# Structure de données: les piles et les files

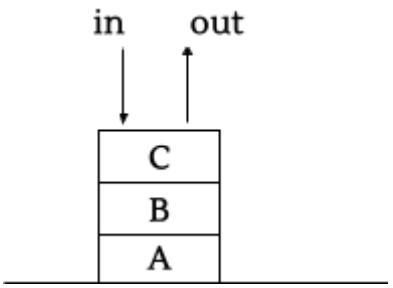
Une structure de données est une organisation logique des données permettant de simplifier ou d'accélérer leur traitement (elle se définit tout simplement comme étant la manière de représenter des données.)

Le but de ce chapitre est de décrire des représentations des structures de données telles : les piles et les files. L'autre but recherché est de voir l'importance de ces structures à travers quelques exemples d'applications.

En python, le type **liste** permet de représenter chacune de ces structures de données.

## 1. Pile

Une pile est une structure de données qui suit le principe **dernier entré - premier sorti** (Ou **LIFO** pour **Last In - First Out** ). Elle correspond exactement à l'image traditionnelle d'une pile de cartes ou d'assiettes posée sur une table. En particulier, on ne peut accéder qu'au dernier élément ajouté, qu'on appelle le **sommet** de la pile. Ainsi, si on a ajouté successivement **A**, puis **B**, puis **C** dans une pile, on se retrouve dans la situation suivante :




**C** est **empilé** sur **B**, lui-même **empilé** sur **A**. On peut soit retirer **C** de la pile (on dit qu'on « **dépile** » C), soit ajouter un quatrième élément D (on dit qu'on « **empile** » D). Si on veut accéder à l'élément A, il faut commencer par **dépiler** C, puis B.

L'opération consistant à insérer un élément dans une pile s'appelle **empiler**. L'opération consistant à supprimer un élément d'une pile s'appelle **dépiler**. Ces deux opérations s'accomplissent du **même côté** de la pile.

### 1.1. Applications

- Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton « **Afficher la page précédente** ».

- La fonction « **Annuler la frappe** » (en anglais « Undo ») d'un traitement de texte mémorise les modifications apportées au texte dans une pile.
- Pile de vérification de la correction des parenthèses d'une chaîne de caractères

### 1.2. Opérations caractérisant une structure de pile

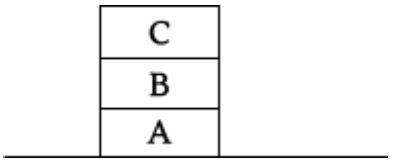
Les trois opérations principales sont réalisées par les fonctions **creer_pile**, **depiler** et **empiler**.

- **creer_pile(c)** renvoie une nouvelle pile de capacité **c**, initialement **vide**.
- **depiler(p)** dépile et renvoie le **sommet** de la pile **p**.
- **empiler(p, v)** empile la valeur **v** sur la pile **p**.

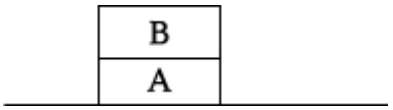
Les opérations **depiler** et **empiler** modifient le contenu de la pile passée en argument. Voici une illustration de l'utilisation de ces trois opérations :

p = creer_pile(10)
empiler(p, A)


empiler(p, B)
empiler(p, C)




depiler(p)




D'autres opérations sont disponibles, mais ne modifient pas la pile :

- **taille(p)** renvoie le nombre d'éléments contenus dans la pile p.
- **est_vide(p)** indique si la pile p est vide.
- **sommet(p)** renvoie le sommet de la pile p, sans modifier p.

### 1.3. Réalisation d'une pile

Nous pouvons exploiter la possibilité d'ajouter ou supprimer des éléments à l'extrémité droite d'un tableau en Python.

La méthode **append** permet d'empiler un élément dans une pile et la méthode **pop** pour dépiler.

```
# Implémentation du type abstrait Pile avec des listes Python
def creer_pile( ):
        return [ ]

def depiler(p):
        assert len(p) > 0
        return p.pop( )

def empiler(p, v):
        p.append(v)
```

```
def   sommet(p):
        assert len(p) > 0
        return p[-1]


def  taille(p):
        return len(p)

def   est_vide(p):
    return taille(p) == 0
```

### 1.4. Application 1 : Reconnaissance des expressions bien parenthésées

On considère le problème suivant : étant donnée une chaîne de caractères ne contenant que des caractères **'('** et **')'**, déterminer s'il s'agit d'un **mot bien parenthésé**.

Un mot bien parenthésé est une succession de parenthèses ouvrantes et fermantes, telle que une ouvrante est toujours refermée par une fermante.

**Exemple :**

- `()()()` et `(())` sont des mots bien parenthésés .
- `)(` et `((()))` sont des mots pas bien parenthésés .

Nous stockons les parenthèses ouvrantes non encore refermées dans une pile de caractères, Le programme lit caractère par caractère le mot entré :

- si c'est une ouvrante, elle est empilée ;
- si c'est une fermante, l'ouvrante correspondante est dépilée.

Le mot est accepté :

- si la pile n'est jamais vide à la lecture d'une fermante ; et
- si la pile est vide lorsque le mot a été lu.

```
# Vérification des parenthèses
def  verifparenthese (texte):
        # On commence avec une pile vide
        p = créer_pile( )
        # Parcours du texte caractère par caractère
        for  i  in  range(0 , len(texte)):
                if  texte[i]=='(':
                # Si le caractère est une parenthèse ouvrante, on empile son indice i
                        empiler(p, i)
                elif (texte[i]==')' ):
                # Sinon, c'est qu'il s'agit d'une parenthèse fermante
                        if (est_vide(p)):
                                # Si la pile est vide le mot n'est pas bien parenthèsé
                                return  False
                        # Sinon, on dépile l'indice j de la dernière parenthèse ouvrante
```

```
                j = depiler(p)
                # On affiche le couple (j,i) : la parenthèse ouvrante de l'indice j
                # correspond à la parenthèse fermante à l'indice i
                print( j , i )
    # Il ne reste plus qu' à vérifier si la pile est vide
    return  est_vide(p)
```

Cette fonction renvoie pour chaque parenthèse ouvrante, la position de la parenthèse fermante correspondante.
Par exemple :

```
>>>  mot = '(())()'
>>> verifparenthese (mot)
(1, 2), (0, 3) et (4, 5)
```

### 1.5. Application 2 : Calcul d'une expression postfixée

La **notation polonaise inverse** consiste tout simplement à utiliser la **notation postfixée** des opérateurs conjointement avec une pile. Une opération binaire est généralement notée de manière **infixée**, c'est-à-dire avec les opérandes de part et d'autre de l'opérateur, **a + b** par exemple. La notation **postfixée** consiste à placer l'opérateur après les deux opérandes, donc **a b +**. Cette écriture en apparence anodine à quelques avantages décisifs en termes de calcul. Les parenthèses deviennent inutiles (les parenthèses sont nécessaires uniquement en notation **infixée**).

**Exemples:**

- l'expression : **(3 + 5) * 2** s'écrira en notation **postfixée** (**notation polonaise**) : **3 5 + 2 ***
- l'expression : **3 +(5 * 2)** s'écrira en notation **postfixée** (**notation polonaise**) : **3 5 2 * +**
- l'expression : **6 5 2 3 + 8 * + 3 + *** = 6*(5 + ((2+3) * 8) + 3) = 6*48 = 288

**Exemple : Evaluation d'une expression postfixée en utilisant une pile**

On suppose que l'on dispose d'une **pile** initialement vide et dont le contenu va évoluer en fonction de la **lecture gauche-droite** de l'expression **postfixée**. Si on lit **une valeur**, elle est **empilée**, si on lit **une opération** #, on **dépile** les deux derniers **opérandes a et b**, et **on empile le résultat** du calcul **a # b.**

Si on prend l'expression (**32 2 12 3 7 2 - * - * -** ) on a l'évolution de la **pile** suivante:

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **P** | 6 | | | | | | | 2 | | | | | |
| **I** | 5 | | | | | | 7 | 7 | 5 | | | | |
| **L** | 4 | | | | | 3 | 3 | 3 | 3 | 15 | | | |
| **E** | 3 | | | | 12 | 12 | 12 | 12 | 12 | 12 | -3 | | |
| | 2 | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | -6 | |
| | 1 | | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | **38** |
| | **lecture** | | **32** | **2** | **12** | **3** | **7** | **2** | **-** | ***** | **-** | ***** | **-** |

**Exercice :**

- Écrire une fonction **EstNombre**(chaine) qui renvoie **True** si la chaîne de caractères en entrée est un nombre (flottant ou entier), **False** sinon.
  (Penser à la méthode split( ) et lstrip( ))
- Écrire une fonction **Operation**(o,a,b) qui prend un opérateur **o (+, -, *, /)** et deux nombres (**a** et **b**) et qui renvoie le résultat de l'opération.

- Écrire une fonction **PostFixe**(expr) qui évalue une expression postfixée (une chaîne) à l'aide d'une pile. On supposera que les opérandes sont uniquement des nombres flottants ou entiers et que les opérateurs se limitent aux quatre opérations arithmétiques **+**, **-**, *****, **/**.

```
def EstNombre(x) :
    y=x.lstrip("-")
    y=y.lstrip("+")
    y=y.split(".")
    if len(y) > 2 :
        return False
    else:
        for k in y:
            if k!="" and k.isdigit()==False:
                return False
    return True
```

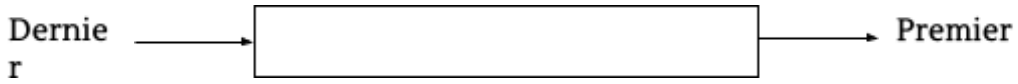
```
def Operation(o,a,b):
    if   o=="+":   return a+b
    elif o=="-":   return a-b
    elif o=="*":   return a*b
    else:
        if b!=0:
            return  a/b
        else:
            return "Erreur, division par zero"
```

```
def postfix(expr):
    expr=expr.split()
    p=creer_pile()
    for k in expr:
        if EstNombre(k)==True:
            empiler(p,float(k))
        else:
            b=depiler(p)
            a=depiler(p)
            r=operation(k,a,b)
            empiler(p,r)
    v=depiler(p)
    assert est_vide(p), 'la pile n est pas vide expression non valide '
    return v
```

## 2. File

Une **file** est une structure de données **dynamique** dans laquelle on **insère** des nouveaux éléments à la **fin** (**queue**) et où on **enlève** des éléments au **début** (**tête** de file). La **file** est basée sur le principe « Premier entré, premier sorti », on parle de mode d'accès **FIFO** (**First In, First Out**), par opposition au mode d'accès **LIFO** (**Last In First Out** ) des piles.

Le fonctionnement ressemble à une **file d'attente** : les premières personnes à arriver sont les premières personnes à sortir de la file. Une file est modifiée à ses deux bouts :




Voici les opérations de bases utilisées pour manipuler des files :

- **FileVide** : créer une file vide
- **EstVide** : indique si la file est vide ou non ;
- **Enfiler** : ajouter un élément à la fin de la file ;
- **Tête** : consulter l'élément à la tête de la file ;
- **Queue** : consulter l'élément en fin de file ;
- **Défiler** : retirer l'élément en tête de file.

**Exemple :**

En Python, les files peuvent toutes être simulées avec la **liste**. Pour ajouter un élément à la fin de la file, utiliser **append( )**. Pour récupérer un élément du devant de la file, utiliser **pop(0)** avec **0** pour indice.

```
>>> file = ["A" , " B"]
>>> file . append ("C")      # "C" est ajouté
>>> file . append ("D")      # "D" est ajouté
>>> file
["A" , " B", " C", " D"]
>>> file . pop (0)           # "A" est  supprimé
"A"
>>> file
[" B", " C", " D"]
>>> file . pop (0)
"B"
```

```
>>> file
[" C", " D"]
```

**Exercice :** **Implémenter** en Python les opérations de bases utilisées pour manipuler les files.