# Les piles

Basé sur un document du lycée Touchard-Washington.

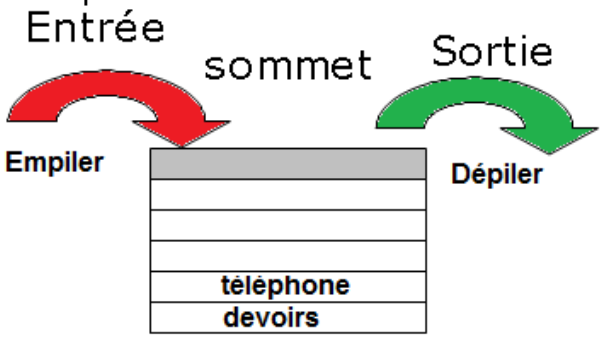
## 1. Mise en situation

Un élève (Hugo) est en train de faire ses devoirs :

- « Hugo, ton frère au téléphone ! »
- Mettre les devoirs en attente (empiler(devoirs)) et répondre au téléphone ;
- « Hugo, il faut ouvrir la porte, il y a le facteur » ;
- Mettre le téléphone en attente (empiler(téléphone)) et aller ouvrir la porte ;
- Le facteur a remis le colis ;
- Dépiler la tâche en attente (téléphone) et terminer la conversation ;
- Dépiler la tâche en attente (devoirs) et la terminer.

En informatique, une pile (en anglais stack) est une structure de données abstraite de type pile LIFO (Last In First Out)

C'est le principe même de la pile d'assiettes : c'est la dernière assiette posée sur la pile d'assiettes sales qui sera lavée en premier.




## Définition du type abstrait Pile LIFO (Last In, First Out : dernier entré, premier sorti)

- **Représentation : ['a', 'b', 'c']**

L'élément 'a' est le dernier de la pile, l'élément 'c' est le premier. Ainsi, le dernier élément ajouté à la pile sera le premier à en être retiré.

- **Opérations :**

- creer(P) : création de la pile P vide
  - → Pile
- empiler(P,element) : ajoute l'élément en dernier dans P
  - Pile x élément → Pile
- depiler(P) : retire l'élément en dernier dans P
  - Pile → Pile // précondition : P n'est pas vide (élément présent)
- longueur(P) : nombre d'éléments dans P
  - Pile → Entier
- estVide : retourne Vrai si P est vide
  - Pile → Booléen
- sommet : retourne le dernier élément de la pile
  - Pile → Pile

➢ **Décontextualisation :** La pile est composée d'éléments représentés sous forme d'une liste

➢ **Spécification :**
- Entrée : Pile d'éléments ; éléments à ajouter ou retirer
- Sortie : Pile d'éléments modifiée
- Rôle : Ajouter ou retirer des éléments de la pile.
- Précondition : L'élément à retirer est présent dans la pile

➢ **Principe de l'algorithme :** On ajoute ou on extrait les éléments de la pile

| Algorithme | Script Python |
|---|---|
| Fonction creer() → Pile<br>pile ← [ ]<br>retourner pile | `def creer():` |
| Fonction longueur(pile) → Entier<br>retourner taille(pile) | `def longueur(pile):` |
| Fonction empiler(pile, element) → Pile<br>empiler(pile) | `def empiler(pile,element):` |
| Fonction depiler(pile) → Pile<br>retourner depiler(pile) | `def depiler(pile):` |
| Fonction estVide(pile) → Booléen<br>vide ← faux<br>si pile = [ ] alors<br>vide ← vrai<br>fin si<br>retourner vide | `def estVide(pile):` |
| Fonction sommet(pile) → Pile<br>Si la pile n'est pas vide alors<br>retourner dernier élément de la pile<br>fin si | `def sommet(pile):` |

| Tests | Résultats dans la console |
|---|---|
| `pile = creer()`<br>`empiler(pile,'A')`<br>`empiler(pile,'B')`<br>`empiler(pile,'C')`<br>`print(pile)`<br>`print('Le sommet de la pile est',sommet(pile))`<br>`print('La liste est vide ?',estVide(pile))`<br>`print('La longueur de la liste est :',longueur(pile))`<br>`print(depiler(pile))`<br>`print(depiler(pile))`<br>`print(depiler(pile))`<br>`print('La liste est vide ?',estVide(pile))`<br>`print('La longueur de la liste est :',longueur(pile))` | ['A', 'B', 'C']<br>Le sommet de la pile est C<br>La liste est vide ? False<br>La longueur de la liste est : 3<br>C<br>B<br>A<br>La liste est vide ? True<br>La longueur de la liste est : 0 |

Vous trouverez en annexe un script Python utilisant la programmation orientée objet.

➢ **Complexité de l'algorithme :** https://wiki.python.org/moin/TimeComplexity

- Il termine car les opérations sur la pile élément se terminent.
- Il est correct car les éléments sont bien ajoutés ou retirés de la pile selon le principe du dernier arrivé, premier sorti.
- Il prend un temps O(nombre d'éléments en attente) car les opérations sur la pile sont en temps O(1) pour l'ajout d'un élément et en O(1) pour le retrait du dernier élément.

| | | |
|---|---|---|
| Opération empiler(P,element) | append | O(1) |
| Opération depiler(P) | pop() | O(1) |

# 2. Partie 2 : exercices

Pour les exercices suivants, utiliser le script Python utilisant la programmation orientée objet en annexe.

## Exercice 2.1

Quel est le contenu de la pile après exécution du programme suivant ?

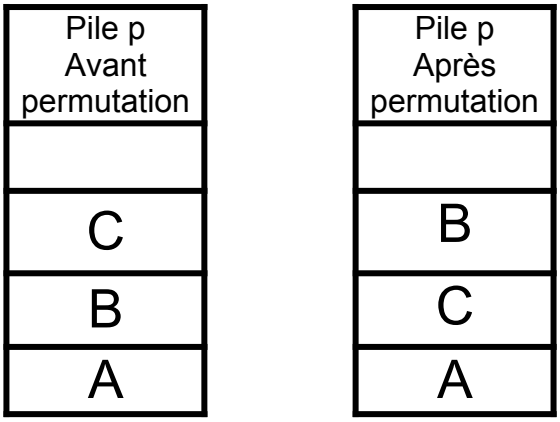
```
p = Pile()

p.empiler('A')
p.empiler('B')
p.empiler('C')
p.depiler()
p.empiler('D')
print(p)
```

| Pile p | | | | |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |

## Exercice 2.2

Compléter le programme suivant qui permute les deux éléments situés au sommet d'une pile de taille au moins égale à 2.

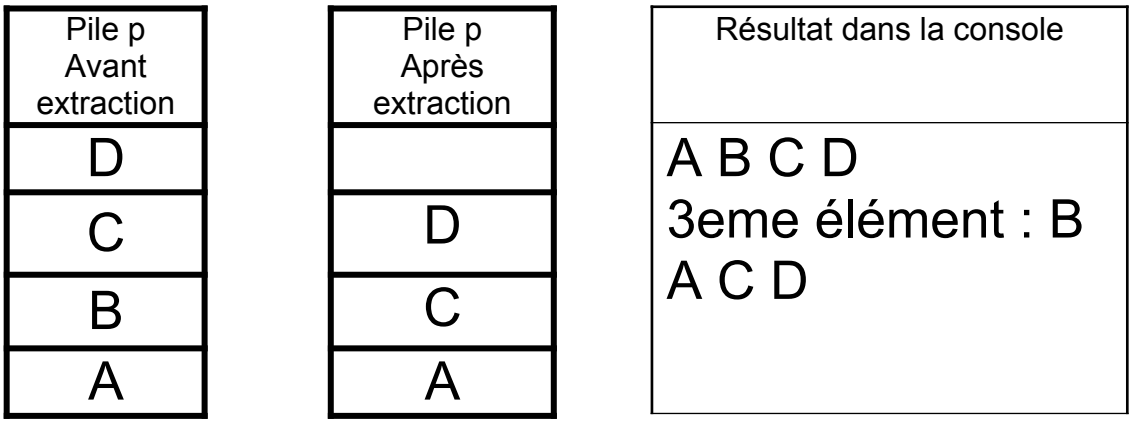
**Exemple :**




| Script python | Résultat dans la console |
|---|---|
| `p = Pile()`<br><br>`p.empiler('A')`<br>`p.empiler('B')`<br>`p.empiler('C')`<br>`print(p)`<br>`# A compléter`<br><br><br>`print(p)` | A B C<br>A C B |

## Exercice 2.3

Ecrire un programme qui dépile et renvoie le troisième élément d'une pile de taille au moins égale à 3. Le premier et le deuxième élément devront rester au sommet de la pile.



| **Script python** | |
|---|---|
| `p = Pile()`<br><br>`p.empiler('A')`<br>`p.empiler('B')`<br>`p.empiler('C')`<br>`p.empiler('D')`<br>`print(p)` | `# A compléter` |

## 3.Partie 3 : La vérification du bon parenthésage

Mr Dupond s'est encore trompé en voulant écrire un calcul sur son logiciel de traitement de texte !!!

Un problème fréquent d'un compilateur et des traitements de textes est de déterminer si les parenthèses d'une chaîne de caractères sont balancées et proprement incluses l'une dans l'autre.

Par exemple, la chaîne ((( ) ) ( ) )( ) est bien parenthésée , tandis que les chaînes )( ) ou ( ) ) ne le sont pas.

Principe :

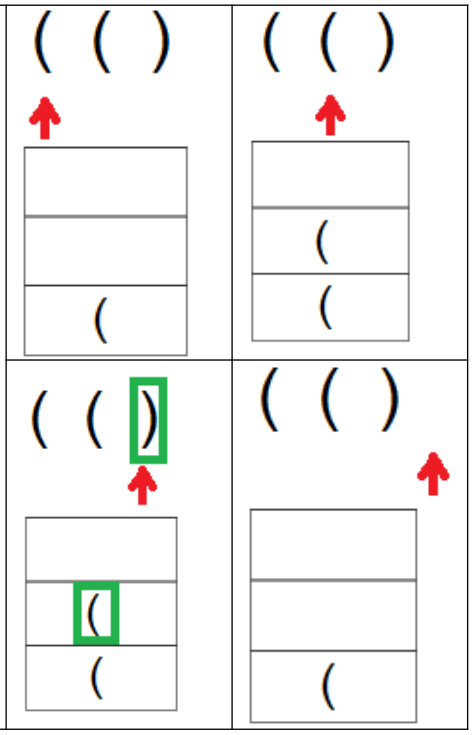
Dans une boucle tant que, on empile les parenthèses ouvrantes, puis, quand on lit une parenthèse fermante, on vérifie qu'elle correspond au sommet de la pile.

- s'il n'y a pas correspondance, il y a une erreur de parenthésage.
- sinon il y a bien une correspondance, on dépile.

A la fin de la boucle, si la pile est n'est pas vide on renvoie faux sinon vrai.

**Algorithme :**

```
fonction bien_parenthese(txt : chaine de caractère) : booleen
    verificateur ←Vrai
    instanciation  de la pile p
    indice ← 0
    tant que verificateur est vrai et indice < longueur(txt):
        si txt[indice] = "(" alors
            p.empiler(txt[indice])
        sinon si txt[indice] = ")" alors
                    si p.sommet() différent de "(" alors
                        verificateur ← False
                     else:
                        p.depiler()
                    fin si
        fin si
        indice ← indice+1
    fin tant que
    si p.estVide()=faux:
        verificateur ← False
    retourner verificateur
```



Compléter la fonction `bien_parenthese`, qui retourne vraie si une chaîne de caractères est bien parenthésée, et faux sinon. Visualiser l'exécution avec python tutor.

| Script python | Résultat dans la console |
|---|---|
| `def bien_parenthese(txt):`<br>`  verificateur = True`<br>`  p = Pile()`<br>`  indice = 0`<br>`  # A compléter`<br><br><br><br><br><br><br><br>`print(bien_parenthese("(3+2)+5*(2+8)"))`<br>`print(bien_parenthese("(3+2)+5*(2+8"))` | True<br>False |

## 4.Partie 4 : La notation polonaise inversée (NPI)

La notation polonaise inversée (NPI) (en anglais RPN pour Reverse Polish Notation), également connue sous le nom de notation post-fixée, permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses.

Dérivée de la notation polonaise présentée en 1924 par le mathématicien polonais Jan Lukasiewicz, elle s'en différencie par l'ordre des termes, les opérandes y étant présentées avant les opérateurs et non l'inverse.

La notation post-fixée d'une expression algébrique consiste à placer les opérateurs après son ou ses opérandes.

Par exemple, l'addition de a et de b sera écrite "a b +" en notation post-fixe,

Par exemple, l'expression « 3 × (4 + 7) » peut s'écrire en NPI sous la forme 3 4 7 + *

L'intérêt majeur de cette notation est qu'une expression post-fixée n'est jamais ambiguë alors que l'expression infixe "1 + 2 x 3" peut avoir deux significations : "(1 + 2) x 3" ou "1 + (2 x 3)",

Ce n'est jamais le cas d'une expression post-fixée, ce qui rend l'usage des parenthèses superflu :

- "1 2 + 3 x " ne peut être compris que de cette façon : "(1 2 +) 3 x "
- "1 2 3 x +" de cette façon : "1 (2 3 x ) +".

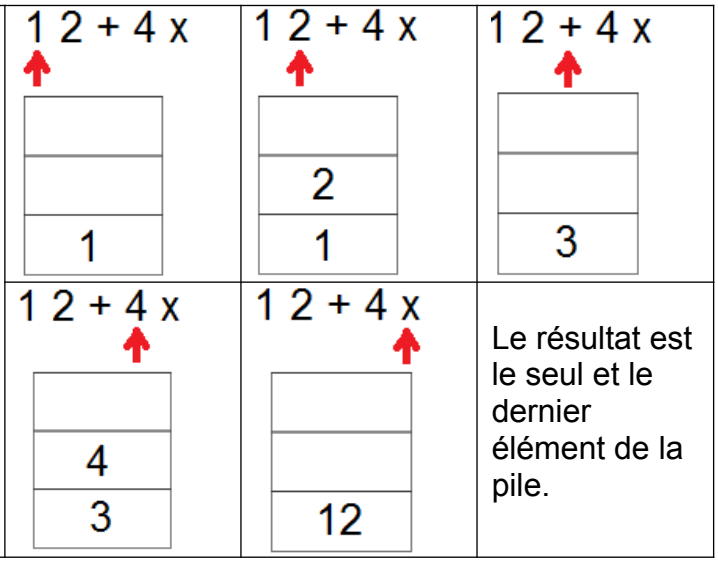
Elle est utilisée dans certains langages de programmation ainsi que pour certaines calculatrices, notamment celles de la marque Hewlett-Packard.

L'évaluation d'une expression postfixée se déroule en parcourant la liste des éléments et en suivant les règles suivantes :

Si l'élément de la liste est un opérateur binaire f alors,
On dépile les deux éléments a et b les plus hauts, et on empile le résultat de f (a;b).
Sinon l'élément est un nombre, on l'empile ;
On retourne le résultat qui est le dernier élément de la pile.

**Algorithme :**

```
fonction polonaise(liste):valeur entière ou réelle
    instanciation  de la pile p
    pour chaque element dans la liste faire :
        si element est parmi ['+','-','*','/'] alors
            b←p.depiler()
            a←p.depiler()
            r←calculer(a,b,element)
            p.empiler(r)
        sinon:
            p.empiler(element)
        fin si
     fin pour
    retourner p.depiler()
ou   retourner p.sommet()
```




Compléter la fonction qui évalue une expression écrite en NPI (on se limitera aux opérateurs binaires). Visualiser l'exécution avec python tutor.

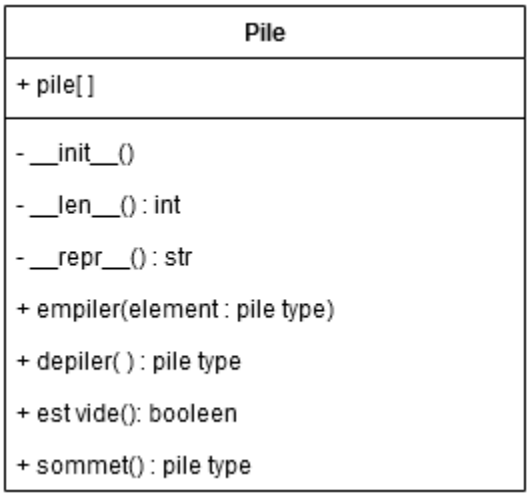
Script python

```
# fonction qui calcule le résultat de
l'opération a op b

def calculer(a,b,op):
    if op=='+':
        resultat=a+b
    elif op=='-':
        resultat=a-b
    elif op=='*':
        resultat=a*b
    elif op=='/':
        if a!=0:
            resultat=a/b
        else:
            print("erreur division par
zéro")
    return resultat
```

```
def polonaise(liste):
    p = Pile()
    # A compléter



print(polonaise([2,3,'-']))
print(polonaise([2,3,'*']))
print(polonaise([2,3,'+',4,'*']))
print(polonaise([4,3,2,'*','/']))
```

```
-1
6
20
0.6666666666666666
```

# Annexe : Piles avec la POO




```
class Pile:
    def __init__(self):
        self.pile = []

    def __len__(self):
        return len(self.pile)

    def __repr__(self):
        return ' '.join([str(i) for i in self.pile])

    def empiler(self, element):
        self.pile.append(element)

    def depiler(self):
        return self.pile.pop()

    def estVide(self):
        vide=False
        if self.pile==[]:
            vide=True
        return vide

    def sommet(self):
        if not self.estVide():
            return self.pile[-1]
```

```
p = Pile()

p.empiler('A')
p.empiler('B')
p.empiler('C')
print(p)
print('Le sommet de la pile est',p.sommet())
print('La liste est vide ?',p.estVide())
print('La longueur de la liste est :',len(p))
print(p.depiler())
print(p.depiler())
print(p.depiler())
print('La liste est vide ?',p.estVide())
print('La longueur de la liste est :',len(p))
```