Structure de pile

Table des matières

I Prinicpe

Le principe de la pile informatique (*stack* en anglais) est résumé par l'acronyme LIFO :

L*ast* **I***n* **F***irst* **O***ut*

C'est le principe d'une pile d'assiettes :

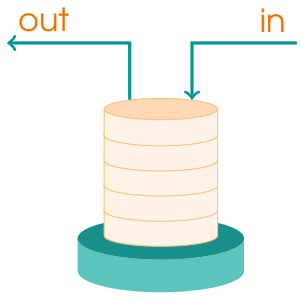


FIGURE 1 – Structure de pile (LIFO)

Il existe également une structure FIFO (**F***irst* **I***n* **F***irst* **O***ut*) de file (*queue* en anglais), hors-programme d'informatique de tronc commun :

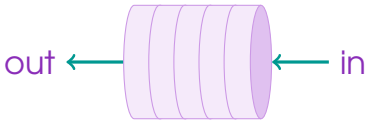
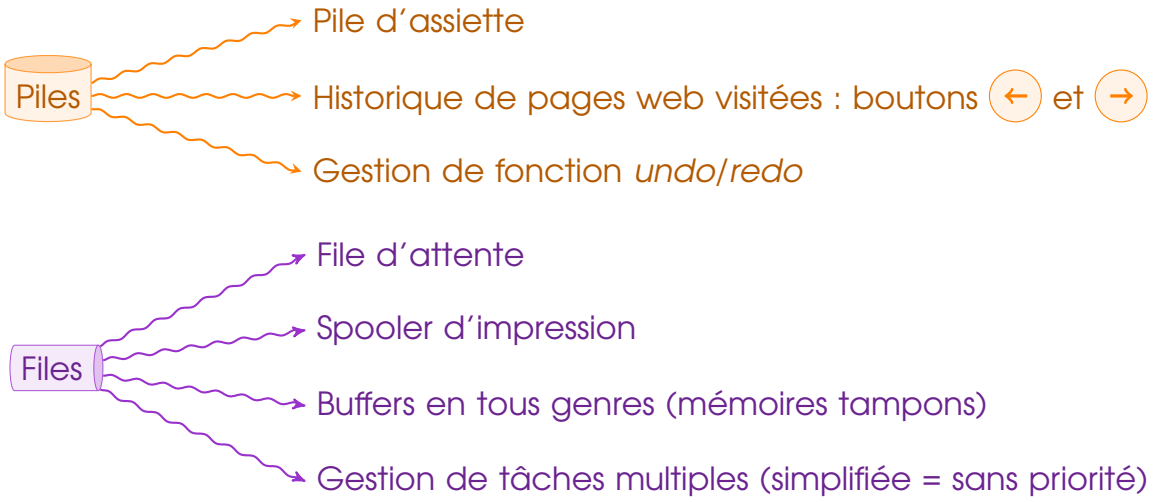


FIGURE 2 – Structure de file (FIFO)

Voici quelques exemples d'usage courant de l'une ou l'autre de ces structures :



Pour implémenter une structure de pile, on a besoin d'un nombre réduit d'opérations de bases :

- Créer une pile vide,
- Tester si une pile est vide,
- Dépiler le dernier élément du haut (**pop**) à temps constant,
- Empiler un élément (**push**) à temps constant.

À cela peuvent éventuellement s'ajouter des fonctions permettant de :

- renvoyer la taille de la pile,
- renvoyer le sommet de la pile sans la modifier (= pop + push).

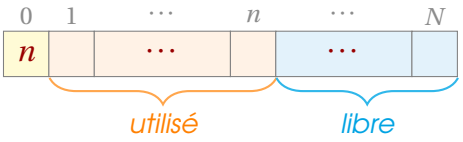
II RÉALISATION

La structure de pile est un concept abstrait. Comment la réaliser dans la pratique ? Voici plusieurs implémentations possibles. L'idée principale étant que les fonctions de bases pourront être utilisées indépendamment de l'implémentation choisie.

1 Piles à capacité finie

Une première solution est d'utiliser un tableau de taille fixée $N+1$.

Pour pouvoir empiler et dépiler, on a besoin de connaître la position du sommet de la pile, autrement dit la taille de la pile, qui est un entier que l'on peut placer en position 0 dans le tableau.



Voici les fonctions de base :

```
def pile_vide(N):
    "Renvoie une pile vide de capacité N."
    T = [None for i in range(N + 1)]
    T[0] = 0
    return T
```

```
def est_vide(p):
    "teste si p est vide et renvoie le booléen obtenu."
    return p[0] == 0
```

```
def depile(p):
    "dépile et renvoie l'élément au sommet de la pile p."
    assert not est_vide(p), "Pile vide"
    n = p[0]  # taille de la pile
    p[0] -= 1 # on dépile
    return p[n]
```

```
def empile(p, x):
   "empile l'élément x sur la pile p."
   N = len(p) - 1
   assert p[0] < N, 'Pile pleine'
   p[0] += 1
   p[p[0]] = x # on empile
```

Auxquelles on peut ajouter :

```
def taille(p):
   "renvoie la taille de pile p."
   return p[0]
```

```
def est_pleine(p):
   "teste si p est pleine et renvoie le booléen obtenu."
   return taille(p) == len(p) - 1
```

```
def sommet(p):
   "renvoie le sommet de pile p."
   assert not est_vide(p), "Pile vide"
   return p[taille(p)]
```

2 Piles non bornées

Une deuxième solution consiste à utiliser les listes Python. Il se trouve que les méthode `append` et `pop` sur les listes jouent déjà le rôle de push et pop sur les piles. Voici les fonctions de base :

```
def pile_vide():
    "Renvoie une pile vide."
    return []
```

```
def est_vide(p):
    "teste si p est vide et renvoie le booléen obtenu."
    return p == []
```

```
def depile(p):
    "dépile et renvoie l'élément au sommet de la pile p."
    assert not est_vide(p), "Pile vide"
    return p.pop()
```

```
def empile(p, x):
   "empile l'élément x sur la pile p."
   p.append(x) # on empile
```

Auxquelles on peut ajouter :

```
def taille(p):
   "renvoie la taille de pile p."
   return len(p)
```

```
def sommet(p):
   "renvoie le sommet de pile p."
   assert not est_vide(p), "Pile vide"
   return p[-1]
```

Dans la pratique, on se contente souvent de simuler une pile à l'aide d'une liste python, initialisée par `[]` et en utilisant `append` et `pop` (sans argument pour cette dernière !)

3 Une pointe de POO : définition de classes Pile

Notons qu'une façon plus propre d'implémenter les piles est d'utiliser de la programmation orientée objet.

L'initialisation se fera alors par `p = Pile(N)` ou `p = Pile()` suivant le choix d'une pile à capacité bornée ou non, puis les fonctions sont remplacées par des méthodes : `p.est_vide()`, `p.depile()` et `p.empile(x)`.

Nous avons également ajouté une méthode pour la représentation des piles dans la console.

On aurait aussi pu imaginer une extension de la capacité lorsque la pile devient pleine, sur le même principe que les listes Python.

Pour les piles à capacité finie, nous avons plutôt choisi d'utiliser un tableau de taille N et un entier à part pour gérer la taille de la pile. `T`, `taille` et `capacite` sont des attributs, tandis que `est_vide`, `est_pleine`, `depile`, `empile` et `sommet` sont des méthodes.

```
class Pile:
    "classe pile à capacité finie"

    def __init__(self, N):
        """initialisation d'une pile vide de capacité finie N"""
        self.T = [None for i in range(N)]
        self.taille = 0
        self.capacite = N

    def est_vide(self):
        """teste si la pile est vide"""
        return self.taille == 0

    def est_pleine(self):
        """teste si la pile est pleine"""
        return self.taille == self.capacite

    def depile(self):
        assert not self.est_vide(), 'Pile vide'
        n = self.taille
        self.taille -= 1
        return self.T[n - 1]

    def empile(self, x):
        "empile x sur la pile"
        assert not self.est_pleine(), 'Pile pleine'
        self.taille += 1
        self.T[self.taille - 1] = x

    def sommet(self):
        "renvoie le sommet de la pile"
        assert self.taille == 0, 'Pile vide'
        return self.T[self.taille - 1]

    def __repr__(self):
        "Pour l'affichage"
        a_afficher = ['T                  T']
        if self.taille != 0:
            for e in self.T[self.taille - 1::-1]:
                chaine = str(e)
                if len(chaine) > 15:
                    chaine = chaine[:10] + '[...]'
                a_afficher.append('| {:^15} |'.format(chaine))
        a_afficher.append('\_________________/')
        return "\n".join(a_afficher)
```

`L` est un attribut, tandis que `est_vide`, `depile`, `empile` et `sommet` sont des méthodes.

```
class Pile:
    "classe pile non bornée"

    def __init__(self):
        """initialisation d'une pile vide"""
        self.L = []

    def est_vide(self):
        """teste si la pile est vide"""
        return self.L == []

    def depile(self):
        assert not self.est_vide(), 'Pile vide'
        return self.L.pop()

    def empile(self, x):
        "empile x sur la pile"
        self.L.append(x)

    def sommet(self):
        "renvoie le sommet de la pile"
        assert not self.est_vide(), 'Pile vide'
        return self.L[-1]

    def __repr__(self):
        """Pour l'affichage"""
        a_afficher = ['T                  T']
        for e in self.L[::-1]:
            chaine = str(e)
            if len(chaine) > 15:
                chaine = chaine[:10] + '[...]'
            a_afficher.append('| {:^15} |'.format(chaine))
        a_afficher.append('\_________________/')
        return "\n".join(a_afficher)
```

III Deux applications classiques

1 Mots bien parenthésés

On souhaite écrire une fonction permettant d'abord de vérifier si une expression est bien parenthésée, et ensuite de renvoyer la liste (ou la pile) des couples correspondant aux indices de chaque parenthèse ouvrante et de la fermante qui lui correspond. Exemples de résultats :

Expression	Résultat
''	`True, []`
'()()'	`True, [(0, 1), (2, 3)]`
'(())()'	`True, [(1, 2), (0, 3), (4, 5)]`
')('	`False, []`
'(()'	`False, []`

Pour ce faire, on parcourt le mot de gauche à droite. À chaque parenthèse ouvrante, on empile son indice, et à chaque parenthèse fermante, on dépile la position de la parenthèse ouvrante correspondant, et on garde en mémoire. Si l'expression est bien parenthésée, la pile doit être vide à la fin du parcours.

On a utilisé la classe `Pile` non bornée, mais seule l'initialisation des piles change.

```
def parentheses(chaine):
    """Renvoie un couple (booleen, couples) où booleen indique si
l'expression dans chaine est bien parenthésée et si c'est le cas, couples
contient une pile de couples correspondant aux positions d'une parenthèse
ouverte et de la fermante correspondante."""
    p = Pile() # p va contenir les positions de parenthèses ouvrante au fur
    # et à mesure qu'on les rencontre.
    couples = Pile()
    for i in range(len(chaine)):
        if chaine[i] == '(':
            p.empile(i)
        elif chaine[i] == ')':
            if p.est_vide():
                return False, Pile()      # Mal parenthésé
            else:
                j = p.depile()
                couples.empile( (j, i) )
    if p.est_vide():
        return True, couples
    else:
        return False, Pile()              # Mal parenthésé
```

Et si on autorise plusieurs types de parenthèses : `(`, `{`, `[` ?

```
def ouvrante(car):
    "renvoie l'ouvrante correspondant à une fermante."
    if car == ')': return '('
    if car == ']': return '['
    if car == '}': return '{'
```

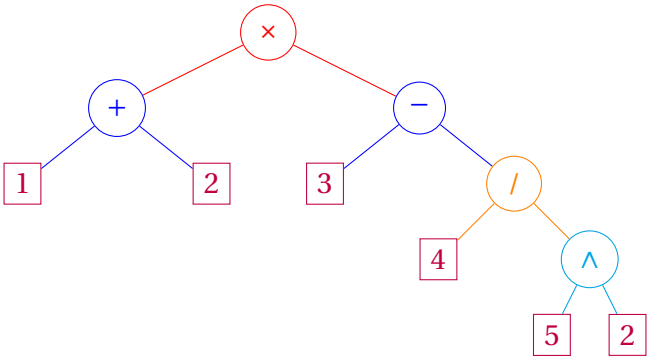
```
def parentheses_multiples(chaine):
    ouvrantes = ('(', '[', '{')
    fermantes = (')', ']', '}')
    p = Pile() # p va contenir les couples (pos. de parent. ouvr., parent.)
    couples = Pile() # au fur et à mesure qu'on les rencontre.
    for i in range(len(chaine)):
        if chaine[i] in ouvrantes:
            p.empile( (i, chaine[i]) )
        elif chaine[i] in fermantes:
            if p.est_vide():
                return False, Pile()       # Mal parenthésé
            else:
                j, parenthese = p.depile()
                if parenthese != ouvrante(chaine[i]):
                    return False, Pile() # Mal parenthésé
                else:
                    couples.empile( (j, i) )
    if p.est_vide():
        return True, couples
    else:
        return False, Pile()              # Mal parenthésé
```

2 Notation polonaise inversée

L'usage d'une pile est naturel lors de l'évaluation post-fixée d'une expression algébrique.

Le principe est le suivant : une expression algébrique, par exemple $(1+2)\times\left(3-4/\left(5^2\right)\right)$ peut être représentée avec un arbre dont les nœuds sont les opérations et les feuilles les nombres.

Ici, il s'agit d'un produit entre une somme et la différence entre un nombre et le quotient d'un nombre avec le carré d'un nombre. Cela donne l'arbre :



Le principe du parcours postfixe d'un arbre consiste à lire d'abord le sous-arbre (appelé fils) gauche, puis le fils droit, puis effectuer l'opération (qui se trouve au nœud).

Ici, cela donne :

$$\Big[\big[1\big]\big[2\big]\big(+\big)\Big]\ \Big[\ \big[3\big]\ \Big[\big[4\big]\ \big[\ \big[5\big]\ \big[2\big]\ \big(\wedge\big)\ \big]\ \big(/\big)\Big]\ \big(-\big)\Big]\ \big(\times\big)$$

L'idée est donc, pour évaluer cette expression, d'utiliser un tableau

```
[1, 2, '+', 3, 4, 5, 2, '**', '/', '-', '*']
```

correspondant à ce parcours de l'arbre.

Un avantage de cette écriture de l'expression est l'affranchissement complet de parenthésage.

Traditionnellement, les calculatrices HP utilis(ai ?)ent cette notation appelée RPN (pour Reverse Polish Notation) à l'origine parce que les machines n'étaient pas assez puissantes pour gérer les parenthésages mais qui s'avère très pratique à l'usage.

La calculatrice affiche (et gère) en permanence une pile (le sommet est affiché en bas de l'écran), et pour calculer l'expression précédente,

(1) on tape [1] [ENTER]

(2) puis [2] [ENTER]

(3) puis [+]

(4) puis [3] [ENTER] [4] [ENTER] [5] [ENTER] [2] [ENTER]

(5) puis [y^x]

(6) puis [÷]

(7) puis [−]

(8) et enfin [×].

Comme les calculatrices HP, nous allons utiliser une pile pour faire les calculs correspondant à la notation polonaise inversée à partir d'entrées stockées initialement dans un tableau.

```
def opere_bin(op, a, b):
    """renvoie le résultat de l'opérateur binaire op entre a et b"""
    if op == '+': return a + b
    if op == '-': return a - b
    if op == '*': return a * b
    if op == '/': return a / b
    if op == '**': return a ** b
```

```python
def evalue_rpn(expr):
    """évaluation postfixe de l'expression expr sous forme d'un tableau"""
    pile = []
    operateurs = ['+', '-', '*', '/', '**']

    for elem in expr:

        if elem not in operateurs:
            pile.append(elem)

        else:
            assert pile != [], "expression mal formée"
            b = pile.pop()
            assert pile != [], "expression mal formée"
            a = pile.pop()

            pile.append( opere_bin(elem, a, b) )

    resultat = pile.pop()
    assert pile == [], "expression mal formée"

    return resultat
```

(Ici, on a utilisé une liste directement pour implémenter la pile.)