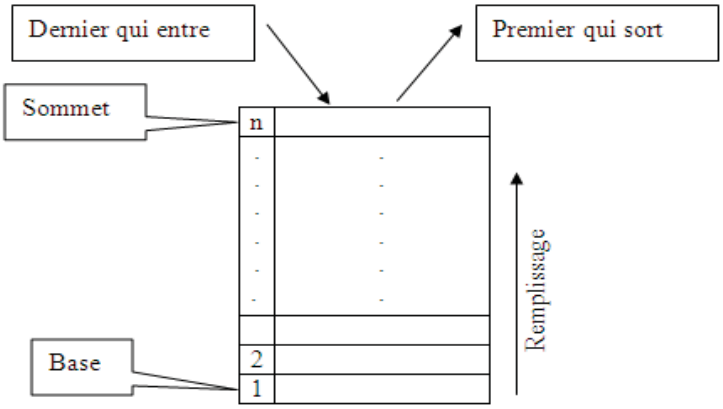
Chapitre 4
Structure de données : les piles

1. Définition

Une pile est une liste ordonnée d'éléments où les insertions et les suppressions d'éléments se font à une seule et même extrémité de la liste appelée le sommet de la pile. Le principe d'ajout et de retrait dans la pile s'appelle LIFO (Last In First Out) : "le dernier qui entre est le premier qui sort". Il est impossible donc d'accéder à un élément au milieu.



Selon le type d'implémentation, on distingue deux types de Piles:

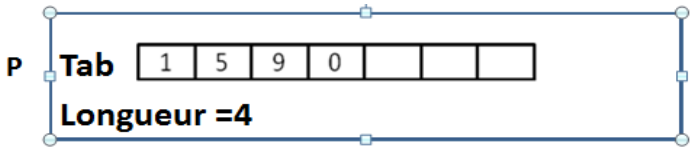
- **Pile statique (contiguë):** Implémentation par un tableau.
- **Pile dynamique:** Implémentation par une liste chainée

2. Piles statiques

Implémentation par un tableau et dans ce cas :

- La capacité de la pile est limitée par la taille du tableau.
- L'ajout à la pile se fait dans le sens croissant des indices, tandis que le retrait se fait dans le ses inverse.

Exemple : Soit la Pile statique P suivante avec un tableau 7 élément (taille fixe) et une variable longueur contient la valeur 4 qui est le nombre des éléments existe dans la liste.



2.1. Définition du type pile

Type Structure Pile
début
Tab: Tableau[MAX] d'Éléments ;
longueur : Entier ;// garde le nombre d'éléments stockés dans la Pile
Fin

Exemple: Pile d'entiers
Type Structure Pile
début
Tab: Tableau[1000] d'entiers ;
longueur : Entier;
Fin

2.2. Déclaration en C++

```
struct pile {
        type_des_élements Tab [Max];
         int   longueur;
    }
```

// type_des_éléments c'est le type des élément de la pile (int, char, float, étudiant,...etc)

Exemple: Déclaration d'une pile d'entiers

```
# include <iostream>
const int Max=1000;
struct pile {
   int Tab [Max];
   int longueur;
}
int main ( )
{   pile L1, L2 ;
. . .
}
```

2.3. Opérations usuelles sur les piles statiques

Dans cette section nous présentons les opérations primitives sur les piles statiques.

1) **La procédure initialiser** : initialise la variable longueur d'une pile à la valeur 0. Elle crée donc une pile vide.

```
Procédure Initialiser(var P: Pile)
Début
      P.longueur ← 0;
Fin
```

En C++ :

```
void initialiser (pile &P) {
   P.longueur =0;
}
```

2) **La fonction est_vide** : une fonction booléen qui reçois une pile en entrée et retourne vrai si la pile est vide (Longueur = 0) et faux sinon.

```
Fonction Est_vide(P: Liste) : Booléen
Début
      Retourne (P.Longueur == 0);
Fin
```

En C++ :

```
bool est_vide (pile L) {
  return (P.longueur == 0);
}
```

3) **La fonction est pleine:** teste si une pile est pleine ou non. Elle retourne vrai si la longueur de la pile égal sa taille maximale (longueur = max).

```
Fonction Est_pleine (P : Pile) : Booléen
Début
     Retourne (P.longueur==Max);
Fin
```

En C++ :

```
bool est_plein (pile L) {
  return(P.longueur==Max);
}
```

4) **La fonction sommet** : retourne l'élément en somment de la pile (le dernier élément empilé).

```
Fonction sommet (P: Pile) entier
Debut
  retourner(P.Tab[P.longueur]);
fin
```

En C++ :

```
type sommet (pile L) {
  return(P.Tab[longueur-1]);
}
```

5) **La procédure empiler** : Si la Pile est non pleine, ajoute un élément en sommet de pile et incrémente la variable longueur de 1 sinon elle ne fait rien.

```
Procédure Empiler(P : Pile, x:  type)
Début
 Si !est_plein(P) alors
   P.Tab [P.Longueur+1] ← x
   P.Longueur ←  P.Longueur+1
  Finsi
Fin
```

En C++ :

```
void empiler (pile &P, type x) {
    if (!est_plein(P))
    {
       L.Tab[L.Longueur]= x;
       L.Longueur++;
    }
}
```

6) **La procédure dépiler** : si la pile est non vide, elle décrémente la variable longueur de 1 sinon elle ne fait rien.

```
Procedure dépiler (var P: Pile)
début
    si !est_vide(P) faire
          P.Longueur←P.Longeur-1;
    finsi
fin
```

En C++ :

```
void depiler (pile &P) {
    if (!est_vide(P))
    {
       L.Longueur--;
    }
}
```

7) **La fonction taille**: retourne le nombre d'éléments déjà empilés dans la pile.

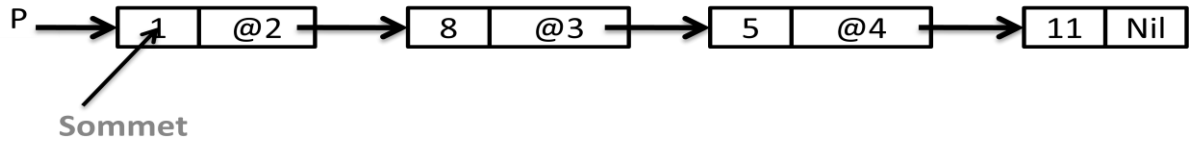
```
Fonction taille (P: Pile): entier
début
 Retourner P.Longueur;
Fin
```

En C++ :

```
int taille (Pile P)
{ return P.longueur ;
}
```

3. Pile dynamique

C'est une liste chainée où l'empilement et le dépilement se font seulement à la tête et de la liste.



3.1. Définition du type Pile

Type

Structure Maillion

Ele : typeq; // typeq désigne un type quelconque (int, float, personne, étudiant ...etc.

suivant: * Maillion**;**

fin

type Pile : * Maillon;

3.2. Définition du type Pile en C++

```
struct maillon {
   type_des_élements  ele;                    // déclaration d'un maillon en C++
   maillon *suivant;
};
typedef maillon *pile;     // définition de type pile
```

Exemple : Déclaration d'une pile chainée d'entier

```
# include <iostream>
struct maillon {
   int ele; // typeqq
   maillon *suivant;
}
typedef maillon *pile;

int main ( )
{   Pile L, L1 ;
. . .
}
```

3.3. Opérations primitives

Les opérations primitives sur les piles dynamiques sont les suivantes :

1) **Est_vide** : retourne vrai si la Pile est vide (P =NULL) sinon retourne faux.

```
Fonction Est_vide(P : Pile) : Booléen
Début
    Si (P=Nil) alors
      Retourner ( vrai);
   Sinon
       Retourner (faux);
   Finsi
Fin
```

En C++ :

```
bool est_vide (pile P)
{
    return (p== NULL);
}
```

⇔

```
bool est_vide (pile P) {
        if (P==NULL)  return true;
        else    return false;
        }
```

2) **Sommet** (P: Pile) : retourne l'élément existe dans l'entête de la Pile (Le dernier élément empilé).

```
Fonction Sommet (P : Pile): type
Début
     Retourner (L -> ele);
Fin
```

En C++ :

```
type sommet (pile P)
{
    return (P -> ele);
}
```

3) **Empiler:** empile un élément en sommet de la pile. cette fonction est équivalent à Ajouter pour une liste chainée.

```
Procedure Empiler(var P: Pile, x : typeq)
 Q: Pile
Début
    Q ← creer_maillon (x);
    Q -> suivant   ← P;
    P←Q;
Fi n
```

En C++ :

```
void Empiler (pile &P, type x)
{pile Q;
    Q = new Maillon ;
    Q->ele =x ;
    Q->suivant =P ;
    P=Q ;
}
```

4) **Dépiler**: retourne la pile sans son sommet. Elle est équivalente à reste pour les listes chainées.

```
Procédure dépiler (var P : Pile)
    Q : pile
debut
  Si ( ! est_vide (p)) alors
    Q←P ;
     P← p->suivant;
     Libérer(Q) ;
  Finsi
```

En C++ :

```
void  dépiler (pile &P)
{
  if ( ! est_vide (p))
    {
     pile Q=P ;
     p= p->suivant;
     delete Q;
    }
}
```

5) **La fonction taille**: retourne le nombre d'éléments déjà empilés dans la pile.

```
fonction Taille (P : Pile) :entier
    Courant : Liste
    n : entier ;
Début
   Courant = P ;  n=0 ;
    Tantque  (courant != NULL)
   {
            n++;
            courant = courant->suivant ;
   }
    Retourner n ;
}
```

En C++ :

```
int longueur (Pile P)
{   liste courant = P ;  int  n=0 ;
    while (courant != NULL)
   {
            n++;
            courant = courant->suivant ;
   }
    return n ;
```

4. Exemple d'utilisation des piles

4.1. Évaluation des expressions post-fixées

Pour l'évaluation des expressions arithmétiques ou logiques, les langages de programmation utilisent généralement les représentations préfixée et postfixée.

Dans la représentation postfixée, on représente l'expression par une nouvelle, où les opérations viennent toujours après les opérandes.

Exemple

1) L'expression **(2 + 3*) * 6*** *est exprimée, en postfixé, comme suit* ***2 3+ 6****

2) L'expression ***(a + (b * c))/(c-d)*** *est exprimée, en postfixé, comme suit :* ***a b c * + cd − /***

Pour l'évaluation des expressions postfixée, les langages de programmation utilisent le type Pile et ses primitives en parcourant l'expression de gauche à droite.

Exemple : L'algorithme suivant permettant d'évaluer l'expression arithmétique suivante en utilisant une pile comme pour un langage de programmation. **(2 + 3*) * 6*** *ou 2 3 + 6 * en postfixé*

```
. . .
 initialiser (P);
Empiler (P, 3); Empiler (P, 2);
a ← Sommet (P); Dépiler (P);
b ← Sommet (P); Dépiler (P);
Empiler (P, a+b);    Empiler (P, 6);
a ← Sommet (p); Dépiler (p);
b ←Sommet (p); Dépiler (p);
Empiler (p, a * b) ;
écrie  (Sommet (p)) ;
…
```

4.2. Appels récursifs

Pour exécuter les fonctions récursives les langages de programmation utilisent des piles pour les rendre itératif. L'exemple suivant représente une version itérative pour la fonction factoriel en utilisant une pile et ses primitives.

```
int factoriel (int n)
{
  pile  p=NULL ;
  for( int i = N, i> =0 ; i--)
   {
       if ( i==0)
          Empiler (P, 1);
       else
          Empiler (P, i);
   }
   while ( Nb_Éléments (P) > 1 )
    {  a = sommet (P); Dépiler (P);
       b = sommet (P); Dépiler (P);
       Empiler (P, a*b) ;
     }
    return (sommet (p));
}
```

Exemple :

Fact (4) =4*Fact(3)=4*3*Fact(2)=4*3*2*Fact(1)= 4*3*2*1*Fact(0)

= 4*3*2*1*1=4*3*2*1=4*3*2=4*6=24