1}$
      - $Sucesseur(L_i) = L_s$
    - ajout en fin de liste :
      - $Successeur(L_n) = L_s$
      - $Successeur(L_s) = Nil$
    - ajout au début :
      - $Successeur(L_s) = L_1$
      - $Premier(L) = (L_s)$

## 13.2. Listes chaînées
## 13.2.4. Définition d'un type liste chaînée

- Chaque élément de la liste est contenu dans une ***cellule*** ou ***maillon***.
- La cellule est de type enregistrement (record) et contient deux champs principaux
  - un champ appelé ***clé*** contenant une information sur l'élément en question
  - un champ ***suivant (pointeur)*** contenant l'adresse de la cellule suivante de la liste chaînée
- Si le champ suivant d'un élément vaut *NIL*, cet élément n'a pas de successeur et est donc le dernier élément ou la ***queue*** de liste.
- Le premier élément est appelé la ***tête*** de la liste.
- Une liste *L* est manipulée via un pointeur vers son premier élément que l'on notera *Tete(L)*.
- Si ***Tete(L)*** vaut *NIL* alors la liste *L* est vide.

## 13.2. Listes chaînées
## 13.2.4. Définition d'un type liste chaînée

```
(*En Algorithme*)

type
maillon =  Enregistrement
             valeur : entier
             suivant: Pointeur sur maillon
           finEnregistrement;

Liste = Pointeur sur maillon;
```

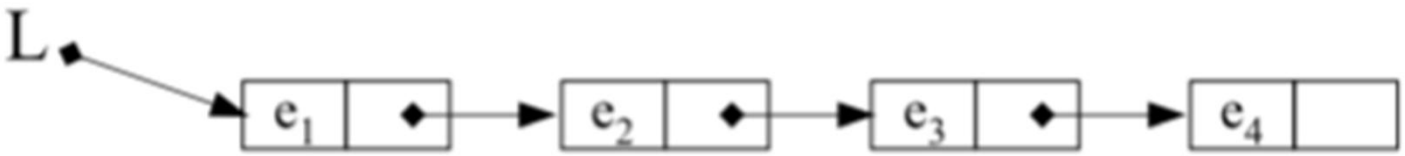
```
(*En Pascal*)

type
maillon =  Record
             valeur : integer;
             suivant: ^maillon
             end;

Liste = ^maillon;
```

- Après avoir défini le type Liste, on peut l'utiliser comme un type normal pour déclarer des variables de ce type.

```
(*En Pascal*)
var
L : Liste;
```

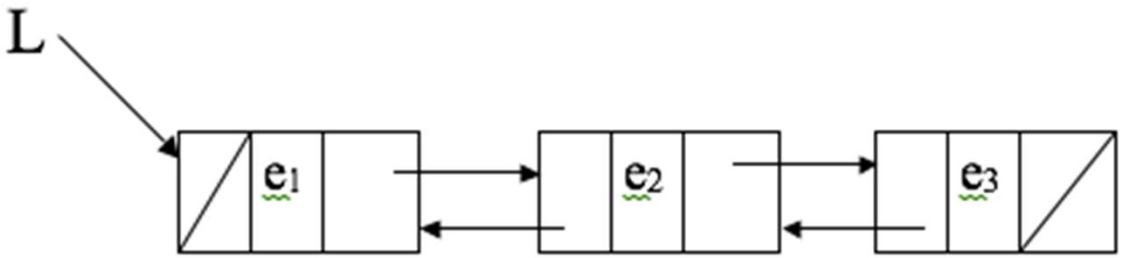
## 13.2. Listes chaînées

### 13.2.5. Représentation graphique d'une liste chaînée



## 13.2. Listes chaînées

- Une liste peut prendre plusieurs formes :
  - **Liste doublement chaînée :**
    - en plus du champ ***suivant***, chaque élément contient un champ ***precedent*** qui est un pointeur sur l'élément précédent dans la liste.
    - Si le champ ***precedent*** d'un élément vaut ***nil***, cet élément n'a pas de prédecesseur et est donc le premier élément ou la ***tête*** de liste.
    - Une liste qui n'est pas doublement chaînée est dite ***simplement chaînée***



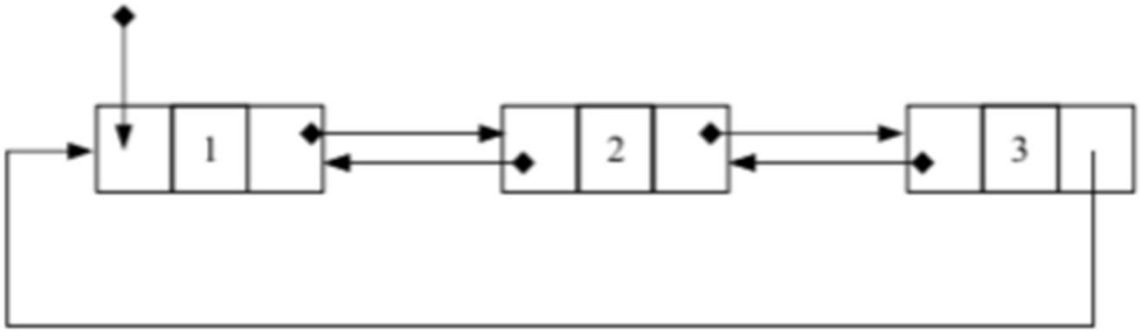
## 13.2. Listes chaînées

- Une liste peut prendre plusieurs formes :
  - **Liste doublement chaînée :**
    - **Déclaration:**

```
                    (*En Algorithme*)

type
maillon =  Enregistrement
                valeur : entier
                suivant: Pointeur sur maillon
                precedent: Pointeur sur maillon
              finEnregistrement;

Liste = Pointeur sur maillon;
```

## 13.2. Listes chaînées

- Une liste peut prendre plusieurs formes :
  - **Liste circulaire :**
    - si le champ prédécesseur de la tête de la liste pointe sur la queue, et si le champ successeur de la queue pointe sur la tête.
    - La liste est alors vue comme un anneau.



## 13.2. Listes chaînées

## 13.2.6. Algorithmes de manipulation de listes chaînées

- **Création d'une liste vide :**

```
(*En Pseudo-code*)

Procedure listeVide(var L: Liste);
Debut
      L:=nil;
Fin
```

## 13.2. Listes chaînées

## 13.2.6. Algorithmes de manipulation de listes chaînées

- Test de la liste vide :

```
(*En Pseudo-code*)

Fonction estVide( L: Liste) : booleen;
Debut
      estVide := (L=nil);
Fin
```

## 13.2. Listes chaînées

## 13.2.6. Algorithmes de manipulation de listes chaînées

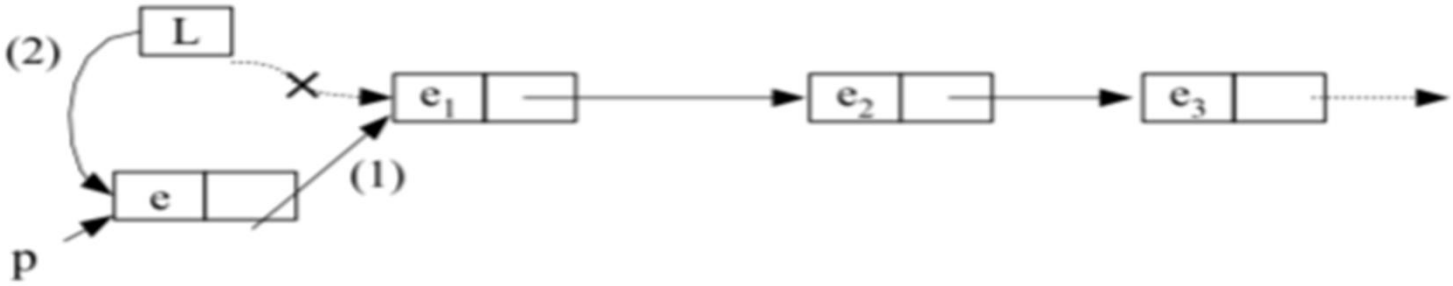
- **Tête de liste :**

```
                    (*En Pseudo-code*)

Fonction teteListe( L: Liste) : integer;  //suppose que la liste contient des entiers
Debut
      if (estVide (L)) then
                        writeln('Erreur, liste vide')
                     else
                        teteListe := L ^.valeur;
Fin
```

## 13.2. Listes chaînées

### 13.2.6. Algorithmes de manipulation de listes chaînées

- Ajout en tête de liste :

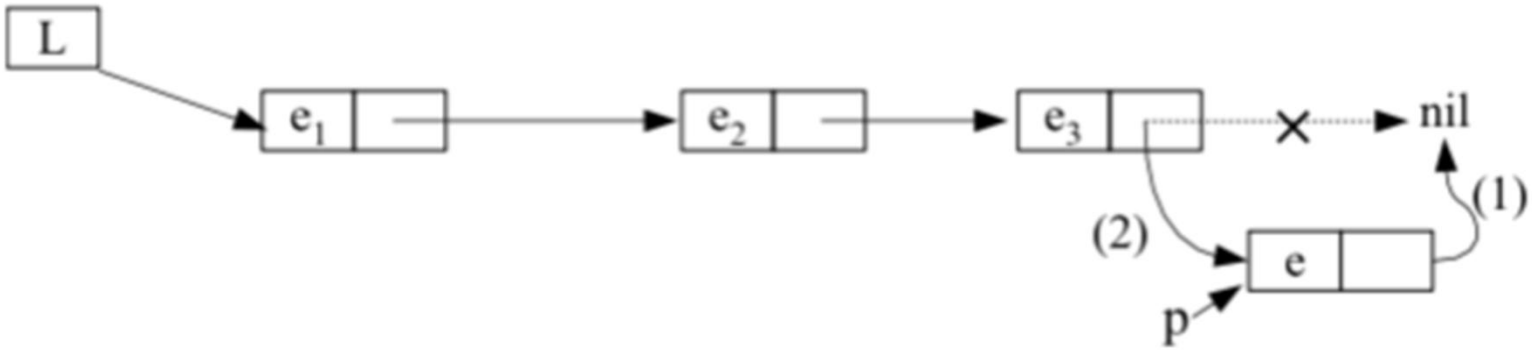



```
Procedure ajoutTete(e : integer; var L: Liste); //La liste est passée par adresse
var
   p : Liste;
Debut
       Allouer(p);
      p^.valeur := e;
      p^.suiv := L;
      L := p;
Fin;
```

## 13.2. Listes chaînées

## 13.2.6. Algorithmes de manipulation de listes chaînées

- **Ajout en queue de liste :**
  - La procédure ***ajoutQueue*** insère l'élément ***e*** en queue de liste, après la queue actuelle.
  - La variable *q* est modifiée itérativement par *q := q^.suiv* de façon à parcourir tous les éléments jusqu'à ce que l'on arrive sur le ***dernier élément*** de la liste (*q^.suiv = nil*).



## 13.2. Listes chaînées

### 13.2.6. Algorithmes de manipulation de listes chaînées

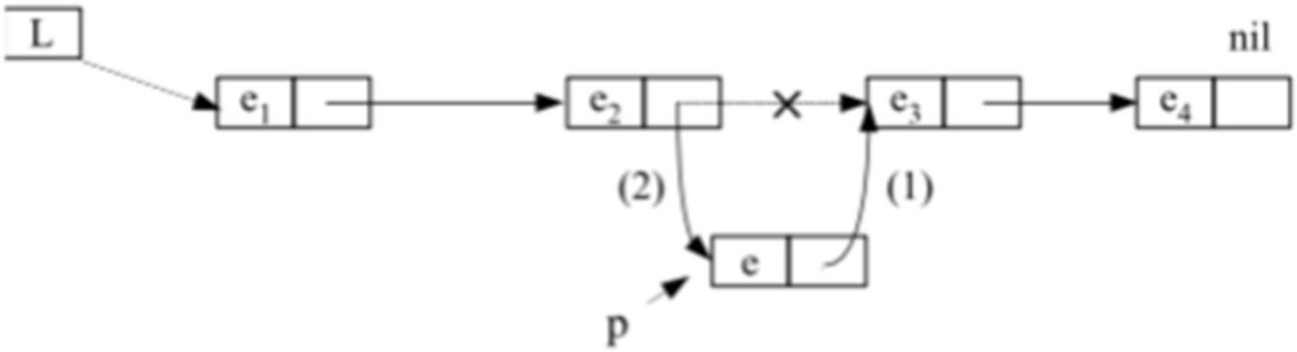
- **Ajout en queue de liste :**

```
Procedure ajoutQueue (e : integer; var L: Liste);
var
   p, q : Liste;
Debut
   Allouer(p);
   p^.valeur := e;
   p^.suiv := nil;
   SI (estVide(L)) ALORS
                        L := p
                   SINON
                        Debut
                            q := L;
                            TANTQUE (q^.suiv<>nil) FAIRE
                                        q := q^.suiv;
                            q^.suiv := p;
                        Fin;
Fin;
```

## 13.2. Listes chaînées

## 13.2.6. Algorithmes de manipulation de listes chaînées

- **Ajout au mileu de liste (entre deux maillons):**
  - Nous ajoutons l'élément à sa bonne place dans une liste déjà triée par ordre croissant.



# Chap13: Les structures de données élémentaires

## 13.2. Listes chaînées

### 13.2.6. Algorithmes de manipulation de listes chaînées

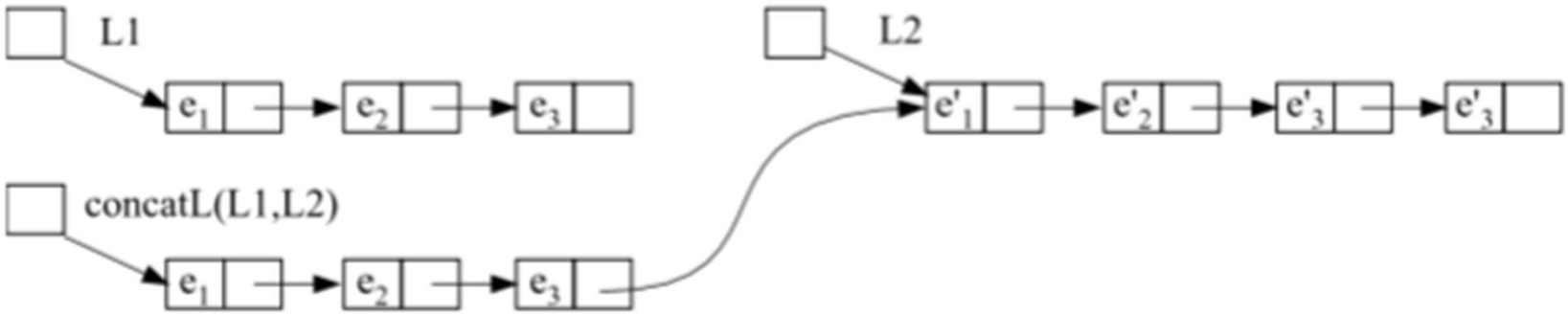
- **Ajout au mileu de liste (entre deux maillons) :**

```
Procedure ajoutMilieu(e : integer; var L: Liste);
var p, courant : Liste;
Debut
    Allouer(p);
    p^.valeur := e;
    p^.suivant := nil;
    SI (estVide(L)) ALORS
                        L:=p
                SINON
                    Debut
                        q:=L;
                        TANTQUE ((q<>nil) ET (q^.valeur < e)) FAIRE
                                    Debut
                                        courant := q;
                                        q := q^.suiv;
                                    Fin;
                        p^suiv := q;
                        courant^.suiv := p;
                    Fin;
Fin;
```

## 13.2. Listes chaînées

## 13.2.6. Algorithmes de manipulation de listes chaînées

- **Concaténation de deux listes:**
  - Donnons une fonction ***concatL*** qui met deux listes bout à bout pour en construire une troisième dont la longueur est égale à la somme des longueurs des deux autres.
  - Les deux listes ne sont pas modifiées



# Chap13: Les structures de données élémentaires

## 13.2. Listes chaînées

### 13.2.6. Algorithmes de manipulation de listes chaînées

- **Concaténation de deux listes:** (version itérative)

```
Fonction concatL(L1: Liste; L2 : Liste) : Liste;
Temp: Liste;
Debut
    SI (estVide(L1)) ALORS
                        concatL := L2
                    SINON
                        debut
                            temp:=L1;
                            TantQue (temp^.suiv <> NIL) FAIRE
                                        temp = temp^.suiv;
                             temp^.suiv=L2;
                             concatL := L1;
                        fin;
Fin
```

# Chap13: Les structures de données élémentaires

## 13.2. Listes chaînées

### 13.2.6. Algorithmes de manipulation de listes chaînées

- **Insertion d'un élément à la k-ieme place:** (version itérative)

```
Fonction ajoutek(e :integer ; k :integer; L: Liste): Liste;
P, temp: Liste
Com: integer;
Debut
    SI (k=0) ALORS
                ajoutek := cons(e, L) // fonction ajout en tête
              SINON
                debut
                 Allouer(p);
                 p^.valeur := e;
                 com:=0;
                 temp := L;
                 Tant Que ((temp^.suiv <> NIL) and (com < k)) FAIRE
                              debut
                             temp = temp^.suiv;
                             com++;
                             fin;
                   p^.suiv := temp^.suiv;
                   temp^.suiv := p;
                          fin;
Fin
```

## 13.2. Listes chaînées

## 13.2.6. Algorithmes de manipulation de listes chaînées

- **Recherche d'un élément:**
  - La fonction ***existeIter*** effectue un parcours itératif de la liste pour rechercher l'élément ***e*** dans la liste ***L***.
  - La variable ***q*** est modifiée itérativement par ***q := q^.suiv*** de façon à parcourir tous les éléments jusqu'à ce que l'on arrive à la fin de la liste (***q^.suiv = nil***)

```
Fonction existeIter(e : integer; L: Liste) : booleen;
var trouve : booleen;
Debut
    trouve := faux;
    TANTQUE ((L<>nil) ET (NON trouve)) FAIRE
            Debut
                trouve := (L^.valeur=e);
                L:=L^.suiv
             Fin;
     existeIter := trouve;
Fin;
```

## 13.2. Listes chaînées

## 13.2.6. Algorithmes de manipulation de listes chaînées

- **Affichage des éléments d'une liste:**
  - version itérative.

```
Procedure afficheListe(L : Liste);
var
   p : Liste;
Debut
      P := L;
      TANTQUE (p<>nil) FAIRE
            Debut
                  Ecrire( p^.valeur);
                  p := p^.suiv;
             Fin;
Fin;
```

## 13.2. Listes chaînées

## 13.2.6. Algorithmes de manipulation de listes chaînées

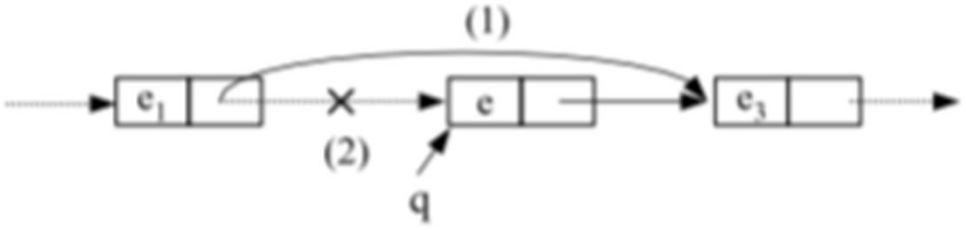
- **Nombre d'éléments d'une liste:**
  - version itérative.

```
Fonction nbElementsRIter(L: Liste) : integer;
var
   nb : integer;
Debut
       nb:=0;
       TANTQUE (L<>nil) FAIRE
                Debut
                       nb := nb + 1;
                       L := L^.suiv;
                Fin;
         nbElementsRIter := nb;
Fin;
```

## 13.2. Listes chaînées

## 13.2.6. Algorithmes de manipulation de listes chaînées

- **Suppression d'un élément d'une liste:**
  - Pour supprimer la cellule contenant ***e***, il faut modifier la valeur du champ ***suiv*** contenu dans le prédecesseur de ***e*** :
    - le sucesseur du prédecesseur de ***e*** devient le successeur de ***e*** ***(Predecesseur(e) ← Sucesseur(e))***.
  - Le traitement est différent si l'élément à supprimer est le premier élément de la liste.



# Chap13: Les structures de données élémentaires

## 13.2. Listes chaînées
## 13.2.6. Algorithmes de manipulation de listes chaînées

- **Suppression d'un élément d'une liste:** version itérative

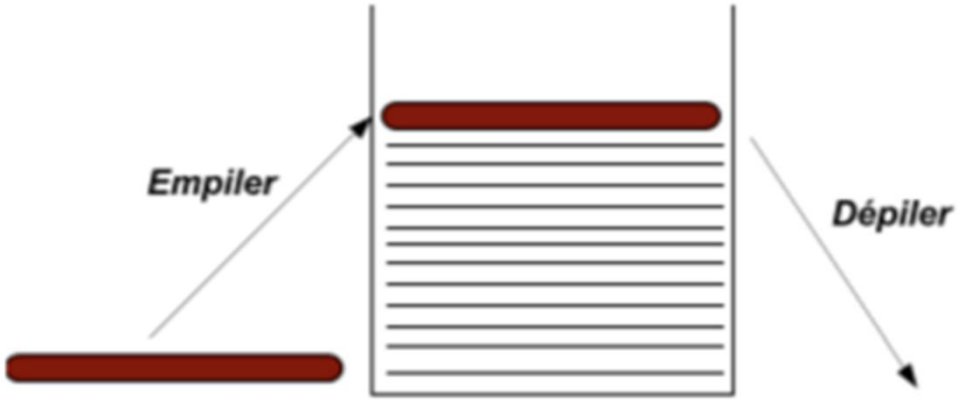
```
Procedure suppressionIter(e : integer; var L: Liste);
var succ, cour, q : Liste;
Debut
    SI (L<>nil) ALORS
                SI (L^.valeur = e) ALORS
                                Debut
                                    q:=L;
                                     L:=L^.suiv;
                                     Liberer(q);
                                Fin
                               SINON
                                  Debut
                                     cour:=L;
                                     succ:=L^.suiv;
                                     TANTQUE (succ<>nil) FAIRE
                                         SI (succ^.valeur<>e) ALORS
                                                          Debut
                                                            cour:=succ;
                                                            succ:=succ^.suiv;
                                                          Fin;
                                                          SINON
                                                             Debut
                                                                cour^.suiv:=succ^.suiv;
                                                                 Liberer(succ);
                                                                 succ:=nil;
                                                             Fin;
                                                  Fin;
fin
```

## 13.3. Les piles
### 13.3.1. Définition

- *Une pile est une structure de données pour laquelle l'ajout et le retrait d'un élément se fait toujours au sommet (en tête).*
- *Elle met en œuvre le principe « dernier entré, premier sorti » (**LIFO : Last-In, First-Out en anglais**).*

## 13.3. Les piles
## 13.3.2. Représentation graphique

- On peut imaginer une pile comme une boîte dans laquelle on place des objets et de laquelle on les retire dans un ordre inverse dans lequel on les a mis :
- les objets sont les uns au dessus des autres et on ne peut accéder qu'à l'objet situé au sommet de la pile.



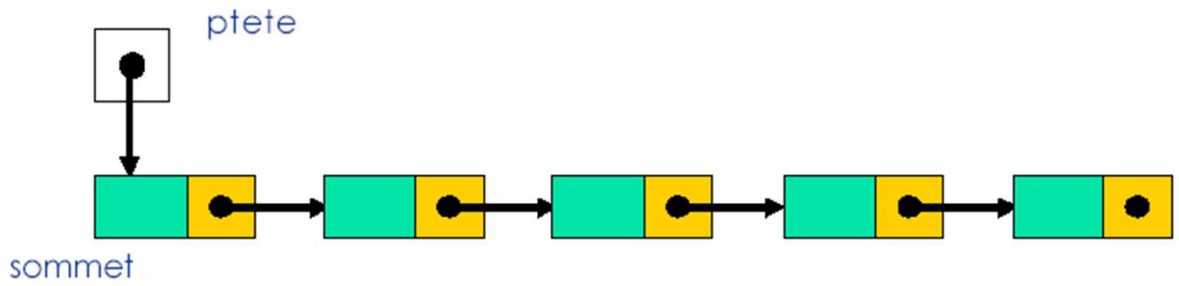
## 13.3. Les piles
## 13.3.3. Opérations sur les piles

- Une pile supporte les opérations suivantes :
  - ***vide(P)*** : est égal à vrai si la pile est vide et faux sinon
  - ***empiler(e, P )*** ou ***push***: son résultat est la pile obtenue à partir de la pile ***P*** en insérant l'élément e au sommet la pile.
  - ***valeur(P)***: cette fonction permet de récupérer l'élément figurant au sommet d'une pile non vide
  - ***depiler(e, P)*** ou ***pop***: son résultat est la pile obtenue à partir de ***P*** en supprimant l'élément qui se situe en son sommet

## 13.3. Les piles

## 13.3.4. Représentation d'une pile sous forme de liste chaînée

- Les piles sont tout à fait isomorphes aux listes, si ce n'est qu'on a traditionnellement tendance à les imaginer verticales, et qu'on donne aux opérations d'*insertion*, de *lecture de la tête* et de *suppression d'un élément* les noms *EMPILER*, *sommet*, *dépiler*.
  - ***Leur codage par des listes chaînées s'effectue de manière immédiate.***



## 13.3. Les piles

### 13.3.5. Représentation d'une pile sous forme de tableau

- Il est facile d'implanter une pile au moyen d 'un tableau qui contient les éléments et un indice qui indiquera la position du sommet de la pile.
- La seule difficulté dans ce cas est la gestion des débordements qui interviennent quand on tente d'éffectuer l'opération depiler sur une pile vide ou l'opération empiler sur un tableau codant la pile qui est déjà pleine.

# 13.3. Les piles

## 13.3.5. Représentation d'une pile sous forme de tableau

```
const tailleMax = 200;
type TPile = Enregistrement
                    taille :integer;
                contenu :Tableau[1..tailleMax] of Element;
        fin ;
var P : TPile;
```

```
Procedure ViderPile (var P : TPile);
debut
    P.taille :=0;
fin ;

Function estVideP (var P : TPile) : boolean;
debut
     estVideP := (P.taille= 0);
fin ;
```

## 13.3. Les piles
### 13.3.5. Représentation d'une pile sous forme de tableau

```
Function valeurPile (var P : TPile) : Element;
debut
    valeurPile := P.contenu[P.taille];
fin ;

Procedure empiler (e : Element; var P : TPile);
debut
   P.taille := P.taille +1;
   P.contenu[P.Taille] := e;
fin ;

Procedure depiler (var P :TPile);
debut
   P.taille := P.taille -1;
fin ;
```

**Exercice d'application**: *Donner les primitives de gestion d'une pile à l'aide d'une liste chaînée.*

## 13.4. Les files

### 13.4.1. Définition

- *Une file (ou queue) est une structure de données mettant en œuvre le principe « **premier entré, premier sorti** » (**FIFO : First-In, First-Out** en anglais).*
- *Dans une file les éléments sont systématiquement ajoutés en queue et supprimés en tête.*
- *La valeur d'une file est par convention l'élément de tête.*

## 13.4. Les files
## 13.3.2. Opérations sur les files

- Une file supporte les opérations suivantes :
  - ***vide(F):*** est égal à vrai si la file est vide et faux sinon
  - ***enfiler(e, F):*** son résultat est la file obtenue à partir de la file ***F*** en ajoutant l'élément ***e*** en queue de file.
  - ***valeur(F):*** cette fonction permet de récupérer l'élément figurant en tête de file.
  - ***defiler(e, F):*** son résultat est la file obtenue à partir de ***F*** en supprimant l'élément en tête de file.

## 13.4. Les files
### 13.4.3. Représentation d'une file sous forme de liste chaînée

- De la même façon que pour la pile, une file peut être implémentée en utilisant un tableau ou une liste chaînée.
- Seulement dans ce cas, il est nécesaire de connaître le début et la fin de la file qui seront respectivement repérés par les variables ***debut*** et ***fin***.
- On note que :
  - la file d'attente est vide si et seulement si *debut* = *fin*.
  - lorsqu'un élément arrive en queue de file on incrémente la variable *fin*
  - lorsque l'élément tête de file est traitée, la variable *debut* est incrémentée si la file n'est pas vide
- *Dans la suite on illustre l'implémentation d'une file grâce à une liste chaînée gardée*
  - *liste dans laquelle on connait à la fois l'adresse du premier et du dernier élément.*

## 13.4. Les files

### 13.4.3. Représentation d'une file sous forme de liste chaînée

- *Dans une file, le premier élément ajouté doit être en tête et le dernier élément ajouté doit être en queue.*
- La tête et la queue d'une file correspondent à la tête et la queue de la liste chaînée qui l'implémente.
- Une file doit être accessible à la fois par la tête pour retirer un élément (***défiler***) et par la queue pour ajouter un nouvel élément (***enfiler***).
- Pour cela, on va utiliser deux pointeurs sur la liste : un pointeur sur la tête et un pointeur sur la queue.

## 13.4. Les files

### 13.4.3. Représentation d'une file sous forme de liste chaînée

```
type
Liste =^maillon ;
maillon =record
            valeur : Element;
            suiv    :Liste;
end;
TFile =Record
          tete : Liste;
          queue : Liste;
end ;
```

```
Procedure ViderFile (var F : TFile);
Debut
   F.queue := F.tete :=Nil;
fin ;

Fonction Successeur (var L : Liste) :Liste;
Debut
    Successeur := L^.suiv ;
fin ;
```

```
Fonction estVideFile (var F : TFile) :boolean;
Debut
    estVideFile := (F.tete = F.queue=nil);
fin ;
```

## 13.4. Les files

### 13.4.3. Représentation d'une file sous forme de liste chaînée

```
Fonction valeurF (var F : TFile) :Element;
Debut
    valeurF := F.tete^.valeur ;
fin ;
```

```
Procedure defiler (var F : TFile);
var q : Liste;
Debut
    si (F.tete = F.queue) alors
            Liberer(F.tete);
             F.tete:=F.queue:=nil
        SI (NON estVideFile(F)) ALORS
              Debut
                q :=F.tete;
                F.tete :=Sucesseur(F.tete);
                Liberer(q) ;
              Fin
fin ;
```

```
Procedure enfiler (e : Element; var F : TFile);
var q : Liste;
Debut
    Allouer(q) ;
    q^.valeur := e;
    q^.suiv := nil;
     si (estVideFile(F) = true) alors
          debut
             F.queue := q;
             F.tete     :=q;
          fin
       Sinon
         debut
            F.queue^.suiv := q;
            F.queue := q;
          fin;
fin ;
```

# FIN CHAP13
# Structures de données élémentaires

## QUESTIONS ??

Dr Ousmane DIALLO