II. Les piles et les files:

Les piles et les files sont des ensembles dynamiques pour lesquels l'élément à supprimer *via* l'opération SUPPRIMER est défini par la nature intrinsèque de la structure.
Dans une ***pile***, l'élément supprimé est le dernier inséré : la pile met en œuvre le principe ***dernier entré, premier sorti***, ou ***LIFO*** (Last-In, First-Out). De même, dans une ***file***, l'élément supprimé est toujours le plus ancien, la file met en œuvre le principe ***premier entré, premier sorti (servi)***, ou ***FIFO*** (First-In, First-Out). Il existe plusieurs manières efficaces d'implémenter des piles et des files dans un programme.

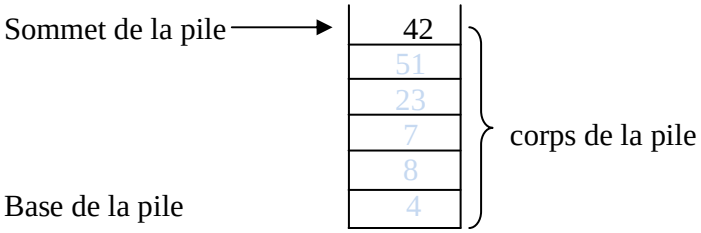
1- les Piles

1.1 Introduction :

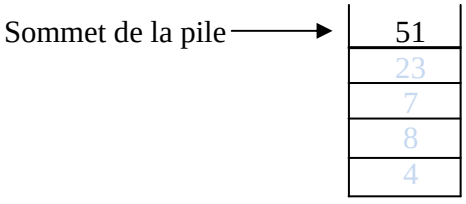
Une pile est une collection d'objets de même type dont le seul élément directement accessible est le dernier élément introduit dans la pile. L'opération INSÉRER dans une pile est souvent appelée EMPILER et l'opération SUPPRIMER, est souvent appelée DÉPILER. Ces noms sont spécifiques aux piles rencontrées dans la vie de tous les jours, comme les piles d'assiettes. L'ordre dans lequel les assiettes sont dépilées est l'inverse de celui dans lequel elles ont été empilées, puisque seule l'assiette empilée en dernier est accessible. Cette dernière est appelée **sommet de la pile**. Pour accéder à élément particulier de la pile, il est donc nécessaire de dépiler (enlever) tous les éléments qui se situent au dessus de lui.

1.2 Caractéristiques de la structure de la pile :

Il en résulte que pour chaque pile, le seul élément directement accessible est le sommet de la pile, quand un nouvel élément est empilé, il est placé par-dessus l'ancien sommet, par conséquent, ce nouvel élément devient sommet. De même un dépilement de l'élément sommet restitue l'état de la pile avant son empilement.
Exemple : soit une pile d'entiers dans la figure suivante où la seule valeur accessible est 42 :



Après dépilement on obtient la pile suivante :



Ce principe de description de la structure de donnée **pile** nous introduit à une classe de raisonnement et d'élaboration d'algorithmes très efficaces permettant la résolution de beaucoup de problèmes tels que :
1) la gestion du système d'exploitation de l'espace mémoire associé aux différents appels de fonction d'un programme (**segment de pile** vu au chapitre sous-programme) et l'implication du calcul de la récursivité.

2) La structure de pile est très bien adaptée à l'analyse des expressions arithmétiques.
3) Mémorisation des pages webs visitées.

1.3 Implémentation d'une Pile :

Il est à noter que indépendamment de son implémentation la pile est une structure de donnée dont le contenue définit un état permettant ou non les deux opérations principales : empiler et dépiler. Comme par exemple une pile-vide ne permet pas de dépiler un élément. Il en découle un ensemble de primitives (fonctions, sous-programmes) sont nécessaires à la gestion et à la manipulation d'une pile pour respecter les contraintes associées à la pile. Principalement les opérations sur une pile sont : Initialisation de la pile à vide, Valeur du sommet, Empiler une valeur, Dépiler une valeur, Vérifier si non pleine à l'empilement, Vérifier si non vide au dépilement. Ces primitives sont adaptées selon la structure mémoire utilisée. Chaque implémentation délivre ces primitives. On implémente une pile soit par une structure statique de tableau ou par une structure dynamique la liste chaînée.

1.3.1 Implémentation d'une pile d'entiers par un tableau

a-Déclaration d'une pile d'entier :

```
#define taillemax 50
 typedef  struct pile
          { int ind;
            int tab[taillemax];
          } pile;
 pile p ;
```

b-Description fonctionnelle:

Ce sont des primitives qui permettent de gérer une pile.

1) Initialisation de la pile à vide.

```
void init(pile *p)
 { p→ind=-1 ;
 }
```

2) Vérifier état de la pile : si la pile est vide.

```
int  pilevide(pile p)
 {return (p.ind== -1) ;
  }
```

1) Vérifier état de la pile si la pile est pleine.

```
int  pilepleine(pile p)
 {return (p.ind== taillemax-1) ;
  }
```

2) Valeur du sommet de la pile.

```
int sommet(pile p)
{ if(p.ind>=0) return p.tab[p.ind];
}
```

3) Empiler une valeur dans une pile.

```
void empiler(pile *p, int v )
{ if(p→ind<taillemax-1) { p→ind=p→ind+1;
                          p→tab[p→ind]=v ;
                        }
}
```

4) Dépiler une valeur d'une pile..

```
void depiler(pile *p, int *v )
{ if(p→ind> -1) { *v= p→tab[p→ind] ;
                  p→ind=p→ind-1;
                }
}
```

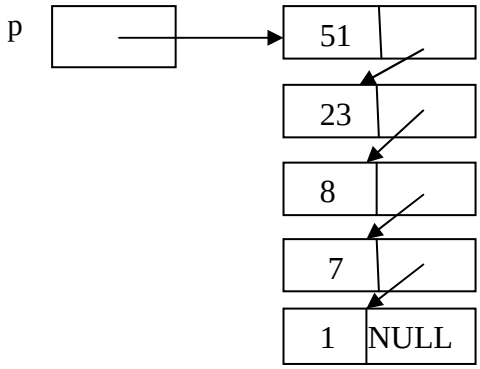
Exemple : Soit une pile d'entiers p à valeurs ordonnées, on désire chercher si une valeur val se trouve dans cette pile ou non, sinon l'insérer.

```
int   rechinsert (pile *p , int val)
  {     pile p1;
        int  v,bool=1;
           init (&p1);
           while (pilevide (*p)==0 && sommetpile(*p) > val )
                                {
                                        depiler (p, &v);
                                        empiler (&p1, v);
                                }
            if (pilevide (*p) || sommetpile(*p) < val)
                                {
                                        empiler (P, val);
                                        bool=0;
                                }

            while (pilevide (p1)==0)
                                {
                                        depiler (&P1, v);
                                        empiler (P, v);
                                }
            return bool;
}
```

1.3.2 Implémentation d'une pile d'entiers par une liste chaînée :

Les limitations d'une implémentation de la pile par un tableau sont liées à la taille limitée du tableau et le nombre de déplacements de données. L'intérêt de représenter une pile avec une liste chaînée permet de résoudre ces contraintes où la tête représente le pointeur vers le sommet, l'empilement est la simple fonction d'ajout en tête et le dépilement est la fonction de suppression en tête que nous avons vue dans les listes chaînées.



a-Déclaration d'une pile d'entiers :

```
typedef struct pile
     {
        int val ;
        struct pile * suivant ;
     } Pile ;
  Pile * p = NULL;
```

b-Description fonctionnelle:

Ce sont des primitives qui permettent de gérer une pile.

1) Initialisation de la pile à vide.

```
void init(Pile **p)
 { *p =NULL;
 }
```

2) Vérifier état de la pile : si la pile est vide.

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

2) Vérifier état de la pile si la pile est **pleine**.
   On ne vérifie rarement on supposant que l'allocation dynamique est toujours possible.
3) Valeur du sommet de la pile.

```
int sommet(Pile *p)
{ if(p!=NULL) return p→val;
}
```

4) Empiler une valeur dans une pile.

```
void empiler(Pile **p, int v )
  {  Pile *nouv;
     nouv=malloc(sizeof(liste))
    nouv→val=v;
    nouv→suivant=*p;
    *p= nouv;          }
```

5) Dépiler une valeur.

```
void depiler(Pile **p, int *v )
  { if(*p!=NULL) { *v= p→val;
                   Pile *d=*P ;
                   *p=(*p)→suivant ;
                    free(d) ;
                 }
  }
```

1.4 Comparaison des deux implémentations

Toutes les implémentations des opérations d'une pile utilisant un tableau ou une liste chaînée ne diffèrent pas en terme de temps d'exécution. Par conséquent, d'un point de vue d'efficacité, aucune des deux n'est meilleur que l'autre. D'un point de vue de complexité mémoire, le tableau doit déclarer une taille fixe initialement. Une partie de cet espace est perdue quand la pile n'est pas entièrement exploitée. La liste peut augmenter ou réduire au besoin mais exige un espace pour mémoriser les adresses des pointeurs.

1.5 Exercice pour Travaux Pratique(TP): Un projet de TP classique qui consiste à implanter un programme de l'évaluation des expressions arithmétiques en adoptant la structure de la pile.

Définitions et principe de passage de l'infixé au postfixé

La notation postfixée (polonaise) d'une expression arithmétique consiste à placer les opérandes devant l'opérateur. La notation infixée (parenthèsée) consiste à entourer les opérateurs par leurs opérandes. Les parenthèses sont nécessaires uniquement en notation infixée.

L'expression infixée (3 + 5) * 2 s'écrira en notation postfixée (notation polonaise): 3 5 + 2 * alors que 3 + (5 * 2) s'écrira: 3 5 2 * +.

La notation infixée: 8*9/6. s'écrira en postfixée est: 8 9 * 6/.

L'évaluation d'une expression postfixée : 3 5 2 * + pour obtenir la valeur dans R suit les opérations suivantes : R=5*2 ; puis

R=3+R.

Définition des règles d'expression en postfixée :

1- Une opérande a est une expression postfixée (pour le tp on ne considèrera que les valeurs constantes).
2- Si a et b sont des expressions postfixées et Φ un opérateur binaire alors abΦ est une expression postfixée.
3- Si a est une expression postfixées et Φ un opérateur unaire alors aΦ est une expression postfixée.
4- Seules les regles 1,2 et 3 peuvent définir une expression postfixée.

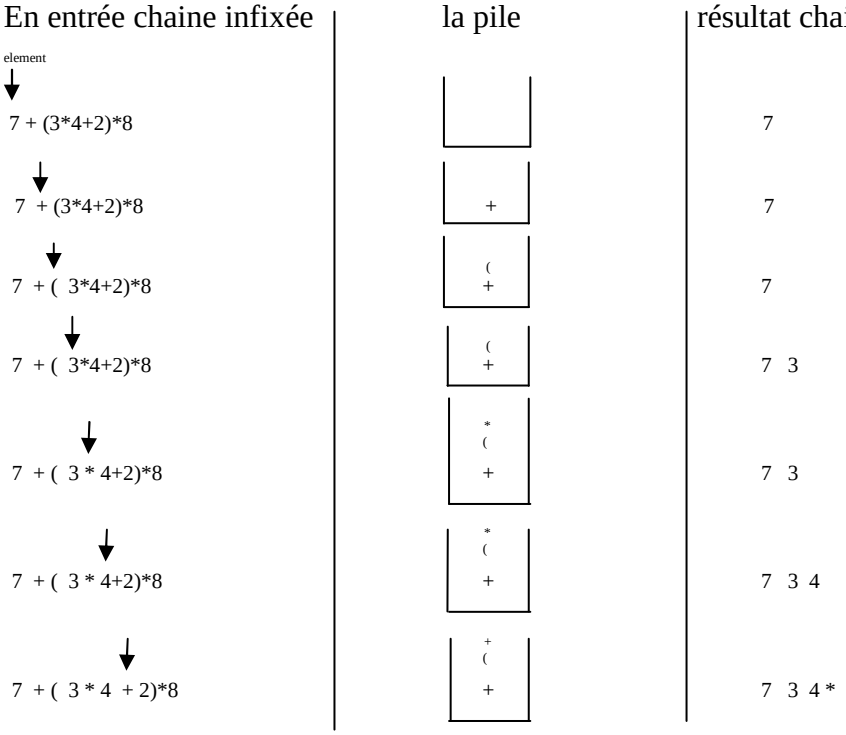
Donc parce que 5 2* est expression postfixée et 3 est une expression postfixée alors 352*+ est une expression postfixée.

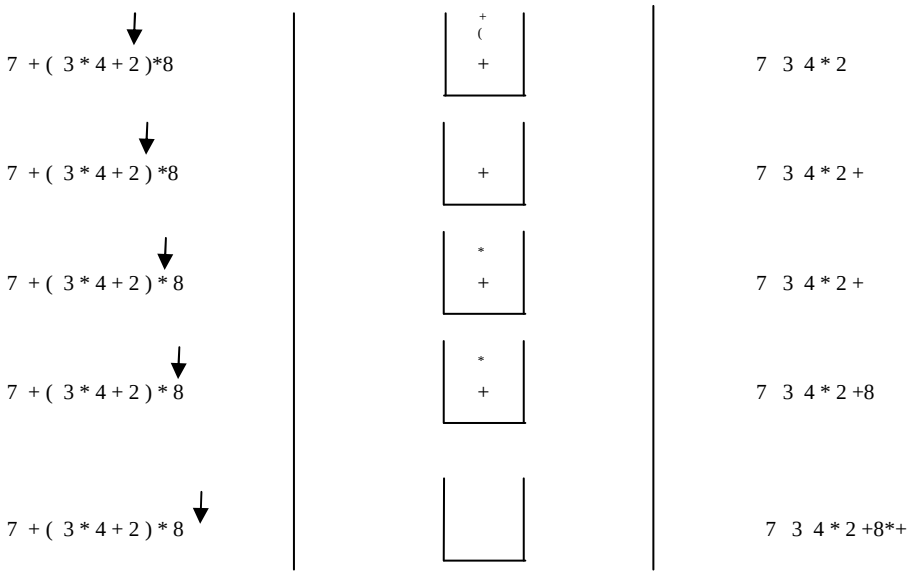
De plus l'expression 3+5*2 s'écrira en postfixée 3 5 2*+ seulement si on introduit la notion de priorité dans l'algorithme de conversion. En attribuant une priorité plus élevée à l'opérateur '*' qu'a '+' on élimine toute ambiguïté(ce qui est valable pour l'ensemble des opérateurs).

L'algorithme de conversion de l'infixé au postfixé, utilise, en plus de la structure de données qui contient l'expression infixée (chaine de caractères par exemple), une **pile** (des opérateurs) qui sert à l'empilement d'opérateurs les moins prioritaires jusqu'à la rencontre d'un autre encore moins prioritaire que ceux de la pile, auquel cas, ces derniers seraient dépilés en les ajoutant dans la structure de données résultat qui contient l'expression postfixée.
Pour la parenthése '(' qui délimite une sous expression il faut donc l'empiler pour pouvoir dépiler tous les opérateurs contenus entre '(' et ')' à la rencontre de la parenthèse fermente ')'. La parenthèse est empilée dans la pile des opérateurs on doit lui associer une priorité la plus faible.
Exemple on doit pouvoir convertir : 5+6*7+(3*4+2)*8 en => 5 6 7*+3 4* 2+ 8 *+
En attribuant et respectant la priorité aux différents opérateurs par exemple :
3 : opérateurs unaires
2 : / *
1 : + -
0 : (
Opérateur binaries: +, -, *, /, etc.,
Opérateur unaires: moins unaire, racine carrée, sin, cos, exp, … etc.

L'objectif final est de parvenir à un programme qui permet d'évaluer des expressions en entrée infixées. L'écriture d'un tel programme doit prendre en compte :
1. La définition du type de données représentant les expressions arithmétique infixées et post fixée.
2. Un analyseur lexicographique permettant l'identification des opérandes (on ne prend que des valeurs constantes pas d'identificateurs de variables) et l'identification des opérateurs.
3. Implémentation de la fonction C de conversion et de la fonction C de l'évaluation.

Exemple de conversion de l'infixé au postfixé :

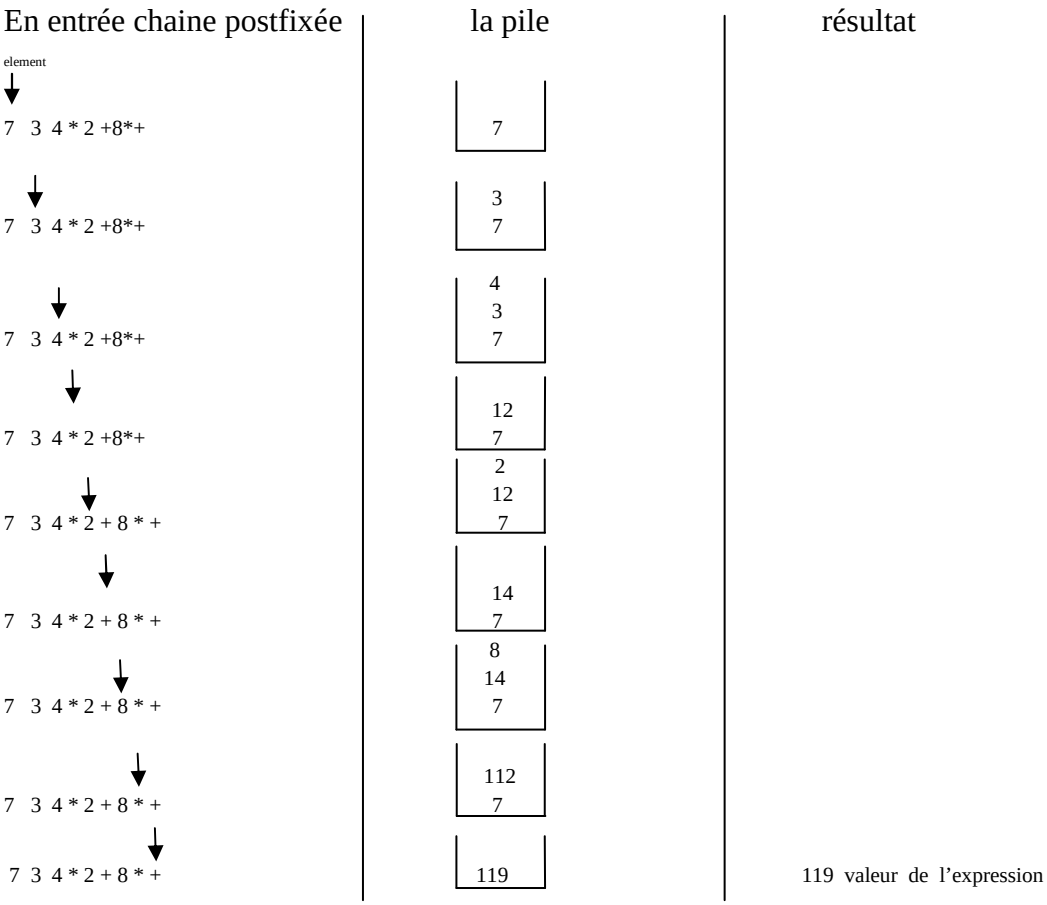




1) Algorithme de conversion de l'infixé au postfixé.

```
Procedure conv_InfixaPostfix(E infixchaine : chaine de caractères ; S postchaine :chaine de caractères)
var
X : varaiable caractére ;
element :{operande,operateur} ;
fonction elementinf(sous-chaine de infixchaine) :{operande,operateur} ;
P : pile des caractéres ;
// Etant donnée une chaine infixée à parcourir de debut de gauche : infixchaine ;
debut
Initialiser la pile P à vide;
tantque(on a pas fini d'explorer infixchaine)
 debut
  element←elementpost(courant de infixchaine);
  si (element='(' ) alors empiler(P,'(')
     sinon si (element= operateur) alors
                                    debut
                                     tantque(nonpilevide(P) et priorite(sommetpile(p))>= priorite(element))
                                          debut depiler(P,X) ;
                                                postchaine←postchaine +X ;
                                          fin
                                      empiler(P,element) ;
                                    fin
              sinon si (element=')') alors
                                    debut
                                     tantque(nonpilevide(P) et sommetpile(p)) != '(')
                                      debut depiler(P,X) ;
                                            postchaine←postchaine +X ;
                                       fin
                                     depiler(P,X) ;
                                    fin
                          sinon
                           postchaine←postchaine +element ;
     fin
   tantque(nonpilevide(P))
     debut depiler(P,X) ;
           postchaine←postchaine +X ;
     fin
fin
```

Exemple d'évaluation d'une expression postfixée :



2) Algorithme d'évaluation d'une expression postfixée:

```
fonction Eval_postfixe( e postchaine : chaine de caractère) :reel  ;
var
// Etant donnée une chaine postfixée à parcourir de debut de gauche : postchaine ;
R ,X,Y variable à valeur réel ;
element :{operande,operateur} ;
fonction elementpost(sous-chaine de postchaine) :{operande,operateur} ;
P : pile des reel ;
debut
Initialiser la pile P à vide;
tantque(on a pas fini d'explorer postchaine)
        debut
              element←elementpost(courant de postchaine);
              si (element =operande) alors empiler(P,element);
                  sinon si (element =operateur binaire )
                                    debut
                                       dépiler (P,X) ;
                                       dépiler (P,Y);
                                       R ← y operateur x;
                                       empiler (P,R) ;
                                     fin
                                sinin si(element =opérateur unaire)
                                          debut
                                            dépiler (P, x);
                                            R← opérateur(x);
                                            empiler (P,R);
                                           fin
        fin
Eval_postfixe=sommetpile(P) ;
fin
```

2- Les Files :

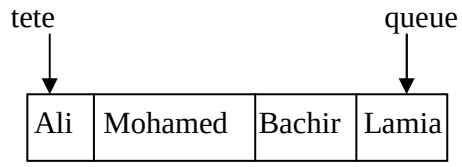
2.1 Introduction

La structure de données file est définie par une ***tête*** et une ***queue*** d'une liste. Lorsqu'un élément est arrivé, il prend place à la queue de la file, comme un client nouvellement arrivé prend sa place à la fin de la file dans la salle d'attente. On appelle ENFILER l'opération INSÉRER sur une file et on appelle DÉFILER l'opération SUPPRIMER assurant la propriété FIFO de gestion dans la file (First In, First Out) ou (premier arrivé, premier sorti). En informatique une file sert essentiellement à stocker des données qui doivent être traitées selon leur ordre d'arrivée. La structure de la file est très utilisé par les programmes des systèmes d'exploitation, l'exemple le plus connu est celui de l'impression de documents reçus par une imprimante qui imprime le premier document arrivé et termine par le dernier. Ce qui fait que les objets quittent la structure de données dans l'ordre de leur ordre d'arrivée.

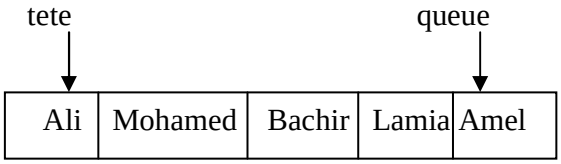
En programmation, une structure de file est caractérisée par un état composé principalement par :
- Une structure de données pour enregistrées les valeurs (elle peut être statique ou dynamique)
- Une variable Debut (tete) qui indique le premier élément de la file.
- Une variable Queue qui indique le dernier élément de la file.

Comme pour les piles, la manipulation d'une file revient à l'utilisation de primitives (fonctions et procédures) dites de bases.
Ces primitives sont :
- Init_File : permet d'initialiser une file à vide lors de sa création ;
- File_vide : pour vérifier si une file est vide ou non et savoir alors s'il reste des valeurs à traiter ou non ;
- File_pleine : pour vérifier s'il est possible de rajouter ou non un nouveau élément (utilisée dans le seul cas des files statiques) ;
- Enfiler : permet d'ajouter un nouvel élément à la file (après le dernier élément de la file qui se trouve au niveau de sa queue et dans le cas d'une file non pleine) ;
- Defiler : permet de supprimer une valeur (se trouvant au début de la file) et de la renvoyer en paramètre. Cette opération n'est possible que si la file n'est pas vide.

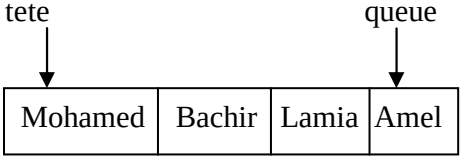
Exemple : soit une file de noms définie par tete et queue



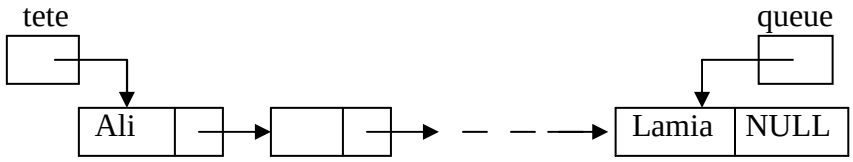
Enfiler (&tete, &queue, "Amel") on obtient :



Defiler (&tete, &queue, &v) on obtient :



2.2 Implémentation d'une file d'entiers par une liste chainée (par structure dynamique):
Une liste chainée plus une variable queue (indiquant l'emplacement où insérer le nouvel arrivé) est très adaptée à l'implémentation d'une file.



a-Déclaration de la structure de données de la file d'entiers :

```
typedef struct file {
                        int val ;
                        struct file *suivant ;
                        } file ;
file *tete,*queue ;   //ou par une structure de deux champs
```

b-Description fonctionnelle:

1) Initialisation de la file à vide.

```
void Init_File(file **t,**q)
   {*t=NULL ;
    *q=NULL ;
   }
```

2) Vérifier état de la file : si la file est vide.

```
int  File_vide(file *t)
 {return (t== NULL) ;
  }
```

3) Enfiler un nouvel élément dans une file.

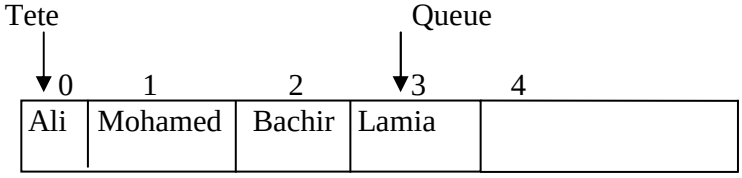
```
void Enfiler(file **t, file**q, int v )
  {  file *nouv;
    nouv=malloc(sizeof(file))
    nouv→val=v;
    nouv→suivant=NULL; //il est dernier
    if (*t==NULL) {*t=nouv ;*q=nouv ;}
       else{(*q)→ suivant=nouv ;
              *q=nouv ;
            }
```

4) Défiler un élément de la file.

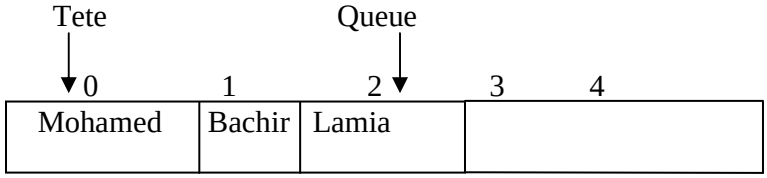
```
void Defiler(file **t, file**q, int *v )
  {  file *d ;
   if (*t !=NULL) {*v= (*t)→val ;
                    d=*t ;
                   *t=(*t)→suivant ;
                    free(d) ;
                   }
```

2.3 Implémentation d'une file d'entiers par un tableau (par structure statique):

2.3.1 Tableau et gestion par décalage : Facile à l'implémentation mais couteuse à chaque défilement, il s'agit de réaliser le défilement à la première position du tableau (l'élément qui se trouve à l'indice 0) et à chaque défilement il faut décaler tous les éléments de la liste pour que le prochain à défiler se mette à la première position comme dans cet exemple :
soit la file



Au défilement :



a-Déclaration de la structure de données tableau de la file d'entiers :

```
#define taille 100
 typedef struct
   {  int[taille] tab;
      int queue, tete=0 ;
   } File;
```

b-Description fonctionnelle:

1) Initialisation de la file à vide.

```
void Init_File(File *f)
   {(*f).queue=-1 ;
    }
```

2) Vérifier état de la file : si la file est vide.

```
int  File_vide(File *f)
 {return (f→queue== -1) ;
  }
```

3) Enfiler un nouvel élément dans une file.

```
void Enfiler(File *f, int v )
 {  f→queue= f→queue +1 ;
   If(f→queue <taille)
        f→tab[f→queue]=v ;
 }
```

4) Défiler un élément de la file.

```
void Defiler(File *f, int *v )
 {  int i;
    *v= f→tab[0] ;
    for(i=0 ;I < f→queue;i++)
       f→tab[i]= f→tab[i+1] ;
    f→queue= f→queue – 1 ;
 }
```

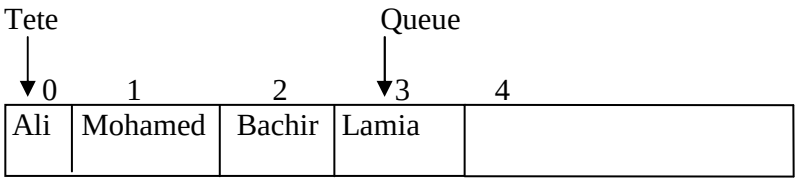
5) Vérifier état de la file : si la file est pleine.

```
  int  File_pleine(File *f)
   {return (f→queue== taille) ;
   }
```

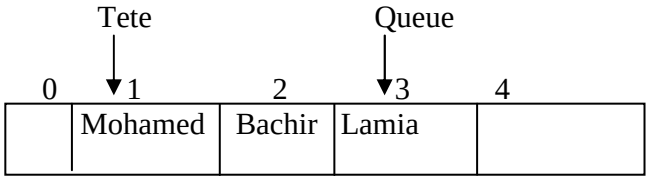
1.3.2 **Tableau et gestion par flot :**

C'est une approche simple et il n'y a pas de boucle de décalage mais, cette approche gaspille de l'espace mémoire, il s'agit d'incrémenter queue à chaque enfilement et incrémenter tête à chaque défilement. La déclaration de cette structure est la suivante avec initialisation de la queue à -1 et tête à 0, ce qui définie l'état de la file à vide si la valeur de tête est supérieur à celle de queue.

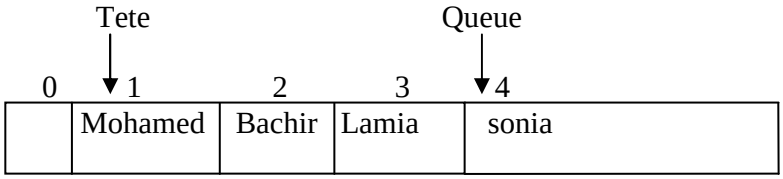
```
#define taille 100
 typedef struct
   {  int[taille] tab;
     int queue=-1, tete=0;
   } File;
```



Au défilement :



A un nouvel enfilement

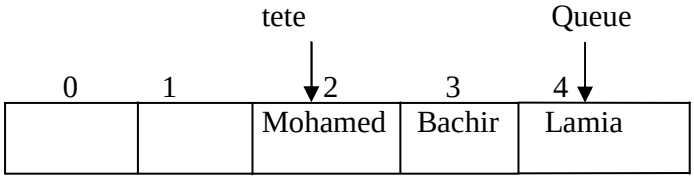


On note qu'on ne récupère pas les espaces utilisés après les défilements, et pourtant, la valeur de queue peut atteindre celle de taille du tableau et indiquer la saturation de l'espace du tableau ce qui donne comme solution ce qui suit.

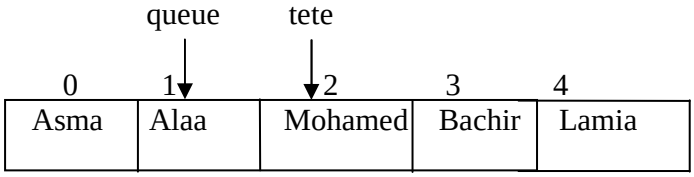
2.3.3 Tableau et gestion circulaire du tableau:

Cette solution assure que lorsque la valeur de la queue atteint la taille du tableau, on effectue les ajouts des éléments à partir du début du tableau, les indices tête et queue progressent maintenant modulo la taille du tableau. Il s'agit du principe de l'approche par flots en utilisant la même déclaration avec les incrémentations se font modulo taille. Toute fois, un problème d'ambiguïté à ne pas négliger et à prendre en compte : examinons par exemple les cas qui se présentent en considérant un tableau de taille 5 :

A) supposons que nous ayons atteint l'état suivant :

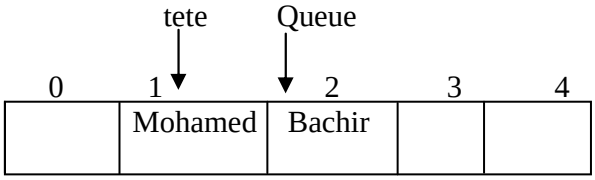


A l'enfilement de « Asma » puis « Alaa »

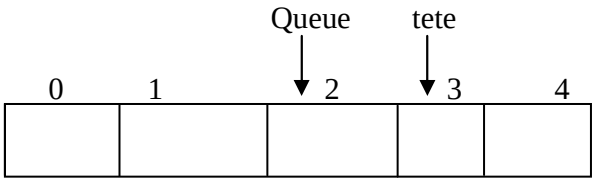


On obtient un état qui définie **la file pleine** avec **tete = queue +1 modulo N**

B) supposons que nous ayons atteint un autre cas de l'état suivant :



Au défilement de « Mohamed» puis « Bachir »



On obtient la valeur de **tete= queue+1 modulo N qui définie aussi l'état vide.**

On peut conclure que si on considère cette approche on doit ajouter un paramètre ou un critère permettant d'éliminer cette ambiguïté. Avec cette conclusion, et à ce niveau de cours en

structure de données et algorithmique, on est en mesure de proposer une solution à ce genre de problèmes. Avec cette dernière phrase et à partir des deux parties du cours, j'espère avoir fait le tour de ce qu'il faut acquérir comme connaissances de base pour l'algorithmique et la programmation.