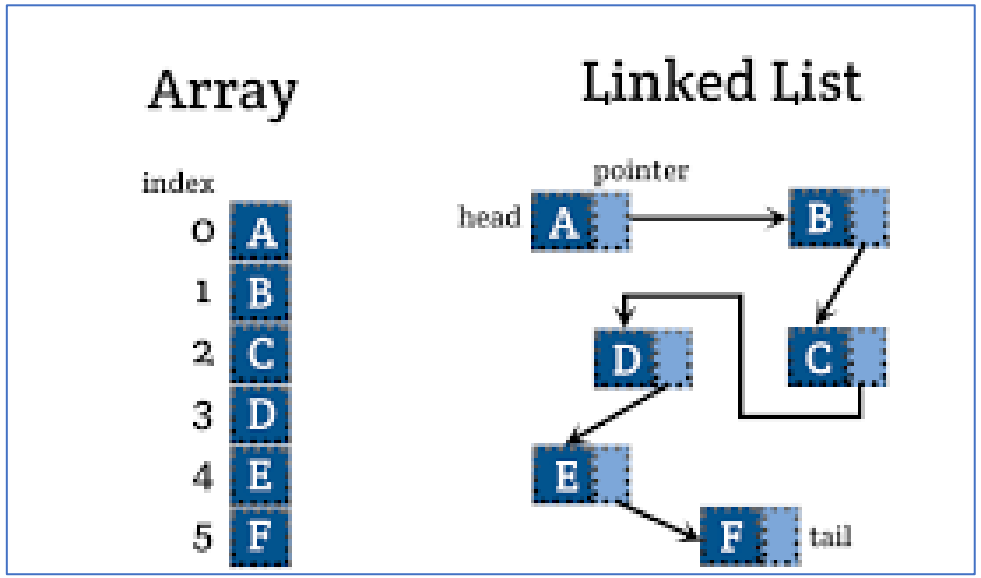
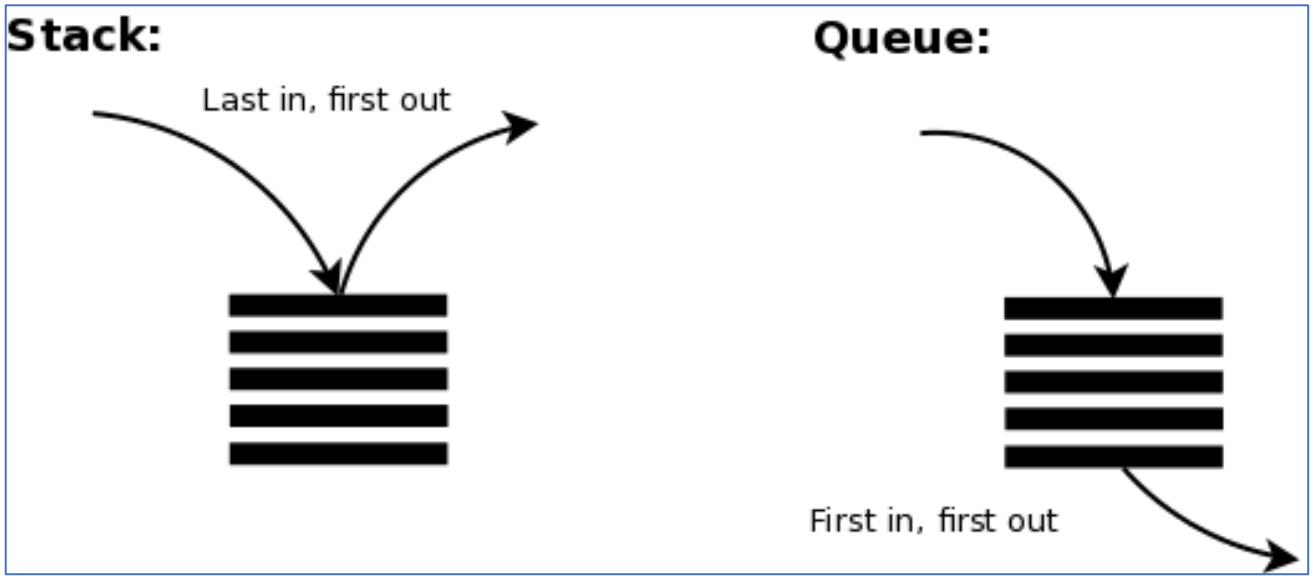
Chapitre 2

Structures Séquentielles

1. STRUCTURE DE DONNÉES : LISTE LINEAIRE
   i. Implémentation contiguë
   ii. Implémentation chainée
2. STRUCTURE DE DONNÉES : PILE
3. STRUCTURE DE DONNÉES : FILE





1.Liste linéaire

Une liste linéaire est la forme la plus courante d'organisation des données. On l'utilise pour stocker des données qui doivent être traitées de manière **séquentielle**. (Structure évolutive, càd pouvoir **ajouter** et **supprimer** des éléments).

Définition : Une *liste* est une suite finie (éventuellement vide) d'éléments **de même type** repérés selon leur **rang(position)** dans la liste.

Primitives : On définit le type abstrait de données par la définition des primitives qui permettent de le manipuler :

- liste **créer_liste()** : création d'une liste vide
- position **début(liste L)** : retourne la position du premier élément de la liste, INCONNUE si la liste est vide
- position **fin(liste L)** : retourne la position du dernier élément de la liste, INCONNUE si la liste est vide
- position **suivante(position p, liste L)** : retourne la position de l'élément qui suit celui en position p, INCONNUE si on sort de la liste
- position **précédente(position p, liste L)** : retourne la position de l'élément qui précède celui en position p, INCONNUE si on sort de la liste

Primitives :

- élément **accès**(position p,liste L) : retourne l'élément en position p
- entier **longueur**(liste L) : retourne le nombre d'éléments contenus dans la liste
- **insérer**(element e, position p, liste L) : rajoute l'élément dans la liste (qui est donc modifiée) à la position p.
- booléen **liste_est_vide**(liste L) : teste si la liste est vide, retourne VRAI ou FAUX
- elément **ieme**(position p, liste L) : retourne l'élément de position p
- **supprimer**(position p, liste L) : supprime l'élément de position p dans la liste (qui est donc modifiée)

Exemples d'utilisation :

```
Algo Afficher_Liste(liste L)
  locales : position i
  debut
    i <- debut(L)
    tant que i<>INCONNUE faire
          afficher(acces(i,L))
          i <- suivante(i,L)
    fin tant que
  fin
```

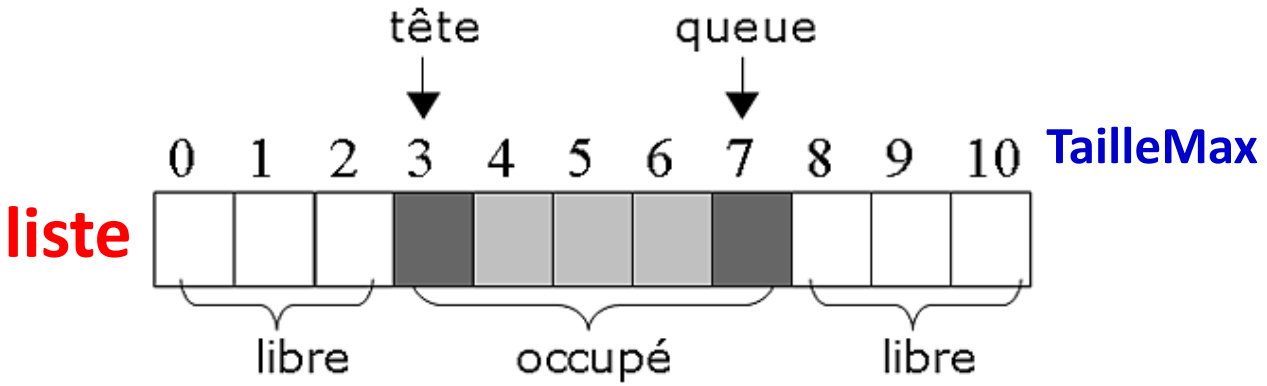
```
Algo Nb_Occurence(liste L, element e):retourne entier
locales : position i , entier nb
  debut
    i <- debut(L) ;    nb <- 0
    tant que i<>INCONNUE faire
          si (e=acces(i,L)) alors nb <- nb+1
                                  i <- suivante(i,L)
    fin tant que
    retourner nb
  fin
```

1.2 Implémentation contiguë :

- Les éléments sont rangés **les uns à côté des autres**, dans un **tableau**. La *i*-ème case du tableau contient le *i*ème élément de la liste.

type position = entier

type liste = tableau[0..N] d'éléments



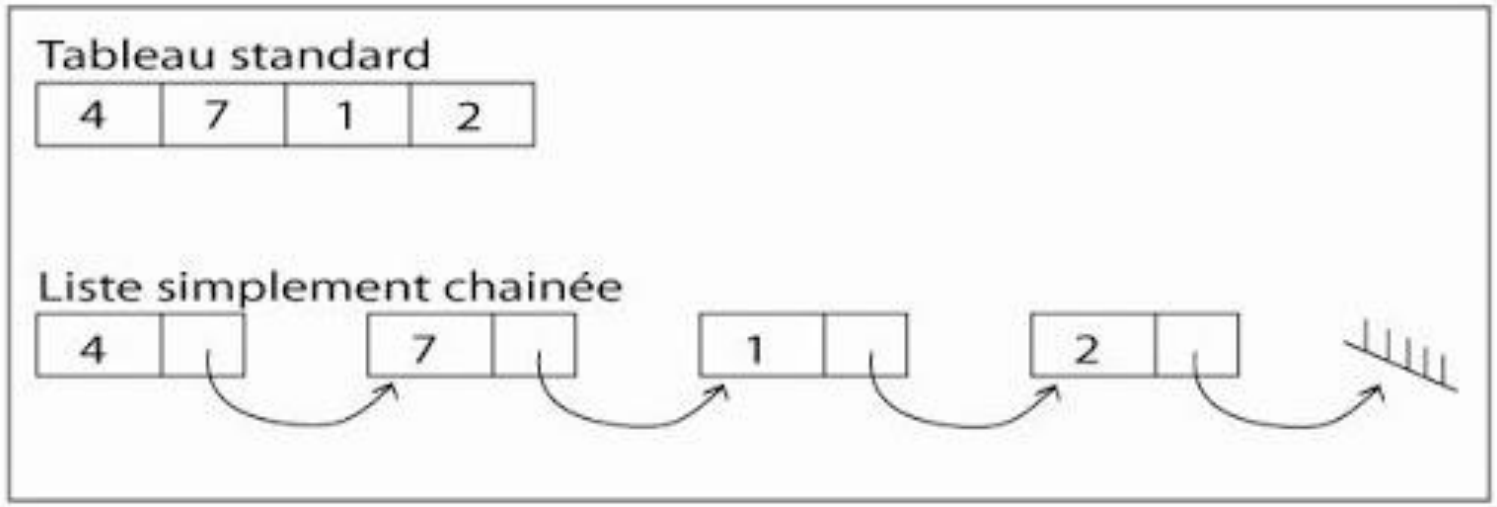
Exemples: liste d'entiers, chaine de caractères, etc…

1.2 Implémentation contiguë :

```
Algo inserer(element e, position p, liste L)
    locales entier i
    debut
        // décaler pour faire un trou
        pour i<-N a p (en décrémentant i) faire
            L[i+1] <- L[i]
        finpour
        // mettre l'element
        L[p] <- creer_element(e)
        // changer la taille car il y a un elt de plus
        ... PROBLEME !!! ...
```

1.2 Implémentation chaînée :

- Les éléments ne sont pas rangés les uns à coté des autres. On a besoin de connaître, pour chaque élément, la position (l'**adresse**) de l'élément qui le suit.
- Une adresse s'appelle aussi un pointeur (L'adresse nulle sera notée VIDE.)



1.2 Implémentation chaînée :

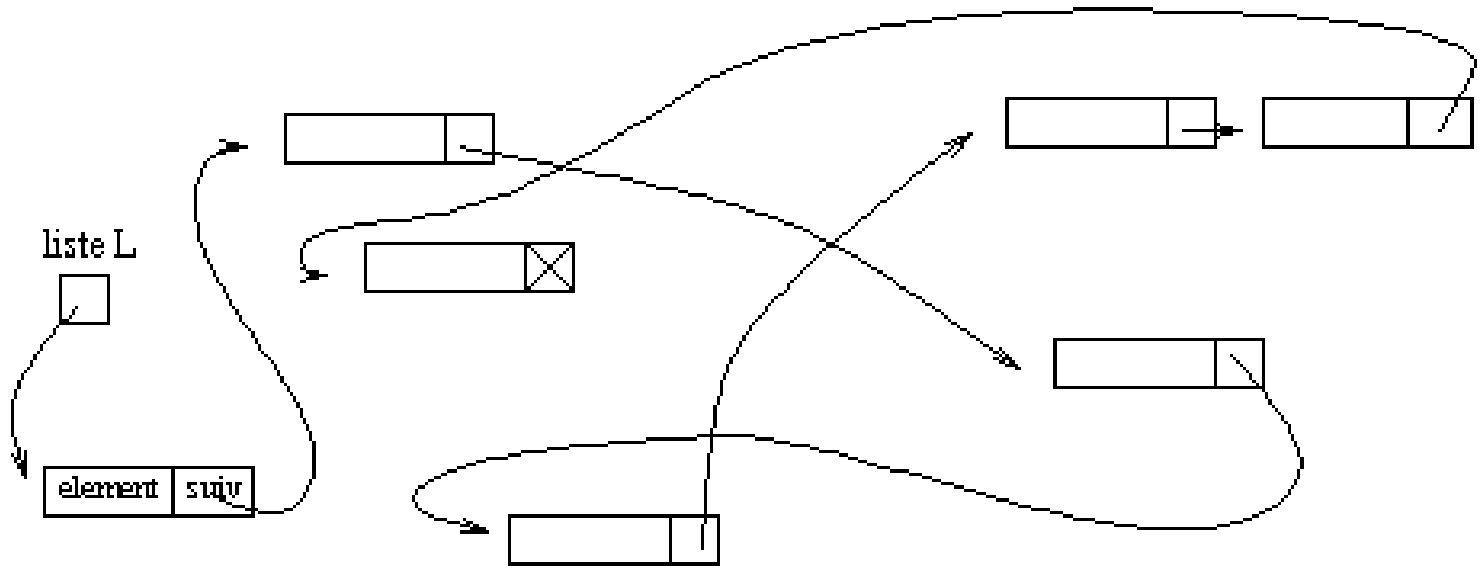
type cellule = {

valeur : element

suivant : adresse d'une cellule

}

type liste = adresse d'une cellule



1.2 Implémentation chaînée :

```
Algo longueur(liste L) : retourne un entier
    locales :  position p
        entier l
    debut
        l <- 0
        p <- L
        tant que p<>VIDE faire
                p <- p.suivant
                l <- l+1
        fintantque
        retourner l
    fin
```

1.2 Implémentation chaînée :

```
Algo inserer(element e, E/S liste L)
        locales :
            position p
            nouv : adresse d'une cellule
        debut
//allouer de la memoire pour une cellule et y mettre e dans le chp valeur
            nouv <- creer_cellule()
            nouv.valeur <- e
// insertion en tete
            nouv.suiv <- L
            L <- nouv


        fin
```

Exemples d'utilisation des listes chainées :

Déclaration :

```
#include <stdlib.h>
struct cellule
        {
         int val;
         struct cellule *nxt;
        };
typedef struct cellule cellule;
typedef cellule* Liste;
```

```
int main()
        {
   /* On Déclare 3 listes chaînées de façons différentes mais équivalentes */

   struct cellule *ma_liste1 = NULL;
    cellule *ma_liste2 = NULL;
    Liste ma_liste3 = NULL;

        return 0;
        }
```

Exemples d'utilisation des listes chainées :

```
void afficherListe(Liste LL)
{
    cellule *tmp = LL;
        while(tmp != NULL)
    {
        /* On affiche */
        printf("%d ", tmp->val);
        /* On avance d'une case */
        tmp = tmp->nxt;
    }
}
```

```
int estVide(Liste LL)
{
    if  (LL == NULL)
            return 0;
else
            return 1;
}
```

Exemples d'utilisation des listes chainées :

Ajout en tête

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

Exemples d'utilisation des listes chainées :

Ajout en fin de liste

```
Liste ajouterEnFin(Liste LL, int valeur)

{    cellule* nouvcell = malloc(sizeof(cellule));

     nouvcell->val = valeur;

     nouvcell->nxt = NULL;

    if(LL == NULL) return nouvcell;

    else
```

```
 { /*on parcourt la liste à l'aide d'un pointeur temporaire */

        cellule* temp=LL;

        while(temp->nxt != NULL)

            temp = temp->nxt;

     temp->nxt = nouvcell;

       return LL;

  }  }
```

Exemples d'utilisation des listes chainées :

Rechercher un élément dans une liste

```
Liste rechercherCellule(Liste LL, int valeur)
{       cellule *tmp=LL;
          while(tmp != NULL)
    {
        if(tmp->val == valeur)   return tmp;
         tmp = tmp->nxt;
    }
    return NULL;
}
```

Exemples d'utilisation des listes chainées :

Rechercher du i-ème élément

```
Liste cellule_i(Liste LL, int indice)

{

   for( int i=0; i<indice && LL != NULL; i++)

        LL = LL->nxt;

   if(LL == NULL) return NULL;

    else

       return LL;

}
```

Exemples d'utilisation des listes chainées :

Compter le nombre d'éléments d'une liste chaînée

Itérative

```
int nombreCellules(Liste LL)
{    int nb = 0;
   cellule* tmp = LL;
   while(tmp != NULL)
   {
      nb++;
      tmp = tmp->nxt;
   }
      return nb;  }
```

Récursive

```
int nombreCellules(Liste LL)
{
      if(LL == NULL)     return 0;
     return 1 + nombreCellules(LL->nxt);
}
```

Exemples d'utilisation des listes chainées :

Compter le nombre d'occurrences d'une valeur

Itérative

```
int nombreOccurences(Liste LL, int valeur)
{  int i=0;
      if(LL == NULL)     return 0;
   while((LL = rechercherCellule(LL, valeur)) != NULL)
   {
      LL = LL->nxt;
       i++;
   }
     return i;
}
```

Exemples d'utilisation des listes chainées :

Compter le nombre d'occurrences d'une valeur

Récursive

```
int nombreOccurences(Liste LL, int valeur)

{

      if(LL == NULL)     return 0;

       if(LL->val  ! = valeur)     return nombreOccurences(LL->nxt, valeur)

           else return 1 + nombreOccurences(LL->nxt, valeur)

}
```

Exemples d'utilisation des listes chainées :

Supprimer un élément en tête

```
Liste supprimerCelluleEnTete(Liste LL)
{
   if(LL != NULL)
   {   cellule* aRenvoyer = LL->nxt;
       free(LL);
       return aRenvoyer;
   }
   else
           return NULL;
   }
```

Exemples d'utilisation des listes chainées :

Supprimer un élément en fin de liste

```
Liste supprimerCelluleEnFin(Liste LL)
{
    if(LL == NULL)  return NULL;
    if(LL->nxt == NULL)
      {   free(LL);
          return NULL;
       }

  cellule* tmp = LL;
  cellule* ptmp = LL;
```

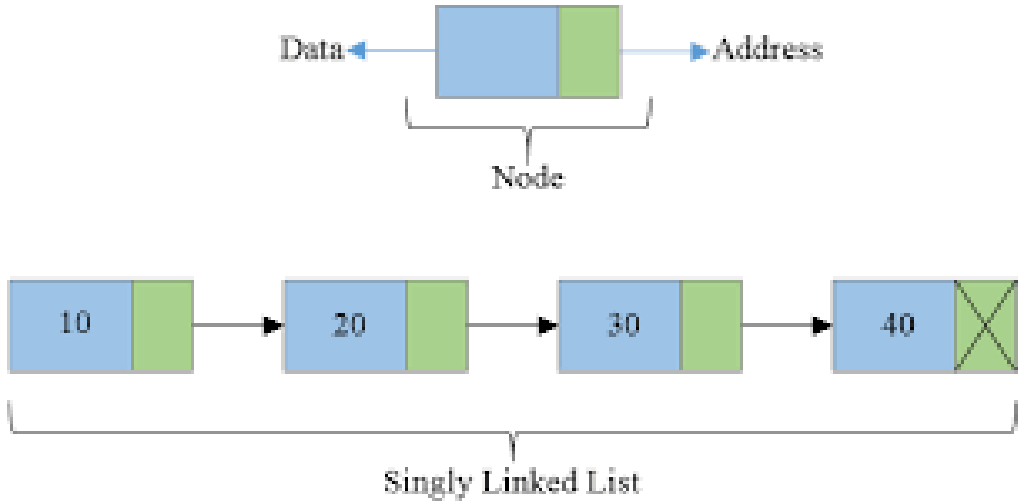
```
while(tmp->nxt != NULL)
   {  ptmp = tmp;
      tmp = tmp->nxt;
   }
      ptmp->nxt = NULL;
      free(tmp);
      return LL;
}
```

1.2 Types de listes chaînées:

Il existe différents types de listes chaînées telles que :

1. Liste simplement chaînée :

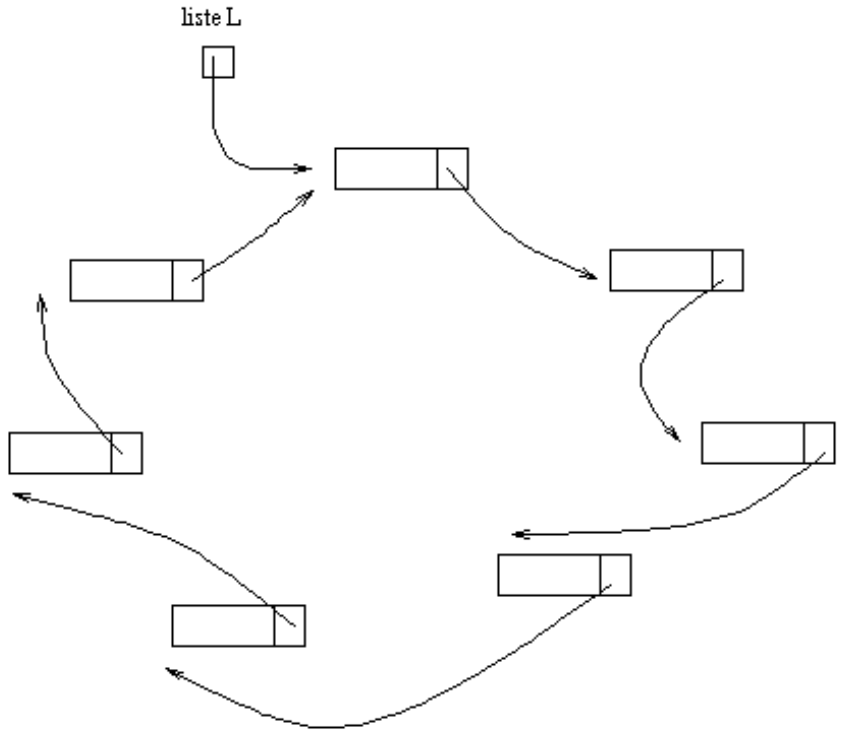
cette liste n'est constituée que d'un pointeur vers le nœud ou l'élément suivant.



1.2 Types de listes chaînées:

2. Liste chaînée circulaire :

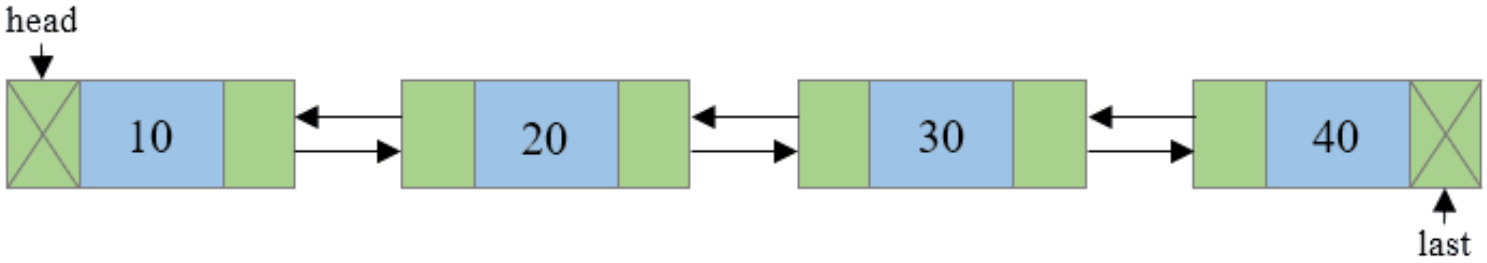
est une liste simplement chainée avec le dernier nœud qui pointe sur le premier (tête de liste)



1.2 Types de listes chaînées:

3. Liste doublement chaînée :

chaque nœud a deux pointeurs, un pointeur **précédent** qui pointe vers le nœud précédent et un pointeur **suivant** pointe vers le nœud suivant(dans le cas du premier nœud, le pointeur précédent est évidemment NULL et le pointeur suivant du dernier nœud pointe vers NULL).



type **cellule** = {
precedent : adresse d'une cellule
valeur : element
suivant : adresse d'une cellule
}
type **liste** = adresse d'une cellule

Comparaison des deux implémentations

Représentation contiguë :

accès facile au k-ème élément, mais insertion et suppression longues à cause des décalages.

Problèmes de mémoire à allouer, re-allouer, etc, lorsque la taille varie.

Représentation chaînée :

pas de longueur fixée (mais plus de mémoire prise (place pour les pointeurs)).

Insertion et suppression rapides, mais pas d'accès direct au k-ème élément.

Conclusion :

ça dépend de ce qu'on a à faire, des primitives que l'on utilise le plus souvent.

Comparaison des deux implémentations

Opération	contiguë	chaînée
▪ création	O(1)	O(1)
▪ Ajout/Suppression en queue	O(1)	O(1)
▪ Ajout/Suppression en tête	O(N)	O(1)
▪ Ajout/Suppression générale	O(N)	O(1)
▪ accès élément de rang k	O(1)	O(N)
▪ concaténation	O(N)	O(1)

2. Structure de données : Pile (en anglais *stack*)

2.1 Définition : Une pile est une liste linéaire particulière : on ne peut accéder (= consulter, ajouter, supprimer) qu'au dernier élément, que l'on appelle le **sommet** de la pile. Dans une pile, il est impossible d'accéder à un élément au milieu ! Les piles sont aussi appelées structures **LIFO** (**L**ast **I**n **F**irst **O**ut, c-a-d Dernier-Entré-Premier-Sorti).

C'est une structure très utilisée en informatique. Par exemple, Undo(éditeurs), les appels de fonctions d'un programme utilisent une pile pour sauvegarder toutes les informations (variables, adresse de l'instruction de retour, etc.).

2.2 Primitives (PILE) : push & pop

- pile créer_pile() : création d'une pile vide
- pile créer_pile(entier taille_max) : création d'une pile vide d'au maximum taille_max éléments
- empiler(element, pile) : met l'élément en sommet de pile, erreur si la taille est limitée et la pile pleine (***push*** en anglais)
- dépiler(pile) : enlève de la pile l'élément en sommet de pile, erreur si la pile est vide
- élément dépiler(pile) : enlève l'élément en sommet de pile et le retourne, erreur si la pile est vide (***pop*** en anglais)
- élément sommet(pile) : retourne l'élément en sommet de pile, erreur si la pile est vide
- booléen pile_est_vide(pile) : teste si la pile est vide
- booléen pile_est_pleine(pile) : teste si la pile est pleine (seulement dans le cas d'une taille limitée)

2.3 Implémentations

2.3.1 Implémentation d'une pile par un tableau :

```
type pile = {
        tab : tableau d'elements
        s : entier // indice du sommet
                }
```

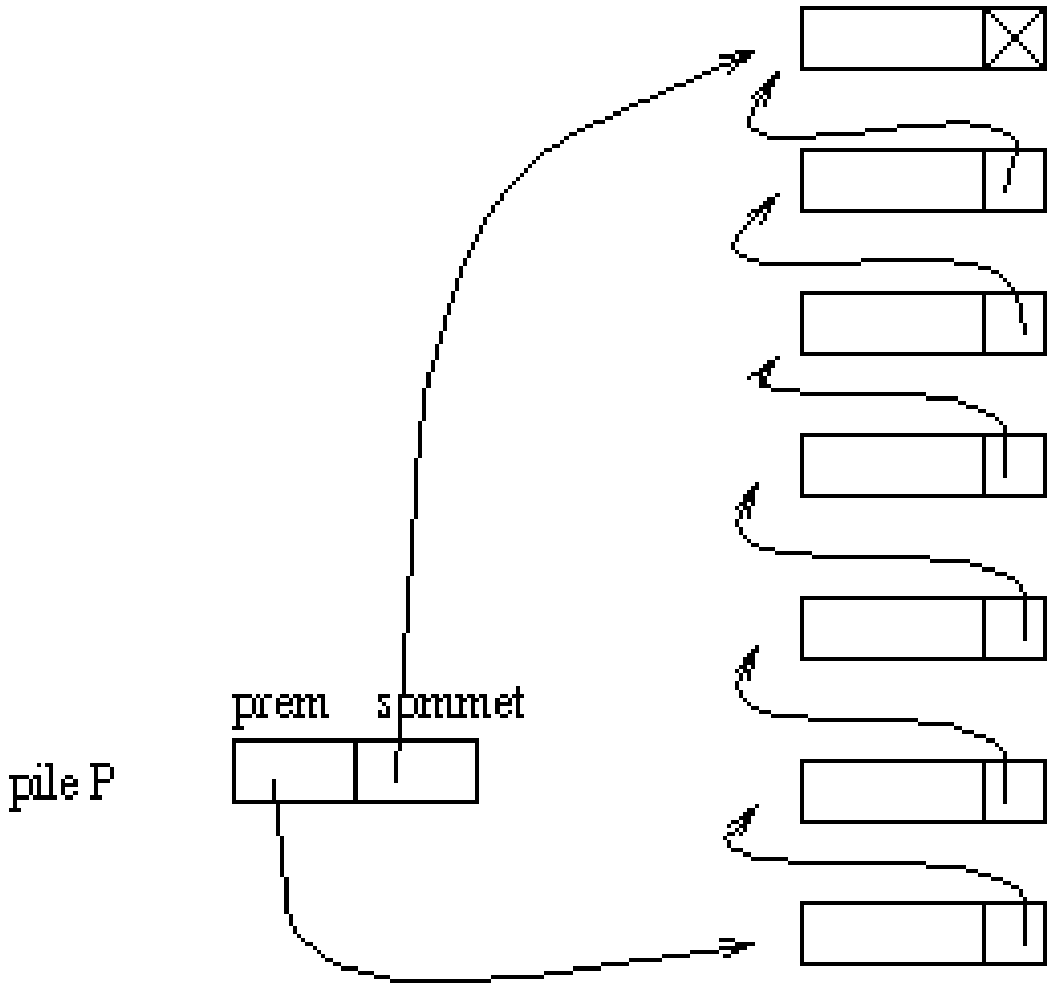
```
Algorithme empiler (element e, E/S pile p)
   debut
      // si la pile n'est pas pleine !
      p.s <- p.s+1
      p.tab[p.s] <- e
   fin
```

```
Algorithme sommet(pile P) ; retourne un element

   debut
      si P.s <> 0 alors
            retourner P.tab[P.s]
      sinon
            erreur "on ne peut pas acceder au sommet d'une pile vide !!"
      finsi
   fin
```

2.3 Implémentations

2.3.2 Implémentation d'une pile par une liste chaînée :

```
type cellule = {
        valeur : element
        suivant : adresse d'une cellule
    }


    type pile = {
        prem : adresse d'une cellule
        sommet : adresse d'une cellule
    }
```



2.3 Implémentations

2.3.2 Implémentation d'une pile par une liste chaînée :

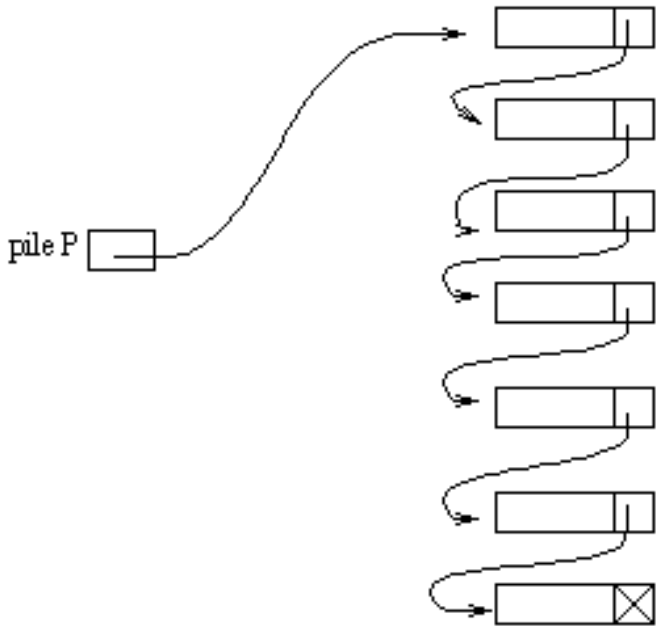
```
Algorithme empiler (element e,  E/S pile p)
  locales        nouv : adresse d'une cellule
  debut
    nouv=creer_cellule()
     nouv.valeur <- e
     nouv.suivant <- VIDE
     // faire les liens
     si p.sommet <> VIDE alors
          // la pile n'est pas vide
          p.sommet.suivant <- nouv
           p.sommet <- nouv
sinon
          // la pile est vide
           p.sommet <- nouv
           p.prem <- nouv
       finsi
    fin
```

2.3 Implémentations

2.3.3 Implémentation d'une pile par une liste chaînée(version2) :

```
type cellule = {
        valeur : element
        suivant : adresse d'une cellule
                }
```

```
type pile = adresse d'une cellule // le sommet
```




```
Algo empiler (element e,  E/S pile p)
Locales  nouv : adresse d'une cellule
   debut
        nouv=creer_cellule()
       nouv.valeur <- e
       nouv.suivant <-  p
       p <- nouv
   fin
```

Exemple d'utilisation d'une pile : expressions arithmétiques postfixées et infixées

Les expressions arithmétiques sont habituellement écrites de manière **infixe**, c'est à dire l'opérateur entre ses deux opérandes (ou avant si c'est un opérateur unaire mais c'est pas le problème). On trouve aussi une notation **postfixée**, que l'on appelle également *notation polonaise* car introduite par le polonais Lukasiewicz : l'opérateur est placé **après** ses opérandes.

Par exemple :
L'intérêt de cette notation est qu'il n'y a plus besoin de connaître les priorités des opérateurs, et donc plus besoin de parenthèses (qui servent à contrer les priorités).
3+2*5=3+(2*5) s'écrit en postfixe : 3 2 5 * +
(3+2)*5 s'écrit en postfixe : 3 2 + 5 *
4+7*(8+2) s'écrit en postfixe : 4 7 8 2 + * +
4 7 + 8 2 * + donne en infixe : (4+7)+(8*2)=4+7+8*2

Comment passer d'une infixe à une postfixe ?

- En empilant les opérateurs !
- Lire l'expression infixe caractère par caractère. Si c'est un opérande, on l'écrit. Sinon (c'est un opérateur), si la pile est vide ou bien si l'opérateur est **moins** prioritaire que l'opérateur en sommet de pile, alors on l'empile. Sinon, on dépile jusqu'à pouvoir l'empiler.

3. Structure de données : File (en anglais Queue)

3.1 Définition :

Une file est une liste linéaire particulière : on ne peut ajouter qu'en queue, consulter qu'en tête, et supprimer qu'en tête. Comme pour une **file d'attente** ... !

Les files sont aussi appelées structures **FIFO** (**F**irst **I**n **F**irst **O**ut), cad premier-entré-premier-sorti.

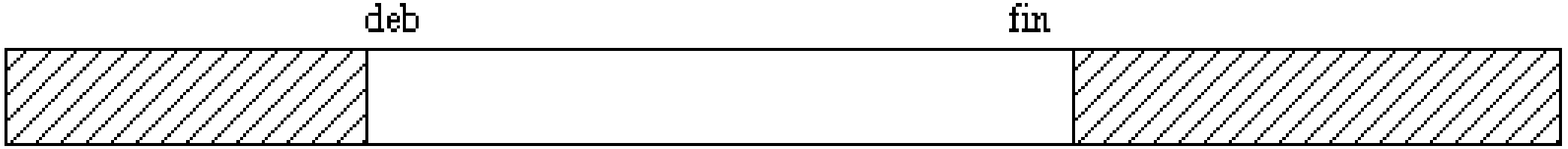
Utilisation : serveurs d'impression(spool), mémoires tampons (buffers), etc…

3.2 Primitives (FILE) : enqueue & dequeue

- file créer_file() : création d'une file vide
- file créer_file(entier taille_max) : creation d'une file vide d'au maximum taille_max éléments
- enfiler(element, file) : met l'élément à la fin de la file, erreur si la taille est limitées et la file pleine
- défiler(file) : enlève le premier élément de la file, erreur si la file est vide
- élément consulter(file) : retourne le premier élément de la file , erreur si la file est vide
- booléen file_est_vide(file) : teste si la file est vide
- booléen file_est_pleine(file) : teste si la file est pleine (seulement dans le cas d'une taille limitée)

3.3 Implémentations

3.3.1 Implémentation d'une file par un tableau :




```
type File = {
        T : tableau[1..MAX] d'elements
        deb : entier
        fin : entier
    }
```

3.3 Implémentations

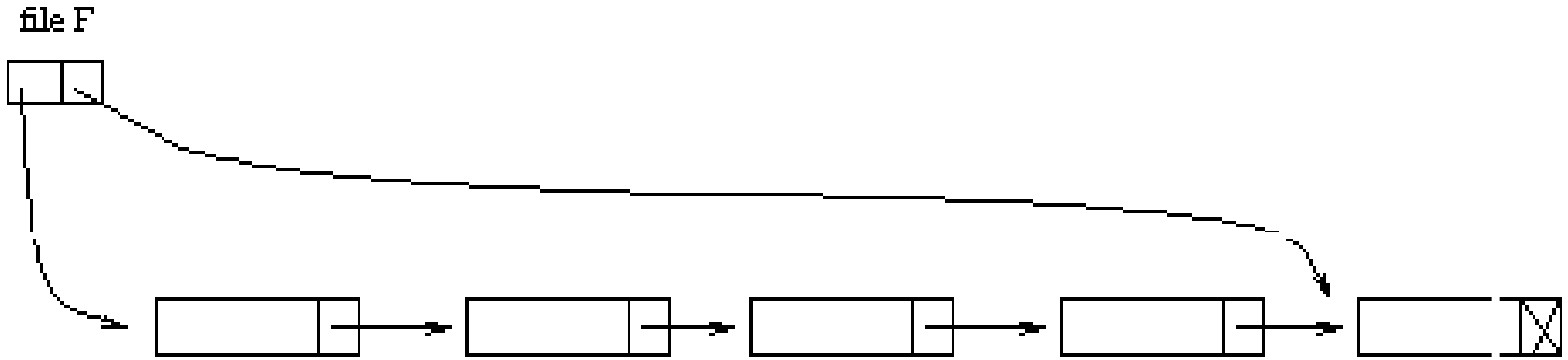
3.3.1 Implémentation d'une file par un tableau :

```
enfiler(element e, E/S file F)
  debut
    si F.fin < MAX alors
        F.fin <- F.fin +1
        F.t[F.fin] <- e
    sinon
        file pleine ???
    finsi
  fin
```

```
defiler(E/S file F)
  debut
    si F.debut <= F.fin MAX alors
        F.debut <- F.debut +1
    sinon
        erreur "file vide"
    finsi
  fin
```

2.3 Implémentations

3.2 Implémentation d'une file par une liste chaînée :



type cellule = {

valeur : element

suivant : adresse d'une cellule

}

type file {

debut : adresse d'une cellule //la première

fin : adresse d'une cellule // la dernière

}