Chapitre II
Piles

Les piles sont des structures de données linéaires dynamiques qui se distinguent par les conditions d'ajout et d'accès aux éléments : elles sont fondées sur le principe du « dernier arrivé, premier sorti » (principe LIFO, pour *last in, first out*). C'est le principe même de la pile d'assiette : c'est la dernière assiette posée sur la pile d'assiettes sales qui sera la première lavée.

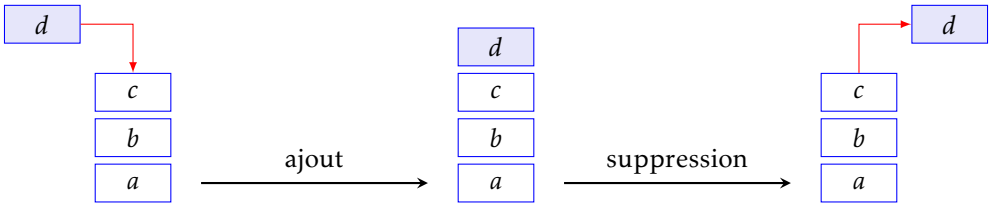

FIGURE 1 – Ajout et suppression dans une pile.

Une réalisation concrète de cette structure de données fournit, outre la structure, les fonctions suivantes :

- une fonction de création d'une pile vide ;
- deux fonctions d'ajout et de suppression *à coût constant* (nommées respectivement `push` et `pop`) ;
- une fonction vérifiant si une pile est vide.

Cette structure de données est assez pauvre, et pour cette raison n'est pas présente en tant que telle dans les langages de programmation de haut niveau comme Python. En revanche, la plupart des microprocesseurs gèrent nativement une pile, qui s'avère être une structure de donnée indispensable dans les langages machine. Nous aurons d'ailleurs l'occasion d'y revenir dans le chapitre consacré à la récursivité.

Nous allons commencer par discuter différentes manières d'implémenter en Python cette structure de données.

1. Implémentation pratique d'une pile

Pour simuler l'utilisation d'une pile en Python, nous allons avoir besoin de définir quatre fonctions :

- une fonction `pile()` qui crée une nouvelle pile vide ;
- une fonction `estVide(p)` qui prend pour argument une pile p et renvoie la valeur `True` si cette dernière est vide, et `False` sinon ;
- une procédure `push(x, p)` qui prend pour arguments un élément x et une pile p et qui *modifie* p en empilant l'élément x à son sommet ;
- une fonction `pop(p)` qui prend pour argument une pile non vide p et renvoie son sommet tout en *modifiant* p en conséquence.

On notera que les deux dernières fonctions doivent réaliser une *mutation* de la pile passée en paramètre (et renvoyer une valeur dans le cas de la seconde) et non pas créer une nouvelle pile.

Implémentation à l'aide d'une liste

Le plus simple est d'utiliser la classe `list` pour simuler une pile, car les éléments de cette classe possèdent des méthodes de mutation bien adaptées à la simulation que l'on souhaite réaliser : les méthodes append et pop.

```
def pile():
    return []

def estVide(p):
    return p == []

def push(x, p):
    p.append(x)

def pop(p):
    return p.pop()
```

On l'aura sans doute compris : avec cette simulation le sommet de la pile est situé en queue de liste ; la méthode append ajoute un élément à cette extrémité, la méthode pop supprime et renvoie la queue de la liste.

Implémentation à l'aide d'un tableau

Dans la pratique, les piles ne sont pas de hauteur infinie, et une erreur se déclenche (*Stack Overflow*) lorsque la capacité totale allouée à la pile est dépassée. Pour modéliser plus précisément ce mécanisme, on peut choisir d'implémenter une pile à l'aide d'un tableau. Puisque ceux-ci sont des structures de données statiques, nous allons devoir ajouter à la structure de données une taille maximale admissible, par exemple déclarée au moment de la création.

Avec cette simulation un indicateur est nécessaire pour connaître la case où se situe le sommet de la pile (illustration figure 2) : la première case du tableau (la case d'indice 0) désignera l'indice q de la première case disponible pour accueillir un nouvel élément dans la pile.

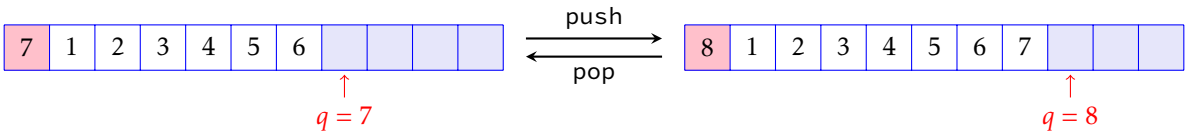

Figure 2 – Une pile implémentée à l'aide d'un tableau, l'ajout et le retrait d'un élément.

Avec cette implémentation une erreur doit se produire lorsqu'on cherche à empiler un nouvel élément sur une pile pleine (ce qu'on appelle un débordement de pile, *Stack Overflow* en anglais).

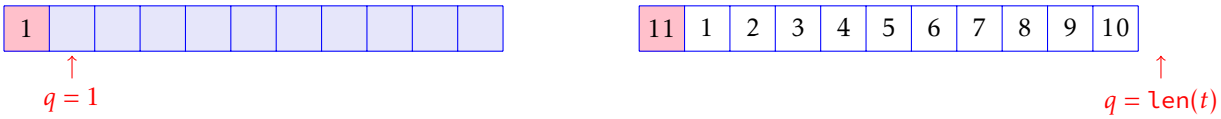

Figure 3 – Une pile vide et une pile pleine

```
def pile(n=256):        # n désigne la taille maximale allouée à la pile
    p = [None] * (n+1)
    p[0] = 1
    return p

def estVide(p):
    return p[0] == 1

def push(x, p):
    if p[0] == len(p): raise RuntimeError('Stack Overflow')
    p[p[0]] = x
    p[0] += 1

def pop(p):
    p[0] -= 1
    return p[p[0]]
```

Il faut bien distinguer la définition d'une structure de donnée abstraite (les piles) de son implémentation concrète, qui peut prendre différentes formes et qui reste en général cachée à l'utilisateur, qui ne peut interagir avec la structure de donnée que par l'intermédiaire des méthodes ou fonctions qui lui sont associées (push et pop ici).

2. Évaluation d'une expression postfixe

La notation *postfixe* 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 postfixe, la racine carrée de a sera écrite « $a\ \sqrt{}$ ». L'intérêt majeur de cette notation est qu'une expression postfixe n'est jamais ambiguë : alors que expression infixe « $1\ +\ 2\ \times\ 3$ » peut avoir deux significations : « $(1\ +\ 2)\ \times\ 3$ » ou « $1\ +\ (2\ \times\ 3)$ », ce n'est jamais le cas d'une expression postfixe, ce qui rend l'usage des parenthèses superflu : « $1\ 2\ +\ 3\ \times$ » ne peut être compris que de cette façon : « $(1\ 2\ +)\ 3\ \times$ » et « $1\ 2\ 3\ \times\ +$ » de cette façon : « $1\ (2\ 3\ \times)\ +$ ».

Nous allons montrer comment, à l'aide d'une pile, on peut évaluer très simplement une expression algébrique postfixe.

Les expressions algébriques seront ici représentée par les listes qui pourront contenir des nombres (de type `int` ou `float`) ou des fonctions à un ou deux arguments.

Ainsi, l'expression $\dfrac{1+2\sqrt{3}}{4}$ sera représentée par la liste `[1, 2, 3, sqrt, multiply, add, 4, divide]`.

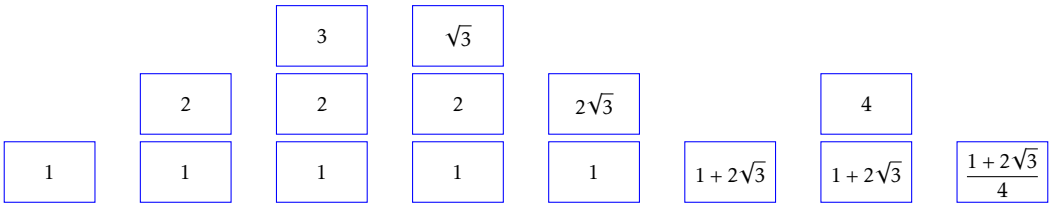

FIGURE 4 – Évolution de la pile associée à l'expression postfixe $1\ 2\ 3\ \sqrt{}\ \times\ +\ 4\ \div$

On suppose données deux listes répertoriant pour l'une les opérateurs unaires autorisés, pour l'autre les opérateurs binaires. Ces deux listes peuvent par exemple être crées en suivant le modèle :

```
from numpy import sqrt, exp, log, add, subtract, multiply, divide

op_uni = [sqrt, exp, log]
op_bin = [add, subtract, multiply, divide]
```

L'évaluation d'une expression postfixe consiste à utiliser une pile initialement vide et à parcourir les éléments de la liste représentant l'expression à évaluer en appliquant les règles suivantes :

- si l'élément est un nombre, il est empilé ;
- si l'élément est un opérateur unaire, le sommet de la pile est dépilé, l'opérateur lui est appliqué et le résultat ré-empilé ;
- si l'élément est un opérateur binaire, deux éléments de la pile sont dépilés, l'opérateur leur est appliqué et le résultat ré-empilé.

Si l'expression postfixe est correcte sur le plan syntaxique (et mathématique), à la fin du traitement de la liste la pile ne contient plus qu'un seul élément égal au résultat de l'évaluation de l'expression.

EXERCICE 1

Rédiger une fonction qui évalue une expression postfixe donnée sous forme de liste. On supposera l'expression passée en argument syntaxiquement correcte.

Un exemple d'utilisation de cette fonction :

```
In [1]: evalue([1, 2, 3, sqrt, multiply, add, 4, divide])
```

```
Out[1]: 1.1160254037844386
```

L'évolution de la pile associée à cet exemple est illustrée figure 4.

3. Parcours d'un graphe

Un *graphe orienté* est constitué d'un ensemble S de *sommets* et d'un ensemble A d'*arcs*, où A est une partie de $S \times S$.

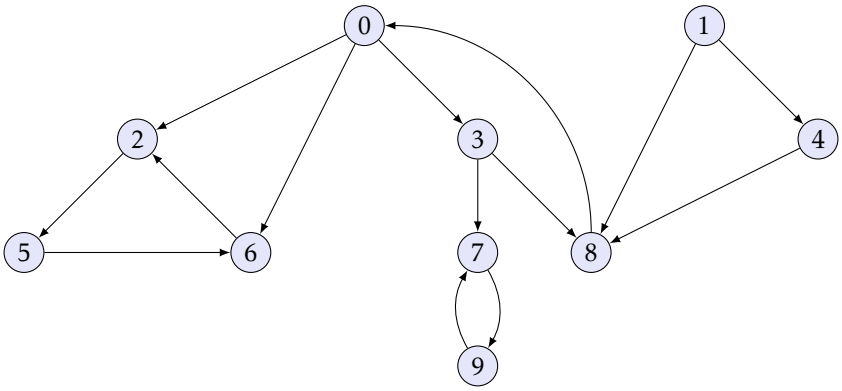

FIGURE 5 – Représentation graphique d'un graphe orienté.

Pour représenter un graphe G = (S, A), on peut utiliser une *matrice d'adjacences* (mais ce n'est pas la seule solution) : on suppose les sommets numérotés arbitrairement : $S = \{0, 1, \dots, n-1\}$ et on représente G par une matrice [1] $M = (a_{ij})$ de taille $n \times n$ pour laquelle

$$a_{ij} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{sinon} \end{cases}$$

$$M = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

FIGURE 6 – Représentations du graphe de la figure 5 par matrice d'adjacences.

Bien évidemment, le graphe sera dit *non orienté* lorsque sa matrice d'adjacences est symétrique.

Parcours en profondeur

Parcourir un graphe, c'est explorer les sommets de ce dernier de proche en proche (c'est-à-dire en suivant les arcs) à partir d'un sommet initial. Il existe plusieurs manières de parcourir un graphe ; comme son nom l'indique, le *parcours en profondeur* consiste à descendre le plus « profondément » possible dans le graphe chaque

1. Attention : contrairement aux matrices mathématiques, les indices de lignes et de colonnes sont indexées entre 0 et $n-1$ pour respecter les conventions PYTHON concernant l'indexation des tableaux.

fois que c'est possible. Lors d'un parcours en profondeur, les arcs sont donc explorés à partir du sommet v découvert le plus récemment et dont on a pas encore exploré tous les arcs qui en partent. Lorsque tous les arcs issus de v ont été explorés, on « revient en arrière » pour explorer les arcs qui partent du sommet à partir duquel v a été découvert. Ce processus se répète jusqu'à ce que tous les sommets accessibles à partir du sommet origine initial aient été découverts.

■ Description sur un exemple

Considérons le parcours en profondeur du graphe représenté figure 5 à partir du sommet 0. À partir de ce sommet, trois autres sommets sont accessibles : les sommets 2, 3 et 6.
L'exploration commence par le sommet 2 ; à partir de ce sommet seul le sommet 5 est accessible donc le parcours se poursuit par ce sommet. Pour les mêmes raisons c'est le sommet 6 qui est exploré ensuite.
À partir du sommet 6 le sommet 2 est accessible, mais ce dernier a déjà été exploré. On revient donc en arrière vers le sommet 5, qui n'a plus de sommet accessible à découvrir. On recule donc encore vers le sommet 2, puis pour les mêmes raisons vers le sommet 0.
À cette étape de l'exploration, sur les trois sommets accessibles à partir du sommet 0, deux ont étés explorés : les sommets 2 et 6. L'exploration ne peut donc se poursuivre que vers le sommet 3, qui conduira dans l'ordre, et pour les mêmes raisons que précédemment, à l'exploration des sommets 7, puis 9 et enfin 8.

On peut le deviner à la lecture de la description ci-dessus, l'exploration en profondeur utilise une pile pour y entasser les sommets en cours d'exploration : un sommet fraichement découvert est empilé, et il ne sera dépilé qu'au moment où tous ses voisins auront été complètement explorés. En outre, il sera nécessaire d'utiliser une structure de données pour y ranger les sommets découverts.

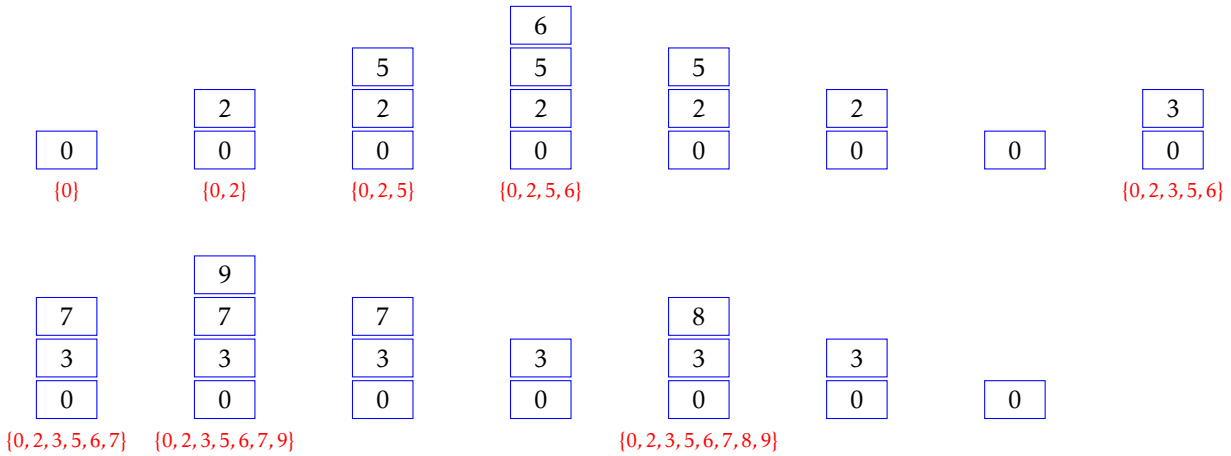

FIGURE 7 – L'évolution de la pile lors d'un parcours en profondeur. Sous la pile, l'évolution de la liste des sommets découverts.

■ Implémentation du parcours en profondeur

EXERCICE 2

Rédiger une fonction python `parcoursEnProfondeur(M, i)` qui prend pour argument un graphe représenté par sa matrice d'adjacences M et un entier i, et qui réalise un parcours en profondeur à partir du sommet i. On affichera chaque sommet au moment de sa découverte (lorsqu'il entre dans la pile) puis à la fin de son traitement (lorsqu'il en sort).

4. Exercices

Dans tous ces exercices, on supposera définies les quatre fonctions `pile`, `estVide`, `push` et `pop` décrites page 1 de ce document.

Exercice 3

Rédiger une fonction `trier(p)` qui prend en argument une pile `p` contenant des nombres entiers et qui modifie l'ordre de ses éléments de sorte qu'en fin de traitement les nombres pairs soient situés sous les nombres impairs.
Indication. Créer et utiliser une ou plusieurs piles auxiliaires.

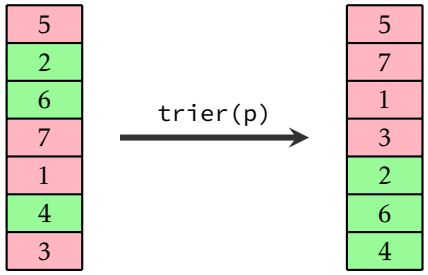

Figure 8 – Un exemple d'utilisation de la fonction `trier`.

Exercice 4

a. À l'aide d'une pile auxiliaire rédiger une fonction `insere(x, p)` qui prend pour arguments un entier x et une pile p formée d'entiers triés par ordre croissant, et qui insère x au sein de p en préservant l'ordre relatif des éléments.

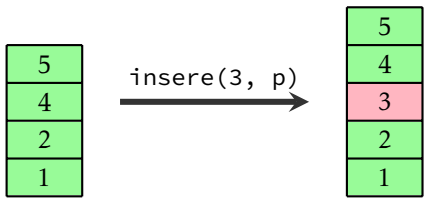

Figure 9 – Un exemple d'utilisation de la fonction `insere`.

b. En déduire une fonction `tri(p)` qui prend pour arguments une pile d'entiers et qui renvoie une nouvelle pile contenant les mêmes éléments mais triés par ordre croissant. Quelle est la complexité temporelle de cette fonction ?

Exercice 5

Une chaîne de caractères S est dite *bien parenthésée* lorsque l'une de ces conditions est vérifiée :

- S est vide ;
- S est de la forme (U) ou [U] où U est une chaîne de caractères bien parenthésée ;
- S est de la forme UV où U et V sont des chaînes de caractères bien parenthésées.

Par exemple, `'[()[]]'` est une expression bien parenthésée tandis que `'([)()]'` ne l'est pas.
Rédiger une fonction `parenthese(S)` qui prend pour argument une chaîne de caractères et renvoie `True` lorsque celle-ci est bien parenthésée, et `False` sinon.

Exercice 6

On dit qu'une permutation $(a_1 \quad a_2 \quad \cdots \quad a_n)$ de $(1 \quad 2 \quad \cdots \quad n)$ peut être engendrée par une pile lorsque il est possible, à partir de la séquence d'entrée $(1 \quad 2 \quad \cdots \quad n)$ et d'une pile (initialement vide), de produire la séquence de sortie $(a_1 \quad a_2 \quad \cdots \quad a_n)$ en utilisant les opérations suivantes :

- empiler l'élément suivant de la séquence d'entrée ;
- ou dépiler le sommet de la pile et l'imprimer à l'écran.

Par exemple, si E et D désignent respectivement chacune des deux opérations permises, la permutation $(2 \quad 3 \quad 1)$ est engendrée par la suite d'opérations : EEDEDD.

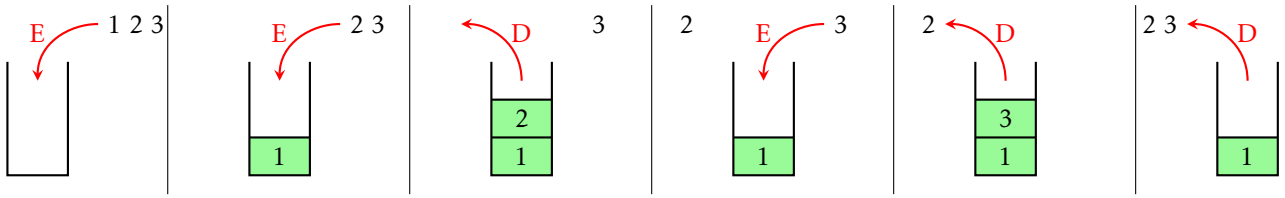

Figure 10 – La permutation $(2 \quad 3 \quad 1)$ est engendrable par une pile.

a. Parmi les deux permutations suivantes, laquelle peut être engendrée par une pile ?

$(3 \quad 5 \quad 7 \quad 6 \quad 8 \quad 4 \quad 9 \quad 2 \quad 10 \quad 1)$ $\qquad$ $(4 \quad 6 \quad 5 \quad 3 \quad 8 \quad 7 \quad 1 \quad 9 \quad 10 \quad 2)$

b. Rédiger une fonction `genere(s)` qui prend pour argument une permutation s représentée par une liste en renvoie un booléen `True` ou `False` suivant que s est engendrable par une pile ou non.